A Multi-Agent Approach to Solving $\mathcal{NP}$-Complete Problems

Christian Agerbeck, Mikael O. Hansen

Kongens Lyngby 2008
IMM-Masters Thesis-2008

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

This master's project concerns the use of multi-agent design principles in making efficient solvers for $\mathcal{NP}$-complete problems. The design of computer programs as multi-agent systems presents a new and very promising software engineering paradigm, where systems are described as individual problem-solving agents pursuing high-level goals. Recently, researchers have started to apply the multi-agent paradigm to the construction of efficient solvers for $\mathcal{NP}$-complete problems. This has resulted in very effective tools for routing problems, graph partitioning and SAT-solving.

The objective of the present project is to make further studies into the application of multi-agent principles to solving $\mathcal{NP}$-complete problems. More specifically, the project has the following two goals. First, it should result in a general discussion of the use of multi-agent approaches to solving $\mathcal{NP}$-complete problems. This should include a discussion of strengths and weaknesses compared to other approaches of solving the same problems. Second, it should result in a concrete software tool for solving $n^2 \times n^2$ Sudoku puzzles, which is known to be an $\mathcal{NP}$-complete problem. The tool should be benchmarked against other solvers for Sudoku.

Resumé

Dette eksamensprojekt beskæftiger sig med multi-agent-systemer og de design principper, der ligger bag effektive løsningmetoder til $\mathcal{NP}$-komplette problemer. Computer programmer, som bygger på multi-agent-systemer, er et nyt og lovende område inden for software udvikling. Systemerne består af individuelle agenterne, hvor hver enkelt agent sammen søger et højereстående mål. Forskerer er fornylig begyndt at anvende multi-agent-systemer til at konstruere effektive løsningsmetoder til $\mathcal{NP}$-komplette problemer. Det har resulteret i effektive værktøjer til ruteplanlægningsproblemer, graf-opdeling og SAT løsningsmetoder.

Formålet med dette projekt er at udforske anvendelse af multi-agent-systemer til at løse $\mathcal{NP}$-komplette problemer yderligere. Mere specifikt, så har projektet de følgende to mål. For det første skal opgaven indeholde en general diskussion af, hvordan multi-agent-systemer bruges til at løse $\mathcal{NP}$-komplette problemer. Dette skal inkludere en diskussion af fordele og ulemper ved at anvende multi-agent-systemer i forhold til andre fremgangsmåder, der løser de samme problemer. For det andet skal der laves et program, der kan løse en Sudoku af størrelsen $n^2 \times n^2$, som er et velkendt $\mathcal{NP}$-komplet problem. Programmet skal testet mod andre løsningsmetoder til Sudoku.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark during the period September 2007 to February 2008.

The thesis deals with $\mathcal{NP}$-complete problems and multi-agent systems. The report consists of two parts. The first part deals with the use of multi-agent approaches to solving $\mathcal{NP}$-complete problems. The second part considers the $\mathcal{NP}$-complete problem Sudoku and utilizes the experiences gathered to develop a Sudoku puzzle solver.

We would like to thank our supervisor Thomas Bolander, who has been very helpful with guidance and critical comments during this work.

Lyngby, February 2008

Christian Agerbeck & Mikael Ottesen Hansen

Contents

Abstract i

Resumé iii

Preface v

1 Introduction 1

2 $\mathcal{NP}$-complete problems 3

2.1 Introduction . . . 3

2.2 Definition . . . 4

2.3 Common $\mathcal{NP}$-complete problems . . . 5

2.4 Solving $\mathcal{NP}$-complete problems . . . 6

3 Multi-Agent Systems and $\mathcal{NP}$-complete problems 11

3.1 What is a multi-agent system? . . . 11

3.2 Multi-Agent Systems and Meta-heuristics . . . 13

3.3 Multi-Agent Systems and distributed constrains . . . 16

3.4 Multi-agent Systems with different agent types 19

3.5 Conclusion 21

4 Multi-agent systems versus common solving techniques 23

4.1 Discussion of strengths and weaknesses 23

4.2 Conclusion 25

5 Sudoku 27

5.1 Behind the puzzle 27

5.2 Solution strategy 30

5.3 Sudoku is $\mathcal{NP}$-complete 32

6 Different approaches to solve Sudoku 33

6.1 Sudoku representations 33

6.2 Solution strategies 37

7 Sudoku solver 51

7.1 Requirements 51

7.2 Analysis 52

7.3 Design 53

7.4 Implementation 60

7.5 Test 65

7.6 Discussion 70

7.7 Conclusion 72

8 Conclusion 73

8.1 Future work . . . 74

A User manual **81**

B Source code **87**

B.1 Agent Environment . . . 87

B.2 Agents . . . 92

B.3 Messaging . . . 117

B.4 Data . . . 121

B.5 Cache . . . 131

Chapter 1

Introduction

$\mathcal{NP}$-complete problems are a class of hard problems, which so far cannot be solved in polynomial time. Many problems from our everyday life are $\mathcal{NP}$-complete, and although we might not be able to find an optimal solution within reasonable time, different methods exist to find a satisfactory solution. These methods include *approximation*, where a near optimal solution can be guaranteed, *randomization*, where an optimal solution can be found with a certain probability, or *heuristics* where a good solution can be found, but where there is no guarantee, it will be found fast. For many problems it are difficult to devise a complete algorithm that guaranties a optimal solution. A higher level of heuristics called meta-heuristics can then be used to combine the solution given by heuristics and solution strategies to obtain a better solution.

From the literature of Artificial Intelligence and Distributed Problem Solving, the concept of Multi-Agent Systems (MAS) have emerged. MAS are inspired by the way humans or other biological entities interact with each other. The idea is that a MAS may be used as an intuitive approach, simulating the way humans think and interact, to construct algorithms and heuristics that can solve $\mathcal{NP}$-complete problems. The idea for using MAS to construct a faster and/or better solution is an ongoing field of research. However, there are already examples in the literature describing different conceptions of the term MAS, in regards to solving $\mathcal{NP}$-complete problems.

Multi-agent systems can be used as a liaison between different heuristics. [13] and [37] explore the advantages of using a MAS to refine and combine the solutions of meta-heuristics. The idea is to let different agents maintain responsibility of a meta-heuristic, and then in a suitable way combine and use the solutions given by

other agents for at better result of the meta-heuristic. Another approach is to use a MAS in solving constraint satisfaction problems. The idea is to divide constraints and variables between agents, and in a suitable manner let the individual agents optimize their local perspective to reach an optimal common global solution. In [16] and [22] this approach is described. The third approach is to analyze a real world instance of the problem and in this way determine what entities are at play, and how they interact to solve the problem, for then afterwards to make a MAS inspired by this interaction.

In the summer of 2005 the Sudoku puzzle took the world with storm and became a huge international hit. Today millions of people around the world are tackling one of the hardest problems in computer science, without even knowing it. Sudoku is a member of the $\mathcal{NP}$-complete subset and is therefore an ideal problem to investigate, when considering multi-agent systems to solve $\mathcal{NP}$-complete problems.

There exist already a number of solution strategies to solve the puzzle, developed by both the Sudoku puzzle enthusiasts and the academic community. The focus from the enthusiasts have been on using logical reasoning, as it is intuitive for humans, and because most of the puzzles are solvable this way. The academic community has looked upon Sudoku as a $\mathcal{NP}$-complete problem, and thereby used techniques already known for solving $\mathcal{NP}$-complete problems, as meta-heuristics and SAT solvers.

This last part of this thesis deals with the considerations in designing a multi-agent Sudoku solver, capable of solving various Sudoku puzzles. To be able to construct the solver, inspiration is first gathered from other solution approaches. This will hopefully reveal strategies that can be helpful in implementing an efficient solver. This finally results in an actual implementation of a Sudoku solver.

Chapter 2

$\mathcal{NP}$-complete problems

In this chapter the class of $\mathcal{NP}$-complete problems is presented by first giving a precise definition of the class. Afterwards examples of different problems, belonging to the class, are listed. Finally different solution strategies are presented.

2.1 Introduction

Problems that can be solved by algorithms in polynomial time are considered to be so called easy problems. For a problem of size n the time needed to find a solution is a polynomial function of n. Harder problems requires on the other hand an exponential function of n, which of course means that the execution time grows much faster than for an easy problem, when the size of the problem increases. $\mathcal{NP}$-complete problems are hard problems to solve. They belong to a class of computational problems, for which no deterministic polynomial algorithm has been found.

$\mathcal{NP}$-complete problems are a subset of the class $\mathcal{NP}$ (Non-Deterministic Polynomial). A Non-deterministic algorithm is able to find a correct solution, but it is not always guaranteed. The solution is found by making a series of guesses, and the algorithm will only arrive at a correct solution, if the right guesses are made along the way. A problem is called $\mathcal{NP}$, if its solution can be found and verified by a non-deterministic algorithm in polynomial time. The class has the following definition according to [9]:

Definition: A yes-no-problem is in $\mathcal{NP}$ if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\mathbf{X}$ the following holds:

$$\begin{array}{lcl} \text{True answer for } \mathbf{X} \text{ is} & \text{YES} & \text{then } \mathbf{P}_R[A(\mathbf{X},R)=\text{YES}]>0 \\ \text{True answer for } \mathbf{X} \text{ is} & \text{NO} & \text{then } \mathbf{P}_R[A(\mathbf{X},R)=\text{YES}]=0 \end{array}$$

where $\mathbf{P}_R[Z]$ denotes the probability of event Z over uniform distribution of R.

With other words this states that the class of $\mathcal{NP}$ problems is the set of all yes-no problems, to which there exist polynomial yes-no algorithms that verifies them. A yes-no problem is a depiction of a input set onto the set $\{yes, no\}$. An example could be the graph coloring problem, with the graph G and the following yes-no problem: is it possible to color G with k colors? Given a yes-no problem $Q : l \rightarrow \{yes, no\}$ and a set J called *certificates*, a yes-no algorithm verifies Q, if either of the two expressions are satisfied:

$$\begin{array}{ll} \forall & x \in l \text{ where } Q(x) = \text{yes} : \text{there exists } y \in J \text{ such that } A(x,y) = \text{yes} \\ \forall & x \in l \text{ where } Q(x) = \text{no} : \text{there does not exists } y \in J \text{ such that } A(x,y) = \text{yes} \end{array}$$

To clarify this further the yes-no problem described above, is considered. In this example the certificates are the different ways, the graph G can be colored. The question is then, given the graph G and k colors (x), is it possible to find a certificate y, such that there exists a k-coloring of the graph. If this is the case, is means that the yes-no algorithm verifies Q ($A(x, y) = \text{yes}$), and i.e. the problem belongs to $\mathcal{NP}$.

2.2 Definition

A precise definition of a $\mathcal{NP}$-complete problem is given in [9], where a problem P is called $\mathcal{NP}$-complete if:

$$\begin{array}{rcl} P & \in & \mathcal{NP} \\ \forall P' & \in & \mathcal{NP} : P' \leq_p P \end{array}$$

The first condition expresses that the problem belongs to the class $\mathcal{NP}$. The second condition expresses that the problem is at least as hard to solve as any problem in

$\mathcal{NP}$. The problem is $\mathcal{NP}$-complete if all other $\mathcal{NP}$ problems, are polynomial-time reducible to it. This means that a instance $p \in P$, is reducible into a new problem L with a instance l, such that the answer to l is yes, if and only if the answer to p is yes.

The fact that $\mathcal{NP}$-complete problems is reducible to other $\mathcal{NP}$-complete problems, is often used to prove that a problem is $\mathcal{NP}$-complete. This is done by first showing that the problem belongs to $\mathcal{NP}$, and then reduce the problem to a new problem that already is shown to be $\mathcal{NP}$-complete.

It has long been the subject of scientific research to determine if $\mathcal{P} \neq \mathcal{NP}$ or $\mathcal{P} = \mathcal{NP}$. It is not known whether any polynomial time algorithms will ever be found for $\mathcal{NP}$-complete problems. If one is found, it means that $P = NP$, since any problem belonging to this class, can be recast into any other member of the class.

2.3 Common $\mathcal{NP}$-complete problems

The list of $\mathcal{NP}$-complete is long, there exits several thousands problems. They are represented within many different areas as graph theory, network, scheduling, games and puzzles etc..

Constraint Satisfaction Problems (CSP) are mathematical problems, where some constraints among a group of variables are given. The goal is to find the value for all variables that satisfy the given constraints. Many problems can be viewed as CSP's. A CSP is not necessarily a $\mathcal{NP}$-complete problem, but many problems in this class is $\mathcal{NP}$-complete.

A *Satisfiability* (SAT) problem is a problem of determining, if the variables in a boolean formula can be assigned a value, so the formula value evaluates to true. The problem can be significantly restricted by the use of different properties and complied into a propositional formula in conjunctive normal form (CNF). A CNF formula is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional variable or its negation. A special case of the SAT problem is the 3-SAT problem, which means that each clause contains three literals. The 2-SAT problem is another special case, where each clause contains two literals.

The *Job Shop Scheduling* problem is another $\mathcal{NP}$-complete problem. It consist of a finite set of n jobs, where each job consists of a chain of operations. In addition it consist of a finite set of machines m, where each machine can handle at most one operation at a time. At the same time each operation needs to be processed during an uninterrupted period of a given length on a given machine. The purpose

is then to find a schedule, that is, an allocation of the operations to time intervals to machines, that has minimal length.

The *Travelling Salesman* problem (TSP) is a well known $\mathcal{NP}$-complete problem. It is the problem of finding the least-cost round-trip route that visits a number of cities exactly once and then returns to the starting city. The given information is the cities and the costs of travelling from any city to any other city. In the *M-TSP* the m-salesman has to cover the given cities and each city must be visited by exactly one salesman. Every salesman starts from the same city, called depot, and must return at the end of his journey to this city again. The Vehicle Routing Problem (VRP) is the m-TSP, where a demand is associated with each city or customer and each vehicle has a certain capacity.

The *k-Graph Partitioning* problem is also a $\mathcal{NP}$-complete problem. It has the following definition. Given a graph $G = (V, E)$, where V is the set of vertex and E the set of edges that determines the connectivity between the nodes. Both vertex and edges can be weighted, where $|v|$ is the weight of a vertex v, and $|e|$ is the weight of edge e. Then, the graph partitioning problem consists on dividing G into k disjoint partitions. The goal is minimize the number of cuts in the edges of the partition, and on the other hand reduce the imbalance of the weight of the sub domains. The weight of a sub domain is the sum of the weights of the vertex allocated in it.

The *vertex cover problem* for an undirected graph $G = (V, E)$ is a subset S of its vertices such that each edge has at least one endpoint in S. In other words, for each edge ab in E, one of vertices, a and b, must be an element of S.

2.4 Solving $\mathcal{NP}$-complete problems

There exits a number of techniques that can be used to solve $\mathcal{NP}$-complete problems. The following list contains some of the well known techniques.

- Approximation
- Randomization
- Heuristics

Approximation

In some $\mathcal{NP}$-complete problems it may be enough to find a near optimal solution to get a satisfactory result. An algorithm that returns a near optimal solutions is called an approximation algorithm cf. [7]. The reason for finding a near optimal solution, instead of an exact solution, is the computation cost, as it may be possible to find a near optimal solution in polynomial time.

The travelling salesman and vertex cover problem are both problems, where a near optimal solution could resolve in a satisfactory result. In section 35.5 in [7] approximation algorithms are presented that can yield a near optimal solution for both problems. The approximation algorithm for the vertex cover problem works the following way: Find an uncovered edge and add both endpoints to the vertex cover and remove them from the graph, until no vertices remain. This is constant factor approximation algorithm with a factor of 2, since the cover is at most twice as large as the minimum cover.

Randomization

A randomized algorithm is an algorithm that can make calls to a random number generator during the execution of the algorithm. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour, in the hope of achieving good performance in the average case (chapter 5 in [7]).

A motivation for using this approach is exemplified in the following. Consider the problem of finding an 1 in an array of n elements, where the one half consists of 1's and the second half of 2's. The obvious approach is to look at each element of the array, but a fast search cannot be guaranteed on all possible inputs. If e.g. the array is ordered with 2's first, it would take $n/2$ before the first 1 is found. On the other hand, if the array elements are checked at random, then a 1 is quickly found with high probability, whatever the input is.

Monte Carlo and *Las Vegas* are two kind of randomized algorithms. A Monte Carlo algorithm runs for a fixed number of steps for each input and produces an answer that is correct with a bounded probability. On the other hand, a Las Vegas algorithm always produces the correct answer, but its runtime for each input is a random variable whose expectation is bounded. See more on these two algorithms and other randomized algorithms in [28].

Heuristics

Heuristics may give nearly the optimal solution or provide a solution for some instances of the problem, but not all. In other words, a heuristic algorithm gives up finding the optimal solution for an improvement in run time.

There is a class of general heuristic strategies called meta-heuristics, which often use randomized search. They can be applied to a wide range of problems, but good performance is never guaranteed. Tabu search, simulated annealing, genetic algorithms, local search and ant colony optimization are examples of different meta-heuristics. See more in [11] that present many different meta-heuristics, but also gives practical guides for implementation.

Examples of solution strategies

Within the area of finding new solution strategies to solve $\mathcal{NP}$-complete problems, there exits many approaches. The objective of this section is to give an insight into two solution strategies for two well known $\mathcal{NP}$-complete problems. The problems described are chosen, because they are similar to the ones that also have a multi-agent solution approach. Additionally the criteria of the selection is that the articles are recent and leading within their field. In the following the articles [26] and [10] are presented.

Vehicle routing problem

In [26] the VRP is studied. The solution strategy is based on, what the paper present as, bilevel programming, which is a heuristic solution strategy. This formulation involves that the original problem is separated in two different sub problems. There exits a generalized assignment problem for the assignment of the customers to vehicles, and a TSP for the routing of the vehicles. In each of the two sub problems two different meta-heuristics are used in continuation of each other. In the first part a genetic algorithm is used for calculating the population of the most promising assignments of customers to vehicles. The second part solves a TSP independently for each member of the population and for each assignment to vehicles. To solve the TSP's, an algorithm called MPNS-GRASP (Multiple Phase Neighborhood Search-Greedy Randomized Adaptive Search Procedure) is used, which is a variant of the GRASP algorithm that again is a modern variant of the DPLL algorithm (explained below).

GRASP is a meta-heuristic used typically for combinatorial problems in which each iteration consists basically of two steps: construction and local search. The con-

struction step builds a feasible solution by a greedy randomized algorithm, and the subsequent search step improves the solution by local search. The best overall result is used. In [8] a detailed description of the algorithm can be found.

After the MPNS-GRASP is used, the solution is improved with yet another meta-heuristic, called ENS (Expanding neighborhood search), which is an advanced variant of local search. This meta-heuristic is explained in [25]. When this phase of the algorithm is finished, it starts all over with the solution found so far. This is continued until a satisfactory result is reached.

The experimental results show that it so far is the fastest algorithm among other meta-heuristics to solve VRP. Compared with other well known algorithms to solve VRP, it is ranked in the tenth place among 36 algorithms.

Satisfiability problem

There exits numerous algorithms to solve SAT problems. Many of them are modern variants of the *DPLL* algorithm, which is a complete backtracking algorithm used to solve SAT problems in CNF. One of these modern variants is the *Chaff* algorithm ([27]), which in resent years has formed the basis of some of fastest SAT solvers, as the *zCahff* algorithm ([10]).

The DPLL algorithm works by first assigning a literal a truth value. Then the formula is simplified, by removing all clauses which become true, and all literals that become false according to the assumption made in the first step. Additionally it uses the two rules, *Unit propagation* and the *Pure literal elimination*, to further simplify the formula. Afterwards a recursive procedure checks whether the formula is satisfied. If the verification succeeds the next literal is assigned a value, else the same recursive procedure is done again, but with the opposite truth value assigned to the literal.

The zChaff algorithm adds a numbers of features to the DPLL algorithm to make the solver more efficient. The *two watched literal scheme* and *clause learning* are two of them. The two watched literal scheme, explained in [27], is used to reduces the total number of memory accesses. Clause learning is to find and record conflict clauses. A conflict clause represents an assignment to a subset of the variables from the problem that can never be part of a solution. That is, finding and recording conflict clauses prunes previously discovered sections of the search tree that can never contain a solution. This can help improve the performance of the search. These are only two of the features, but there exits more initiatives, which are explained in [10].

The zChaff algorithms has won the prize for best complete solver in the industrial category in the 2005 SAT Competition. It was also among the three best algorithms

in the most recent competition in 2007.

Chapter 3

Multi-Agent Systems and $\mathcal{NP}$-complete problems

The design of computer programs as multi-agent systems to solve $\mathcal{NP}$-complete problems presents a new and very promising software engineering paradigm. Sofar there exits only few articles deling with the subject ([13], [4], [37], [14], [16] and [19]). These articles deals with the following $\mathcal{NP}$-complete problems: k-Graph Partitioning Problem, Job Shop Scheduling Problem, Travelling Salesman, SAT problem and Vehicle Routing Problem with Time Window. In this chapter a sort description of each article will be given, but their will also be an overall analysis of the potential a MAS processes to solve $\mathcal{NP}$-complete problems.

3.1 What is a multi-agent system?

It is assumed that the reader has some knowledge about agents and *multi-agent systems* (MAS), and therefore this section will only give a brief description of agents and multi-agent systems. Since a multi-agent system is a system consisting of individual agents, its necessary first to define what an agent is. The following definition is given in [36]:

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.

A MAS is a composed system of several agents which interact and work together in order to archive certain goals. Their interactions can be either co-operative or selfish. That is, the agents can share a common goal, or they can pursue their own interests. The typical characteristics of MAS's are that each agent has incomplete information or capabilities for solving the problem. I.e. each agent can have a local perception of the global state and need to co-operate in an autonomous and asynchronous way with other agents in order to meet the goals of the global system.

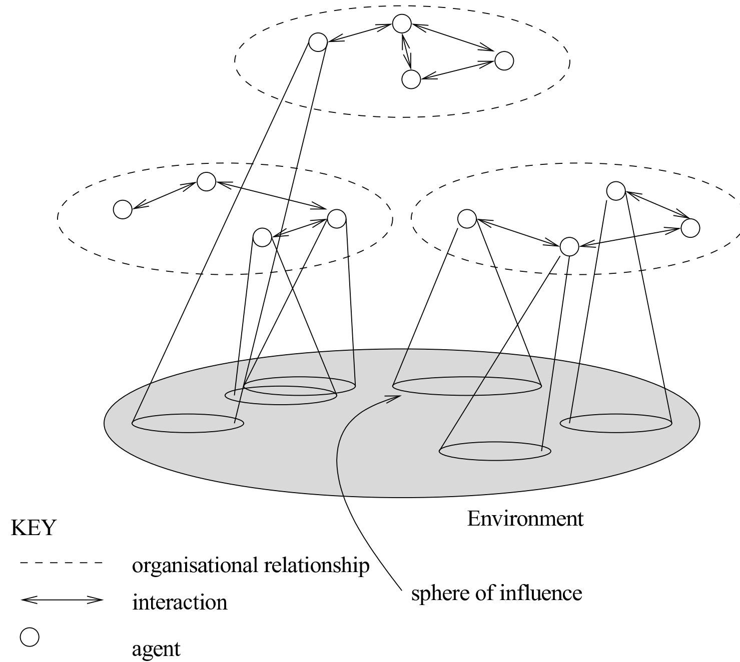


Figure 3.1: Typical structure of a MAS. Figure from [36].

To get a more illustratively presentation of a MAS, see figure 3.1. Here is the typical structure of a MAS shown. The system contains multiple agents who interact through a communication protocol, which are indicated with the double pointing arrows. The agents are able to act in the environment (grey sphere), but with different influence on the environment. The *spheres of influence* show the different parts of the environment the agents have influence over. These spheres may coincide in some cases, which may give rise to dependency relationships between the agents. In [36] they give the example that two agents may both be able to move through a door, but may not be able to do so simultaneously. Additionally agents will typically be linked by other relationships, which is show with the punctuated sphere. This could be that an agent is the boss of another.

In the following we try to analyze the common characteristics when using MAS in solving $\mathcal{NP}$-complete problems. The usage of MAS when solving $\mathcal{NP}$-complete problems may differ from the intuitive perception of MAS. In the following we give

an overview of the different categories of multi-agent systems when solving $\mathcal{NP}$-complete problems.

3.2 Multi-Agent Systems and Meta-heuristics

3.2.1 Introduction

As previously described meta-heuristics are often used when solving $\mathcal{NP}$-complete constraint optimization problems. But although they provide the possibility of finding a feasible solution, they might not always do so within a reasonable time. The reason for this is that heuristic search algorithms on large optimization problems often get trapped in local optima (or minima), which they can not escape in reasonable time. These local optima (or minima) are not necessarily the same for the different heuristic search algorithms, as they may have different search neighbourhoods ([4] and [37]).

An approach to improve the behaviour of a heuristic search algorithm is to use the multi-agent paradigm to combine different meta-heuristics, so they in a co-operative manner can complement and help each other in avoiding local optima. This approach is described in [13], [4] and [37], which will be analysed in the following.

3.2.2 Analysis

The three articles ([13], [4] and [37]) are all similar in the aspect that they all try to develop a new approach to solving a specific $\mathcal{NP}$-complete problem. Furthermore they all have identified that meta-heuristics are able to solve the given problem, but has some shortcomings in terms of getting captured in local optima (or minima). All three articles suggests a multi-agent system to find a better and quicker solution.

In [13] two approaches to solving the *k-Graph Partitioning Problem* (k-GPP) are presented: the COSATS and the X-COSATS system. The idea in both is that the multi-agent system consists of two agents: an agent implementing Simulated Annealing and an agent implementing Tabu Search. The two agents co-operate by providing each other with its local information about the search landscape. In each agent iteration the meta-heuristic agents values its best found solution from some evaluation criteria, and then gives its best solution to the other agent. The other agent then takes the best from its own solution and the best from the received solution and uses the new combined solution, as a starting point for its next search. The idea is then that the agents are going to converge their search into a search

area, where both Simulated Annealing and Tabu Search find best solutions. The other approach, X-COSATS, extends COSATS by adding a third agent, which acts as a middleman between the two original agents. The crossover agent creates a new starting point for both agents based on their found solutions and a crossover operator. The test results show that the co-operation between the two meta-heuristics actually yields a better solution, than can be obtained by the two meta-heuristics separately. When also applying the genetic crossover principle, the solutions found are not always better than the solutions found by COSATS. The article concludes that the co-operation scheme has proven to be a good method for avoiding being trapped in a local optima (or minima).

In [4] a similar approach is presented, in order to solve a *Job Shop Scheduling problem*(JSSP). Here a MAS called ATeams is described. The principle in ATeams is that a number of asynchronous autonomous agent co-operate on a shared solution variable. The solution variable is in fact a population of solutions to the JSSP problem, where each agent independent of each other can pick a solution from the population and try to improve it. Each agent implements a meta-heuristic algorithm, which it uses to try to improve the solution. In this implementation a couple of meta-heuristic algorithms such as Simulated Annealing, Tabu Search and Genetic Algorithms are suggested as individual agents. Each agent modifies a solution from the population concurrently and afterwards returns the solution into the population, based on some predefined rules. A configuration of the system is e.g that the solutions are put back into the population in a hill-climbing manner, meaning that only improved solutions can be returned to the population. Another configuration is that solutions are put back no matter the improvement. In order to ensure that the solution population does not contain bad solutions, a destroyer agent is introduced to remove *bad* solutions in respect to some criteria. The ATeams can be regarded as an advanced technique for creating hybrid algorithms. In the experiments in [4] it is stated that the ATeams are not always effective. This is shown by a single agent that performs better than a combined team. [4] suggests that this is due to the similarity in the implemented agents. For a ATeam system to be effective, it is necessary that the agents are diverse and should explore the search space in different ways in order to avoid being captured in the same local minima.

The last of the three articles ([37]) is designing a multi-agent system for solving the Travelling Salesman Problem (TSP). The ideas presented are to use the multi-agent system as a way to combine the three meta-heuristics: Ant Colony Optimization (ACO), Genetic Algorithms (GA) and Local Search (LS). The responsibility of the different meta-heuristics are divided into agent tasks. The agents then work in a sequential manner on a group of solutions. The work flow is that the ACO agent, given a list of cities, creates a group of solutions to the TSP. Afterwards a group of agents, implementing GA principles such as crossover and mutation, optimize the solutions in the solution group. Lastly the local search agent picks the best found solution from the group and tries to optimize it by applying a search heuris-

tic in local parts of the solution. In this sequential manner the system generates a group of solutions, optimize the entire solution group and then picks the best solution and tries to optimize it further. As long as no found solution fulfils the end condition, the process is started over. The system can therefore be described as a MAS implementing a hybrid meta-heuristic between ACO, GA and local search. The results show that the described systems performs well compared to other ACO systems. However the test do not show significant better performance of the hybrid multi-agent system over the other ACO systems.

The three articles show how multi-agent systems can be interpreted as a paradigm to combine different heuristic algorithms in solving $\mathcal{NP}$-complete problems, in the following the aspects of this will be discussed.

3.2.3 Discussion

In all three articles it is described how multi-agent systems can be used to combine meta-heuristics into a co-operative hybrid system. The reasons for combining meta-heuristics, are to use the different strengths of the individual meta-heuristics in order to minimize the weaknesses. [31] argues that a common reason for hybridization is to ensure exploration, and exploitation:

> Two competing goals govern the design of a metaheuristic: exploration and exploitation. Exploration is needed to ensure that every part of the space is searched enough to provide a reliable estimate of the global optimum. Exploitation is important since the refinement of the current solution will often produce a better solution.

Population based meta-heuristics, such as Genetic Algorithms and Ant Colony Optimization, are very powerful in exploring the search space, where they are weak in exploiting the found solutions. Whereas local search heuristics such as Simulated Annealing and Tabu Search, are powerful in exploiting solutions. Since the two types of meta-heuristics have complementary strengths and weaknesses, combining them may yield a better solution.

Both [4] and [37] combine a population based meta-heuristic with a local search heuristic, in their multi-agent system, and gain the advantages of combining exploration and exploitation. In [13] the approach is slightly different, since the co-operation is between two local search heuristics.

Is hybrid meta-heuristics multi-agent systems?

Hybrid meta-heuristics need not to be implemented as a multi-agent system, but one obvious benefit of using multi-agent system for this, is the ability to quickly change the meta-heuristics used for the hybridization. The important thing to notice though, is that although all implementations shows a better performance, a similar improvement in performance could probably have been obtained, by implementing the hybrid result as a single agent. E.g. in [37] its is obvious that the sequential manner of the system could be obtained by one single agent, performing different tasks during the solution process. In [13] it is possible that a single agent running Simulated Annealing and then Tabu Search in a relay manner, could show the same type of improvements, over Simulated Annealing and Tabu Search run separately. Last the ATeams, in [4], could also be implemented as single agents performing the different heuristic algorithms in some sequence, and probably also yield similar good results.

What is the motivation for using MAS?

The obvious in [13] and [4] is that you receive a gain in computational performance by running the algorithms in parallel. Furthermore the systems is easier to maintain, since a single agent should be independent of the others, and could be replaced by an agent implementing a different heuristic without affecting the stability of the system. This could be more difficult if the heuristics where all combined in a single hybrid agent.

3.3 Multi-Agent Systems and distributed constrains

3.3.1 Introduction

Distributed or multi-agent problem solving extends classical problem solving techniques to domains, where several agents can plan and act together. There exist many recent developments in this field that range over different approaches for distributed solving algorithms and distributed plan execution processes. One of the reasons for using distributed solving is that it is the most appropriate way to tackle certain kind of problems. Specially those where a centralized solving is infeasible. An area where this approach has been used is to solve $\mathcal{NP}$-complete CSP's. If a CSP problem is distributed among a number of agents it is called a *distributed constraint satisfaction problem* (DCSP) cf. [33]. In a DSCP each agent is given the responsibility for setting the value of its own allocated variable. The agents do not know the values of any other variable, but can communicate with other agents to determine

the correct value for its variable. Some of the most popular DCSP algorithms are the *asynchronous backtracking* (ABT) and *asynchronous weak-commitment search* (AWC).

The *distributed constraint optimization problem* (DCOP) is similar to the DSCP except that the goal is to minimize the value of the constraint violations. The definition for the problem is the following. Given a set of variables $x_1, x_2, \ldots, x_n$ with domains $D_1, D_2, \ldots, D_n$ and a set of constraints, then find a assignment for all the variables such that the sum of the constraints is minimized. The most common approach to solving DCOP's is to use a branch and bound algorithm. See more on DCSP and DCOP in see [33].

3.3.2 Analysis

To clarify the use of this type of multi-agent systems to solve CSP problems, two articles are analyzed. They are chosen because both of them describe a multi-agent system that can solve a $\mathcal{NP}$-complete problem. Both articles, [14] and [16], work with the SAT problem that is a well know $\mathcal{NP}$-complete problem.

In [14] the authors have developed a algorithm for solving a SAT problem. They have distributed the variables and constraints among multiple agents to transform the original SAT problem into a DCSP. It is not a new idea to distribute the problem, there are numerous distributed constraint satisfaction algorithms. ABT and AWC are two of them as mentioned above. These algorithms and their descendants contain two problems. Firstly the agents do not mutually exclude their undesirable values, and simultaneously decide the values to their variables. Secondly both algorithms can produce a huge number of *nogood* messages, when the problem gets critical hard (the nogood messages are used by both algorithms to report to the other agents that it cannot find a value for its variable). This means that both can consume a lot of memory to record these nogoods. The algorithm described in [14] does not have neither of these problems.

The algorithm works in the following way. Initially each agent is assigned multiple local variables and the relevant clauses to the variables. Then a local search procedure is performed to determine values that give a possible improvement in the weighted sum of violated constraints. Afterwards the agents exchange these values to resolve any emerged conflicts. This is done by redrawing any values that increase the total weighted sum of violated clauses over the agents. The remaining values are then sat. This is then repeated until a solution is found. This approach is closely released to the *distributed breakout* algorithms mentioned in [33] and is different from the backtracking idea used in ABT and AWC, in the sense that this is a hill-climbing strategy. All hill-climbing algorithms suffer from the problem of local minima (or maxima), but they have bypassed this problem by finding what

they call a quasi-local minimum. From [33] they give the following definition:

> An agent x_i is in a quasi-local minimum if it is violating some constraint and neither it nor any of its neighbors can make a change that results in lower total cost for the system.

They authors have later on improved their algorithm to make a even better hill-climbing strategy, by applying a *random walk* (see [15]). This decreases the chance for the algorithm to get stuck in a local minima (or maxima).

The experimental results show that both algorithms perform least as good and often better than the well known algorithms. The improved one always find a solution for 3-SAT problems, whereas the old one cannot solve all 3-SAT problems.

The algorithm described in [16] has many similarities with the previous one. The authors also tries to solve a SAT problem by means of dividing variable into groups, and then represent each group with an agent. After the agent has been randomly assigned a group, the agent system chooses their movement in their local search space by assigning them one of three different search strategies: *random-move*, *best-move* and *better-move*. The agent system will keep on dispatching agent until a solution is found, or a certain threshold value is reached. The difference between this approach and the previous one is the way the agents search in their local space. So all in all the general approach are more or less the same. In this article they also obtain comparable result with other popular algorithms.

The major difference between the two described algorithms and the solution strategy for a DCSP is that instead of using one agent per variable, they use multiple. This means that the communication cost is lower than for a classic DCSP.

3.3.3 Discussion

The essence of the different algorithms described is that they all perform a distributed search, where each agent has some local information. The goal is to get all agents to set themselves to a state, such that the set of states in the system are optimal. The agents can at any time take any of their available actions, but the utility of their actions depends on the actions of others.

A problem is easy to represent by a MAS, if the problem has a structure that makes it easy to transform to either DCSP or DCOP. Imagine a dinner party where the guests are told to sit next to persons of the opposite sex. Each guest must find a seat around the table, but the seat depends on the location of the other guests. In this example each guest can be regarded as an agent with local information, where

its actions depends on the actions of the other agents. This is a problem that is easily transformed to a distributed constraint satisfaction problem.

Applying this type of multi-agent system could on the other hand also be a choice, if the problem is to hard to solve single handed. Splitting the problem up in smaller sub problems could help solve the problem quicker as each agent would run in parallel. This is perhaps not always an advantage, since the communication cost between the agents can overshadow the effect of running the program in parallel. This is e.g. a big topic in [14], where it is attempted to use multiple agents per variable to decrease the communication cost.

3.4 Multi-agent Systems with different agent types

3.4.1 Introduction

Another approach when using multi-agent systems to solving $\mathcal{NP}$-complete problems, is to analyse a real world instance of the problem. In this way determine what entities are at play and how they interact to solve the problem, for then afterwards to make a MAS inspired by this interaction. This method is closely related to the abstraction of DCSP and DCOP, in the sense that the problem is distributed among multiple agents, with responsibility for local parts of the problem.

For most $\mathcal{NP}$-complete problems it is not hard to determine a real world instance. In [19] the *Vehicle Routing Problem with Time Window* (VRPTW) is an example of a $\mathcal{NP}$-complete problem, which has a real world instance. This is used to describe a MAS for solving the VRPTW, which will be analyzed in the following.

3.4.2 Analysis

In [19] the multi-agent system is meant as an optimization scheme on an existing solution, already determined by use of a heuristic. The VRPTW considered has three types of entities: customers, routes and a central depot. The MAS proposed looks at each of these entities and regards them as autonomous agents. It threats each of the customers, the routes and the global planner as an unique agent and allows them to exchange information.

Each separate agent (customers and route agent) has control of its own state, and is thereby provided with some functions, it can perform in order to change its state in the direction of a local objective. The goal of the complete system is that all the agents have a local objective that they pursue, and in doing so the global system

should also approach its objective. To help the local agents in co-operating towards the common global objective, the planner agent controls the global environment state, by guiding the local agents in the most desired direction. In practice this is done by a *move-pool*, where the planner agent collects desired moves from the local agents. A move is a change of a local agents state. An example of this is that a customer agent makes a complete search through the solution space, and determines at which route and time slot it minimizes the global objective function. Every agent makes a proposition of where and how it can minimize the objective function best. The planner agent then collects these suggestions and chooses the move that is best for the global solution and lets the corresponding agent change its state. Additionally the planner agent possesses the ability to optimize the global solution, by optimizing the routes with a heuristic and trimming the problem by removing *bad* routes.

In order to maintain a reasonable time complexity of the entire system, the planner agent only requests move propositions in a given interval, and then caches the result. This is done since the agent performs a complete search, in each agent proposition. Likewise the planner agent performs its optimization heuristics periodically in order to improve the solution from a global perspective.

To test the performance of the algorithm it is tested against a number of well known solvers which implement meta-heuristics as simulated annealing, tabu search, ant colony optimization, and genetic algorithms. The tests show that the system is able to obtain comparable results with the listed well known solvers, but do not show a remarkable improvement.

3.4.3 Discussion

At the first glance this approach looks very similar to the distributed constraint optimization approach. However there are a number of significant differences. The construction of the solution differs, since this approach optimizes on an initial solution in contrast to the DCOP which constructs the solution it self. Secondly the structure of the MAS differs in the way agents maintain their own state. In the DCSP and DCOP approach the agents communicate directly without any global control but in this system a planner agent maintains the control of the system. This means that the rest of the agents are not completely autonomous, since their actions are dependent on the choices of the planner agent.

The system described shows good results and give the indication that other related problems could be solved in a similar way. It is obvious that the system has room for improvement. E.g. the cached propositions, which are connected to a certain state of the environment. This state changes after a move has been performed, hence will the moves proposed to the planner agent gradually be outdated, since the state of

the environment changes every time, an agent performs a move. It would be optimal if the moves, the planner bases its decisions upon, where on a updated environment. However in this configuration, it is difficult to see a way to solve this without causing a dramatically increase in the computational cost. Likewise it might be proven that it is unnecessary to have up-to-date moves for every environment state, in order to find a satisfactory solution.

Generally this approach provides a apprehensible method for dividing the solution of a $\mathcal{NP}$-complete problem into a MAS, although the abstraction of agents may be different of what one might expect. E.g. its maybe not intuitively obvious that a route could be regarded as an agent, however when looking at the complete system it makes perfect sense. Although the system in the article is not matured, it provides an indication that multi-agent abstractions in solving $\mathcal{NP}$-complete problems need not to be inspired by either hybrid meta-heuristics or distributed constraints, but can be inspired by the problem entities and the real world instances of the problem.

3.5 Conclusion

It is obvious that multi-agent systems can be used in solving of $\mathcal{NP}$-complete problems. The different applications of multi-agent systems handled in this chapter show that multi-agent systems is still a very broad an abstract term.

In multi-agent systems and meta-heuristics it is important to notice, that the advantage of creating hybrid heuristics only exists if the different meta-heuristic parts can be chosen in a manner where they complement each other. It is only interesting to create a hybridization if a synergistic effect can be seen, otherwise the best of the meta-heuristic parts could have been used alone instead.

Often it is obvious to regard $\mathcal{NP}$-complete problems as DCSP and DCOP, which means that the advantages of a fully distributed algorithm can be used. That is allocating different parts of the problem between the multiple agents and let them run asynchronous to help one another solve the problem. The MAS paradigm, if used properly, provides the possibility that a solution is found quicker than with a single agent sequential approach. The disadvantage is the communication cost. Therefore when using the MAS paradigm it is important to consider how much the problem is divided and distributed, since a to fine-grained division might result in an overhead in communication cost.

The last type of MAS approaches shows, that it is also possible to get satisfactory solutions to a $\mathcal{NP}$-complete problem, by using a different abstraction of the MAS, when dividing the problem. This approach contains aspects of the two other approaches. It combines the functionality of the MAS with a heuristic local-search,

which is somehow similar to section 3.2, in the way that it uses heuristics to improve the solution when the environment has changed. Likewise the division of the problem entities into agents have similarities with section 3.3, in the aspect that each entity has responsibility over its own constraints.

The use of the multi-agent paradigm when designing algorithms for $\mathcal{NP}$-complete problems is an ongoing and new field of study. In our study of the subject, we have found that multi-agent solutions to $\mathcal{NP}$-complete problems fall into the three mentioned categories. The application of the multi-agent paradigm in the three method varies, however they share a common abstraction of the definition of multi-agent systems.

Chapter 4

Multi-agent systems versus common solving techniques

In chapter 2 and 3 different solving techniques to $\mathcal{NP}$-complete problems are presented. The systems are designed on the basis of respectively single-agent systems (common techniques) and multi-agent systems. The following contains a discussion of the strength and weaknesses of a multi-agent approach compared to the common solving techniques. The discussion consist of two parts. The first part concerns the difference between the approaches described in the two chapters and the second part highlights some of the general strengths and weaknesses in a MAS.

4.1 Discussion of strengths and weaknesses

The algorithm described in section 2.4 has a strong correlation with the algorithms presented in section 3.2, as all of them use a combination of meta-heuristics to solve the problems. In fact, the basis principle is the same, except that the algorithms in section 3.2 run the meta-heuristics in parallel. However, this could have been done with a sequential approach likewise, as mentioned previously. Running the meta-heuristics in parallel could result in a a gain in computational performance, but the results achieved in section 2.4 produces no reason to do so. It is one of the fastest approaches yet, compared with other meta-heuristics.

The problem worked on in section 2.4 is almost similar to the problem solved in section 3.4, but the way to go about it is completely different. Instead of representing each meta-heuristic with an agent, a different abstraction of the MAS is used, where

each problem entity is represented by an agent. It is therefore irrelevant to compare the two solution strategies. It is though important to mention that this system design in section 3.4 is innovative, in the sense that it uses a different abstraction of the MAS, when dividing the problem. It is undoubtedly slower than the algorithm in 2.4, but could be an interesting approach in the future, when optimized further.

The zChaff algorithm, described in section 2.4, is also comparable, with some of the MAS approaches from section 3.3, in the sense that they solve the SAT problem. There is though a big difference between the general approaches. The zChaff algorithm is highly optimized to solve the SAT problem. It uses advanced tools to solve the problem, whereas the multi-agent systems are based on more simple backtracking procedures. However, one might think that the distributed system with time could make use of some of these advanced procedures, but now it seems that MAS cannot catch up with the SAT solvers.

In section 3.2 it was argued that some of the approaches likewise could have been sequential approaches. However, in some domains it it necessary to use a MAS, when designing the system.

Domains

It is not always obvious to use a MAS when designing a system. There are some situations for which it is particular appropriate, and for other where it is not. However in some cases the domain requires it. Consider the problem described in section 3.3.3, but now each guest independently decides who they want to sit next to. This can only be modelled as a MAS, since the MAS is needed to handle their interaction. Additionally each person has different criterion to where they will sit, which must be represented by different agents, if their criterion are to be justly considered.

An example of a domain that does not require a MAS, but where it could be appropriate, is the problem described in section 3.4. Here the problem can be divided among multiple agents fairly straight forward. However, in situations where this subdivision is not obvious, it would be foolish to force the system design to be a MAS. This is due to that a single agent probably could do the same job as fast as a MAS and with a simpler system design. That is, a MAS should only be used on a domain that can benefit from it.

Parallelism

A MAS has a obvious advantage, if the problem can be spilt up in smaller sub problems, since the sub problems can be assigned to the multiple agents. This can

help solve the problem quicker, as each agent run in parallel. In the algorithms described in section 3.3, this approach is used. It is however not always possible to subdivide the problem, but even in domains that are not distributable, there could be advantages of using a MAS. Having multiple agents could speed up a systems operations by providing a method for parallel computation.

The one major drawback using a MAS is the communication cost. It can diminish the effect of running the program in parallel and perhaps even cause the strategy to be slower than a solution strategy, using only a single agent. In the algorithms described in section 3.3, they are aware of this issue, and have tried to decrease the communication cost by assigning more variables to each agent.

Scalability

Even though the communication cost could be a problem, a MAS possess an advantage compared to a single agent system. A MAS is scalable. Since each agent is independent of the others, it is easier to add new agents to a MAS, than it is to add new capabilities to a single agent system. Additionally agents are easily replaced by an agent implementing a different strategy. This could be an advantage in the algorithms described in section 3.3, compared to the SAT solver presented in section 2.4. As different solution strategies could be tested simply by replacing a number of agents. This could also be an advantage in the algorithms from section 3.2, where it would be possible to test different heuristic, simply by replacing an agent.

Robustness

Robustness is a benefit of MAS that have agents that complement each other. If control and responsibilities are sufficiently shared among different agents, the system can tolerate failures by one or more of the agents. However, in none of the above MAS approaches, this is used.

4.2 Conclusion

Designing systems as MAS to solve $\mathcal{NP}$-complete problem, is clearly a new software paradigm. However, it is not always an advantages to use a MAS, compared to the well known solvers to $\mathcal{NP}$-complete problems, as they are more optimized and specialized. This is perhaps not that big of an issue, since the system likewise can be optimized and specialized in the time to come. The key issue is that a MAS is new way to represent the problems, not similar to any other solver. This is obvious

in section 3.4, where the VRP is presented in an entirely new way. The advantage in using a MAS is therefore not the computation speed, but the fact that the problem is presented in a different way compared to the old strategies, which in the long run could lead to effective solvers. It is important to mention that the system design should only be MAS, if the system can benefit from it, but it is sometime necessary to try to use a MAS, even though it is not obvious, in order to spot new effective solution strategies.

Multi-agent systems own a number strengths and weaknesses compared to other strategies that solves the same problem. First of all if the problem can be partioned, it is possible to receive a gain in computational performance, but it is important that the communication cost is kept down, to get a fast solution. Additionally a MAS is scalable, which means that it is easy to change or remove agents. A MAS is also robust, if it has agents that supplement each other.

Chapter 5

Sudoku

In this chapter a brief introduction to Sudoku is given, by presenting the terminology and the rules in the puzzle. Then a solution strategy is presented that can help solve the puzzle. Finally it is shown that Sudoku belongs to the class of $\mathcal{NP}$-Complete problems.

5.1 Behind the puzzle

Sudoku is not an entirely new invention. It started already in 1783 with the Swiss mathematician Leonard Euler, who invented Latin Squares, but Sudoku puzzles, as we know them today, were first published in 1979 under the name *Number Place*. Later on the Japanese gave the puzzle the name Sudoku.

A Latin Square problem is not entirely the same as a Sudoku puzzle, but they have many similarities. A Latin Square is a grid of size $n \times n$, which contains all the numbers from 1 through n exactly once in every row and column. The Sudoku problem has the following definition.

General Sudoku:

A general Sudoku puzzle consists of a $n^2 \times n^2$ grid, which is divided into $n \times n$ squares, where n is the order of the puzzle. Throughout this thesis the following notation of the Sudoku properties is used.

- A *cell* refers to one of the n^4 entries in the grid.
- The *value* of a cell refers to the number placed in the cell.
- A *row*-, *column*- or *square-domain* refers to the rows, columns and squares of the grid.
- A *candidate* is a number that could be placed in the cell.
- A *clue* is a value in a cell that is already given in the problem instance.
- The *order* of the puzzle refers to the size of n. A typical 9×9 grid, will therefore be a puzzle of order 3.

A Sudoku puzzle has the following constraints:

- Each cell must have a value from 1 to n^2.
- Each of the row, column and square domains must contain the values from 1 to n^2 exactly once.

These constraints will trough out the report be referred to, as the Sudoku constraints.

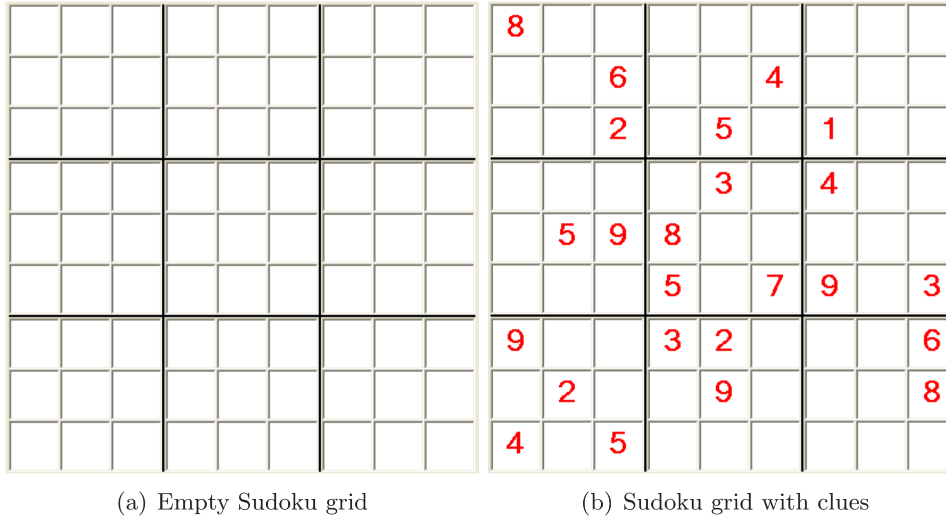
The difference between a Latin Square and a solved Sudoku puzzle is therefore that a Sudoku is more constrained than the Latin Square, as a valid Sudoku must also satisfy the square constraints. The fact that each row and column constraint must be satisfied in a valid Sudoku puzzle shows that every solved Sudoku is also a Latin Square, but not the other way around.

At figure 5.1 a classical order 3 Sudoku grid is displayed, with and without initial values (clues).

Not every order 3 problem instance is considered to be a real Sudoku puzzle. It is therefore often in the literature defined that a proper problem instance satisfies the condition:

Condition: A Sudoku problem instance must have a unique solution, which can be determined by stepwise making logical conclusions based on the values already present in the puzzle.

A puzzle is only considered a proper Sudoku problem instance, throughout this thesis, if this holds.



(a) Empty Sudoku grid (b) Sudoku grid with clues

Figure 5.1: Sudoku problem instances.

A Sudoku problem instance contains a number of predefined cell values, clues. The clues provide the player with the initial information to start determining the solution to the puzzle.

According to [24] an empty classical grid have 5.472.730.538 possible solutions, if the puzzle does not have any clues, and symmetry is taken into account.

Rating the Sudoku puzzle

A Sudoku puzzle is usually associated with a difficulty rating, which indicates how easy or difficult a given puzzle is to solve. The rating of a Sudoku puzzle is however not a standardized method. The obvious factors in determining a Sudoku puzzles rating would be the number of clues and the placement of the clues. One may argue that intuitively a Sudoku with few clues are more difficult that a Sudoku with many clues. Although this is right in many cases, it is not true for all Sudoku puzzles. There exists puzzles that have many clues, but which are more difficult than some puzzles with few clues. Likewise the placement of the clues will give rise to different difficulties. A Sudoku puzzle with a certain placement of the clues can be very different in rating, compared to a Sudoku puzzle with the same placement of clues, but where the values of the clues are different. This concludes that neither the number of clues nor the placement, is enough to determine the difficulty rating of a puzzle.

In the Sudoku community a good rating reflects how difficult a puzzle is for a human to solve. This could be reflected by the number and difficulty of the strategies

necessary to solve the puzzle. Recently a puzzle has been developed, known as *Qassim Hamza*, which has proven to be extremely difficult to solve, because all basic solution methods will not advance this puzzle towards a solution. See the puzzle in figure 5.2.

			7			8		
				4			3	
					9			1
6			5					
	1			3			4	
		5			1			7
5			2			6		
	3			8			9	
		7						2

Figure 5.2: Puzzle known as Qassim Hamza. Very hard puzzle to solve.

The number of clues and their placement, are not useful for determining difficulty, but instead they are believed to be the main factors in determining if a puzzle has a unique solution. It is not known what the minimum number of clues are, to ensure that the puzzle has one unique solution. It is argued in [35] that the minimum number in a classical Sudoku is 17, but this is not proved. In [24] they give an example on a puzzle with 17 clues with one solution, but underline that the minimum number could be even smaller. One might think that a puzzle with many given clues is likelier to have a unique solution, but this is not necessarily the case. In [24] they give an example of a puzzle with 29 clues that actually have two different solution.

An important part of solving a Sudoku puzzles is to keep track of the candidates in the undetermined cells. In figure 5.3 both the determined values and the candidates are displayed with respectively large and small font. The candidate values are the key for using clever strategies that can help solve the puzzle.

5.2 Solution strategy

The strategy for solving a puzzle by hand can be divided into a number of levels. The first level is the scanning level. This level uses the two strategies *Counting in domains* and *Cross-hatching*, which we call the *0-level strategies*. The first strategy is straight forward. The procedure is to look in each domain, to see whether all but one has a undetermined value. If such a cell is found, it is obvious what the value

8	13479	1347	12679	167	12369	23567	2345679	24579
1357	1379	6	1279	178	4	23578	235789	2579
37	3479	2	679	5	3689	1	346789	479
1267	1678	178	1269	3	1269	4	125678	1257
12367	5	9	8	146	126	267	1267	127
126	1468	148	5	146	7	9	1268	3
9	178	178	3	2	158	57	1457	6
1367	2	137	1467	9	156	357	13457	8
4	13678	5	167	1678	168	237	12379	1279

Figure 5.3: Classical Sudoku grid with clues and candidates.

of the cell should be. The second strategy is also straight forward, but it requires more searching. In figure 5.4 the approach for this strategy is shown. In the top right square the green cell must contain a 5, since every other cell in the square cannot contain a 5, because of the location of a 5 in every row and column, the square is a part of.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 5.4: The Cross-hatching technique.

Often a Sudoku is not solvable by using only 0-level strategies. Therefore more advanced strategies have been developed to harder Sudoku puzzles. Some of these will be explained later.

5.3 Sudoku is $\mathcal{NP}$-complete

The Sudoku problem can be expressed with the notation used in 2.1 the following way:

Let l be the set of all $n^2 \times n^2$ Sudoku grids with a number of clues, with the values between 1 and n^2. Additionally let $Q : l \rightarrow \{yes, no\}$ be the problem. For the answer to be *yes* to the problem, the following should hold:

$$Q(x) = yes \leftrightarrow \text{the grid } x \text{ is filled in without violating any constrains}$$

To show that Sudoku is $\mathcal{NP}$-complete, it is necessary first to show that it belongs to $\mathcal{NP}$. This is fairly easy, since it is possible to verify in polynomial time that a Sudoku grid is filled correctly, by running through each domain (rows, columns and squares) to determine if any Sudoku constrains are violated. This take $(3 \times n^2)n^2$, since the number of domains that should be visited equals $3 \times n^2$ and the number of cells in each domain are n^2. This is clearly polynomial.

It is harder to show that it is $\mathcal{NP}$-complete. As mentioned previously it can be done by a reduction one a problem that is already proven to be $\mathcal{NP}$-complete problem to Q. It will not be shown here, as it has already be proven in [30].

In [30] it is shown that the problem of solving a Sudoku puzzle on a $n^2 \times n^2$ grid is a $\mathcal{NP}$-complete problem. This is done by using a reduction on the Latin Square problem, which has already be proven to be $\mathcal{NP}$-complete cf. [6]. Even though the Sudoku puzzle is $\mathcal{NP}$-complete the small puzzles are easily solved by any computer by means of a simple brute-force, backtracking or using some sort of optimization method as simulated annealing. It is therefore interesting, when developing a solution method to the Sudoku problem, not only to focus on the small puzzles, but also experiment on the larger puzzles, as it is here the $\mathcal{NP}$-complete characteristics really can be seen.

It is important to mention that not all Sudoku puzzles belong to the class of $\mathcal{NP}$-complete problems. Puzzles that can be solved be use of only 0-level strategies are polynomial time solvable (see [23]).

Chapter 6

Different approaches to solve Sudoku

Sudoku has had an enormous public interest due to the immediate attraction of logic puzzles. Sudoku puzzle enthusiasts have developed numerous strategies and solution methods for the puzzle. The academic community has also given focus to the Sudoku problem, as it is a $\mathcal{NP}$-complete problem, which therefore is solvable by some of the already known solving techniques for $\mathcal{NP}$-complete problems. In this chapter we try to examine, which approaches that have been used for solving Sudoku puzzles, both in the academic literature and in the Sudoku enthusiasts community.

6.1 Sudoku representations

When solving a given problem, the first step is to choose a representation for the problem. It is obvious that a Sudoku puzzle can be represented by a Sudoku puzzle instance, but the literature also shows that the Sudoku puzzle can be represented, as both a CSP and a SAT problem.

6.1.1 Basic

The basic representation of a Sudoku puzzle, would be to use the Sudoku puzzle direct as the representation. In this representation a Sudoku puzzle of order n, would consists of n^4 cells placed in a grid, where i is the row, j is the column and

k is the square index. The following constraints ensure the Sudoku representation:

- There is a number between 1 and n^2 in every cell:

$$\forall_{i,j}(cell_{i,j} \in \{1, .., n^2\}) \tag{6.1}$$

- Every row contains the number 1 to n^2 once:

$$\forall_i\big((\cup\forall_j cell_{i,j}) = \{1, .., n^2\}\big) \tag{6.2}$$

- Every column contains the number 1 to n^2 once:

$$\forall_j\big((\cup\forall_i cell_{i,j}) = \{1, .., n^2\}\big) \tag{6.3}$$

- Every square contains the number 1 to n^2 once:

$$\forall_k\big((\cup\forall_{i,j\in k} cell_{i,j}) = \{1, .., n^2\}\big) \tag{6.4}$$

6.1.2 CSP

Sudoku is a CSP problem. CSP problems are mathematical problems, where one must find states or objects that satisfy a number of constraints or criterias. In Sudoku, these constrains are the so-called Sudoku constrains that was mentioned in chapter 5.

There are numerous approaches to solving CSP problems, but only a general outline of these will be described in context of solving Sudoku puzzles.

Constraint Programming

In Constraint Programming (CP), a model of the problem is created in terms of variables belonging to the given domains and constraints that must be satisfied. In CP a solution is found by trying all possible variable values and check if the constraints are satisfied. Since this leads to a search space of exponential size, a number of pruning schemes are applied in order to minimize the search space. E.g. every constraint is associated with a filtering algorithm, which is used repeatedly in order to filter the domains, thus called *domain filtering algorithms*. When a domain filtering algorithm reduces a variable domain, all other domains containing the same variable updates the variable domain in order to be consistent. This is ensured by using *constraint propagation algorithms*. A more thorough explanation of CP and propagation algorithms, can be found in [32].

In [29] the Sudoku is modeled as a CP problem, and different constraint propagation algorithms are tried in order to solve the Sudoku puzzles. The focus of the article

is to use CP to solve the problem, without using search. This means that the domain filtering and constraint propagation algorithms should be able to determine a solution by only assigning unambiguous variable values. The evaluation shows that CP is able to solve all the presented Sudoku puzzles, without using search.

Integer Programming

In relation to the CP programming, the Sudoku can also be modeled by an Integer Programming (IP) model. This is shown by [5], but since IP in itself is $\mathcal{NP}$-complete, it is again necessary to minimize the search space. In IP this is often done by applying *Cutting Planes*. The article however, does not elaborate on the fact that solving a Sudoku puzzle by an IP model, is also $\mathcal{NP}$-complete. Therefore we will not go into detail about optimizing the IP model in order to minimize the search space, but just state the fact that IP models also are able to solve Sudoku puzzles.

Belief Propagation

Another approach, similar to CP, is Belief Propagation (BP) that instead of propagating constraints, propagates probabilities. It is however out of the scope of this thesis to go into details about BP and applications hereof. It is therefore only noted that in [12], BP are suggested as an approach to solving Sudoku puzzles. BP is able to solve the majority of the puzzles presented without search.

6.1.3 SAT

Sudoku can also be expressed as a boolean satisfiability (SAT) problem. In [23] and [34], Sudoku is encoded and solved as a SAT problem. In this section we give a brief introduction to a SAT encoding of Sudoku, as it provides an intuitive mathematical understanding of the structure of the Sudoku constraints.

In [23] the authors presents two SAT encodings of the Sudoku problem. The first is the minimal encoding, which is sufficient for describing a Sudoku problem. However, when adding a number of redundant constraints in an extended encoding the resolution of the encoding is increased.

Below is found a minimal Sudoku encoding, as described in [23]. Here variable s_{xyz} is assigned true, if the entry in row x and column y has the value z:

- There is at least one number in each entry:

$$\bigwedge_{x=1}^{9}\bigwedge_{y=1}^{9}\bigvee_{z=1}^{9} s_{xyz} \tag{6.5}$$

- Each number appears at most once in each row:

$$\bigwedge_{y=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{x=1}^{8}\bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz}) \tag{6.6}$$

- Each number appears at most once in each column:

$$\bigwedge_{x=1}^{9}\bigwedge_{z=1}^{9}\bigwedge_{y=1}^{8}\bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz}) \tag{6.7}$$

- Each number appears at most once in each 3x3 square:

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=y+1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z}) \tag{6.8}$$

$$\bigwedge_{z=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{x=1}^{3}\bigwedge_{y=1}^{3}\bigwedge_{k=y+1}^{3}\bigwedge_{l=1}^{3}(\neg s_{(3i+x)(3j+y)z} \vee \neg p_{(3i+k)(3j+l)z}) \tag{6.9}$$

The constraint 6.8 ensures that each number appears at most once in the rows of the square, and 6.9 ensures that each number appears at most once in the columns of the square. Together this ensures that each number appears at most once in the entire sub-grid for every sub-grid.

This is the minimal encoding of a Sudoku puzzle. Each pre-given value is encoded as a unit clause, e.g. the unit clause $s_{111} = 1$ will denote that the value 1 is given in the entry $(1, 1)$.

In [23] an extended encoding is also presented. This has constraints that ensures that:

- There is at most one number in each entry.
- Each number appears at least once in each row.
- Each number appears at least once in each column.
- Each number appears at least once in each 3x3 sub-grid.

This gives a number of redundant constraints, but during the evaluation of the different encodings, it is noticed that the extended encoding yields the best results. This is probably due to the increased resolution of the encoding.

When solving a SAT problem the a number of inference techniques are usually used in order to filter and minimize the search space. Sometimes this is done by adding more clauses, and or eliminating variables from clauses. A number of inference techniques are presented in [23], where the most well-known is unit propagation. Unit propagation is based on the unit clause rule, which means that whenever a unit clause s is identified all other clauses are updated after the following rules:

- Every clause containing s is removed.
- In every clause that contains $\neg s$, the literal is deleted.

This approach, to minimize the search space, is very similar to the ones used in both constraint and belief propagation for CSP problems.

In [23] all the presented puzzles can be solved with the use of inference techniques.

6.2 Solution strategies

When solving a Sudoku problem there are different approaches. As mentioned above the representation often gives rise to a number of techniques for minimizing the search space. If the Sudoku puzzle is solvable by using only logical reasoning, then these techniques are often sufficient. On the other hand, if no obvious logical reasoning or deductions can be performed, the Sudoku can still be solved using either search or meta-heuristics. This section describes the common techniques, which the literature describes for solving Sudoku. Common for all these techniques are that they all use some sort of direct representation of the Sudoku problem.

6.2.1 Search

Brute-force

When given a Sudoku puzzle, the brute force approach enumerates all the possible permutations of values in the empty cells of the Sudoku puzzle. A candidate solution is then created, when all cells have a value. If the candidate solution does not violates the Sudoku constraints, a solution is found, otherwise the next permutations of values are tried. It is obvious, that this approach of enumerating all permutations, is computationally expensive. One may notice that a violation of the Sudoku constraints can occur early in the creation of the candidate solution, hence making the rest of the creation a waste of resources. The brute-force approach is therefore often combined with a simple backtracking scheme: Instead of creating the entire candidate solution, every decision towards the candidate solution is eval-

uated against the Sudoku constraints, and if it violates the constraints the decision is discarded, and the next decision is tried.

A simple recursive brute-force depth first search can then be used to solve a Sudoku. When presented a Sudoku, it finds a solution using the recursive procedure presented in Algorithm 1.

Algorithm 1 BFS(*puzzle*)

```
if puzzle has empty cells then
  r, c = row and column of next empty cell
else
  return true {Solution path fully expanded}
end if
for i = 1 to i < n^2 do
  puzzle(r, c) ⇐ i
  if the assignment doesn't violate any constraints then
    if BFS(puzzle) is true then
      return true
    end if
  end if
end for
puzzle(r, c) ⇐ empty
return false
```

The procedure works by placing the value 1 in the first empty cell and checks if its violates any constraints. If there are no violations, then the algorithm advances to the next cell (expanding the search tree), and places the value 1 in that cell. If there is a violation in the next value, it tries a new value until all values have been tried. This is then repeated for every empty cell. The algorithm expands recursively, until the value of the last empty cell is discovered.

One of the advantaged by using this method is that a solution is guaranteed, if there exists one. Since it is a search, the solution time is not necessarily dependent of the difficulty of the puzzle, since these are often based on logical strategies needed in order to solve the puzzle. However, the solution time can be exponential to the order of n^4. An example of a Sudoku grid that causes a long execution time, can be seen in figure 6.1.

Solving this puzzle by brute force requires 641.580.843 iterations ([35]), because it only has 17 clues, and on average has 5 value choices in every empty cell. This means that the number of choices in the worst case is $5^{64} = 5 \times 10^{44}$, which is enormous. However the search minimizes the search tree by checking for violations of the constraints in each iteration, so the actual worst case running time is much smaller, but still exponential.

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Figure 6.1: The brute force algorithm used on a grid that causes a long execution time for the algorithm.

Backtracking

Although the described brute-force approach is a variant of the backtrack search, backtracking is often a further refinement of the brute-force depth first search. The refinement often consists of heuristic choices that ensures a better traversal of the nodes in the search tree. An good idea would be to start placing values in the empty cell with fewest candidate values, which increases the possibility of making the right decision in creating the solution.

6.2.2 Meta-heuristics

Meta-heuristics have proven useful when solving $\mathcal{NP}$-complete problems. It is therefore natural to explore the possibility that meta-heuristics are also suitable for solving Sudoku puzzles. In the literature we have only found one example of the use of meta-heuristics to solve Sudoku. Although many meta-heuristics such as Tabu Search, Ant Colony Optimization and Genetic Algorithms probably could be used on the Sudoku problem, it is beyond the scope of this thesis to go further into the detail with this. In the following, it will be described, how [20] uses Simulated Annealing to solve the Sudoku problem.

Simulated Annealing

Simulated Annealing is often used in optimization of combinatorial problems. Simulated Annealing starts with a candidate solution and an objective function. It

then tries to minimize the objective function by iteratively changing the candidate solution. It accepts changes that decreases the objective function, but also some changes that increases the objective function in order to avoid being trapped in local minima/optima. The choices are accepted with a probability dependent of the change and a factor called the temperature of the system. The temperature decreases as the computation proceeds, making the variation of the changes lesser and lesser as the temperature approaches zero. For a more thorough explanation of Simulated Annealing see [17] and [18].

In [20] the general idea is to construct a random candidate solution, which then is modified iteratively, until a valid solution is found. In constructing the candidate solution a Sudoku is represented by a grid of cells. A value of a cell in the ith row and jth column is denoted by $cell_{i,j}$. If a cell is already given a value, in the problem instance, that cell is fixed, and every empty cell is non-fixed. The non-fixed cells are then assigned random values in a manner that ensures that the square constraints of the puzzle are satisfied. The remaining constraints for the rows and columns are then used as the basis for the objective function. The objective function is then the sum of every violation of the row and column constraints. A solved Sudoku puzzle is therefore a puzzle where the objective function is equal to zero.

In order to ensure that the square constraints remains satisfied, the random modifications of the candidate solution only exchanges the value of two non-fixed cells inside the same square. At each modification the change in the objective function is calculated, and based on that value and the temperature of the system, the modification is either accepted or discarded. By making modifications iteratively the Sudoku converges towards a solution, but in contrast to most optimization problems, we are only interested in the solution, if the objective function reaches zero. It is possible that the system gets trapped in a local minima/optima late in the process, where the temperature is near zero, making it difficult to escape. Therefore the algorithm is not complete, as it does not guarantee a solution. In order to fix this, a mechanism is added. When trapped in a local minima/optima, that can not be escaped over a fixed number of iterations, the algorithm is re-started with a new candidate solution. This mechanism is called a re-heat, and ensures that the algorithm is complete. The system presented in [20] is capable of solving the presented Sudoku puzzles of order 3 and 4, using only few re-heats.

6.2.3 Reasoning

When solving Sudoku puzzles by hand one of the most used approaches is logical reasoning. That is, determining values of cells based solely on the already present values in the Sudoku puzzle. This is a continuation of the solution strategy explained in the section 5.2. As explained the next levels are more advanced, than the first level.

In contrast to the 0-level strategies, described earlier, which only looked at properties in a single cell, the next level of strategies (1-level) evaluate sets of cells in order to determine some global properties, which can result in a logical deduction. These strategies consists of *Naked Pairs*, *Naked Triples*, *Hidden Pairs*, *Hidden Triples* and *Intersection Removal* (see www.scanraid.com). The 1-level strategies differ from the 0-level strategies in another important aspect, namely that they do not necessarily determine a cell value, but instead make use of simple logic to reduce the number of candidates in the cells. It is therefore important, after having used the 1-level strategies successfully, to step back and use the 0-level strategies once again in case the 1-level strategies indirectly revealed a step towards the solution.

Naked Pairs and Triples

Naked pairs is a set of two cells, belonging to the same domain, where the number of candidates in the two cells are maximum two, and the candidates share the same values. If a set of cells with this property exits, it can be logically concluded that the two candidate values must be placed in either of the two cells. It is therefore possible to eliminate the candidates present in the naked cells in every other cell belonging to the same domain. If the naked cells share two domains, e.g. row/square or column/square, it is possible to eliminate the candidates present in the common domains. In figure 6.2 the Naked Pairs strategy is shown. The two cells $A2$ and $A8$ is the naked pair. It is clear that these two cells only can contain the candidates 1 and 6, hence all other candidates with value 1 and 6 can be removed in the row (cell A3, A4, A5, A6 and A9).

	1	2	3	4	5	6	7	8	9
A	7	1 6	1 2 4 6 9	1 2 3 4 8 9	1 3 4 9	2 3 6 8	5	1 6	2 3 4 6

Figure 6.2: Naked Pair strategy

This strategy can then be expanded to triples, i.e. a set of three cells. A Naked Triple is therefore a set of three cells, all belonging to the same domain, where the number of candidates in the three cells is maximum three, and the candidates share the same values. An example could be three cells with the following candidates: (1 2 3), (2 3) and (1 3). It can therefore again be logically concluded that the three candidate values must be placed in either of the three cells, hence be eliminated in every other cell in the common domains.

It can be seen that the Naked strategies can be extended to sets of size four up to $n^2 - 1$. It is therefore from here on regarded as a single strategy called *Naked Sets*. The *Naked Sets* strategy can therefore be generalized as:

Naked Set: If C is the set of cells in a given domain, a Naked set is a subset, C' of C, with cardinality n, where the following holds:

- If S is the union of all the candidates present in C', S must have cardinality n.
- The candidates present in S can be eliminated from the cells belonging to the set $C \setminus C'$.

It is obvious to see how a Naked Set is used to eliminate values, but it is also interesting to show its equivalence in another representation. For this we have chosen to show, the equivalent inference technique in a SAT representation.

Mathematical definition

The above description of the different levels is a very illustrative way to describe the approach, but it can also be described in a mathematical fashion in terms of boolean logic.

The elimination of candidates is equivalent to a removal of variables in the SAT representation, causing the constraints to be simplified.

To help show the Naked Set inference, the pseudo Sudoku in figure 6.3 is used for the SAT representation. For simplicity the pseudo Sudoku only consists of a single row with five cells. It is imagined that the cells, in the presented row, is affected by other domains, e.g. row and column domains, yielding a restriction of the candidates as shown in figure 6.3. It is seen that there is a Naked Set containing the two cells $i = 1$ and $i = 3$.

i	1	2	3	4	5
	2 5	1 2 3 4 5	2 5	1 2 3 4 5	1 2 3 4 5

Figure 6.3: Pseudo Sudoku with candidates shown

The notation in figure 6.3 contains a lot of irrelevant information, when representing it as a SAT problem, as the candidate notation cannot be directly transferred to the SAT problem. Instead we introduce the notation of negative candidates. A negative candidate is defined as a value that cannot be placed in a given cell, and is denoted with a bar over the candidate. The pseudo Sudoku can then be expressed as in figure 6.4.

The Sudoku constraints in this simple example, are defined in the following way:

Figure 6.4: Pseudo Sudoku with negative candidates shown

$$\bigwedge_{i=1}^{5} \bigvee_{j=1}^{5} s_{ij} \tag{6.10}$$

$$\bigwedge_{j=1}^{5} \bigwedge_{i=1}^{4} \bigwedge_{k=i+1}^{5} (\neg s_{ij} \vee \neg s_{kj}) \tag{6.11}$$

The boolean variable s_{ij} is true, if the cell at position i has the value j. The constraint 6.10 states that each cell should contain one value between 1 and 5. The constraint 6.11 ensures that each number appears at most once in the Sudoku row.

From the figure 6.4 it is obvious that the information in the Naked Set, can be expressed directly by the negative candidates:

$$\neg s_{11} \wedge \neg s_{13} \wedge \neg s_{14} \wedge \neg s_{31} \wedge \neg s_{33} \wedge \neg s_{34} \tag{6.12}$$

The first three variables belong to the first cell and the last three variables to the third cell. As we only observe the pseudo Sudoku, we assume that 6.12 is obtained by influences from other domains. E.g. by unit propagation if a 1 was sat in a domain containing the first cell. As s_{11} must be false, $\neg s_{11}$ is true, and so forth.

If the expression 6.10 is written for the two Naked Set cells, and expression 6.12 is used, it produces the following expression:

$$\begin{aligned} s_{11} \vee s_{12} \vee s_{13} \vee s_{14} \vee s_{15} &\rightsquigarrow \quad s_{12} \vee s_{15} \\ s_{31} \vee s_{32} \vee s_{33} \vee s_{34} \vee s_{35} &\rightsquigarrow \quad s_{32} \vee s_{35} \end{aligned} \tag{6.13}$$

The variables contained in expression 6.12 can be ignored, since they all must be false for the expression to evaluate to true. The expression obtained in 6.13 is the basis of the Naked Set strategy:

$$(s_{12} \vee s_{15}) \wedge (s_{32} \vee s_{35}) \tag{6.14}$$

The conjunction 6.14 is then combined with 6.11 for $j = 2$ and $j = 5$. This gives the following expression. For simplicity only the clauses which contain the variables $\neg s_{12}$,$\neg s_{15}$,$\neg s_{32}$ and $\neg s_{35}$ are observed initially:

$$(\neg s_{12} \vee \neg s_{32}) \wedge (\neg s_{15} \vee \neg s_{35}) \wedge (s_{12} \vee s_{15}) \wedge (s_{32} \vee s_{35})$$

Using the distributive law the following expression is obtained:

$$\Big((\neg s_{15} \vee \neg s_{35}) \wedge (\neg s_{12} \vee s_{15}) \wedge (s_{32} \vee s_{35})\Big) \vee$$
$$\Big((\neg s_{15} \vee \neg s_{35}) \wedge (s_{32} \vee s_{35}) \wedge (s_{12} \vee s_{15}) \wedge \neg s_{32}\Big)$$

With the absorption-, commutative- and associative law the expression can be reduced to:

$$\Big(s_{15} \wedge s_{32} \wedge \neg s_{12} \wedge \neg s_{35}\Big) \vee \Big(\neg s_{15} \wedge \neg s_{32} \wedge s_{12} \wedge s_{35}\Big) \tag{6.15}$$

This states exactly the Naked Set inference, i.e. that $\mathcal{S}(1) = 5$ and $\mathcal{S}(3) = 2$ or $\mathcal{S}(1) = 2$ and $\mathcal{S}(3) = 5$.

The remainder of 6.11 for $j = 2$ is therefore:

$$\begin{aligned} \underbrace{(\neg s_{12} \vee \neg s_{22})}_{a} \wedge \underbrace{(\neg s_{12} \vee \neg s_{42})}_{b} \wedge \underbrace{(\neg s_{12} \vee \neg s_{52})}_{c} \wedge \\ \underbrace{(\neg s_{22} \vee \neg s_{32})}_{a} \wedge (\neg s_{22} \vee \neg s_{42}) \wedge \\ (\neg s_{22} \vee \neg s_{52}) \wedge \underbrace{(\neg s_{32} \vee \neg s_{42})}_{b} \wedge \underbrace{(\neg s_{32} \vee \neg s_{52})}_{c} \wedge (\neg s_{42} \vee \neg s_{52}) \end{aligned} \tag{6.16}$$

The clauses marked with a are then combined with 6.15, which using the distributive law yields:

$$\neg s_{22} \wedge \left(\Big(s_{15} \wedge s_{32} \wedge \neg s_{12} \wedge \neg s_{35}\Big) \vee \Big(\neg s_{15} \wedge \neg s_{32} \wedge s_{12} \wedge s_{35}\Big)\right) \tag{6.17}$$

It is now possible in the same way to combine the clauses marked with b and c with 6.17 yielding:

$$\begin{aligned} \neg s_{22} \wedge \neg s_{42} \wedge \neg s_{52} \wedge \\ \left(\Big(s_{15} \wedge s_{32} \wedge \neg s_{12} \wedge \neg s_{35}\Big) \vee \Big(\neg s_{15} \wedge \neg s_{32} \wedge s_{12} \wedge s_{35}\Big)\right) \end{aligned} \tag{6.18}$$

It is then obvious, when using the absorption law that the unmarked clauses in 6.16 can be eliminated, leaving back equation 6.18. This reduction is also performed for $j = 5$, giving the complete constraint expression for $j = 2$ and $j = 5$:

$$\neg s_{22} \wedge \neg s_{42} \wedge \neg s_{52} \wedge \neg s_{25} \wedge \neg s_{45} \wedge \neg s_{55} \wedge \left(\left(s_{15} \wedge s_{32} \wedge \neg s_{12} \wedge \neg s_{35} \right) \vee \left(\neg s_{15} \wedge \neg s_{32} \wedge s_{12} \wedge s_{35} \right) \right) \tag{6.19}$$

This is the conclusion of the Naked Set, which states that $\mathcal{S}(2) \neq 2$, $\mathcal{S}(2) \neq 5$, $\mathcal{S}(4) \neq 2$, $\mathcal{S}(4) \neq 5$, $\mathcal{S}(5) \neq 2$ and $\mathcal{S}(5) \neq 5$. This proves that the Naked Set inference is able to determine the conclusion in 6.19. This information is also showed in figure 6.5.

i	1	2	3	4	5
	1 3 4	2 5	1 3 4	2 5	2 5

Figure 6.5: Naked Set conclusion

Hidden Pairs and Triples

The principles in the Hidden Pairs strategy are similar to the Naked Sets. In Hidden Pairs the objective is also to determine a set of cells that must contain a certain set of candidates, hence revealing possible eliminations. A Hidden Pair is a set of two cells, belonging to the same domain, where two of the candidates present in the cells only appear in those cells in the domain.

The difference is that the set of candidates are hidden amongst other candidates, whereas they before where the sets of all candidates present in the Naked Set cells. The effect of this strategy is therefore different, as the elimination takes place in the cells in the chosen set. An example of a Hidden Pair is shown in figure 6.6. It is seen that the only two cells in the domain that have the candidates 3 and 5 are cell $A4$ and $A5$. Hence all other candidates in $A4$ and $A5$ can be eliminated.

As with Naked Pairs, this strategy can also be extended to sets of cells with size three and four up to $n^2 - 1$. It is therefore from here on regarded as a single strategy called *Hidden Sets*. The Hidden Sets strategy can be generalized as:

> **Hidden Set:** If C is the set of cells in a given domain, a Hidden Set is a subset C' of C with cardinality n, where the following holds:

	1	2	3	4	5	6	7	8	9
A	2 78	2 89	1	23 5 789	23 5 7 9	2 7 9	89	4	6

Figure 6.6: Hidden Pair

- If S is the union of all the candidates present in C', there exits a subset S' with cardinality n, for which the candidates in S' only exists in the cells in C' in the given domain.
- The candidates belonging to $S \setminus S'$ can be eliminated from the cells belonging to C'.

As with the Naked Sets, it is possible to show the inference of the Hidden Sets mathematically, however as they are very similar, this is left up to the reader.

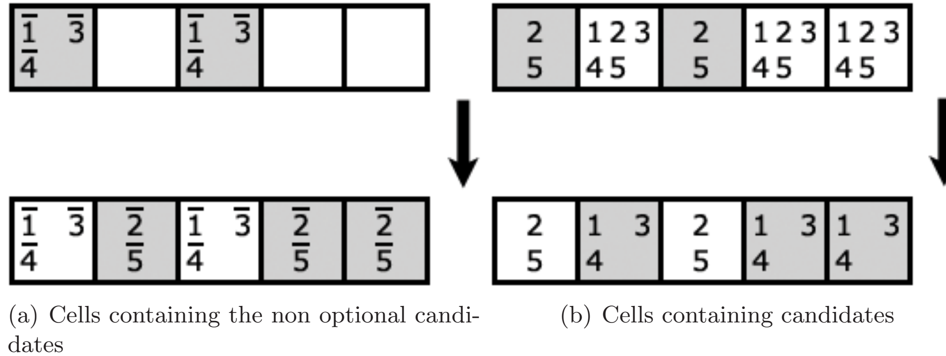
It is important to notice that there exists a strong connection between the Naked and Hidden Set strategies. E.g. a Hidden Set becomes a Naked Set, when eliminating the surplus candidates. In figure 6.6 it is also seen that there is a Naked Set with cardinality four in the cells A1, A2, A6 and A7. The Naked Set yields the same eliminations as the Hidden Set. This means that it is unimportant which of the two strategies that are used, the result is the same. This dualism will be explained more precisely in the following.

Duality

As mentioned above, there exits a duality between the Naked and Hidden strategy. To explain this duality the simple pseudo Sudoku from earlier is used, which can be seen in figure 6.7 and figure 6.8. The pseudo Sudoku consist of only a single row, which should contain the values from 1 to 5. In order to help the visualization of the duality, the concept of negative candidates is used again. A negative candidate is a value that is not a option in a given cell.

In the figure 6.7a top, it is seen that the candidate values 1, 3 and 4 are not possible in the cells marked with grey. In figure 6.7b top, it is obvious that there is a Naked Set in cell 1 and 3, with the candidates 2 and 5. In figure 6.7a and b bottom, the conclusions of the Naked Set are added to the Sudoku. It is seen that when using the negative candidates, information is added to the Sudoku, but when using positive candidates information is removed.

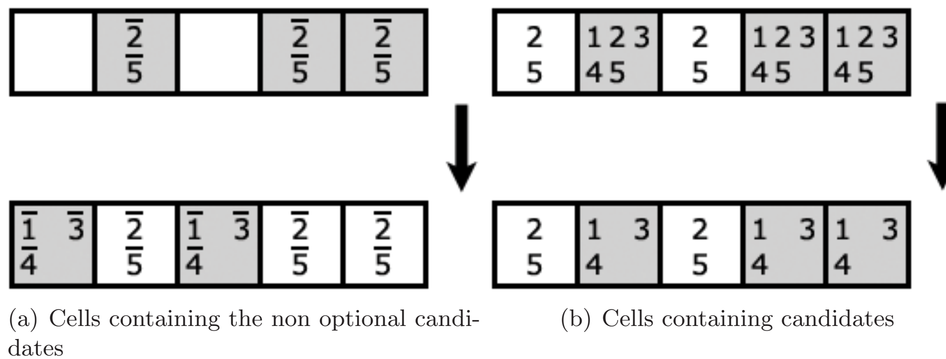
In figure 6.8 the Hidden strategy is used to find a hidden triple in exactly the same complementary cells to the the Naked Set. The duality with the Naked Set is seen



(a) Cells containing the non optional candidates

(b) Cells containing candidates

Figure 6.7: Grids revealing naked candidates.



(a) Cells containing the non optional candidates

(b) Cells containing candidates

Figure 6.8: Grids revealing hidden candidates.

in figure 6.8a, which shows that the cells affected by the conclusion of the Hidden Set, is exactly the same cells from the Naked Set, and in both the negative and positive candidate notation the conclusion obtained is the same. That is, if a Naked Set strategy is used, it is always possible also to use the Hidden Set strategy, to get the same result.

Intersection Removal

The last strategy 1-level strategy is the Intersection Removal strategy. The Intersection Removal strategy states that if a set of cells share two domains, i.e. row/square or column/square, and one of the candidates present in the cells are unique within one of the domains, the same candidate can be eliminated from the other domain.

The four types of Intersection Removal are therefore:

Intersection type 1 A candidate with value n is unique in a square - If the cells containing the candidate are aligned on a row, n can be removed from the rest of the row.

Intersection type 2 A candidate with value n is unique in a square - If the cells containing the candidate are aligned on a column, n can be removed from the rest of the column

Intersection type 3 A candidate with value n is unique in a row - If the cells containing the candidate are in the same square, n can be removed from the rest of the square.

Intersection type 4 A candidate with value n is unique in a column - If the cells containing the candidate are in the same square, n can be removed from the rest of the square.

In figure 6.9 the strategy is illustrated. In the cell $A4$ and $A5$ there is a intersection of type 3 with the candidate 2, which means that it can be eliminated in all other cells in the grey square, i.e. cell $C5$. In cell $C7$ and $C8$ another intersection of type 1 yields the same elimination in cell $C5$.

	1	2	3	4	5	6	7	8	9
A	3 4 5	3 4 5	6	2 4 5	2 3	7	1	8	5 9
B	4 5	2	1	4 5	9	8	7	3	6
C	3 4 5 7 9	8	3 5 7	1	2 3	6	2 4 5	2 4 5	5 9

Figure 6.9: The Intersection Removal strategy.

The Intersection Removal strategy can be generalized as:

Intersection Removal: If $C1$ is the set of cells in a given domain, and $C2$ is the set of cells in another domain, a Intersection Set C' is the intersection of $C1$ and $C2$ with cardinality n, where the following holds:

- If S is the union of all the candidates present in C', there exits a subset S', for which the candidates in S' only exists in the cells in either $C1$ or $C2$.
- If $\exists cell \in C1$ with a $candidate \in S'$ the candidate can be eliminated from the cells belonging to $C2 \setminus C'$.
- Else if $\exists cell \in C2$ with a $candidate \in S'$ the candidate can be eliminated from the cells belonging to $C1 \setminus C'$.

Advanced strategies

In this section we have only covered the most basic strategies. In the Sudoku community various more advanced strategies have been developed. All describing possible logic conclusions on the basis of advanced connections made on the the current information contained in the Sudoku. They do not have any obvious common characteristics, but they all use more advanced reasoning to eliminate candidates than the described strategies. The *X-Wing* strategy is an example of such an advanced strategy. It can be used when a candidate appears exactly twice in two different rows, and the four candidates lie in the same two columns. This produces that all other candidates with this value that lie in the columns can be eliminated.

Figure 6.10: The X-Wing strategy.

In figure 6.10 the X-Wing strategy is shown. The four yellow cells marked with the letters A, B, C and D tell that this is a X-Wing pattern with the value 6. This is due to the their mutual location and because they are the only cells that can contain 6 in that row where they are located. From this it is possible conclude that if A turns out to be a 6, then B cannot be a 6, and vice versa. Likewise if C turns out to be a 6 then D cannot be, and vice versa. This again means that if A has the value 6 both C and B cannot have it, but then D must also have the value. Therefore 6 must be present at AD or BC then any other candidates with the value 6 along the edge of the rectangle the four cells stretch are redundant. The candidates marked with the red square are the eliminated candidates.

As mention before there exits many advanced strategies, but we will not go trough them here. To get a insight into the advanced strategies see www.scanraid.com or www.setbb.com.

Chapter 7

Sudoku solver

In this chapter we try to utilize our experiences from the previous chapters in designing a multi-agent Sudoku solver. The chapter consists of a general description of the system design, the most important implementation details and numerous tests of the system, both individually and against others solvers.

7.1 Requirements

In making a satisfactory solver, a few goals is defined that should be met:

- It should be able to solve a Sudoku of sizes $n^2 \times n^2$ (order n), with focus on the sizes 9×9 (order 3) and 16×16 (order 4).
- It should try to solve any Sudoku without search (guessing).
- It should use some of the human strategies in determining a solution.

The focus is going to be on order 3 and 4 puzzles, since these are the common Sudoku sizes, which is also comprehensible for a human. Larger puzzles are difficult to obtain, as the Sudoku community has had focus on the human solvable puzzles. Additionally the aim is to make the solution progress of the Sudoku solver similar to the way humans attack the problem. That is, trying to avoid using search (guessing) to solve the problem, but instead use the available strategies.

7.2 Analysis

The goal is to use the experiences learned previous to design a Sudoku solver. Since the main focus has been on how multi-agent systems can solve $\mathcal{NP}$-complete problems, it is obvious that we should try to design a MAS, which is capable of solving a Sudoku.

The first consideration is therefore, which of the multi-agent design paradigms should be used in making a satisfying solver. The problem it self gives rise to a natural partition of the problem, as Sudoku consists of multiple domains (rows, columns and squares), which are connected in a complex manner. E.g in an order n Sudoku each row coincides with n^2 columns and n squares. That is, a change in a cell will have impact on three domains, namely a row, column and square. This change may propagate new changes in three domains, which again may propagate changes in other domains, thereby expressing a complex chain of connections between the domains. To manage this interaction it could be advantageous to regard the different domains as agents, where each Sudoku domain is capable of detecting obvious steps towards a solution, within its own domain. This would be very similar to the agent abstraction in section 3.4, with the entities at play being the Sudoku domains.

There are a number of considerations to be made, before the MAS is implemented. The domains should be managed, so the individual state changes are performed in a synchronized manner. The easiest way to achieve this, is to centralize the control of the domains to a coordinator agent. This is also in line with the system proposed in section 3.4. This means that if a cell contains a single candidate, it is clear that the candidate must be placed inside the cell, and then the domain would suggest this state change to the coordinator agent, who would coordinate the proposed state changes.

Our knowledge of Sudoku puzzles show that most puzzles cannot be solved completely by only looking at simple characteristics of the domains. More advanced observations and strategies are necessary to detect possible state changes towards a global solution. Therefore the MAS should use heuristics based on the previous discussed Sudoku strategies, in order to determine possible state changes. The chosen strategies are therefore:

Naked Set determines if candidates can be eliminated outside a given set on the basis of the candidates in the set. See 6.2.3 for an explanation.

Hidden Set determines if candidates can be eliminated inside a given set on the basis of the connection of the candidates in the cells. See 6.2.3 for an explanation.

Intersection Set determines if candidates can be eliminated outside a given set on the basis of the connection of the candidates in the set. See 6.2.3 for an explanation.

In order to ensure scalability of the system, an agent implementation of each strategy is appropriate. This also shares some characteristics with the approaches used in section 3.2. Here each strategy agent can be regarded as a part of a hybrid solution strategy. This approach also makes the system flexible, as the optimal configuration of the solution strategies is not known before hand.

The goal of the system is to solve the Sudoku puzzles without search (guessing). However, the strategies provided might show insufficient, when trying to solve a Sudoku. Since we wish to be able to solve all Sudoku puzzles, a last resort mechanism should also be provided by the system. This mechanism should use search in order to determine the solution to the puzzle. Since Sudoku also is a SAT problem, as described earlier, an obvious choice would be to use some of the same techniques used in SAT solvers. SAT solvers are often based on a backtracking search, which also could prove efficient in finding solutions to Sudoku puzzles, since the Sudoku constraints also induce a propagation scheme, which minimizes the overall search space.

To summarize, this means that the system is a hybrid between the algorithms mentioned in section 3.2 and 3.4. It is similar to the approach in section 3.4 in the aspect of the partition of the domains between the agents (*domain agents*). And similar to the approaches in section 3.2, because each solution strategy is controlled by an agent (*strategy agents*), which can be compared with the division of heuristics between agents.

7.3 Design

In this section we concretize the above considerations in designing the system.

7.3.1 The overall system

The system should be able to contain different agent types. Furthermore it should facilitate communication between the agents in an appropriate manner. One of the common approaches, when handling agent communications, is to use the FIPA Agent Communication Language (ACL) [36]. Finally, the system should also be able to register and manage the state of the puzzle. To represent the puzzle a direct

representation is chosen. This gives the possibility of representing each cell as an entity of the puzzle.

The functionality of the system should be handled by the different agents. Therefore the system consists of three types as descriped previously. The three types are:

Domain agent which is responsible for a domain in the Sudoku puzzle. Its main task is to ensure that the domain constraint is satisfied. Furthermore the domain agent has the responsibility to inform the coordinator agent about possible steps, it can take towards a solution of the puzzle (*solution steps*). The two possible solution steps, which the agent should recognize, is:

- A cell, where the state of the domain results in a value that is unambiguous (defined as a value solution step):
  - A cell containing only one possible candidate, indicating that the candidate must be placed inside this cell. This is also equivalent to a Naked Set with cardinality 1.
  - A candidate which can only be placed inside a single cell in the domain. This is equivalent to a Hidden Set with cardinality 1.

  When the domain agent suggests possible solution steps, it should use the FIPA ACL performative `Propose`.
- If a value has been sat in one of its cells, it should suggest to eliminate the candidate from every other cell in the domain (defined as an elimination solution step).

Strategy agent which is responsible for a solution strategy heuristic. Its task is to use its strategy to suggest possible solution steps to the coordinator agent. The steps towards a solution, proposed by strategy agents, will always be elimination solution steps.

Coordinator agent which maintains the state of the puzzle cell entities. It should also manage the progress of the solution, by cooperation between the domain and strategy agents.

The design of the three agent types will be covered in the following.

7.3.2 Coordinator Agent

The role of the coordinator agents role is to manage the solution and the state of the puzzle. Initially, it should assign the given values to the corresponding puzzle cells. Thereafter it executes the suggested solution steps. If the agent does not have a possible solution step, it must first try to request one from the predefined strategy agents. This should be done by a message using the FIPA ACL performative

`Request`. If the puzzle is not solvable by the implemented strategies, the agent must still be able to solve the Sudoku. Therefore it should perform a backtrack search. The backtrack search follows the procedure:

1. Choose a cell, where the number of candidates are minimal.
2. Save this cell as the decision basis, which is saved on the decision stack.
3. Try the next candidate (decision) in the decision basis, starting with the first candidate.
4. Let the system proceed as normal, by letting the domain agents and strategy agents suggest possible solutions. While doing this, record the implications (solution steps) for this decision basis.
5. If an agent detects a conflict, e.g. a value has already been used in a domain, or a cell in a domain has no possible candidates, the coordinator agent undoes all the implications recorded. If the current decision basis still have untried candidates jump to step 3, otherwise continue to step 6. If the system solves the puzzle, stop.
6. The current decision basis is discarded, and
   - if the decision stack is non-empty the previous decision basis is popped from the decision stack. If the decision basis has untried candidates jump to step 3, otherwise continue to step 6.
   - else if the decision stack is empty, then no solution can be found to the given puzzle.

The worst-case running time of the procedure is exponential, since it uses a backtrack search, which is a refinement of the brute force search explained earlier in section 6.2.1. However, the ensuring of the Sudoku constraints minimizes the search space.

The coordinator agent should also be responsible for determining, if the Sudoku has been successfully solved, and should notify interested listeners that a solution was found.

7.3.3 Strategy Agents

The strategies used should be implemented as separate agents. This ensures scalability and flexibility, when later determining the optimal configuration and cooperation between the strategies. The strategy agents should be equivalent in functionality,

so that they operate in the same manner, but features different ways to interpret the puzzle in terms of possible solution steps. Therefore they should be able to interact with the other agents, in order to acquire the necessary information for their strategies. When determining possible solution steps, they should be able to pass this on to the coordinator agent. In the following sections the separate strategy agents will be described.

Naked Agent

The Naked Set strategy bases its elimination on knowledge about the number of candidates inside a set of cells, and the value of these candidates. Therefore when the Naked Set agent is requested to perform a search[1] for a Naked Set, it should search through a given set of cells and determine, if they contain a Naked Set. It should not search the entire puzzle at once, but instead search small parts of the puzzle. Because the search is divided, it is possible to ensure that the agent only searches the parts of the puzzle that are relevant. The division of the puzzle into domain agents comes in handy at this point, as the strategy agent can request the relevant domain agents for a list of cells, in which to search. In order to avoid requesting the same domain multiple times, the agent should only request cells from domains that it has not searched before, or domains that have changed since the last time it 'visited'.

Given a list of cells, the agent should search the cells following the recursive procedure shown in Algorithm 2, with the following parameters:

cell is the cell that is examined, to determine if it is contained in a Naked Set. When the procedure first is called, this is chosen as a cell with a minimum number of candidates.

neighbours is the neighbour cells. The neighbour cells are the remaining cells, in which to search for a Naked Set.

choices is the candidates in the Naked Set. When the procedure is first called this is equal to the candidates of the starting cell.

length is the number of cells in the Naked Set

maxlength is the limit size of the Naked Set.

[1]Note that we use the term search in to different interpretations. When solving a Sudoku we wish to avoid using trial and error, also called guessing or search. E.g. the last resort backtrack procedure is a search, which is based on trial and error. However, when using the word search in the aspect of strategies, it means searching the entire puzzle domain for an unambiguous step towards a global solution.

Algorithm 2 NakedSetSearch(*cell*, *neighbours*, *choices*, *length*, *maxlength*)

  {*length* is the number of cells in the set}
  {Check if we have n cells with only n possible candidates, meaning that we are at the endpoint of a set of naked cells}
  if $length = $ count($choices$) **then**
    return MAKE-SET($cell$)
  end if
  while count($neighbours$) > 0 **do**
    $neighbour \Leftarrow$ first($neighbours$) {Get the first cell in neighbours}
    if common($neighbour$, $choices$) > 0 AND $length < maxlength$ **then**
      {Determine the candidates which are different}
      $extra \Leftarrow$ different($neighbour$, $choices$)
      $choices' \Leftarrow choices + extra$
      $neighbours' \Leftarrow neighbours - neighbour$
      $nakedset \Leftarrow$
         NakedSetSearch($neighbour$, $neighbours'$,$choices'$,$length + 1$,$maxlength$)
      if $nakedset$ is not empty **then**
        return UNION($nakedset$,MAKE-SET($cell$))
      end if
    end if
    remove $neighbour$ from $neighbours$
  end while
  return MAKE-SET($empty$) {Nothing was found}

The procedure determines if a Naked Set exists in a given list of cells. To illustrate the search procedure, consider the following example. Given the following list of cells, where X denotes the cell and [x,y,z] denotes the candidates belonging to the cell:

```
A[2,3], B[1,2,3], C[4,5], D[1,3], E[1,4,5]
```

A cell with a minimum number of candidates is chosen, as the possible starting point of a Naked Set. In this example cell A, C or D could be chosen. Lets chose A, and call the procedure with the following parameters:

```
NakedSetSearch(A, {B, C, D, E}, {2,3}, 1, 5)
```

The procedure first checks if the length is equal to the number of choices. Here there are two choices and only one possible cell in the set, so the procedure continues to the while-loop. The first neighbour, B, is then selected, and it is seen that B and A have two candidates in common (candidates 2 and 3). Thereafter the candidates that are different, is added to the possible choices and the neighbour is removed from the neighbour cells. The procedure is then called recursively:

```
NakedSetSearch(B, {C, D, E}, {1, 2, 3}, 2, 5)
```

The procedure again checks if the length is equal to the number of choices. This is still not the case. The next neighbour, C, is then examined, and it is seen that it has no candidates in common with cell A and B. The neighbour is discarded and the next neighbour, D, is considered. Since the candidates in D is already contained in the choices, D is removed from the neighbours and the procedure is called recursively:

```
NakedSetSearch(D, {E}, {1, 2, 3}, 3, 5)
```

The procedure checks if the length is equal to the number of choices, which this time is the case. This means that there exists 3 cells that share 3 common candidates, hence revealing a Naked Set. The procedure therefore starts creating the Naked Set, by making a set containing the cell D, and when returning the set, it is expanded with the other cells in the Naked Set. Finally, the procedure will return a Naked Set containing the cells A, B and D.

The search for a Naked Set on an input of length n is a $O(n^3)$ operation. In the worst-case the procedure needs to compare the candidates in every cell with every

other cell, but in each recursion a cell is removed yielding:

$$\sum_{k=1}^{n}(k-1)n = \frac{1}{2}n^3 - \frac{1}{2}n^2 \approx O(n^3)$$

This is however only the worst case running time. In order to ensure a reasonable running time, the agent should only search for Naked Sets of length $\frac{1}{2}n$. This is due to the fact that the Naked and Hidden Set strategies are dual, hence a Naked Sets, with cardinality larger than $\frac{1}{2}n$, could also be determined by a Hidden Set, with cardinality less or equal to $\frac{1}{2}n$. It is also important to mention that a search in the entire puzzle yields a n times $\mathrm{O}(n^3)$ operation. This is the case for all the strategy agents.

If the agent determines that a Naked Set exits, the agent should inform the coordinator agent about the possible steps towards a solution. This should be done using the FIPA ACL performatives `Propose`. If no set is found in the part of the puzzle searched the agent should continue to another relevant part. If no such part exists, it should inform the coordinator agent that it is refusing to find a Naked Set, taking advantage of the FIPA ACL performative `Refuse`.

Hidden Agent

The hidden agent should be similar to the naked agent. Therefore the considerations about the design will be briefly covered. The difference between the hidden and naked agent, is the strategy that they use. The hidden agent tries to determine Hidden Sets and in doing so, it requires a different input than the naked agent. Instead of looking on cells and candidates, it should consider the connection of candidates in different cells. A connection of candidates is defined as a unique chain, e.g. if the value 1 can only be placed in cell A, B, C and D, they form a unique chain of value 1 in the considered domain. The hidden agent therefore uses unique chains, as its search data instead of cells. However, the search procedure is equivalent to the one of the naked agent. Starting with a unique chain with few cells, it considers the neighbouring chains and tries to build a set of unique chains, which represents a Hidden Set. Since the search procedure is equivalent to the naked agent, the running time is also the same. The duality between the two agents is also a factor in the hidden agent, and therefore it only considers unique chains of length $\frac{1}{2}n$.

Intersection Agent

The intersection agent uses a slightly different strategy, than the previous two agents. In order to determine a Intersection Set, the agent needs to know the

unique chains within a domain, and thereafter determine if any of the unique chains also share a second domain. As explained earlier, the Intersection Set always is a intersection between a square domain, and either a row or column domain. Therefore the relevant parts to search in this agent, is the square domains, since all intersection sets are also within a square domain. The agent should therefore acquire the unique chains from a relevant square domain, and determine if one of the chains also is part of a row or column domain. This can be determined fairly simple by running through the unique chains, and explore if all the cells share two common domains. On an input of n unique chains, only the chains with a length of $\sqrt{n}$ is considered. The reason for this is that longer chains do not share more than one domain. The operation is therefore a $O(n^{\frac{3}{2}})$ operation.

7.3.4 Domain Agent

The row, column and square domains are all represented by a domain agent. To ensure that the domain constraint is satisfied, the agent should hold a reference to all the cells in its domain. When a value is sat in a cell, the domain agent should inform the coordinator agent about all the elimination solution steps, it can perform. E.g. if a value of 2 was sat in cell A, and cell B and C had 2 as a possible candidate value, the agent would propose to eliminate 2 from cell B and C.

In order to recognize possible value solution steps, the domain agent should manage the connections between the cells. It is simplest described by an example: if a candidate of value 1 can only be placed in cell A, B and C in a domain, there would be a unique chain (value dependency) on the value 1 between the cells A, B and C. It is important that the unique chains are up-to-date, therefore they should be updated every time the domain registers a state change. This can be done in $O(n)$ time, if the state change is a elimination, since the chain containing the elimination value needs to be updated. The chain can be found in constant time, with an appropriate data structure. The removal of the cell from the unique chain can be done in linear time, which in the worst-case is an $O(n)$ operation. If the state change is a value solution step the update is an $O(n^2)$ operation. This is because the cell, in the worst-case, needs to be removed from every unique chain.

The last task of the domain agent is to act on conflicts of the domain constraint. This minimizes the search space for the backtracking search.

7.4 Implementation

In this section the implementation of the Sudoku solver will be described. Only the interesting aspects will be covered, and the GUI plus the testing environment will

not be covered, as the implementation is straightforward. The entire source code is however included on the attached CD.

The implementation is made in *C#* 3.0, since it provides a good developing and debugging environment through Visual Studio. Furthermore, it is our preferred programming language, so it was natural to use it for this project as well.

7.4.1 Implementation of MultiAgentSudokuSolver

The solver is named MultiAgentSudokuSolver (MASS), and the implementation is structured as shown in figure 7.1. An arrow from a class A to a class B indicates that the class A contains an instance of class B.

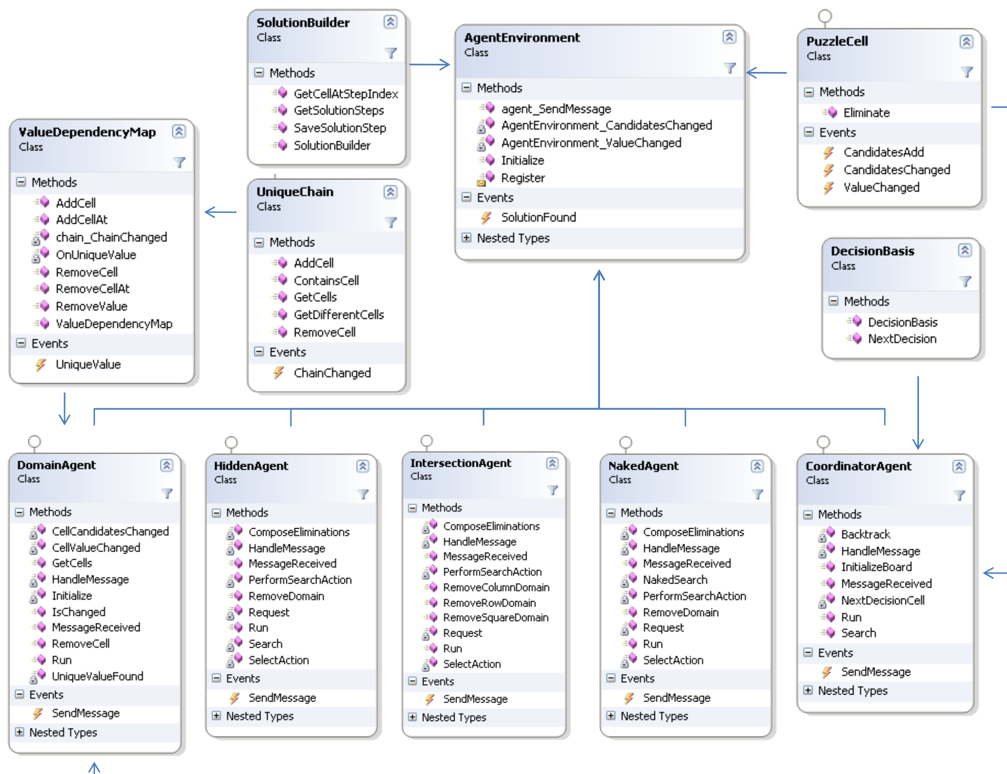


Figure 7.1: Class diagram.

In the different classes not all the functions are listed, but the most important aspects are shown. It is not a complete image of the class structure, but it is an overview of the most important classes and members.

The overall system

The system is constructed as a dynamic linked library (dll), which can be used by other assemblies, such as the GUI or test engine. The agent system is based on the class `AgentEnvironment`, which both initializes all the agent threads, and handles the communication between them. Each agent runs in a separate thread, making the system asynchronous. The communication is therefore handled in the following manner. An agent can send a message by raising the `SendMessage` event. This event is handled by the `AgentEnvironment`, which invokes the `MessageReceived` method on the recipient agent. The `MessageReceived` method enqueues the incoming message on the agents message queue, which is processed continuously by the agent. When an agent processes a message the method `HandleMessage` is used to determine the action to take in response to the message.

The `AgentEnvironment` also represents the Sudoku puzzle as an array of `PuzzleCell` objects. These objects manages the events in every puzzle cell, which means they are able to raise events, whenever the cells state changes. Since every domain agent must be able to propose state changes, they must have knowledge about the state of the puzzle cells inside their domains. Therefore the `DomainAgent` objects have a reference to the puzzle cells in their own domain, to be able to handle the state change events in a proper manner, in order to suggest a state change to the `CoordinatorAgent`. The `CoordinatorAgent` handles the coordination of the domain and strategy agents. Furthermore it is capable of performing a backtracking search, if the system is unable to solve a given puzzle. The detailed implementation of the separate agents, will be covered in the following sections.

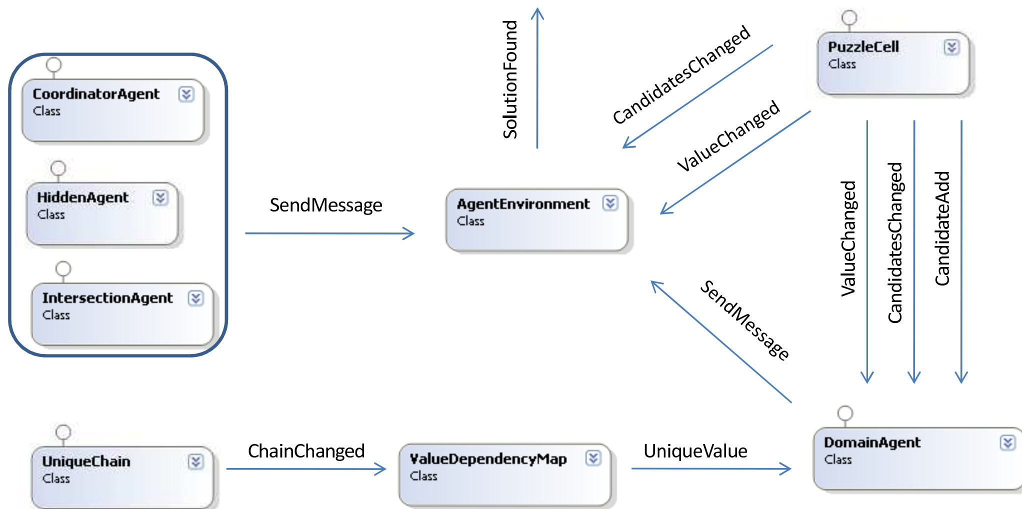


Figure 7.2: Event and message system

The Domain agents

The agent holds a reference to all the cells in its domain, and in order to satisfy the domain constraint the domain catches events, when a state change occurs in a cell. When a value is sat in a cell, the `ValueChanged` event is caught and handled by the `DomainAgent`. Every cell is contained in 3 domain agents, so the event is caught in 3 different agents, all performing the same evaluation in respect to their own domain. When noticing a `ValueChanged` event, the agent composes a message containing all the eliminations it can perform in its domain. This proposition is then send to the `CoordinatorAgent`. See figure 7.2 for an overview of the event and message flow.

In order to recognize when a cell is eliminating a value, the `CandidatesChanged` event is caught, as seen in figure 7.2. If a cell only has one candidate value left, the `DomainAgent` creates a `ValueSolutionStep` and proposes this to the `CoordinatorAgent`. The detection of a value solution step is therefore done in linear time.

To manage the unique chains, the `DomainAgent` holds a reference to a special object, namely the `ValueDependencyMap`. The `ValueDependencyMap` is a construct, which is used to manage the connections between the cells. The unique chain is encapsulated in a `UniqueChain` object. The `ValueDependencyMap` therefore has a reference to all the value dependencies inside a domain. It is important that this map is up-to-date, therefore it is updated every time the domain registers a `CandidatesChanged` or `ValueChanged` event. In every update the `ValueDependencyMap` throws an `UniqueValueFound` event, if a `UniqueChain` only contains a single cell. If this event is caught, the `DomainAgent` creates a `ValueSolutionStep` and proposes this to the `CoordinatorAgent`. The detection is therefore done in $O(n^2)$ time, since it takes either $\mathrm{O}(n)$ time to update the map if an elimination occurred, or $\mathrm{O}(n^2)$ if a value was sat according to the Design section.

Finally, the `DomainAgent` is able to respond to requests from the strategy agents, by returning either a `CellMessage`, containing the appropriate cells for the naked agent, or by returning a `ValueDependencyMessage` containing the unique chains of the domain to the hidden or intersection agent.

The Coordinator agent

The `CoordinatorAgent` manages the entire agent system. The steps towards the solution are executed in a synchronized manner. This is ensured by only executing solution steps, when it receives a `NextStepMessage`, indicating that the system is ready to perform the next step. It is however the coordinator agent itself, which sends these messages in appropriate situations. E.g. when the domain agent executes a value solution step, it waits to execute any other value solution steps, until it has received a `EliminationStrategyMessage` from the three affected `DomainAgent`

threads. The different messages and performatives can be seen in table 7.1

Content message	Performative	Sender	Reciever
`NextStepMessage`	Request	CA	CA
`StrategyMessage`	Request	CA	HA, NA, IA
	Refuse	HA, NA, IA	CA
`CellMessage`	Request	NA	DA
	Inform	DA	NA
`ValueDependencyMessage`	Request	HA, IA	DA
	Inform	DA	HA, IA
`ConflictMessage`	Inform	DA	CA
`SolutionStepMessage`	Propose	DA	CA
`EliminationStrategyMessage`	Propose	HA, NA, IA, DA	CA
`SolutionMessage`	Inform	CA	-

Table 7.1: Message types. The following abbreviations of the agents is used. CoordinatorAgent = CA, HiddenAgent = HA, NakedAgent = NA, IntersectionAgent = IA, DomainAgent = DA

When the coordinator agent is no longer able to execute a solution step, it requests a solution step from one of the configured strategy agents. This is done by sending a `StrategyMessage` to the appropriate strategy agent.

If the strategy agents are unable to find further solution steps, the backtracking procedure is started. Here the domain agent sends a `ConflictMessage` to the `CoordinatorAgent`, if a conflict is detected in the `DomainAgent`.

When a solution is found, the coordinator sends the `SolutionMessage`, which has no receiver, but causes the `AgentEnvironment` to raise the `SolutionFound` event. The coordinator determines that a solution is found, when every cell in the puzzle has a value. Since the domain agents ensures the domain constraints, it is certain that it is a valid solution.

In figure 7.3 the flow of a single solution step is shown. The figure shows the events and messages caused, when a value is sat in a `PuzzleCell`.

The Strategy agents

The implementation of the strategy agents closely follows the description in the Design section. When a strategy agent receives a `StrategyMessage`, it tries to request a `DomainAgent` for the appropriate information. The selection of domains to search is random, however the following criteria is also applied: The state of the domain chosen, must have changed since the last time the agent 'visited'. This

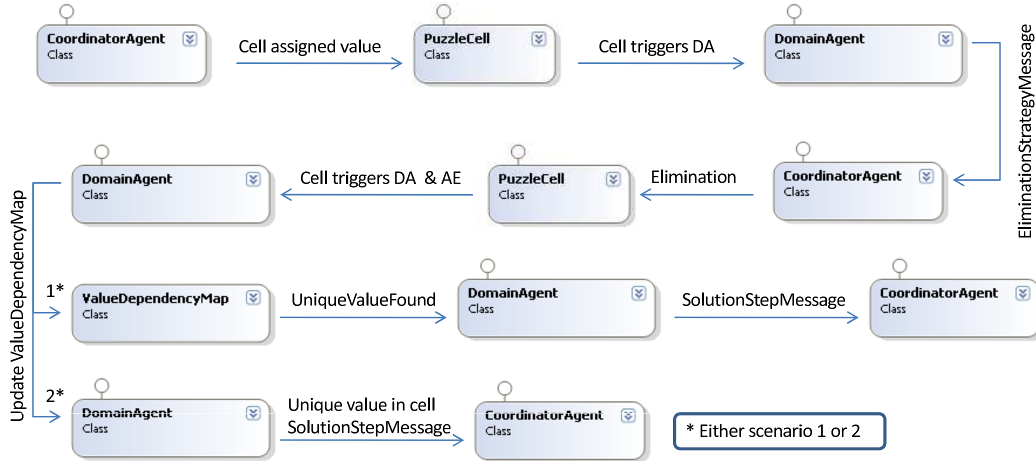


Figure 7.3: Flow of a single solution step. DomainAgent = DA, AgentEnvironment = AE

ensures that no domain is searched unnecessary. If it has searched all domains, or the domains have not changed, the agent refuses the `StrategyMessage` giving the `CoordinatorAgent` a chance to pursue other strategies or backtrack search.

7.5 Test

The main goal of this section is to firstly evaluate the performance of the multi-agent Sudoku solver with different configurations, and secondly compare it to other solution methods.

First, the Sudoku solver is run with different combinations of the strategies used. This will indicate how well the strategies perform separately and hopefully show that a combination has a synergistic effect on the performance.

Since Sudoku is a fairly new research subject, only few benchmark sets are available. However, solving Sudoku puzzles and creating solvers for Sudoku has been the goal for many people in the Sudoku community. The data sets used is therefore found through the Sudoku community [2]. Since most order 3 Sudoku puzzles are quickly solved by an average Sudoku solver, the following two benchmark sets are used, which contain some of the hardest order 3 puzzles:

sudoku17 Gordon Royle's collection of Sudoku Puzzles with 17 clues. As mentioned earlier, 17 clues is believed to be the minimum number of clues. The

[2]http://www.setbb.com/phpbb/ - sudoku programmers forum

data set is used throughout the community, as a common benchmark ([1]).

top1465 A list which is claimed to be the top 1465 hardest Sudoku puzzles of order 3. This data set is also a common benchmark in the community. ([3])

The order 4 Sudoku puzzles or higher, do not get as much attention in the Sudoku community, because they are a lot more difficult to solve for a human. Secondly, the main focus of the Sudoku community is to discover new ways to aid humans in solving classical Sudoku puzzles. This means that it is difficult to find benchmark sets containing the bigger puzzles. However, the following benchmark sets have been chosen, but it only contains a few puzzles compared to the benchmarks sets for the order 3 puzzles:

menneske.no Data set constructed from the order 4 puzzles presented at http://menneske.no

7.5.1 Test of configuration

In this test the configuration of the agent system is evaluated. The test is performed with the following combinations of strategies. All the tests are run on a Pentium Core2Duo 2,17 Mhz with 2 GB of RAM:

- Hidden Set Strategy (H)
- Naked Set Strategy (N)
- Intersection Set Strategy (I)
- Hidden- and Intersection Set Strategies (HI)
- Naked- and Intersection Set Strategies (NI)
- Hidden-, Naked- and Intersection Set Strategies (HNI)

The test is performed on a representative part of the sudoku17 benchmark set, namely the 1500 first entries. The results are shown in figure 7.2.

It is obvious that the optimal configuration, in regards to puzzles solved without search, is the NHI configuration, although only marginally better than HI. The strategy that alone performs best is the Intersection Set strategy. It is seen that the three combinations of strategies all have a synergistic effect with respect to avg. guesses and the percentage solved without search. However, the avg. solution time does not receive a corresponding synergistic effect. This is due to the fact that the more strategies used, the bigger is the actual search space, since every strategy searches the entire puzzle for occurrences of its target strategy set.

Strategy	Avg. time (ms)	Avg. guesses	% solved without search
H	14,8	4	64,9
N	14,8	6	64,5
I	10,9	4	75,5
HI	14,3	4	83,5
NI	14,3	4	82,0
NHI	17,2	3	85,3

Table 7.2: Test results on the first 1500 puzzles of the sudoku17 benchmark set

7.5.2 Test of order 3 puzzles

Since the foremost goal is to solve the Sudoku puzzles quickly without the use of search, the optimal configuration is NHI. This configuration is therefore used on a significantly more difficult benchmark set, the top1465. The results are shown in figure 7.3.

Strategy	Avg. time (ms)	Avg. guesses	% solved without search
NHI	76,5	11	15,6
I	31,7	18	2,7

Table 7.3: Test results on the first 1500 puzzles of the sudoku17 benchmark set

It is clearly seen that the solver is unable to solve the majority of the puzzles without search. This is most possible due to the fact that the puzzles in this benchmark set does not contain very many clues, which the three strategies can detect. Here much more advanced strategies are needed.

It is worth to note that the puzzles in the top1465 benchmark are significantly more difficult than a puzzle rated difficult in the local newspaper. This argument is based on the fact that the majority of a random sample of the puzzles in the benchmark set, could not be solved by any of the currently known strategies that does not involve guessing. The test was carried out on www.scanraid.com/sudoku.htm, which implements the majority of the known Sudoku strategies.

7.5.3 Test of order 4 Sudoku

With the optimal configuration in place, the same test is performed on the order 4 benchmark set. The result is seen in figure 7.4.

It is seen that the solver is able to solve order 4 Sudoku puzzles within reasonable time. The test also shows that the benchmark set is difficult to solve without

Strategy	Avg. time (ms)	Avg. guesses	% solved without search
NHI	311	7	35,7
I	118	29	3,6

Table 7.4: Test results on the menneske.no benchmark set

search. This indicates that the complexity of the benchmark set is more similar to the top1465 benchmark set, than the sudoku17 benchmark set.

7.5.4 Comparison with other solvers

In order to evaluate how efficient the solver is, it is measured against other Sudoku solvers and a single SAT solver in the following. Thereafter the results from the previous tests are compared with test results presented in [23].

The Sudoku solvers are chosen through the Sudoku community, which recommends the following solvers:

Sudoku1 The solver uses depth first and/or breadth first tree search with constraint propagation to prune the search for the next best move (forms of forward checking). The common characteristic for all constraints, here and elsewhere, is that they avoid trial and error ([2]).

Suexg Based on a Dancing Links Engine. In computer science, Dancing Links, also known as DLX, is the technique suggested by Donald Knuth to efficiently implement his Algorithm X. Algorithm X is a recursive, nondeterministic, depth-first, brute-force algorithm that finds all solutions to the exact cover problem. [3]

Before the SAT solver can be used to solve any puzzle, it has to be converted to a SAT instances in CNF format using the tool provided in [3]. The solver chosen for the comparison is:

SATZ Is based on a DPLL backtracking algorithm, which as explained earlier is a common SAT solving technique. The SATZ implementation is based on [21].

With three other solvers a comparison with ours can be made. The three solvers have tried to solve the sudoku17, top1465 and menneske.no benchmark sets. The results are shown in figure 7.5.

Solver	No. puzzles	Benchmark	Avg. time (ms)
Solver1	1500	sudoku17	1
Solver1	1465	top1465	60
Solver1	28	mennesko.no	N/A
Suexg	1500	sudoku17	1
Suexg	1465	top1465	2
Suexg	28	menneske.no	6
SATZ	1500	sudoku17	57
SATZ	1465	top1465	64
SATZ	28	menneske.no	200
MASS	1500	sudoku17	18(11)
MASS	1465	top1465	80(32)
MASS	28	menneske.no	311(118)

Table 7.5: Comparison of solvers

It can be seen that our solver (MASS) cannot compete with the specialized Sudoku solvers when all strategies are used, but when only the intersection strategy is used (shown in the brackets), it performs better than the Sudoku1 solver for the top1465 benchmark. Additionally it is noticed that our solver, when using all strategies, in one case outperforms the SAT solver. When using the fast configuration it outperforms the SAT solver on all the benchmarks. It can therefore be concluded that our solver is able to solve Sudoku puzzles in comparable time or better than the SAT solver, but stands little chance against the specialized solvers.

In [23] the performance of the solver is measured on percentage of Sudoku puzzles solved without search. The article uses the sudoku17 benchmark and four different propagation schemes, namely:

- Unit propagation (up);
- Unit propagation + failed literal rule (up+flr);
- Unit propagation + hyper-binary resolution (up+hypre);
- Unit propagation + binary failed literal rule (up+binflr).

For an explanation see [23].

When using the minimal encoding (described earlier), the maximum encoding (described in the article), and the four propagation schemes, the following results are obtained. See table 7.6.

	% solved without search
Minimal, up	0
Minimal, up+flr	1
Minimal, up+hypre	25
Minimal, up+binflr	69
Maximal, up	47
Maximal, up+flr	100
Maximal, up+hypre	100
Maximal, up+binflr	100
MASS (NHI)	85

Table 7.6: Comparison of solvers

It is seen that our solver actually is better than the SAT solver to determine solutions without search, when the minimum encoding is used. It is also comparable to the SAT solver, when the maximal encoding is used.

7.6 Discussion

The tests show that our Sudoku solver is able to solve every puzzle, we have presented. Furthermore it shows that the configuration of the system can be adjusted to either accommodate solution without search using all the strategies implemented, or accommodate a fast solution by only using the Intersection Set strategy.

The configuration which utilize all the implemented strategies are not able to solve the most difficult Sudoku puzzles without search. This is due to the limitations in the implemented strategies. However, some of the puzzles are so difficult to solve that even an implementation of all known strategies would not yield a solution. It is therefore obvious that our system has a shortcoming, when solving puzzles where the strategies are not sufficient. If a puzzle has few or no solution steps, which can be detected by the strategies, the work of the strategies is redundant. It is however as we know, impossible to know beforehand, if a strategy can be used. A possible area of research is to improve this shortcoming by classifying, when certain strategies are suitable in respect to certain puzzle characteristics such as the puzzles difficulty. This has however been outside the scope of this thesis. The other approach is to abandon the requirement of solving the Sudoku without search and instead use a dedicated search algorithm.

The test also shows that compared with the other solvers, our solver was not superior in respect to running time. Some of the important reasons for this are:

- Our strategies are to specific, as they only search for specific patterns. It is very time consuming to use these strategies, if no solution steps are found. This could perhaps be improved, if the strategies where defined on a higher level of abstraction, meaning that a search could satisfy more than one strategy.

- Even though our system is a multi-agent system, the strategy agents act in a sequential manner. Running them in parallel could most likely result in a gain in performance. This is however only the case when the system is running on a machine with multiple processors.

- The need for communication and synchronization between the agents is also a possible bottleneck for the systems performance.

Even though the specialized Sudoku solvers are faster than our implementation in most cases, the tests shows that in the fastest configuration our solver outperforms the SAT solver for the given benchmarks. This indicates that the system is actually capable of competing with some of the well known solving techniques for $\mathcal{NP}$-complete problems. Furthermore, the solver is able to solve almost as many puzzles without search as the SAT solver covered in [23]. This aspect of our solver could obviously be improved, if more strategies where implemented.

Building and testing our solver has given some insights into the subject of multi-agent systems versus single-agent systems. In our case the multi-agent system was chosen for scalability and flexibility, however if this was irrelevant, it would undoubtedly be more efficient to implement the system as a single agent. This would improve the performance by removing the synchronization issues and the communication costs. The decision of whether it should be a multi- or single-agent system is therefore dependent of the requirements of the system. The advantages by using a multi-agent system is that the system is not as strongly coupled as a single-agent system. This gives more flexibility to test different configurations. This could be useful in the process of determining a good solution strategy or heuristic to a problem. In our example it gave us the possibility to test different configurations of the strategies, which showed that if the goal was to solve the puzzles without search all of the strategies worked together in a synergistic way. However, if the goal was to solve the puzzles quickly, the fastest strategy (Intersection Set) was sufficient.

Another important aspect of our solution is to evaluate how general it is in aspect of solving a $\mathcal{NP}$-complete problem. It is obvious that it is not capable of competing with the most evolved Sudoku solvers, instead the approach is a more generalized solution strategy similar to the ones mentioned in Chapter 3. The structure of a Sudoku puzzle is very similar to a SAT problem therefore with a few changes our system would most likely be able to solve certain SAT instances.

7.7 Conclusion

In this chapter we have successfully designed, implemented and tested a multi-agent Sudoku solver. The solver has proven efficient in respect to solve the puzzles without search. It has also shown a comparable performance against a SAT solver, but it is outperformed by specialized Sudoku solvers.

We have elaborated on the aspects of using a multi-agent system for our solver, and can conclude that it has advantages in respect to scalability and flexibility, but has some disadvantages in respect to performance and simplicity. A single-agent system could have been constructed in a more clear fashion. Our system is not matured, and several improvements can be made, which is mentioned in the discussion and will be covered in the final conclusion.

Chapter 8

Conclusion

The project had two goals. Firstly, it should include a general discussion of the use of multi-agent approaches to solving $\mathcal{NP}$-complete problems, and additionally a discussion of the strengths and weaknesses compared to other approaches of solving the same problems. Secondly, a concrete software tool should be implemented for solving the $\mathcal{NP}$-complete problem Sudoku. The following conclusion will therefore consists of two parts; one for each of the two objectives.

From the analysis of the different MAS approaches to solving $\mathcal{NP}$-complete problems, it was obvious that a MAS can be used in solving these problems, but we found that there is different conceptions of the term MAS in regards to solving the problems. A multi-agent system can be used as a liaison between different meta-heuristics, where each heuristic is represented by an agent. The meta-heuristic agents then cooperate by sharing their solutions with each other, to achieve a better result. Another approach is to use a MAS in solving constraint satisfaction problems. The idea is to divide constraints and variables between agents, and in a suitable manner let the individual agents optimize their local perspective to reach an optimal common global solution. The third and last approach was a new way to represent the problems, not similar to any other solvers. It works by analysing a real world instance of the problem and in this way determine what entities are at play and how they interact to solve the problem, for then afterwards to make a MAS inspired by this interaction. The conclusion is therefore that the term multi-agent system can be used as a design paradigm when solving complex problems. The multi-agent paradigm attacks the problem from a high level of abstraction and can therefore result in many different interpretations.

The comparison between the multi-agent systems and the common approaches showed that multi-agent systems possess a number of advantages. First of all if the problem can be divided into smaller problems, you can receive a gain in computational performance. Additionally a MAS is scalable, which means that it is easy to change or remove agents and thereby test new strategies or heuristics. A MAS is also robust if it has agents that complement each other, meaning that a part of the system can fail, but the overall system can still succeed. However, the communication cost can be factor for the performance of the system. It is therefore important that the communication cost is kept down, to get a fast solution. Besides these general considerations, the comparison also showed that the common approaches are more optimized and specialized than the MAS, hence undoubtedly faster than the MAS algorithms. However, it is to soon to abandon the MAS approach, since it is a new way to represent the problem, which in the long run could lead to effective solvers, when further optimized and specialized.

Finally it can be concluded that a MAS can successfully be applied in designing a Sudoku solver. The solver is capable of solving the presented puzzles, which includes the most common benchmarks. It has proven efficient in solving these puzzles, but the system is not fully matured. As it exists today, it is outperformed by the specialised Sudoku solvers, however the solver was not intended to be a specialized solver, but a more general approach in solving a $\mathcal{NP}$-complete problem. The advantages of our system is that it is scalable and flexible. A new strategy can be added to the system with a minimum amount of effort. Likewise it is possible to change the configuration to fit certain problem instances fairly easy. Some of the disadvantages of our system is the performance compared to specialized solvers.

8.1 Future work

As discussed previously our solution gives rise to a couple of improvements. First of all the current strategies could be optimized. An obvious optimization is to improve the way the strategy agents searches for new solution steps. A more clever heuristic could perhaps minimize the search space that the strategy agents traverse, by determining when it is undesirable to continue the search. Another improvement is to make the strategy agents more cooperative, e.g. by sharing knowledge about their individual search space, and perhaps helping each other in fulfilling their design objectives.

The human way of solving a Sudoku puzzle is sequential, and as our solver tries to mimic human solving techniques the construction of the solution is done in a sequential manner. This does not exploit the full potential of a multi-agent system. Therefore an improvement would be to try and run parts of the system in parallel. The performance of the strategy agents could for instance be improved by running

them in parallel.

In this scope, not all of our plans where possible within the time frame. In the future it could therefore be interesting to both implement more advanced strategies, look into training of our agents and consider even larger Sudoku puzzles.

Finally the overall architecture of the system could possibly be improved. It could also be interesting to transfer our system to an existing multi-agent platform, in order to receive the benefits of an matured agent environment.

Bibliography

[1] http://people.csse.uwa.edu.au/gordon/sudokumin.php.

[2] http://www.research.att.com/ gsf/.

[3] http://magictour.free.fr/sudoku.htm, 2005.

[4] M. Emin Aydin and Terence C. Fogarty. Teams of autonomous agents for job-shop scheduling problems: An experimental study. *Journal of Intelligent Manufacturing*, 15(4):455–462, 2004.

[5] Andrew C. Bartlett and Amy N. Langville. An integer programming model for the sudoku problem, 2006.

[6] Charles J. Colbourn. The complexity of completing partial latin squares. *Discreate Applied Mathematics*, (8):25–30, 1984.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[8] T. Feo and M. Resende. Greedy randomized adaptive search procedures, 1995.

[9] Paul Fischer. *Computationally Hard Problems*. IMM, DTU, 2006.

[10] Zhaohui Fu, Yogesh Marhajan, and Sharad Malik. zchaff sat solver. Technical report, Princeton University, 2005.

[11] Fred W. Glover and Gary A. Kochenberger. *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer, January 2003.

[12] Jacob Goldberger. *Solving Sudoku Using Combined Message Passing Algorithms*. PhD thesis, School of Engineering, Bar-Ilan University.

[13] Moez Hammami and Khaled Ghédira. Cosats, x-cosats: Two multi-agent systems cooperating simulated annealing, tabu search and x-over operator for the k-graph partitioning problem. In *KES (4)*, pages 647–653, 2005.

[14] Katsutoshi Hirayama and Makoto Yokoo. Local search for distributed sat with complex local problems. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1199–1206, New York, NY, USA, 2002. ACM Press.

[15] Katsutoshi Hirayama and Makoto Yokoo. The distributed breakout algorithms. *Artif. Intell.*, 161(1-2):89–115, 2005.

[16] Xiaolong Jin and Jiming Liu. Multiagent sat (massat): Autonomous pattern search in constrained domains. In *IDEAL '02: Proceedings of the Third International Conference on Intelligent Data Engineering and Automated Learning*, pages 318–328, London, UK, 2002. Springer-Verlag.

[17] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing, 1983.

[18] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[19] Hon Wai Leong and Ming Liu. A multi-agent algorithm for vehicle routing problem with time window. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 106–111, New York, NY, USA, 2006. ACM.

[20] Rhyd Lewis. Metaheuristics can solve sudoku puzzles. *Journal of Heuristics*, 13(4):387–401, 2007.

[21] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.

[22] JyiShane Liu and Katia Sycara. Distributed problem solving through coordination in a society of agents. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, 1994.

[23] Inês Lynce and Jöel Ouaknine. Sudoku as a sat problem. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (AIMATH 2006)*, January 2006.

[24] Agnes M. Herzberg and M. Ram Murty. Sudoku squares and chromatic polynomials. *Notices of the AMS*, 54(6), 2007.

[25] Yannis Marinakis, Athanasios Migdalas, and Panos M. Pardalos. Expanding neighborhood search – grasp for the probabilistic traveling salesman problem. *Optimization Letters*, 2007.

[26] Yannis Marinakis, Athanasios Migdalas, and Panos M. Pardalos. A new bilevel formulation for the vehicle routing problem and a solution method using a genetic algorithm. *J. of Global Optimization*, 38(4):555–580, 2007.

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[28] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[29] Helmut Simonis. Sudoku as a constraint problem. In Brahim Hnich, Patrick Prosser, and Barbara Smith, editors, *Proc. 4th Int. Works. Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005.

[30] Yato Takayuki and Seta Takahiro. Complexity and completeness of finding another solution and its application to puzzles. Technical report, The University of Tokyo, 2003.

[31] E.-G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8(5):541–564, 2002.

[32] W. van Hoeve. The alldifferent constraint: A survey, 2001.

[33] José M. Vidal. *Fundamentals of Multiagent Systems: Using NetLogo Models*. Unpublished, 2006. http://www.multiagent.com/fmas.

[34] Tjark Weber. A SAT-based Sudoku solver. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings*, pages 11–15, December 2005.

[35] Wikipedia. http://en.wikipedia.org/wiki/Mathematics_of_Sudoku, 2007.

[36] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2002.

[37] T. Zhou, Yihong Tan, and L. Xing. A multi-agent approach for solving traveling salesman problem. *Wuhan University Journal of Natural Sciences*, 11(5), 2005.

Appendix A

User manual

To visualize the function of the multi-agent Sudoku solver a graphical user interface is constructed. The following is a user manual for the GUI that illustrates the different functionalities and describes how to use them. The program can be found in the attached CD in the folder *Multi-agent Sudoku Solver*.

Load and solve a puzzle

When the program is executed the window in figure A.1 is displayed. From here it is possible to load a puzzle by clicking on the button *Load*. To test the program numerous puzzles are available in a folder called *Data* placed in the same location as the program. There are only puzzles of order 3 and 4, as the GUI only handles these sizes even though the solver is capable of solving puzzles of any order.

When the puzzle is loaded the clues are displayed in a grid. In figure A.2 a puzzle is loaded containing 18 clues, where the number of clues can be found in the shaded box called *Info*. If the button *Solve* is pressed, the puzzle is solved.

Analyze the result

In figure A.3 a solution is found to the puzzle loaded. A number of information concerning the solution can be found in the info box. In the lower left corner of

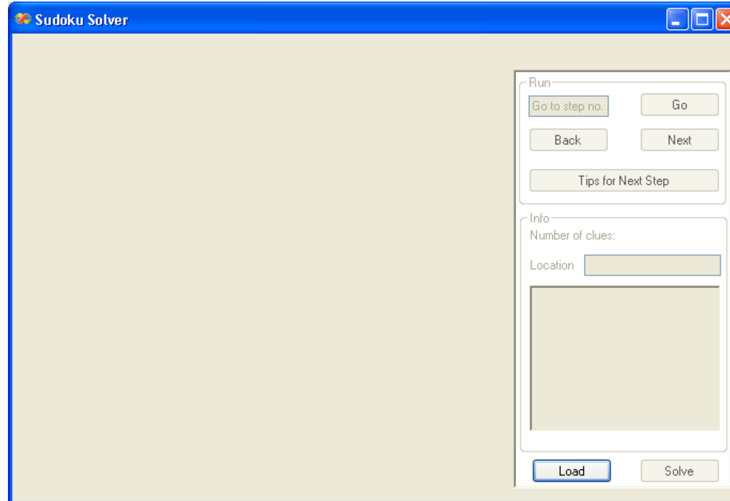


Figure A.1: The solver is started.

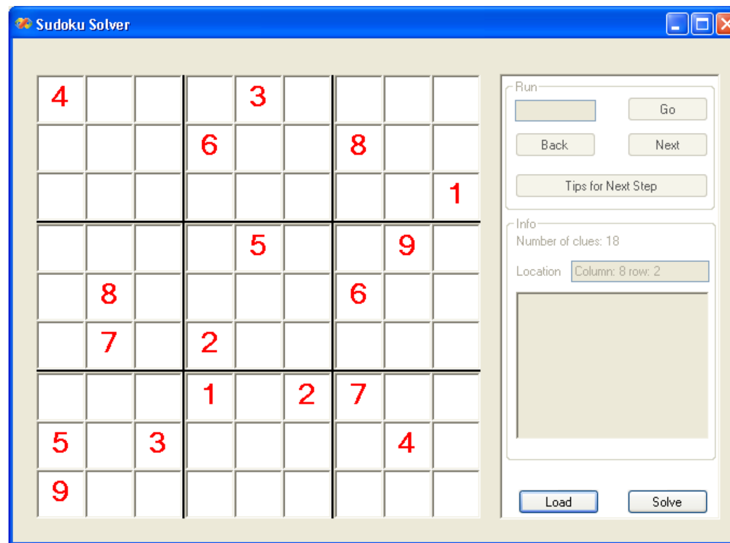


Figure A.2: A puzzle is loaded.

the info box the correctness of the solution is displayed. In this case the solution is correct. Above the solution verification, the execution time can be found in a text box. It is important to mention that this is not the correct execution time, as it includes both the solution time for the Sudoku solver and the caching during the computation of the solution. In the same text it it additionally displayed whether the solution is found using only strategies or with search. It is seen that the current solution is found with search, since it was only possible to solve the puzzle using strategies to step 21. A step contains all the eliminations performed until it is

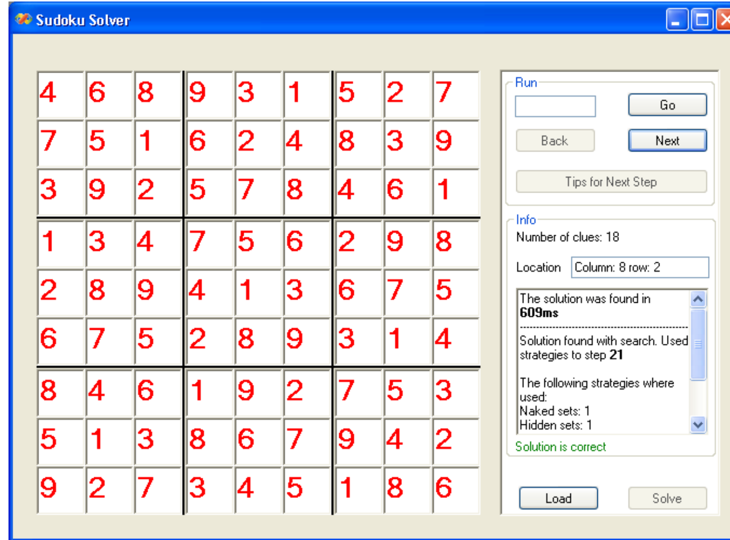


Figure A.3: The puzzle is solved.

possible to place an unambiguous value in a cell. That is, a order 3 puzzle has 81 steps and a order 4 puzzle has 256 steps if search is not used. In the text box there are furthermore information about how many times the different strategies are used. The number of strategies used only reflect the steps where the strategies cause eliminations. However, there can be more hidden, naked and intersection sets found in the steps not yielding any eliminations for the strategies.

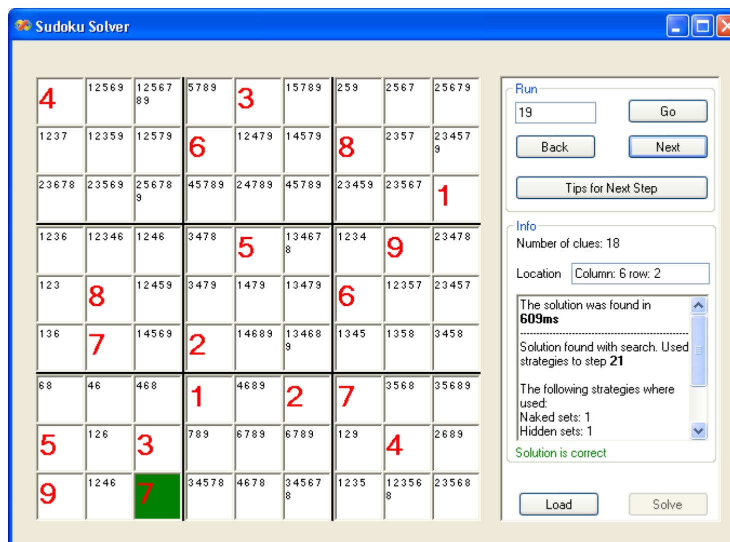


Figure A.4: The Run box is used.

Go trough the solution

In the run box it it possible to go back and forth between the different steps as well as going to a specific step. To go to a specific step you need to type in the desired step number in the text box left to the button *go* and press the button afterwards. In figure A.4 the step number 19 is displayed, where the green cell indicates that this cell was the latest cell assigned a value in the puzzle.

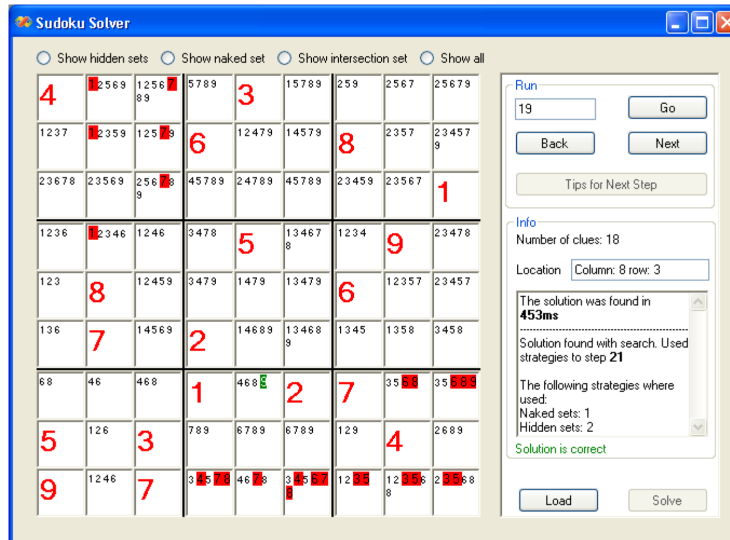


Figure A.5: Tips for the next step.

To get information about the next step, it is possible to press the button *Tips for next step* to show the eliminations performed to determined the next value. In figure A.5 the result is shown after the button is pressed for the step showed in the previous figure. The candidate highlighted with green indicates the next value, and the candidates highlighted with red indicate the eliminated candidates that cause the value to be sat.

A deeper explanation to the eliminations can be displayed, if the strategies are used, by clicking on one of the buttons above the grid that was enabled, when the tip button was clicked. In the step in figure A.4 every strategy is in use, as all of the buttons are enabled. In figure A.6, A.7 and A.8 respectively the hidden set strategy, the naked set strategy and the intersection set strategy is displayed. In figure A.6 two hidden sets are shown both with 3 and 5 as the hidden values. The candidates highlighted with the dark green are the eliminated candidates in the strategy. In figure A.7 one naked candidate set is shown with the value set 3 and 5. The eliminated candidates are highlighted with dark green. In figure A.8 one intersection set is shown with the value 1. The eliminated candidates are highlighted with dark green. To show all the eliminations again you can click on the button

Show all.

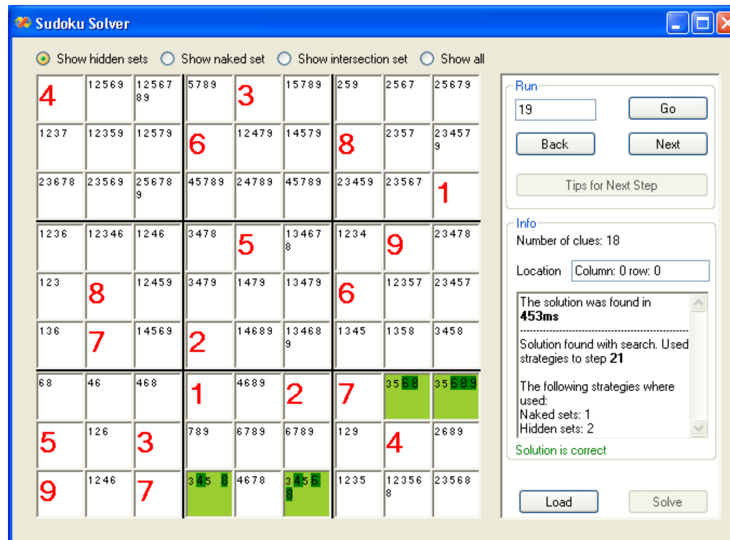


Figure A.6: Hidden strategy.

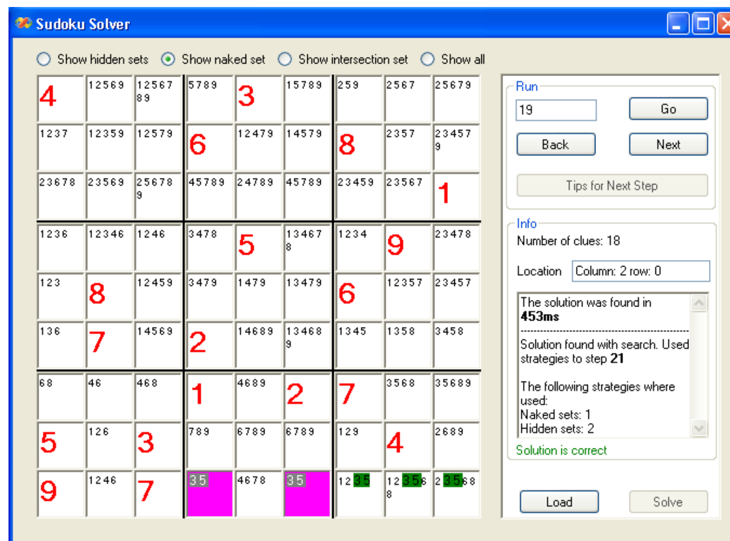


Figure A.7: Naked strategy.

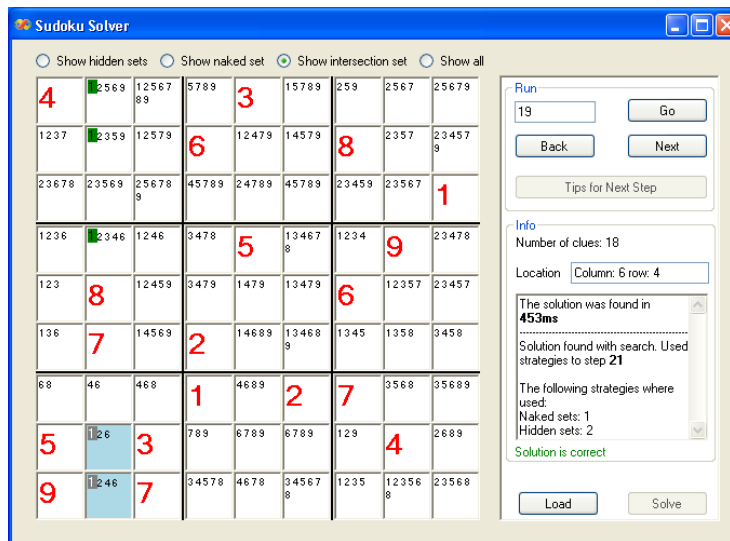


Figure A.8: Intersection strategy.

Appendix B

Source code

B.1 Agent Environment

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Collections;
5  using System.Collections.ObjectModel;
6  using MultiAgentSudokuSolver.Messaging;
7  using MultiAgentSudokuSolver.Agents;
8  using MultiAgentSudokuSolver.Data;
9  using System.Threading;
10 using MultiAgentSudokuSolver.Cache;
11
12
13 namespace MultiAgentSudokuSolver
14 {
15     /// <summary>
16     /// Class that handles the communication between agents, and records the state of the puzzle
17     /// </summary>
18     public class AgentEnvironment
19     {
20         #region Variables
21         private readonly int puzzleSize, puzzleOrder;
22         private string[] data;
23         private PuzzleCell[,] cells;
24         private Queue<EventArgs<FIPAAclMessage>> messageQueue = new Queue<EventArgs<FIPAAclMessage>>();
25
26         private bool useCache;
27         private SolutionBuilder solution;
28         private CacheSolutionStep currentSolutionStep;
29
30         #endregion Variables
31
32         #region Agents
33         private CoordinatorAgent coordinatorAgent;
34         private NakedAgent nakedAgent;
35         private HiddenAgent hiddenAgent;
36         private IntersectionAgent intersectionAgent;
37         private List<DomainAgent> squareAgents;
38         private List<DomainAgent> rowAgents;
39         private List<DomainAgent> columnAgents;
40         #endregion Agents
41
42         Thread coordinatorThread;
43         Thread nakedThread;
44         Thread hiddenThread;
```

```
45          Thread intersectionThread;
46
47          List<Thread> domainThreads;
48
49          private delegate void SendMessageDelegate(object sender, EventArgs<FIPAAclMessage> e);
50
51          public AgentEnvironment(string[] data, bool useCache)
52          {
53              this.data = data;
54              puzzleSize = (int)Math.Sqrt(data.Length);
55              puzzleOrder = (int)Math.Sqrt(puzzleSize);
56              this.useCache = useCache;
57              solution = new SolutionBuilder(puzzleSize);
58              currentSolutionStep = new CacheSolutionStep(puzzleSize);
59              this.Initialize();
60          }
61
62          public void InvokeSendMessage(object sender, EventArgs<FIPAAclMessage> e)
63          {
64              SendMessageDelegate del = new SendMessageDelegate(this.agent_SendMessage);
65              del(sender, e);
66          }
67
68          internal void Register(IAgent agent)
69          {
70              agent.SendMessage += new EventHandler<EventArgs<FIPAAclMessage>>(agent_SendMessage);
71          }
72
73          public PuzzleCell[,] GetPuzzleCells()
74          {
75              return cells;
76          }
77
78          public void Initialize()
79          {
80              // Set up agents
81              coordinatorAgent = new CoordinatorAgent(puzzleSize);
82              nakedAgent = new NakedAgent(puzzleSize);
83              hiddenAgent = new HiddenAgent(puzzleSize);
84              intersectionAgent = new IntersectionAgent(puzzleSize);
85
86              Register(coordinatorAgent);
87              Register(nakedAgent);
88              Register(hiddenAgent);
89              Register(intersectionAgent);
90
91              coordinatorThread = new Thread(new ThreadStart(coordinatorAgent.Run));
92              coordinatorThread.Start();
93
94              nakedThread = new Thread(new ThreadStart(nakedAgent.Run));
95              nakedThread.Start();
96
97              hiddenThread = new Thread(new ThreadStart(hiddenAgent.Run));
98              hiddenThread.Start();
99
100             intersectionThread = new Thread(new ThreadStart(intersectionAgent.Run));
101             intersectionThread.Start();
102
103             squareAgents = new List<DomainAgent>(puzzleSize);
104             rowAgents = new List<DomainAgent>(puzzleSize);
105             columnAgents = new List<DomainAgent>(puzzleSize);
106
107             DomainAgent rowAgent, columnAgent, squareAgent;
108             for (int i = 0; i < puzzleSize; i++)
109             {
110                 rowAgent = new DomainAgent(puzzleSize);
111                 columnAgent = new DomainAgent(puzzleSize);
112                 squareAgent = new DomainAgent(puzzleSize);
113
114                 squareAgents.Add(squareAgent);
115                 rowAgents.Add(rowAgent);
116                 columnAgents.Add(columnAgent);
117
118                 hiddenAgent.AddDomain(squareAgent);
119                 hiddenAgent.AddDomain(rowAgent);
120                 hiddenAgent.AddDomain(columnAgent);
121                 nakedAgent.AddDomain(squareAgent);
122                 nakedAgent.AddDomain(rowAgent);
123                 nakedAgent.AddDomain(columnAgent);
124                 intersectionAgent.AddSquareDomain(squareAgent);
125                 intersectionAgent.AddColumnDomain(columnAgent);
126                 intersectionAgent.AddRowDomain(rowAgent);
127
128                 Register(squareAgent);
129                 Register(rowAgent);
130                 Register(columnAgent);
```

```
131             }
132
133             PuzzleCell cell;
134             cells = new PuzzleCell[puzzleSize, puzzleSize];
135             int square;
136
137             // Add PuzzleCells to the correct domain agents, and register events
138             for (int i = 0; i < puzzleSize; i++)
139             {
140                 for (int j = 0; j < puzzleSize; j++)
141                 {
142                     square = (int)((j / puzzleOrder) + (i / puzzleOrder) * puzzleOrder);
143                     cell = new PuzzleCell((int)puzzleSize, i, j, square);
144                     cells[j, i] = cell;
145                     cells[j, i].CandidatesChanged += new EventHandler<EventArgs<int>>(
                             AgentEnvironment_CandidatesChanged);
146                     cells[j, i].ValueChanged += new EventHandler<EventArgs<Nullable<int>>>(
                             AgentEnvironment_ValueChanged);
147                     squareAgents[square].AddCell(cell);
148                     rowAgents[i].AddCell(cell);
149                     columnAgents[j].AddCell(cell);
150                 }
151             }
152
153             Thread agentThread;
154             domainThreads = new List<Thread>(3 * puzzleSize);
155             // Start the domain agent threads
156             for (int i = 0; i < puzzleSize; i++)
157             {
158                 agentThread = new Thread(new ThreadStart(((DomainAgent)squareAgents[i]).Run));
159                 agentThread.Start();
160                 domainThreads.Add(agentThread);
161                 agentThread = new Thread(new ThreadStart(((DomainAgent)rowAgents[i]).Run));
162                 agentThread.Start();
163                 domainThreads.Add(agentThread);
164                 agentThread = new Thread(new ThreadStart(((DomainAgent)columnAgents[i]).Run));
165                 agentThread.Start();
166                 domainThreads.Add(agentThread);
167             }
168
169             coordinatorAgent.InitializeBoard(cells, data, (int)puzzleSize);
170
171         }
172
173         /// <summary>
174         /// Cleanup
175         /// </summary>
176         public void DestroyAgents()
177         {
178             coordinatorThread.Interrupt();
179             nakedThread.Interrupt();
180             hiddenThread.Interrupt();
181             intersectionThread.Interrupt();
182
183             coordinatorThread.Join();
184             nakedThread.Join();
185             hiddenThread.Join();
186             intersectionThread.Join();
187
188             foreach (Thread agent in domainThreads)
189             {
190                 agent.Interrupt();
191                 agent.Join();
192             }
193
194             domainThreads.Clear();
195
196             for (int i = 0; i < puzzleSize; i++)
197             {
198                 for (int j = 0; j < puzzleSize; j++)
199                 {
200
201                     cells[j, i].CandidatesChanged -= new EventHandler<EventArgs<int>>(
                             AgentEnvironment_CandidatesChanged);
202                     cells[j, i].ValueChanged -= new EventHandler<EventArgs<Nullable<int>>>(
                             AgentEnvironment_ValueChanged);
203                     cells[j, i] = null;
204                 }
205             }
206
207             cells = null;
208             coordinatorAgent.SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(
                     agent_SendMessage);
209             nakedAgent.SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(agent_SendMessage);
210             hiddenAgent.SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(agent_SendMessage);
```

```
211             intersectionAgent.SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(
                    agent_SendMessage);
212
213             coordinatorAgent = null;
214             nakedAgent = null;
215             hiddenAgent = null;
216             intersectionAgent = null;
217
218             for (int i = 0; i < puzzleSize; i++)
219                         {
220                 squareAgents[i].SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(
                        agent_SendMessage);
221                 rowAgents[i].SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(
                        agent_SendMessage);
222                 columnAgents[i].SendMessage -= new EventHandler<EventArgs<FIPAAclMessage>>(
                        agent_SendMessage);
223
224                 squareAgents[i] = null;
225                 rowAgents[i] = null;
226                 columnAgents[i] = null;
227                         }
228
229             squareAgents.Clear();
230             rowAgents.Clear();
231             columnAgents.Clear();
232
233         }
234
235         #region EventHandler methods
236         /// <summary>
237         /// EventHandler method that handles the mapping of FIPAAclMessage objects to the correct
                agents
238         /// </summary>
239         public void agent_SendMessage(object sender, EventArgs<FIPAAclMessage> e)
240         {
241             switch (e.Value.MessagePerformative)
242             {
243                 case FIPAAclMessage.Performative.Inform:
244                     if (e.Value.Content is ValueDependencyMessage)
245                     {
246                         if (e.Value.Receiver is HiddenAgent)
247                         {
248                             hiddenAgent.MessageReceived(sender, e);
249                         }
250                         else if (e.Value.Receiver is IntersectionAgent)
251                         {
252                             intersectionAgent.MessageReceived(sender, e);
253                         }
254                     }
255                     else if (e.Value.Content is SolutionMessage)
256                     {
257                         OnSolutionFound(((SolutionMessage)e.Value.Content).Log);
258                     }
259                     else if (e.Value.Content is CellMessage)
260                     {
261                         nakedAgent.MessageReceived(sender, e);
262                     }
263                     else if (e.Value.Content is ConflictMessage)
264                     {
265                         coordinatorAgent.MessageReceived(sender, e);
266                     }
267                     break;
268                 case FIPAAclMessage.Performative.Propose:
269                     if (e.Value.Content is SolutionStepMessage)
270                     {
271                         coordinatorAgent.InvokeMessageReceived(sender, e);
272                     }
273                     else if (e.Value.Content is EliminationStrategyMessage)
274                     {
275                         EliminationStrategyMessage content = (EliminationStrategyMessage)e.Value.
                                Content;
276                         switch (content.StrategyType)
277                         {
278                             case "NakedAgent":
279                                 solution.NakedCount++;
280                                 break;
281                             case "HiddenAgent":
282                                 solution.HiddenCount++;
283                                 break;
284                             case "IntersectionAgent":
285                                 solution.IntersectionCount++;
286                                 break;
287                             case "DomainAgent":
288                                 break;
289                             default:
290                                 break;
```

```
291                             }
292
293                             if (useCache)
294                             {
295                                 currentSolutionStep.AddEliminationStep(content);
296                             }
297                             coordinatorAgent.InvokeMessageReceived(sender, e);
298                         }
299                         break;
300                     case FIPAAclMessage.Performative.Request:
301
302                         if (e.Value.Content is NextStepMessage)
303                         {
304                             if (((NextStepMessage)e.Value.Content).IsSearch)
305                             {
306                                 useCache = false;
307                                 solution.Guesses++;
308                                 solution.IsSearched = true;
309                             }
310                             coordinatorAgent.InvokeMessageReceived(sender, e);
311
312                         }
313                         else if (e.Value.Content is StrategyMessage)
314                         {
315                             if (((StrategyMessage)e.Value.Content).Strategy.Equals("Hidden"))
316                             {
317                                 hiddenAgent.InvokeMessageReceived(sender, e);
318                             }
319                             else if (((StrategyMessage)e.Value.Content).Strategy.Equals("Naked"))
320                             {
321                                 nakedAgent.InvokeMessageReceived(sender, e);
322                             }
323                             else if (((StrategyMessage)e.Value.Content).Strategy.Equals("Intersection"))
324                             {
325                                 intersectionAgent.InvokeMessageReceived(sender, e);
326                             }
327                         }
328
329                         else if (e.Value.Content is ValueDependencyMessage)
330                         {
331                             if (e.Value.Receiver != null)
332                             {
333                                 ((DomainAgent)e.Value.Receiver).InvokeMessageReceived(sender, e);
334                             }
335                         }
336                         else if (e.Value.Content is CellMessage)
337                         {
338                             if (e.Value.Receiver != null)
339                             {
340                                 ((DomainAgent)e.Value.Receiver).InvokeMessageReceived(sender, e);
341                             }
342                         }
343                         break;
344                     case FIPAAclMessage.Performative.Refuse:
345                         if (e.Value.Content is StrategyMessage)
346                         {
347                             coordinatorAgent.InvokeMessageReceived(sender, e);
348                         }
349                         break;
350                     default:
351                         break;
352                 }
353             }
354
355             void AgentEnvironment_ValueChanged(object sender, EventArgs<Nullable<int>> e)
356             {
357                 if ((sender as PuzzleCell).CellValue.HasValue)
358                 {
359                     if (useCache)
360                     {
361                         PuzzleCell cell = (PuzzleCell)sender;
362                         // Save solutionstep (i.e. candidate events, strategy events and value event).
363                         currentSolutionStep.AddValueStep(cell);
364                         solution.SaveSolutionStep((CacheSolutionStep)currentSolutionStep.Clone());
365                         currentSolutionStep.RemoveStrategies();
366                     }
367                 }
368             }
369
370
371             void AgentEnvironment_CandidatesChanged(object sender, EventArgs<int> e)
372             {
373                 if (useCache)
374                 {
375                     currentSolutionStep.AddCandidateStep((PuzzleCell)sender);
376                 }
```

```
377         }
378         #endregion EventHandler methods
379 
380         #region Events
381         public event EventHandler<EventArgs<List<Object>>> DisplayEvent;
382 
383         private void OnDisplayEvent(List<Object> e)
384         {
385             if (DisplayEvent != null)
386             {
387                 this.DisplayEvent(this, new EventArgs<List<Object>>(e));
388                 e.Clear();
389             }
390         }
391 
392         public event EventHandler<EventArgs<Solution>> SolutionFound;
393 
394         private void OnSolutionFound(LogElement log)
395         {
396             if (SolutionFound != null)
397             {
398                 Solution finalSolution = new Solution(solution.GetSolutionSteps(),solution.Guesses,
                         solution.IsSearched, solution.NakedCount, solution.HiddenCount, solution.
                         IntersectionCount);
399                 this.SolutionFound(this, new EventArgs<Solution>(finalSolution));
400             }
401         }
402 
403         #endregion Events
404     }
405 }
```

B.2 Agents

IAgents.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Collections.ObjectModel;
5  using MultiAgentSudokuSolver.Messaging;
6  using MultiAgentSudokuSolver.Data;
7 
8  namespace MultiAgentSudokuSolver.Agents
9  {
10     public interface IAgent
11     {
12         Guid AgentID { get; }
13 
14         event EventHandler<EventArgs<FIPAAclMessage>> SendMessage;
15 
16         void OnSendMessage(EventArgs<FIPAAclMessage> e);
17 
18         void MessageReceived(object sender, EventArgs<FIPAAclMessage> e);
19 
20         void InvokeMessageReceived(object sender, EventArgs<FIPAAclMessage> e);
21 
22         void Run();
23 
24     }
25 }
```

DomainAgent.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Collections;
5  using System.Collections.ObjectModel;
```

```
6   using MultiAgentSudokuSolver.Messaging;
7   using MultiAgentSudokuSolver.Data;
8   using System.Threading;
9   using System.Diagnostics;
10
11
12  namespace MultiAgentSudokuSolver.Agents
13  {
14      /// <summary>
15      /// Handles the domains of the puzzle, e.g. row, columns and squares
16      /// </summary>
17      public class DomainAgent : IAgent
18      {
19          private const string AGENT_TYPE = "DomainAgent";
20
21          #region Variables
22          private Guid agentID = Guid.NewGuid();
23          private List<PuzzleCell> cells;
24          private ValueDependencyMap valueDependencies;
25          private int freeCells;
26          private Queue<EventArgs<FIPAAclMessage>> messageQueue = new Queue<EventArgs<FIPAAclMessage>>();
27          private delegate void MessageReceivedDelegate(object sender, EventArgs<FIPAAclMessage> e);
28          private bool[] usedValues;
29          Dictionary<IAgent, bool> changeMap = new Dictionary<IAgent, bool>();
30          #endregion
31
32          #region Properties
33
34          public int FreeCells
35          {
36              get { return freeCells; }
37          }
38
39          /// <summary>
40          /// Returns a boolean indicating if the domain is satisfied.
41          /// </summary>
42          public bool Satisfied
43          {
44              get { return freeCells == 0; }
45          }
46          #endregion
47
48          public DomainAgent(int puzzleSize)
49          {
50              cells = new List<PuzzleCell>(puzzleSize);
51              valueDependencies = new ValueDependencyMap(puzzleSize);
52              usedValues = new bool[puzzleSize];
53          }
54
55          #region Public Methods
56          public void AddCell(PuzzleCell cell)
57          {
58              cells.Add(cell);
59              // Subscribe to events
60              cell.ValueChanged += new EventHandler<EventArgs<Nullable<int>>>(InvokeValueChanged);//
                    CellValueChanged);
61              cell.CandidatesChanged += new EventHandler<EventArgs<int>>(CellCandidatesChanged);
62              cell.CandidatesAdd += new EventHandler<EventArgs<int>>(CellCandidatesAdd);
63              // Make sure that valueDependencies are initialized
64              if (valueDependencies != null)
65              {
66                  valueDependencies.AddCell(cell);
67              }
68
69              // Add domain reference to cell
70              cell.AddDomain(this);
71
72              if (!cell.CellValue.HasValue)
73              {
74                  freeCells++;
75              }
76          }
77
78          public void RemoveCell(PuzzleCell cell)
79          {
80              cells.Remove(cell);
81              // Unsubscribe events
82              cell.ValueChanged -= new EventHandler<EventArgs<Nullable<int>>>(CellValueChanged);
83              cell.CandidatesChanged -= new EventHandler<EventArgs<int>>(CellCandidatesChanged);
84              cell.CandidatesAdd -= new EventHandler<EventArgs<int>>(CellCandidatesAdd);
85              valueDependencies.RemoveCell(cell);
86
87              // Remove domain reference from cell
88              cell.RemoveDomain(this);
89
90              if (!cell.CellValue.HasValue)
```

```
91              {
92                  freeCells--;
93              }
94          }
95
96          /// <summary>
97          /// Returns a snapshot of the cells belonging to the domain agent.
98          /// This collection can be modified without affecting the internal cell collections of the
                domain agent
99          /// </summary>
100         /// <returns>A collection of cells belonging to the domain agent. </returns>
101         public Collection<PuzzleCell> GetCells()
102         {
103             return new Collection<PuzzleCell>(new List<PuzzleCell>(cells));
104         }
105
106         /// <summary>
107         /// Returns a boolean value indicating if the domain has changed since last time
108         /// a given strategy agent visited the environment. Return true if the domain has changed,
109         /// or the strategy agent has not visited the domain. Otherwise false.
110         /// </summary>
111         public bool IsChanged(IAgent strategyAgent)
112         {
113             if (changeMap.ContainsKey(strategyAgent))
114             {
115                 return changeMap[strategyAgent];
116             }
117             return true;
118         }
119         #endregion
120
121         #region Private Methods
122         private void Initialize()
123         {
124             // Initialize used values array
125             usedValues = new bool[cells.Count];
126             foreach (PuzzleCell cell in cells)
127             {
128                 if (cell.CellValue.HasValue)
129                 {
130                     usedValues[cell.CellValue.Value - 1] = true;
131                 }
132             }
133
134             if (valueDependencies != null)
135             {
136                 // Subscribe to valuedependencies
137                 this.valueDependencies.UniqueValue += new EventHandler<EventArgs<SolutionStep>>(
                        UniqueValueFound);
138             }
139         }
140
141         private void HandleMessage(EventArgs<FIPAAclMessage> e)
142         {
143             switch (e.Value.MessagePerformative)
144             {
145                 case FIPAAclMessage.Performative.Request:
146                     // A strategy agent has requested some information.
147
148                     // Record, that a strategy agent has visited the domain.
149                     if (changeMap.ContainsKey((IAgent)e.Value.Sender))
150                     {
151                         changeMap[(IAgent)e.Value.Sender] = false;
152                     }
153                     else
154                     {
155                         changeMap.Add((IAgent)e.Value.Sender, false);
156                     }
157
158                     if (e.Value.Content is ValueDependencyMessage)
159                     {
160                         ValueDependencyMessage content = new ValueDependencyMessage();
161
162                         foreach (UniqueChain chain in valueDependencies.Values)
163                         {
164                             if (chain.Count > 0)
165                             {
166                                 content.AddValueDependency(chain);
167                             }
168                         }
169                         // Return the value dependencies
170                         OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                                Performative.Inform, this, e.Value.Sender, content)));
171                     }
172                     else if (e.Value.Content is CellMessage)
173                     {
```

```
174                         CellMessage content = new CellMessage();
175                         foreach (PuzzleCell cell in cells)
176                         {
177                             if (!cell.CellValue.HasValue)
178                             {
179                                 content.AddCell(cell);
180                             }
181                         }
182                         OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                                Performative.Inform, this, content)));
183                     }
184                     break;
185                 default:
186                     break;
187             }
188         }
189
190         #endregion
191
192         #region Event Handlers
193
194         private delegate void ValueChangedDelegate(object sender, EventArgs<Nullable<int>> e);
195         public void InvokeValueChanged(object sender, EventArgs<Nullable<int>> e)
196         {
197             ValueChangedDelegate del = new ValueChangedDelegate(this.CellValueChanged);
198             del(sender, e);
199         }
200
201
202         // O(n^2)
203         private void CellValueChanged(object sender, EventArgs<Nullable<int>> e)
204         {
205             PuzzleCell senderCell = (PuzzleCell)sender;
206             Message content;
207
208             // If senderCell has been given a value
209             if (!e.Value.HasValue)
210             {
211                 int value = senderCell.CellValue.Value;
212
213                 // If the value just sat, has already been used in the domain we have a conflict
214                 if (usedValues[value - 1])
215                 {
216                     // Notify the CoordinatorAgent that a conflict has occured
217                     OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                            Performative.Inform, this, new ConflictMessage())));
218                     content = new EliminationStrategyMessage(AGENT_TYPE, new List<
                            EliminationSolutionStep>(), new List<PuzzleCell>(), new List<int>());
219                     OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                            Performative.Propose, this, content)));
220                     return;
221                 }
222                 else // Otherwise remember that the value has been used in the domain
223                 {
224                     usedValues[value - 1] = true;
225                     freeCells--;
226                 }
227
228                 // Update the valuedependencies
229                 if (valueDependencies != null)
230                 {
231                     valueDependencies.RemoveValue(value);
232                     // O(n^2)
233                     valueDependencies.RemoveCell(senderCell);
234                 }
235
236                 List<EliminationSolutionStep> eliminations = new List<EliminationSolutionStep>();
237
238                 // Compose the eliminations that occur on basis of the value sat in the current cell
239                 foreach (PuzzleCell cell in cells)
240                 {
241                     if (cell != senderCell && !cell.CellValue.HasValue)
242                     {
243                         eliminations.Add(new EliminationSolutionStep(cell, value));
244                     }
245                 }
246
247                 // Notify the CoordinatorAgent about the possible eliminations
248                 content = new EliminationStrategyMessage(AGENT_TYPE, eliminations, new List<PuzzleCell
                        >(), new List<int>());
249                 OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                        Performative.Propose, this, content)));
250             }
251             else // Otherwise if sender has not been given a value, it must be because the cell has
                    been "undone" by backtrack search
252             {
```

```
253                     // Undo the value of the cell.
254                     usedValues[e.Value.Value - 1] = false;
255 
256                     // Make sure that valueDependencies are initialized
257                     if (valueDependencies != null)
258                     {
259                         valueDependencies.AddCell(senderCell);
260                     }
261                 }
262             }
263 
264             /// <summary>
265             /// Eventhandler called when a cell candidate is changed
266             /// </summary>
267             private void CellCandidatesChanged(object sender, EventArgs<int> e)
268             {
269                 PuzzleCell cell = sender as PuzzleCell;
270 
271                 // If the cell no longer contains candidates, there is a conflict
272                 if (cell.Candidates.Count == 0)
273                 {
274                     OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                            Performative.Inform, this, new ConflictMessage())));
275                 }
276 
277                 // The domain has changed, record the change so that strategy agents can
278                 // aquire the information later on.
279                 List<IAgent> keys = new List<IAgent>(changeMap.Keys);
280                 foreach (IAgent key in keys)
281                 {
282                     changeMap[key] = true;
283                 }
284 
285                 // Make sure that valueDependencies are initialized
286                 if (valueDependencies != null)
287                 {
288                     // Update valuedependencies
289                     valueDependencies.RemoveCellAt(e.Value, cell);
290                 }
291                 if (cell.UniqueValue.HasValue)
292                 {
293                     // If the elimination has revealed a unique candidate value, inform the
                            CoordinatorAgent about it
294                     SolutionStepMessage content = new SolutionStepMessage(new ValueSolutionStep(cell, cell.
                            UniqueValue.Value));
295                     OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                            Performative.Propose, this, content)));
296                 }
297             }
298 
299             /// <summary>
300             /// EventHandler called when a candidate is added to a cell.
301             /// Is used in Backtrack search.
302             /// </summary>
303             private void CellCandidatesAdd(object sender, EventArgs<int> e)
304             {
305                 PuzzleCell cell = sender as PuzzleCell;
306                 // Make sure that cellDependencies are initialized
307                 if (valueDependencies != null)
308                 {
309                     // Update the valuedependencies, so that the domain is up to date
310                     valueDependencies.AddCellAt(e.Value, cell);
311                 }
312             }
313 
314             void UniqueValueFound(object sender, EventArgs<SolutionStep> e)
315             {
316                 SolutionStepMessage content = new SolutionStepMessage(e.Value);
317                 OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.Performative.
                        Propose, this, content)));
318             }
319             #endregion
320 
321             #region IAgent Members
322 
323             public Guid AgentID
324             {
325                 get { return agentID; }
326             }
327 
328             public event EventHandler<EventArgs<FIPAAclMessage>> SendMessage;
329 
330             public void OnSendMessage(EventArgs<FIPAAclMessage> e)
331             {
332                 SendMessage(this, new EventArgs<FIPAAclMessage>(e.Value));
333             }
```

```
334
335         public void InvokeMessageReceived(object sender, EventArgs<FIPAAclMessage> e)
336         {
337             MessageReceivedDelegate del = new MessageReceivedDelegate(this.MessageReceived);
338             del(sender, e);
339         }
340
341         public void MessageReceived(object sender, EventArgs<FIPAAclMessage> e)
342         {
343             lock (messageQueue)
344             {
345                 messageQueue.Enqueue(e);
346                 Monitor.Pulse(messageQueue);
347             }
348         }
349
350         public void Run()
351         {
352             EventArgs<FIPAAclMessage> message = null;
353             bool interrupted = false;
354
355             Initialize();
356
357             while (!interrupted)
358             {
359                 try
360                 {
361                     lock (messageQueue)
362                     {
363                         while (messageQueue.Count == 0)
364                         {
365                             Monitor.Wait(messageQueue);
366
367                         }
368                         message = messageQueue.Dequeue();
369                     }
370                     HandleMessage(message);
371                 }
372                 catch (ThreadInterruptedException)
373                 {
374                     // Clean up
375                     interrupted = true;
376                     foreach (PuzzleCell cell in cells)
377                     {
378                         cell.ValueChanged -= new EventHandler<EventArgs<Nullable<int>>>(
                                 CellValueChanged);
379                         cell.CandidatesChanged -= new EventHandler<EventArgs<int>>(
                                 CellCandidatesChanged);
380                         cell.CandidatesAdd -= new EventHandler<EventArgs<int>>(CellCandidatesAdd);
381
382                         // Remove domain reference from cell
383                         cell.RemoveDomain(this);
384                     }
385
386                     if (valueDependencies != null)
387                     {
388                         valueDependencies.UniqueValue -= new EventHandler<EventArgs<SolutionStep>>(
                                 UniqueValueFound);
389
390
391                         for (int i = 0; i <= 9; i++)
392                         {
393                             valueDependencies.RemoveValue((int)i);
394                         }
395                         valueDependencies = null;
396                     }
397                     cells.Clear();
398                 }
399             }
400         }
401         #endregion
402
403         public override string ToString()
404         {
405             StringBuilder s = new StringBuilder();
406
407             foreach (PuzzleCell c in cells)
408             {
409                 s.Append(c.ToString());
410             }
411             return s.ToString();
412         }
413     }
414 }
```

NakedAgent.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using System.Collections.ObjectModel;
5   using System.Threading;
6   using MultiAgentSudokuSolver.Data;
7   using MultiAgentSudokuSolver.Messaging;
8
9   namespace MultiAgentSudokuSolver.Agents
10  {
11      /// <summary>
12      /// Agent implementing the Naked Set strategy
13      /// </summary>
14      public class NakedAgent : IAgent
15      {
16          private const string AGENT_TYPE = "NakedAgent";
17
18          #region Variables
19          private Guid agentID = new Guid();
20          private Queue<EventArgs<FIPAAclMessage>> messageQueue = new Queue<EventArgs<FIPAAclMessage>>();
21          private List<DomainAgent> excluded;
22          private Dictionary<DomainAgent, List<PuzzleCell>> usedCells;
23          private List<DomainAgent> domains;
24          private LogElement log = new LogElement("NakedAgent");
25          #endregion Variables
26
27          private delegate void MessageReceivedDelegate(object sender, EventArgs<FIPAAclMessage> e);
28
29          public NakedAgent(int puzzleSize)
30          {
31              excluded = new List<DomainAgent>(3*puzzleSize);
32              usedCells = new Dictionary<DomainAgent, List<PuzzleCell>>(3*puzzleSize);
33              domains = new List<DomainAgent>(3*puzzleSize);
34          }
35
36          public void AddDomain(DomainAgent domain)
37          {
38              domains.Add(domain);
39              usedCells.Add(domain, new List<PuzzleCell>(domain.FreeCells));
40          }
41
42          public void RemoveDomain(DomainAgent domain)
43          {
44              domains.Remove(domain);
45              usedCells.Remove(domain);
46          }
47
48          /// <summary>
49          /// Recursive search for a chain of naked cells
50          /// </summary>
51          /// <param name="cell">Current focus cell</param>
52          /// <param name="neighbours">Possible neighbours</param>
53          /// <param name="choices">The current number of candidate choices in the chain</param>
54          /// <param name="length">The length of the chain</param>
55          /// <param name="maxLength">The max. length of the chain</param>
56          /// <returns>Return a collection of cells representing a chain of naked cells</returns>
57          private Collection<PuzzleCell> NakedSearch(PuzzleCell cell, Collection<PuzzleCell> neighbours,
                Collection<int> choices, int length, int maxLength)
58          {
59              PuzzleCell neighbour;
60              List<int> temp;
61              List<PuzzleCell> cells;
62
63              if (length == choices.Count)
64              {
65                  cells = new List<PuzzleCell>(length);
66                  cells.Add(cell);
67                  return new Collection<PuzzleCell>(cells);
68              }
69
70              // O(3(n-1)) = O(n)
71              while (neighbours.Count > 0)
72              {
73                  neighbour = neighbours[0];
74
75                  // O(n)
76                  Collection<int> extra = neighbour.CandidatesDifferent(choices);
77
78                  int candidates = neighbour.Candidates.Count; // No. of candidates in neighbour cell
79                  int common = candidates - extra.Count; // No. of common candidates
80
81                  // Check if we have n cells with only n possible candidates, meaning that we are at the
                        endpoint
```

```
82                  // of a chain of naked cells.
83                  if (length == (choices.Count + extra.Count))
84                  {
85                      cells = new List<PuzzleCell>(length);
86                      cells.Add(cell);
87                      return new Collection<PuzzleCell>(cells);
88                  }
89
90                  // Neighbour cell has candidates in common with parent cell
91                  if (common > 0)
92                  {
93                      if (length < maxLength)
94                      {
95                          temp = new List<int>(choices);
96                          temp.AddRange(extra);
97                          cells = new List<PuzzleCell>(maxLength);
98
99                          List<PuzzleCell> neighbourList = new List<PuzzleCell>(neighbours);
100                         neighbourList.Remove(neighbour);
101
102                         // Recursive call
103                         cells.AddRange(NakedSearch(neighbour, new Collection<PuzzleCell>(neighbourList)
                                , new Collection<int>(temp), length + 1, maxLength));
104
105                         // The chain is starting to build, meaning that this cell is also part of it
106                         if (cells.Count > 0)
107                         {
108                             cells.Add(cell);
109                             return new Collection<PuzzleCell>(cells);
110                         }
111                     }
112                 }
113
114                 // this neighbour gave nothing, remove it and try another
115                 neighbours.Remove(neighbour);
116             }
117
118             // Return empty collection
119             return new Collection<PuzzleCell>();
120         }
121
122         /// <summary>
123         /// Heuristic for the NakedAgent
124         /// </summary>
125         private Collection<PuzzleCell> PerformSearchAction(Collection<PuzzleCell> cells, int maxLength)
126         {
127             // Find the puzzle cell with minimum candidates
128             PuzzleCell start = SetFunctions.Min<PuzzleCell>(cells);
129
130             Collection<PuzzleCell> nakedCells = new Collection<PuzzleCell>();
131             List<PuzzleCell> neighbours;
132
133             // Searching for a list of naked cells
134             while (nakedCells.Count < 1 && cells.Count > 0)
135             {
136                 // Remove chosen cell as it should only be treated once
137                 cells.Remove(start);
138
139                 neighbours = new List<PuzzleCell>(cells.Count);
140                 neighbours.AddRange(cells);
141
142                 nakedCells = NakedSearch(start, new Collection<PuzzleCell>(neighbours), start.
                        Candidates, 1, maxLength);
143
144                 if (cells.Count > 0)
145                 {
146                     start = cells[0];
147                 }
148             }
149
150             return nakedCells;
151         }
152
153
154         private void Request()
155         {
156             // if agent has searched all domains
157             if (excluded.Count == domains.Count)
158             {
159                 // Reset excluded domains - coordinater agent ensures that this agent is
160                 // only called again when the state of the puzzle has changed.
161                 excluded.Clear();
162
163                 // Refuse the search message
164                 StrategyMessage content = new StrategyMessage("Naked");
```

```
165                 FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative.Refuse, this,
                        content);
166                 OnSendMessage(new EventArgs<FIPAAclMessage>(message));
167             }
168             else
169             {
170                 // Start with one domain agent
171                 Random random = new Random();
172                 DomainAgent start = domains[random.Next() % domains.Count];
173                 while (excluded.Contains(start))
174                 {
175                     start = domains[random.Next() % domains.Count];
176                 }
177                 FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative.Request, this,
                        start, new CellMessage());
178                 OnSendMessage(new EventArgs<FIPAAclMessage>(message));
179             }
180         }
181 
182         private void SelectAction(DomainAgent sender, CellMessage message)
183         {
184             Collection<PuzzleCell> cells = message.Cells;
185 
186             // Only search cells that are not already found as naked in the given domain
187             // O(n)
188             foreach (PuzzleCell cell in usedCells[sender])
189             {
190                 cells.Remove(cell);
191             }
192 
193             // If there is cells to search
194             if (cells.Count > 1)
195             {
196                 Collection<PuzzleCell> nakedCells = PerformSearchAction(cells,(int)Math.Ceiling((double
                        )(cells.Count / 2)));
197 
198                 if (nakedCells.Count == 0)
199                 {
200                     // We found nothing
201                     // Don't bother to search this domain again before the global state changes.
202                     excluded.Add(sender);
203                     OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                            Performative.Request, this, new StrategyMessage("Naked"))));
204                 }
205                 else
206                 {
207                     // We have found some naked cells
208                     usedCells[sender].AddRange(nakedCells);
209                     // Compose and send elimination message
210                     EliminationStrategyMessage content = ComposeEliminations(nakedCells);
211                     if (content != null)
212                     {
213                         OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                                Performative.Propose, this, content)));
214                         OnLog(log);
215                     }
216                     else
217                     {
218                         // If no eliminations where possible, continue search.
219                         SelectAction(sender, message);
220                     }
221 
222                 }
223 
224             }
225             else // Otherwise try another domain
226             {
227                 excluded.Add(sender);
228 
229                 Request();
230             }
231         }
232 
233         private EliminationStrategyMessage ComposeEliminations(Collection<PuzzleCell> nakedCells)
234         {
235             // Helper lists, to determine which candidates to eliminate, and in which domains
236             List<int> eliminateCandidates;
237             List<DomainAgent> domains;
238             Collection<DomainAgent> commonDomains;
239 
240             eliminateCandidates = new List<int>();
241             domains = new List<DomainAgent>();
242             commonDomains = new Collection<DomainAgent>();
243 
244             // Determine how many domains the cells have in common; must be either 1 or 2.
245             // and determine the union candidate set for the cells.
```

```
246             // O(maxCount*(2+maxCount)) = O(n^2)
247             foreach (PuzzleCell cell in nakedCells)
248             {
249                 if (commonDomains.Count == 0)
250                 {
251                     commonDomains = cell.Domains;
252                 }
253                 else
254                 {
255                     IEnumerable<DomainAgent> o = SetFunctions.Intersect<DomainAgent>(commonDomains,
                            cell.Domains);
256                     commonDomains = new Collection<DomainAgent>(new List<DomainAgent>(o));
257                 }
258 
259                 foreach (int candidate in cell.Candidates)
260                 {
261                     if (!eliminateCandidates.Contains(candidate))
262                     {
263                         eliminateCandidates.Add(candidate);
264                     }
265                 }
266             }
267 
268             List<PuzzleCell> eliminationCells = new List<PuzzleCell>();
269             List<EliminationSolutionStep> eliminations = new List<EliminationSolutionStep>();
270 
271             log.Information = "Naked␣cells␣found␣in:";
272             // Ensure that found naked cells are remembered, so we don't search them again.
273             // O(2*maxcount) = O(n)
274             foreach (DomainAgent domain in commonDomains)
275             {
276                 foreach (PuzzleCell cell in domain.GetCells())
277                 {
278                     if (nakedCells.Contains(cell))
279                     {
280                         log.Information = String.Concat(log.Information, "\n", cell.ToString());
281                     }
282                     else
283                     {
284                         if (!eliminationCells.Contains(cell) && cell.Candidates.Count > 0 && !cell.
                                CellValue.HasValue)
285                         {
286                             eliminationCells.Add(cell);
287                             foreach (int candidate in eliminateCandidates)
288                             {
289                                 eliminations.Add(new EliminationSolutionStep(cell, candidate));
290                             }
291                         }
292                     }
293                 }
294             }
295 
296             log.StopTimer();
297             if (eliminations.Count > 0)
298             {
299                 EliminationStrategyMessage content = new EliminationStrategyMessage(AGENT_TYPE,
                        eliminations, new List<PuzzleCell>(nakedCells), eliminateCandidates);
300                 return content;
301             }
302             else return null;
303         }
304 
305         /// <summary>
306         /// Parses and handles a given FIPA acl message
307         /// </summary>
308         /// <param name="e">The message to be handled</param>
309         private void HandleMessage(EventArgs<FIPAAclMessage> e)
310         {
311             switch (e.Value.MessagePerformative)
312             {
313                 case FIPAAclMessage.Performative.Inform:
314                     if (e.Value.Content is CellMessage)
315                     {
316                         SelectAction((DomainAgent)e.Value.Sender, (CellMessage)e.Value.Content);
317                     }
318                     break;
319                 case FIPAAclMessage.Performative.Request:
320                     if (e.Value.Content is StrategyMessage)
321                     {
322                         if (e.Value.Sender is CoordinatorAgent)
323                         {
324                             // Ask each domain if it has changed since last time the strategy visited
325                             foreach (DomainAgent domain in domains)
326                             {
327                                 if (domain.IsChanged(this))
328                                 {
```

```
329                                 excluded.Remove(domain);
330                             }
331                         }
332                     }
333                     Request();
334                 }
335                 break;
336             default:
337                 break;
338 
339         }
340     }
341 
342     #region IAgent Members
343 
344     public Guid AgentID
345     {
346         get { return agentID; }
347     }
348 
349     public event EventHandler<MultiAgentSudokuSolver.Data.EventArgs<MultiAgentSudokuSolver.
            Messaging.FIPAAclMessage>> SendMessage;
350 
351     public void OnSendMessage(MultiAgentSudokuSolver.Data.EventArgs<MultiAgentSudokuSolver.
            Messaging.FIPAAclMessage> e)
352     {
353         if (SendMessage != null)
354         {
355             SendMessage(this, e);
356         }
357     }
358 
359     public event EventHandler<EventArgs<LogElement>> Log;
360 
361     protected void OnLog(LogElement log)
362     {
363         if (Log != null)
364         {
365             Log(this, new EventArgs<LogElement>(log));
366         }
367     }
368 
369     public void InvokeMessageReceived(object sender, EventArgs<FIPAAclMessage> e)
370     {
371         MessageReceivedDelegate del = new MessageReceivedDelegate(this.MessageReceived);
372         del(sender, e);
373     }
374 
375     public void MessageReceived(object sender, EventArgs<FIPAAclMessage> e)
376     {
377         lock (messageQueue)
378         {
379             messageQueue.Enqueue(e);
380             Monitor.Pulse(messageQueue);
381         }
382     }
383 
384     public void Run()
385     {
386         EventArgs<FIPAAclMessage> message = null;
387         bool interrupted = false;
388         while (!interrupted)
389         {
390             try
391             {
392                 lock (messageQueue)
393                 {
394                     while (messageQueue.Count == 0)
395                     {
396                         Monitor.Wait(messageQueue);
397 
398                     }
399                     message = messageQueue.Dequeue();
400                 }
401                 HandleMessage(message);
402             }
403             catch (ThreadInterruptedException)
404             {
405                 interrupted = true;
406             }
407         }
408     }
409     #endregion
410 }
411 }
```

HiddenAgent.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Collections.ObjectModel;
5  using System.Threading;
6  using MultiAgentSudokuSolver.Data;
7  using MultiAgentSudokuSolver.Messaging;
8
9  namespace MultiAgentSudokuSolver.Agents
10 {
11     /// <summary>
12     /// Agent implementing search for Hidden Sets
13     /// </summary>
14     public class HiddenAgent : IAgent
15     {
16         private const string AGENT_TYPE = "HiddenAgent";
17
18         #region Variables
19         private Guid agentID = new Guid();
20         private Queue<EventArgs<FIPAAclMessage>> messageQueue = new Queue<EventArgs<FIPAAclMessage>>();
21         private List<DomainAgent> excluded;// = new List<DomainAgent>();
22         private Dictionary<DomainAgent, List<UniqueChain>> usedChains;// = new Dictionary<DomainAgent,
               List<UniqueChain>>();
23         private List<DomainAgent> domains;// = new List<DomainAgent>();
24         private LogElement log = new LogElement("HiddenAgent");
25         #endregion Variables
26
27         private delegate void MessageReceivedDelegate(object sender, EventArgs<FIPAAclMessage> e);
28
29         public HiddenAgent(int puzzleSize)
30         {
31             excluded = new List<DomainAgent>(3*puzzleSize);
32             usedChains = new Dictionary<DomainAgent, List<UniqueChain>>(3*puzzleSize);
33             domains = new List<DomainAgent>(3*puzzleSize);
34         }
35
36         // Add the domains to be searched
37         public void AddDomain(DomainAgent domain)
38         {
39             domains.Add(domain);
40             usedChains.Add(domain, new List<UniqueChain>(domain.FreeCells));
41         }
42
43         public void RemoveDomain(DomainAgent domain)
44         {
45             domains.Remove(domain);
46             usedChains.Remove(domain);
47         }
48
49         // Search for Hidden Sets
50         private Collection<UniqueChain> Search(UniqueChain chain, Collection<UniqueChain>
               neighbourChains, Collection<PuzzleCell> choices, int length, int maxLength)
51         {
52             UniqueChain neighbour;
53             List<PuzzleCell> temp;
54             List<UniqueChain> chains;
55
56             // If we have found x (= length) chains which share y (= choices.Count) cells we have a
                   Hidden Set
57             if (length == choices.Count)
58             {
59                 // Add this chain to the Hidden Set
60                 chains = new List<UniqueChain>();
61                 chains.Add(chain);
62                 return new Collection<UniqueChain>(chains);
63             }
64
65             // We have not yet found a Hidden Set, but still have neighbour chains to examine
66             while (neighbourChains.Count > 0)
67             {
68                 neighbour = neighbourChains[0];
69
70                 Collection<PuzzleCell> extra = neighbour.GetDifferentCells(choices);
71
72                 int candidates = neighbour.Count;
73                 int common = candidates - extra.Count;
74
75                 // If the chosen chain has cells in common with the allready chosen cells
76                 if (common > 0)
77                 {
78                     if (length < maxLength)
79                     {
80                         temp = new List<PuzzleCell>(choices);
```

```
81                          // Add the new cells from the chain to the allready chosen cells
82                          temp.AddRange(extra);
83                          chains = new List<UniqueChain>();
84
85                          List<UniqueChain> neighbourList = new List<UniqueChain>(neighbourChains);
86                          neighbourList.Remove(neighbour);
87
88                          // Recursive call
89                          chains.AddRange(Search(neighbour, new Collection<UniqueChain>(neighbourList),
                                 new Collection<PuzzleCell>(temp), length + 1, maxLength));
90
91                          // The chain is starting to build, meaning that this cell is also part of it
92                          if (chains.Count > 0)
93                          {
94                              chains.Add(chain);
95                              return new Collection<UniqueChain>(chains);
96                          }
97                      }
98                  }
99
100                 // this neighbour gave nothing, remove it and try another
101                 neighbourChains.Remove(neighbour);
102             }
103
104             // Return empty collection
105             return new Collection<UniqueChain>();
106         }
107
108         private Collection<UniqueChain> PerformSearchAction(Collection<UniqueChain> chains, int
             maxLength)
109         {
110             Collection<UniqueChain> hiddenChains = new Collection<UniqueChain>();
111             List<UniqueChain> neighbours;
112
113             // Find the shortest chain
114             UniqueChain start = SetFunctions.Min<UniqueChain>(chains);
115
116             if (start.Count > chains.Count)
117             {
118                 // No reason to continue the search
119                 return hiddenChains;
120             }
121
122             // Map each puzzlecell to all the unique chains it is a part of
123             Dictionary<PuzzleCell, List<UniqueChain>> chainMap = new Dictionary<PuzzleCell, List<
                 UniqueChain>>();
124             foreach (UniqueChain chain in chains)
125             {
126                 foreach (PuzzleCell cell in chain.GetCells())
127                 {
128                     if (!chainMap.ContainsKey(cell))
129                     {
130                         chainMap.Add(cell, new List<UniqueChain>());
131                     }
132                     chainMap[cell].Add(chain);
133                 }
134             }
135
136             while (hiddenChains.Count < 1 && chains.Count > 0)
137             {
138                 chains.Remove(start);
139
140                 neighbours = new List<UniqueChain>();
141
142                 // Find neighbour chains to the starting UniqueChain
143                 foreach (PuzzleCell cell in start.GetCells())
144                 {
145                     foreach (UniqueChain chain in chainMap[cell])
146                     {
147                         if (!neighbours.Contains(chain)) //&& !usedChains.Contains(chain))
148                         {
149                             neighbours.Add(chain);
150                         }
151                     }
152                 }
153                 neighbours.Remove(start);
154
155                 // Start the search for a Hidden Set
156                 hiddenChains = Search(start, new Collection<UniqueChain>(neighbours), start.GetCells(),
                     1, maxLength);
157
158                 if (chains.Count > 0)
159                 {
160                     start = chains[0];
161                 }
162             }
```

```
163
164                 return hiddenChains;
165             }
166
167             private void Request()
168             {
169                 // if hidden has searched all domains
170                 if (excluded.Count == domains.Count)
171                 {
172                     // Reset excluded domains - coordinater agent ensures that this agent is
173                     // only called again when the state of the puzzle has changed.
174                     excluded.Clear();
175
176                     // Refuse the search message
177                     StrategyMessage content = new StrategyMessage("Hidden");
178                     FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative.Refuse, this,
                             content);
179                     OnSendMessage(new EventArgs<FIPAAclMessage>(message));
180                 }
181                 else
182                 {
183                     // Start with one domain agent
184                     Random random = new Random();
185                     DomainAgent start = domains[random.Next() % domains.Count];
186                     while (excluded.Contains(start))
187                     {
188                         start = domains[random.Next() % domains.Count];
189                     }
190                     FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative.Request, this,
                             start, new ValueDependencyMessage());
191                     OnSendMessage(new EventArgs<FIPAAclMessage>(message));
192                 }
193             }
194
195             private void SelectAction(DomainAgent sender, ValueDependencyMessage message)
196             {
197                 Collection<UniqueChain> chains = message.GetValueDependencies();
198
199                 foreach (UniqueChain chain in usedChains[sender])
200                 {
201                     chains.Remove(chain);
202                 }
203
204                 if (chains.Count > 1)
205                 {
206                     Collection<UniqueChain> hiddenChains = PerformSearchAction(chains, (int)Math.Ceiling((
                             double)(sender.FreeCells / 2)));
207
208                     if (hiddenChains.Count == 0)
209                     {
210                         // We found nothing
211                         // Don't bother to search this domain again before the global state changes.
212                         excluded.Add(sender);
213                         OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                                 Performative.Request, this, new StrategyMessage("Hidden"))));
214                     }
215                     else
216                     {
217                         // We have found some naked cells
218                         usedChains[sender].AddRange(hiddenChains);
219                         // Compose and send elimination message
220                         EliminationStrategyMessage content = ComposeEliminations(hiddenChains);
221                         OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                                 Performative.Propose, this, content)));
222                     }
223                 }
224                 else // Otherwise try another domain
225                 {
226                     excluded.Add(sender);
227
228                     Request();
229                 }
230             }
231
232             /// <summary>
233             /// Compose the eliminations which is a result of the found Hidden Set
234             /// </summary>
235             /// <param name="hiddenChains">A collection of UniqeChain objects containing the Hidden Set</
                     param>
236             /// <returns>An EliminationStrategyMessage containing all possible elimination on basis of the
                     Hidden Set</returns>
237             private EliminationStrategyMessage ComposeEliminations(Collection<UniqueChain> hiddenChains)
238             {
239                 List<PuzzleCell> eliminationCells = new List<PuzzleCell>();
240                 List<int> excludedCandidates = new List<int>();
241                 List<int> eliminationCandidates = new List<int>();
```

```
242
243             foreach (UniqueChain chain in hiddenChains)
244             {
245                 excludedCandidates.Add(chain.Value);
246
247                 foreach (PuzzleCell cell in chain.GetCells())
248                 {
249                     if (!eliminationCells.Contains(cell) && !cell.CellValue.HasValue)
250                     {
251                         eliminationCells.Add(cell);
252                     }
253                 }
254             }
255
256             List<EliminationSolutionStep> eliminations = new List<EliminationSolutionStep>();
257             foreach (PuzzleCell cell in eliminationCells)
258             {
259                 foreach (int candidate in cell.Candidates)
260                 {
261                     if (!excludedCandidates.Contains(candidate))
262                     {
263                         eliminations.Add(new EliminationSolutionStep(cell, candidate));
264                     }
265                 }
266             }
267
268             EliminationStrategyMessage content = new EliminationStrategyMessage(AGENT_TYPE,eliminations
                     , eliminationCells, eliminationCandidates);
269             return content;
270         }
271
272
273         private void HandleMessage(EventArgs<FIPAAclMessage> e)
274         {
275             switch (e.Value.MessagePerformative)
276             {
277                 case FIPAAclMessage.Performative.Inform:
278                     if (e.Value.Content is ValueDependencyMessage)
279                     {
280                         SelectAction((DomainAgent)e.Value.Sender, (ValueDependencyMessage)e.Value.
                             Content);
281                     }
282                     break;
283                 case FIPAAclMessage.Performative.Request:
284                     if (e.Value.Content is StrategyMessage)
285                     {
286                         if (e.Value.Sender is CoordinatorAgent)
287                         {
288                             // Start using strategy
289
290                             // First check if some domains are excluded
291                             if (excluded.Count > 0)
292                             {
293                                 // Ask each domain if it has changed since last time the strategy
                                     visited
294                                 foreach (DomainAgent domain in domains)
295                                 {
296                                     if (domain.IsChanged(this))
297                                     {
298                                         excluded.Remove(domain);
299                                     }
300                                 }
301                                 if (excluded.Count > 0)
302                                 {
303                                     int i = excluded.Count;
304                                 }
305                             }
306
307                         }
308                         Request();
309                     }
310                     break;
311                 default:
312                     break;
313             }
314         }
315
316         #region IAgent Members
317
318         public Guid AgentID
319         {
320             get { return agentID; }
321         }
322
323         public event EventHandler<MultiAgentSudokuSolver.Data.EventArgs<MultiAgentSudokuSolver.
                 Messaging.FIPAAclMessage>> SendMessage;
```

```
324
325         public void OnSendMessage(MultiAgentSudokuSolver.Data.EventArgs<MultiAgentSudokuSolver.
                Messaging.FIPAAclMessage> e)
326         {
327             if (SendMessage != null)
328             {
329                 SendMessage(this, e);
330             }
331         }
332
333         public void MessageReceived(object sender, MultiAgentSudokuSolver.Data.EventArgs<
                MultiAgentSudokuSolver.Messaging.FIPAAclMessage> e)
334         {
335             lock (messageQueue)
336             {
337                 messageQueue.Enqueue(e);
338                 Monitor.Pulse(messageQueue);
339             }
340         }
341
342         public void InvokeMessageReceived(object sender, MultiAgentSudokuSolver.Data.EventArgs<
                MultiAgentSudokuSolver.Messaging.FIPAAclMessage> e)
343         {
344             MessageReceivedDelegate del = new MessageReceivedDelegate(this.MessageReceived);
345             del(sender, e);
346         }
347
348         public void Run()
349         {
350             EventArgs<FIPAAclMessage> message = null;
351             bool interrupted = false;
352             while (!interrupted)
353             {
354                 try
355                 {
356                     lock (messageQueue)
357                     {
358                         while (messageQueue.Count == 0)
359                         {
360                             Monitor.Wait(messageQueue);
361
362                         }
363                         message = messageQueue.Dequeue();
364                     }
365                     HandleMessage(message);
366                 }
367                 catch (ThreadInterruptedException)
368                 {
369                     interrupted = true;
370                 }
371             }
372         }
373
374         #endregion
375     }
376 }
```

IntersectionAgent.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using MultiAgentSudokuSolver.Data;
5   using MultiAgentSudokuSolver.Messaging;
6   using System.Collections.ObjectModel;
7   using System.Threading;
8
9   namespace MultiAgentSudokuSolver.Agents
10  {
11      /// <summary>
12      /// Agent implementing the Intersection Set strategy
13      /// </summary>
14      public class IntersectionAgent : IAgent
15      {
16          struct Intersection
17          {
18              public DomainAgent domain;
19              public UniqueChain chain;
20          }
21
```

```
22          private const string AGENT_TYPE = "IntersectionAgent";
23
24          #region Variables
25          private Guid agentID = new Guid();
26          private Queue<EventArgs<FIPAAclMessage>> messageQueue = new Queue<EventArgs<FIPAAclMessage>>();
27          private List<DomainAgent> excluded;
28          private Dictionary<DomainAgent, List<UniqueChain>> usedChains = new Dictionary<DomainAgent,
                List<UniqueChain>>();
29          private List<DomainAgent> rows;
30          private List<DomainAgent> columns;
31          private List<DomainAgent> squares;
32          private LogElement log = new LogElement("IntersectionAgent");
33
34
35          #endregion Variables
36
37          public IntersectionAgent(int puzzleSize)
38          {
39              excluded = new List<DomainAgent>(3 * puzzleSize);
40              rows = new List<DomainAgent>(puzzleSize);
41              columns = new List<DomainAgent>(puzzleSize);
42              squares = new List<DomainAgent>(puzzleSize);
43          }
44
45          private delegate void MessageReceivedDelegate(object sender, EventArgs<FIPAAclMessage> e);
46
47          public void AddRowDomain(DomainAgent domain)
48          {
49              rows.Add(domain);
50              usedChains.Add(domain, new List<UniqueChain>(domain.FreeCells));
51          }
52          public void RemoveRowDomain(DomainAgent domain) { rows.Remove(domain); }
53
54          public void AddColumnDomain(DomainAgent domain)
55          {
56              columns.Add(domain);
57              usedChains.Add(domain, new List<UniqueChain>(domain.FreeCells));
58          }
59          public void RemoveColumnDomain(DomainAgent domain) { columns.Remove(domain); }
60
61          public void AddSquareDomain(DomainAgent domain)
62          {
63              squares.Add(domain);
64              usedChains.Add(domain, new List<UniqueChain>(domain.FreeCells));
65          }
66          public void RemoveSquareDomain(DomainAgent domain) { squares.Remove(domain); }
67
68
69          private Intersection PerformSearchAction(Collection<UniqueChain> chains, DomainAgent sender)
70          {
71              UniqueChain pointingChain = null;
72              DomainAgent intersectionDomain = null;
73              List<DomainAgent> commonDomains;
74              UniqueChain start;
75
76              while (pointingChain == null && chains.Count > 0)
77              {
78                  start = chains[0];
79                  chains.Remove(start);
80
81                  commonDomains = new List<DomainAgent>();
82
83                  Collection<PuzzleCell> chainCells = start.GetCells();
84                  commonDomains.AddRange(chainCells[0].Domains);
85
86                  foreach (PuzzleCell cell in chainCells)
87                  {
88                      commonDomains = new List<DomainAgent>(SetFunctions.Intersect<DomainAgent>(cell.
                            Domains, new Collection<DomainAgent>(commonDomains)));
89                  }
90
91                  if (commonDomains.Count == 2)
92                  {
93                      // Pointing pair found
94                      commonDomains.Remove(sender);
95                      pointingChain = start;
96                      intersectionDomain = commonDomains[0];
97                  }
98
99              }
100             Intersection intersection;
101             intersection.chain = pointingChain;
102             intersection.domain = intersectionDomain;
103             return intersection;
104         }
105
```

```
106
107         private void SelectAction(DomainAgent sender, ValueDependencyMessage message)
108         {
109             Collection<UniqueChain> chains = message.GetValueDependencies();
110             List<UniqueChain> reduced = new List<UniqueChain>();
111             Intersection intersection;
112             intersection.chain = null;
113             intersection.domain = null;
114
115             foreach (UniqueChain chain in chains)
116             {
117                 if (chain.Count <= Math.Sqrt(sender.GetCells().Count) && !usedChains[sender].Contains(
                        chain))
118                 {
119                     reduced.Add(chain);
120                 }
121             }
122
123             if (reduced.Count > 0)
124             {
125                 // If we have some chains to examine, search for a intersection set
126                 intersection = PerformSearchAction(new Collection<UniqueChain>(reduced), sender);
127             }
128
129             if (intersection.chain == null)
130             {
131                 // We found nothing
132                 // Don't bother to search this domain again before the global state changes.
133                 excluded.Add(sender);
134                 OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                        Performative.Request, this, new StrategyMessage("Intersection"))));
135             }
136             else
137             {
138                 usedChains[sender].Add(intersection.chain);
139                 // Compose and send elimination message
140                 EliminationStrategyMessage content = ComposeEliminations(intersection.chain,
                        intersection.domain, sender);
141                 if (content != null)
142                 {
143                     OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                            Performative.Propose, this, content)));
144                 }
145                 else
146                 {
147                     // Continue with search of intersections
148                     SelectAction(sender, message);
149                 }
150             }
151         }
152
153         private void Request()
154         {
155             // if agent has searched all domains
156             if (excluded.Count == squares.Count)
157             {
158                 // Refuse the search message
159                 StrategyMessage content = new StrategyMessage("Intersection");
160                 FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative.Refuse, this,
                        content);
161                 OnSendMessage(new EventArgs<FIPAAclMessage>(message));
162             }
163             else
164             {
165                 // Start with one domain agent
166                 Random random = new Random();
167                 DomainAgent start = squares[random.Next() % squares.Count];
168                 while (excluded.Contains(start))
169                 {
170                     start = squares[random.Next() % squares.Count];
171                 }
172                 FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative.Request, this,
                        start, new ValueDependencyMessage());
173                 OnSendMessage(new EventArgs<FIPAAclMessage>(message));
174             }
175         }
176         private EliminationStrategyMessage ComposeEliminations(UniqueChain intersectionChain,
                DomainAgent intersection, DomainAgent excluded)
177         {
178             List<EliminationSolutionStep> eliminations = new List<EliminationSolutionStep>();
179
180             List<int> eliminationCandidates = new List<int>();
181
182             eliminationCandidates.Add(intersectionChain.Value);
183             Collection<PuzzleCell> excludedCells = excluded.GetCells();
184
```

```
185                 foreach (PuzzleCell cell in intersection.GetCells())
186                 {
187                     if (!excludedCells.Contains(cell) && cell.Candidates.Contains(intersectionChain.Value)
                             && !cell.CellValue.HasValue)
188                     {
189                         eliminations.Add(new EliminationSolutionStep(cell, intersectionChain.Value));
190                     }
191                 }
192 
193                 if (eliminations.Count > 0)
194                 {
195                     EliminationStrategyMessage content = new EliminationStrategyMessage(AGENT_TYPE,
                             eliminations, intersectionChain.GetCells(), eliminationCandidates);
196                     return content;
197                 }
198 
199                 return null;
200             }
201 
202             /// <summary>
203             /// Parses and handles a given FIPA acl message
204             /// </summary>
205             /// <param name="e">The message to be handled</param>
206             private void HandleMessage(EventArgs<FIPAAclMessage> e)
207             {
208                 switch (e.Value.MessagePerformative)
209                 {
210                     case FIPAAclMessage.Performative.Inform:
211                         if (e.Value.Content is ValueDependencyMessage)
212                         {
213                             SelectAction((DomainAgent)e.Value.Sender, (ValueDependencyMessage)e.Value.
                                     Content);
214                         }
215                         break;
216                     case FIPAAclMessage.Performative.Request:
217                         if (e.Value.Content is StrategyMessage)
218                         {
219                             if (e.Value.Sender is CoordinatorAgent)
220                             {
221                                 // Ask each domain if it has changed since last time the strategy visited
222                                 foreach (DomainAgent domain in squares)
223                                 {
224                                     if (domain.IsChanged(this))
225                                     {
226                                         excluded.Remove(domain);
227                                     }
228                                 }
229                             }
230                             Request();
231                         }
232                         break;
233                     default:
234                         break;
235                 }
236             }
237 
238             #region IAgent Members
239 
240             public Guid AgentID
241             {
242                 get { return agentID; }
243             }
244 
245             public event EventHandler<MultiAgentSudokuSolver.Data.EventArgs<MultiAgentSudokuSolver.
                     Messaging.FIPAAclMessage>> SendMessage;
246 
247             public void OnSendMessage(MultiAgentSudokuSolver.Data.EventArgs<MultiAgentSudokuSolver.
                     Messaging.FIPAAclMessage> e)
248             {
249                 if (SendMessage != null)
250                 {
251                     SendMessage(this, e);
252                 }
253             }
254 
255             public event EventHandler<EventArgs<LogElement>> Log;
256 
257             protected void OnLog(LogElement log)
258             {
259                 if (Log != null)
260                 {
261                     Log(this, new EventArgs<LogElement>(log));
262                 }
263             }
264 
265             public void InvokeMessageReceived(object sender, EventArgs<FIPAAclMessage> e)
```

```
266         {
267             MessageReceivedDelegate del = new MessageReceivedDelegate(this.MessageReceived);
268             del(sender, e);
269         }
270
271         public void MessageReceived(object sender, EventArgs<FIPAAclMessage> e)
272         {
273             lock (messageQueue)
274             {
275                 messageQueue.Enqueue(e);
276                 Monitor.Pulse(messageQueue);
277             }
278         }
279
280         public void Run()
281         {
282             EventArgs<FIPAAclMessage> message = null;
283             bool interrupted = false;
284             while (!interrupted)
285             {
286                 try
287                 {
288                     lock (messageQueue)
289                     {
290                         while (messageQueue.Count == 0)
291                         {
292                             Monitor.Wait(messageQueue);
293                         }
294                         message = messageQueue.Dequeue();
295                     }
296                     HandleMessage(message);
297                 }
298                 catch (ThreadInterruptedException)
299                 {
300                     interrupted = true;
301                 }
302             }
303         }
304         #endregion
305     }
306 }
```

CoordinatorAgent.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Collections.ObjectModel;
4   using System.Globalization;
5   using System.Text;
6   using System.Threading;
7   using MultiAgentSudokuSolver.Messaging;
8   using MultiAgentSudokuSolver.Data;
9
10  namespace MultiAgentSudokuSolver.Agents
11  {
12      /// <summary>
13      /// CoordinatorAgent coordinates the MultiAgent system.
14      /// It controls when a value should be sat in the Sudoku puzzle.
15      /// It controls when and which strategies to try
16      /// It controls the backtrack search which is used as a last resort.
17      /// </summary>
18      public class CoordinatorAgent : IAgent
19      {
20          private enum Strategies { Hidden, Naked, Intersection };
21
22          #region Variables
23          private Guid agentID = Guid.NewGuid();
24          private Queue<SolutionStep> queue;
25          private Queue<SolutionStep> eliminationQueue;
26          private Queue<EventArgs<FIPAAclMessage>> messageQueue = new Queue<EventArgs<FIPAAclMessage>>();
27          private List<PuzzleCell> cells;
28          private int domainAnswers;
29          private Queue<Strategies> availableStrategies = new Queue<Strategies>();
30
31          // Backtrack variables
32          private Stack<DecisionBasis> decisions;
33          private Dictionary<int, Collection<SolutionStep>> implicationsMap;
34          private List<SolutionStep> implications;
35          private bool recordImplications = false;
36          private bool conflict = false;
```

```
37
38          // Statistics
39          private int backtrackcount;
40          private int searchcount;
41          private LogElement log = new LogElement("CoordinatorAgent");
42          #endregion Variables
43
44          private delegate void MessageReceivedDelegate(object sender, EventArgs<FIPAAclMessage> e);
45
46          public CoordinatorAgent(int puzzleSize)
47          {
48              cells = new List<PuzzleCell>(puzzleSize);
49              queue = new Queue<SolutionStep>(puzzleSize * puzzleSize);
50              eliminationQueue = new Queue<SolutionStep>(puzzleSize * puzzleSize * puzzleSize);
51              decisions = new Stack<DecisionBasis>(puzzleSize * puzzleSize);
52              implications = new List<SolutionStep>(puzzleSize * puzzleSize * puzzleSize);
53              implicationsMap = new Dictionary<int, Collection<SolutionStep>>(puzzleSize * puzzleSize);
54          }
55
56          #region Public Methods
57          public void InitializeBoard(PuzzleCell[,] cells, string[] data, int gridSize)
58          {
59              // Add the possible strategies
60              availableStrategies.Enqueue(Strategies.Intersection);
61              availableStrategies.Enqueue(Strategies.Hidden);
62              availableStrategies.Enqueue(Strategies.Naked);
63
64              // Create a list containing all PuzzleCell objects of the puzzle
65              foreach (PuzzleCell cell in cells)
66              {
67                  this.cells.Add(cell);
68              }
69
70              int column = 0, row = 0;
71
72              // Parse the data array
73              foreach (string n in data)
74              {
75                  if (!n.Equals(".") && !n.Equals("0"))
76                  {
77                      queue.Enqueue(new ValueSolutionStep(cells[column, row], int.Parse(n.ToString())));
78                  }
79
80                  column++;
81                  if (column == gridSize)
82                  {
83                      column = 0;
84                      row++;
85                  }
86              }
87          }
88          #endregion
89
90          #region Backtrack search
91
92          /// <summary>
93          /// Determine the next cell to use as decision basis. Sort the cell list, in respect to number
                of candidates
94          /// Choose the cell with the fewest candidates and no value.
95          /// </summary>
96          /// <returns>Returns a cell with fewest candidates</returns>
97          private PuzzleCell NextDecisionCell()
98          {
99              cells.Sort(delegate(PuzzleCell a, PuzzleCell b) { return a.Candidates.Count.CompareTo(b.
                     Candidates.Count); });
100             return cells.Find(delegate(PuzzleCell cell) { return cell.Candidates.Count > 0 && !cell.
                     CellValue.HasValue; });
101         }
102
103         private void Backtrack()
104         {
105             backtrackcount++;
106
107             // Undo implications
108             foreach (SolutionStep step in implications)
109             {
110                 step.Undo();
111             }
112
113             if (decisions.Count > 0)
114             {
115                 NextStepMessage content;
116
117                 // Try decision basis in all ways.
118                 if (!decisions.Peek().IsEmpty())
119                 {
```

```
120                     conflict = false;
121                     implications.Clear();
122                     queue.Clear();
123                     queue.Enqueue(decisions.Peek().NextDecision());
124                     content = new NextStepMessage(true);
125                 }
126                 else
127                 {
128                     // Backtrack to previous.
129                     conflict = true;
130                     queue.Clear();
131                     decisions.Pop();
132                     //if (decisions.Count == 0)
133                     //    return;
134                     implications = new List<SolutionStep>(implicationsMap[decisions.Count]);
135                     content = new NextStepMessage();
136                 }
137 
138 
139                 OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                        Performative.Request, this, content)));
140             }
141             else
142             {
143                 // No solution!
144             }
145         }
146 
147         public void Search()
148         {
149             searchcount++;
150 
151             // Reset availableStrategies.
152             if (availableStrategies.Count == 0)
153             {
154                 if (!availableStrategies.Contains(Strategies.Intersection)) availableStrategies.Enqueue
                        (Strategies.Intersection);
155                 if (!availableStrategies.Contains(Strategies.Hidden)) availableStrategies.Enqueue(
                        Strategies.Hidden);
156                 if (!availableStrategies.Contains(Strategies.Naked)) availableStrategies.Enqueue(
                        Strategies.Naked);
157 
158             }
159 
160             PuzzleCell start = NextDecisionCell();
161 
162             // If no decisioncells can be found, all cell has values, and a solution is found.
163             // otherwise if no decision cells can be found and not all cell has values, the given
                    sudoku is invalid
164             if (start == null)
165             {
166                 // Solution found
167                 OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                        Performative.Inform, this, new SolutionMessage(true, null))));
168                 return;
169             }
170 
171             int dl = decisions.Count; // Decision Level
172             // Save the implications of the current decision level.
173             if (dl > 0)
174             {
175                 if (implicationsMap.ContainsKey(dl))
176                 {
177                     implicationsMap.Remove(dl);
178                 }
179                 implicationsMap.Add(dl, new Collection<SolutionStep>(new List<SolutionStep>(
                        implications)));
180                 implications.Clear();
181             }
182 
183             DecisionBasis basis = new DecisionBasis(start);
184 
185             // Push the next decision basis on the decisions stack.
186             decisions.Push(basis);
187 
188             recordImplications = true;
189 
190             // Start with the first decision in the decision basis, meaning a guess of which value
                    should be sat in the cell.
191             // The guess is represented by a ValueSolutionStep.
192             queue.Enqueue(basis.NextDecision());
193             OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.Performative.
                    Request, this, new NextStepMessage(true))));
194         }
195         #endregion
196 
```

```
197         #region Private Methods
198         private void HandleMessage(EventArgs<FIPAAclMessage> e)
199         {
200             switch (e.Value.MessagePerformative)
201             {
202                 case FIPAAclMessage.Performative.Inform:
203                     if (e.Value.Content is ConflictMessage)
204                     {
205                         conflict = true;
206                     }
207                     break;
208                 case FIPAAclMessage.Performative.Propose:
209                     if (e.Value.Content is SolutionStepMessage)
210                     {
211                         SolutionStepMessage content = e.Value.Content as SolutionStepMessage;
212 
213                         // In the case of eliminations execute them right away, since an elimination
214                         // cannot trigger any further SolutionStepMessages.
215                         if (content.Step is EliminationSolutionStep)
216                         {
217                             (content.Step as EliminationSolutionStep).Execute();
218                             if (recordImplications)
219                             {
220                                 implications.Add(content.Step);
221                             }
222                         }
223                         else
224                         {
225                             // In the case of any other solution step (currently there is only one
                                    other, namely
226                             // ValueSolution step, enque it for later execution. A ValueSolution step
                                    can trigger
227                             // further SolutionStepMessages, therefore is it later executed in a
                                    syncronized manner.
228                             if (!queue.Contains(content.Step as SolutionStep))
229                             {
230                                 queue.Enqueue(content.Step as SolutionStep);
231                             }
232                         }
233                     }
234                     else if (e.Value.Content is EliminationStrategyMessage)
235                     {
236                         EliminationStrategyMessage content = e.Value.Content as
                                 EliminationStrategyMessage;
237 
238                         bool eliminated = false;
239                         foreach (EliminationSolutionStep elimination in content.Eliminations)
240                         {
241                             if (elimination.Execute())
242                             {
243                                 // If a candidate has been eliminated
244                                 eliminated = true;
245                                 if (recordImplications)
246                                 {
247                                     implications.Add(elimination);
248                                 }
249                             }
250                         }
251 
252                         // If sender is DomainAgent, the elimination message, is based on an event in a
                                cell.
253                         if (e.Value.Sender is DomainAgent)
254                         {
255                             domainAnswers++;
256                             // Each cell belongs to three domains, therefore wait until all domains
                                    have processed
257                             // the event.
258                             if (domainAnswers == 3)
259                             {
260                                 domainAnswers = 0;
261                                 // Send NextStepMessage indicating that the execution of solutionsteps
                                        safely can continue.
262                                 FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.Performative
                                        .Request, this, new NextStepMessage());
263                                 OnSendMessage(new EventArgs<FIPAAclMessage>(message));
264                             }
265                         }
266                         else // If the sender is a strategy agent
267                         {
268                             // If nothing has been eliminated, signal the running strategy agent to
                                    continue its search
269                             if (!eliminated)
270                             {
271                                 FIPAAclMessage message;
272                                 if (e.Value.Sender is HiddenAgent)
273                                 {
```

```
274                                 message = new FIPAAclMessage(FIPAAclMessage.Performative.Request,
                                        this, new StrategyMessage("Hidden"));
275                             }
276                             else if (e.Value.Sender is NakedAgent)
277                             {
278                                 message = new FIPAAclMessage(FIPAAclMessage.Performative.Request,
                                        this, new StrategyMessage("Naked"));
279                             }
280                             else if (e.Value.Sender is IntersectionAgent)
281                             {
282                                 message = new FIPAAclMessage(FIPAAclMessage.Performative.Request,
                                        this, new StrategyMessage("Intersection"));
283                             }
284                             else
285                             {
286                                 message = null;
287                             }
288
289                             if (message != null)
290                             {
291                                 OnSendMessage(new EventArgs<FIPAAclMessage>(message));
292                             }
293                         }
294                         else
295                         {
296                             // If something has been eliminated, add the eliminating strategy to
                                    available strategies
297                             if (e.Value.Sender is HiddenAgent)
298                             {
299                                 availableStrategies.Enqueue(Strategies.Hidden);
300                             }
301                             else if (e.Value.Sender is NakedAgent)
302                             {
303                                 availableStrategies.Enqueue(Strategies.Naked);
304                             }
305                             else if (e.Value.Sender is IntersectionAgent)
306                             {
307                                 availableStrategies.Enqueue(Strategies.Intersection);
308                             }
309                             // Send NextStepMessage indicating that the execution of solutionsteps
                                    safely can continue.
310                             OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(
                                    FIPAAclMessage.Performative.Request, this, new NextStepMessage()))
                                    );
311                         }
312                     }
313                 }
314                 break;
315             case FIPAAclMessage.Performative.Request:
316                 if (e.Value.Content is NextStepMessage)
317                 {
318                     if (!conflict)
319                     {
320                         // There is solution steps to execute
321                         if (queue.Count > 0)
322                         {
323                             SolutionStep step = queue.Dequeue();
324                             if (step.Cell.CellValue.HasValue)
325                             {
326                                 // if the cell in a ValueSolutionStep already has a value, there is
                                        a conflict.
327                                 conflict = true;
328                                 OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(
                                        FIPAAclMessage.Performative.Request, this, new NextStepMessage
                                        ())));
329                                 return;
330                             }
331
332                             step.Execute();
333
334                             if (recordImplications)
335                             {
336                                 implications.Add(step);
337                             }
338                         }
339                         else // there is no solution steps to execute
340                         {
341                             if (availableStrategies.Count > 0)
342                             {
343                                 // If we have a strategy available use it.
344                                 Strategies strategy = (Strategies)availableStrategies.Dequeue();
345
346                                 Message content = new StrategyMessage(Enum.GetName(typeof(
                                        Strategies), strategy));
347                                 FIPAAclMessage message = new FIPAAclMessage(FIPAAclMessage.
                                        Performative.Request, this, content);
```

```
348                                     OnSendMessage(new EventArgs<FIPAAclMessage>(message));
349                                 }
350                                 else
351                                 {
352                                     // If there is no strategies to try, then the only option is to
                                            start the backtrack search.
353                                     Search();
354                                 }
355                             }
356                         }
357                         else // if there is a conflict
358                         {
359                             // Backtrack, and undo the implications.
360                             Backtrack();
361                         }
362                     }
363                     break;
364                 case FIPAAclMessage.Performative.Refuse:
365                     if (e.Value.Content is StrategyMessage)
366                     {
367                         OnSendMessage(new EventArgs<FIPAAclMessage>(new FIPAAclMessage(FIPAAclMessage.
                                Performative.Request, this, new NextStepMessage())));
368                     }
369                     break;
370                 default:
371                     break;
372             }
373         }
374         #endregion
375 
376         #region IAgent Members
377 
378         public Guid AgentID
379         {
380             get { return agentID; }
381         }
382 
383         public event EventHandler<EventArgs<FIPAAclMessage>> SendMessage;
384 
385         public void OnSendMessage(EventArgs<FIPAAclMessage> e)
386         {
387             SendMessage(this, new EventArgs<FIPAAclMessage>(e.Value));
388         }
389 
390         public void InvokeMessageReceived(object sender, EventArgs<FIPAAclMessage> e)
391         {
392             MessageReceivedDelegate del = new MessageReceivedDelegate(this.MessageReceived);
393             del(sender, e);
394         }
395 
396         public void MessageReceived(object sender, EventArgs<FIPAAclMessage> e)
397         {
398             lock (messageQueue)
399             {
400                 messageQueue.Enqueue(e);
401                 Monitor.Pulse(messageQueue);
402             }
403         }
404 
405         public void Run()
406         {
407             EventArgs<FIPAAclMessage> message = null;
408             bool interrupted = false;
409             while (!interrupted)
410             {
411                 try
412                 {
413                     lock (messageQueue)
414                     {
415                         while (messageQueue.Count == 0)
416                         {
417                             Monitor.Wait(messageQueue);
418 
419                         }
420                         message = messageQueue.Dequeue();
421                     }
422                     HandleMessage(message);
423                 }
424                 catch (ThreadInterruptedException)
425                 {
426                     interrupted = true;
427                 }
428             }
429         }
430         #endregion
431     }
```

```
432 }
```

B.3 Messaging

Message.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MultiAgentSudokuSolver.Messaging
6  {
7      public abstract class Message
8      {
9
10     }
11 }
```

ConflictMessage.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MultiAgentSudokuSolver.Messaging
6  {
7      public class ConflictMessage : Message
8      {
9      }
10 }
```

FIPAACLMessage.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MultiAgentSudokuSolver.Messaging
6  {
7      public class FIPAAclMessage
8      {
9          private Performative performative;
10         private Message content;
11         private object sender;
12         private object receiver;
13
14         public Performative MessagePerformative
15         {
16             get { return performative; }
17         }
18
19         public Message Content
20         {
21             get { return content; }
22         }
23
24         public object Sender
25         {
26             get { return sender; }
27         }
28
29         public object Receiver
30         {
```

```
31              get { return receiver; }
32          }
33
34          //String aclReceiver, aclReplyTo, aclContent, aclLanguage, aclEncoding;
35          //String aclOntology, aclProtocol, aclConversationId, aclReplyWith, aclInReplyTo, aclReplyBy;
36
37          /// <summary>
38          ///
39          /// </summary>
40          public enum Performative { Inform, Propose, AcceptProposal, RejectProposal, Request, Refuse };
41
42          /// <summary>
43          /// Constructor
44          /// </summary>
45          /// <param name="?"></param>
46          public FIPAAclMessage(Performative performative, object sender, Message content)
47          {
48              this.performative = performative;
49              this.sender = sender;
50              this.content = content;
51          }
52
53          public FIPAAclMessage(Performative performative, object sender, object receiver, Message
                content)
54          {
55              this.performative = performative;
56              this.sender = sender;
57              this.receiver = receiver;
58              this.content = content;
59          }
60
61
62      }
63  }
```

NextStepMessage.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4
5   namespace MultiAgentSudokuSolver.Messaging
6   {
7       public class NextStepMessage : Message
8       {
9           private bool isSearch;
10
11          public bool IsSearch { get { return isSearch; } }
12
13          public NextStepMessage()
14          {
15              isSearch = false;
16          }
17
18          public NextStepMessage(bool isSearch)
19          {
20              this.isSearch = isSearch;
21          }
22
23      }
24  }
```

SolutionStepMessage.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using MultiAgentSudokuSolver.Data;
5
6   namespace MultiAgentSudokuSolver.Messaging
7   {
8       class SolutionStepMessage : Message
9       {
10          private SolutionStep step;
```

```
11
12          public SolutionStep Step
13          {
14              get { return step; }
15
16          }
17
18          public SolutionStepMessage(SolutionStep step)
19          {
20              this.step = step;
21          }
22      }
23  }
```

ValueDependencyMessage.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using MultiAgentSudokuSolver.Data;
5   using System.Collections.ObjectModel;
6
7   namespace MultiAgentSudokuSolver.Messaging
8   {
9       public class ValueDependencyMessage : Message
10      {
11          List<UniqueChain> chains = new List<UniqueChain>();
12
13          public void AddValueDependency(UniqueChain chain)
14          {
15              chains.Add(chain);
16          }
17
18          public Collection<UniqueChain> GetValueDependencies()
19          {
20              return new Collection<UniqueChain>(chains);
21          }
22      }
23  }
```

CellMessage.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using MultiAgentSudokuSolver.Data;
5   using System.Collections.ObjectModel;
6
7   namespace MultiAgentSudokuSolver.Messaging
8   {
9       public class CellMessage : Message
10      {
11          private List<PuzzleCell> cells = new List<PuzzleCell>();
12
13          public void AddCell(PuzzleCell cell) {
14              cells.Add(cell);
15          }
16
17          public Collection<PuzzleCell> Cells
18          {
19              get { return new Collection<PuzzleCell>(cells); }
20          }
21      }
22  }
```

EliminationStrategyMessage.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using MultiAgentSudokuSolver.Data;
5  using System.Collections.ObjectModel;
6
7  namespace MultiAgentSudokuSolver.Messaging
8  {
9      public class EliminationStrategyMessage : Message
10     {
11         private string strategyType;
12
13         public string StrategyType { get { return strategyType; } }
14
15         private ReadOnlyCollection<PuzzleCell> strategyCellSet;
16
17         public ReadOnlyCollection<PuzzleCell> StrategyCellSet
18         {
19             get { return strategyCellSet; }
20         }
21
22         private ReadOnlyCollection<int> strategyCandidateSet;
23
24         public ReadOnlyCollection<int> StrategyCandidateSet
25         {
26             get { return strategyCandidateSet; }
27         }
28
29         private ReadOnlyCollection<EliminationSolutionStep> eliminations;
30
31         public ReadOnlyCollection<EliminationSolutionStep> Eliminations
32         {
33             get { return eliminations; }
34         }
35
36         public EliminationStrategyMessage(string strategyType, IList<EliminationSolutionStep>
               eliminations, IList<PuzzleCell> strategyCellSet, IList<int> strategyCandidateSet )
37         {
38             this.strategyType = strategyType;
39             this.eliminations = new ReadOnlyCollection<EliminationSolutionStep>(eliminations);
40             this.strategyCellSet = new ReadOnlyCollection<PuzzleCell>(strategyCellSet);
41             this.strategyCandidateSet = new ReadOnlyCollection<int>(strategyCandidateSet);
42         }
43     }
44 }
```

SolutionMessage.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using MultiAgentSudokuSolver.Data;
5
6  namespace MultiAgentSudokuSolver.Messaging
7  {
8      public class SolutionMessage : Message
9      {
10         private bool solved;
11         private LogElement log;
12
13         public bool Solved { get { return solved; } }
14         public LogElement Log { get { return log; } }
15
16         public SolutionMessage(bool solved, LogElement log)
17         {
18             this.solved = solved;
19             this.log = log;
20         }
21     }
22 }
```

StrategyMessage.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Text;
 4
 5  namespace MultiAgentSudokuSolver.Messaging
 6  {
 7      public class StrategyMessage : Message
 8      {
 9          private readonly string strategy;
10
11          public string Strategy { get { return strategy; } }
12
13          public StrategyMessage(string strategy)
14          {
15              this.strategy = strategy;
16          }
17
18      }
19  }
```

B.4 Data

EliminationSolutionStep.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Text;
 4
 5  namespace MultiAgentSudokuSolver.Data
 6  {
 7      public class EliminationSolutionStep : SolutionStep
 8      {
 9          public EliminationSolutionStep(PuzzleCell cell, int value) : base(cell, value) { }
10
11          public override bool Execute()
12          {
13              return Cell.Eliminate(Value);
14          }
15
16          public override void Undo()
17          {
18              Cell.Add(value);
19          }
20      }
21  }
```

LogElement.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Text;
 4
 5  namespace MultiAgentSudokuSolver.Data
 6  {
 7      public class LogElement
 8      {
 9          private String agentId;
10          private DateTime startTime, endTime;
11          private String information;
12
13          public LogElement(String agentId)
14          {
15              this.agentId = agentId;
16          }
17
18          public void StartTimer()
19          {
20              startTime = DateTime.Now;
21          }
```

```
22
23          public void StopTimer()
24          {
25              endTime = DateTime.Now;
26          }
27
28          public TimeSpan ExecutionTime
29          {
30              get
31              {
32                  if (startTime != null && endTime != null)
33                  {
34                      return endTime - startTime;
35                  }
36              }
37          }
38
39          public String Information
40          {
41              get { return information; }
42              set { information = value; }
43          }
44
45      }
46  }
```

SetFunctions.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using System.Collections;
5
6   namespace MultiAgentSudokuSolver.Data
7   {
8
9       /// <summary>
10      /// Helper functions
11      /// </summary>
12      public static class SetFunctions
13      {
14          /// <summary>
15          /// Determine the minimum element of a collection, e.g. the chain with the least cells
16          /// or the cell with the least candidates.
17          /// </summary>
18          public static T Min<T>(IEnumerable<T> collection) where T :System.IComparable<T>
19          {
20              T minimum = default(T);
21              foreach (T element in collection)
22              {
23                  if (minimum == null)
24                  {
25                      minimum = element;
26                  }
27                  else if (element.CompareTo(minimum) < 0)
28                  {
29                      minimum = element;
30                  }
31              }
32              return minimum;
33          }
34
35          /// <summary>
36          /// Determine the intersection between two sets
37          /// </summary>
38          public static IEnumerable<T> Intersect<T>(IEnumerable<T> first, IEnumerable<T> second)
39          {
40              Dictionary<T, object> dict = new Dictionary<T, object>();
41              foreach (T element in first) dict[element] = null;
42              foreach (T element in second)
43              {
44                  if (dict.ContainsKey(element)) dict[element] = dict;
45              }
46              foreach (KeyValuePair<T, object> pair in dict)
47              {
48                  if (pair.Value != null) yield return pair.Key;
49              }
50          }
51      }
52  }
```

UniqueChain.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Text;

namespace MultiAgentSudokuSolver.Data
{
    public class UniqueChain : IComparable<UniqueChain>
    {
        private List<PuzzleCell> list;
        private int value;
        private int maxLength;

        /// <summary>
        /// Represents the connection between puzzle cells that lie in the same
        /// domain and shares a candidate value.
        /// </summary>
        /// <param name="value">The value of the chain, e.g. the common candidate value</param>
        /// <param name="maxLength">The maximum chain length, e.g the size of the domain</param>
        public UniqueChain(int value, int maxLength)
        {
            this.value = value;
            this.maxLength = maxLength;
            list = new List<PuzzleCell>(maxLength);
        }

        public Collection<PuzzleCell> GetCells()
        {
            return new Collection<PuzzleCell>(list);
        }

        /// <summary>
        /// Compares the given puzzlecells to the cells in the UniqueChain, and returns
        /// a collection containing the cells in the UniqueChain which is not present in
        /// the given collection of puzzlecells
        /// </summary>
        /// <param name="chain">A collection of cells to compare with</param>
        /// <returns>The cells from this UniqueChain which is not present in the given chain</returns>
        public Collection<PuzzleCell> GetDifferentCells(Collection<PuzzleCell> chain)
        {
            List<PuzzleCell> different = new List<PuzzleCell>();
            foreach (PuzzleCell cell in list)
            {
                if (!chain.Contains(cell))
                {
                    different.Add(cell);
                }
            }
            return new Collection<PuzzleCell>(different);
        }

        public PuzzleCell this[int index]
        {
            get { return list[index]; }
        }

        public int Value
        {
            get { return value; }
        }

        public int Count
        {
            get { return list.Count; }
        }

        public bool ContainsCell(PuzzleCell cell)
        {
            return list.Contains(cell);
        }

        // O(n)
        public void AddCell(PuzzleCell cell)
        {
            if (!list.Contains(cell))
            {
                list.Add(cell);
                OnChainChanged();
            }
        }

        // O(n)
```

```
84          public void RemoveCell(PuzzleCell cell)
85          {
86              list.Remove(cell);
87              OnChainChanged();
88          }
89
90          public event EventHandler ChainChanged;
91
92          private void OnChainChanged()
93          {
94              if (ChainChanged != null)
95              {
96                  this.ChainChanged(this, EventArgs.Empty);
97              }
98          }
99
100         public override string ToString()
101         {
102             return list.ToString();
103         }
104
105         #region IComparable<AdvancedUniqueChain> Members
106
107         public int CompareTo(UniqueChain other)
108         {
109             return this.Count - other.Count;
110         }
111
112         #endregion
113     }
114 }
```

ValueSolutionStep.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4
5   namespace MultiAgentSudokuSolver.Data
6   {
7       public class ValueSolutionStep : SolutionStep
8       {
9           public ValueSolutionStep(PuzzleCell cell, int value)
10              : base(cell, value)
11          {
12          }
13          public override bool Execute()
14          {
15              Cell.SetPrevious(Cell.CellValue);
16              Cell.CellValue = Value;
17              return true;
18          }
19
20          public override void Undo()
21          {
22              if (Cell.CellValue.HasValue)
23              {
24                  Cell.SetPrevious(value);
25                  Cell.CellValue = null;
26              }
27          }
28      }
29  }
```

DecisionBasis.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using System.Collections.ObjectModel;
5
6   namespace MultiAgentSudokuSolver.Data
7   {
8       /// <summary>
```

```
    /// Represents the basis of a decision.
    /// </summary>
    public class DecisionBasis
    {
        private PuzzleCell cell;
        private Collection<int> unusedCandidates;

        public DecisionBasis(PuzzleCell cell)
        {
            this.cell = cell;
            this.unusedCandidates = new Collection<int>(new List<int>(cell.Candidates));
        }

        public bool IsEmpty()
        {
            return (unusedCandidates.Count == 0);
        }

        // Returns the next possible decision in the basis
        public ValueSolutionStep NextDecision()
        {
            if (unusedCandidates.Count == 0) throw new Exception("DecisionBasis␣is␣empty");

            int value = unusedCandidates[0];
            unusedCandidates.RemoveAt(0);
            return new ValueSolutionStep(cell, value);
        }
    }
}
```

EventArgs.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MultiAgentSudokuSolver.Data
{
    public class EventArgs<T> : EventArgs
    {
        public EventArgs(T value)
        {
            this.value = value;
        }

        private T value;

        public T Value
        {
            get { return value; }
        }
    }
}
```

PuzzleCell.cs

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using System.Threading;
using System.Collections.ObjectModel;
using MultiAgentSudokuSolver.Agents;

namespace MultiAgentSudokuSolver.Data
{
    public class PuzzleCell : IComparable<PuzzleCell>
    {
        private Nullable<int> cellValue;
        private Nullable<int> previous;
        private List<int> candidates;
        private List<DomainAgent> domains;

        private readonly int puzzleSize;
```

```
19          private readonly int row, column, square;
20
21          private Nullable<int> uniqueValue;
22
23          public Nullable<int> UniqueValue
24          {
25              get { return uniqueValue; }
26
27          }
28
29          public int Row { get { return row; } }
30          public int Column { get { return column; } }
31          public int Square { get { return square; } }
32
33          public PuzzleCell(int puzzleSize, int row, int column, int square)
34          {
35              this.puzzleSize = puzzleSize;
36              this.row = row;
37              this.column = column;
38              this.square = square;
39              candidates = new List<int>(puzzleSize);
40              for (int i = 0; i < puzzleSize; i++)
41              {
42                  candidates.Add((int)(i + 1));
43              }
44              domains = new List<DomainAgent>(3);
45          }
46
47          public void AddDomain(DomainAgent domain)
48          {
49              domains.Add(domain);
50          }
51
52          public void RemoveDomain(DomainAgent domain)
53          {
54              domains.Remove(domain);
55          }
56
57          public Collection<DomainAgent> Domains
58          {
59              get { return new Collection<DomainAgent>(domains); }
60          }
61
62          public Nullable<int> CellValue
63          {
64              get { return cellValue; }
65              set
66              {
67                  cellValue = value;
68                  OnValueChanged(previous);
69              }
70          }
71
72          public void SetPrevious(Nullable<int> prev)
73          {
74              previous = prev;
75          }
76
77          public Collection<int> Candidates
78          {
79              get { return new Collection<int>(candidates); }
80          }
81
82          public bool Add(int candidate)
83          {
84              if (candidate <= 0 || candidate > puzzleSize)
85              {
86                  throw new ArgumentOutOfRangeException("number", "number must be between 1 and " + this.
                        puzzleSize);
87              }
88
89              if (!candidates.Contains(candidate))
90              {
91                  candidates.Add(candidate);
92                  if (candidates.Count == 1)
93                  {
94                      uniqueValue = candidates[0];
95                  }
96                  else uniqueValue = null;
97                  OnCandidatesAdd(candidate);
98                  return true;
99              }
100             // Do not make a changed event, if no candidates has been eliminated
101             return false;
102         }
103
```

```
104
105         public bool Eliminate(int candidate)
106         {
107             if (candidate <= 0 || candidate > puzzleSize)
108             {
109                 throw new ArgumentOutOfRangeException("number", "number␣must␣be␣between␣1␣and␣" + this.
                        puzzleSize);
110             }
111 
112             if (candidates.Contains(candidate))
113             {
114                 candidates.Remove(candidate);
115                 if (candidates.Count == 1)
116                 {
117                     // If there is only one candidate, it must be placed in this cell
118                     uniqueValue = candidates[0];
119                 }
120                 OnCandidatesChanged(candidate);
121                 return true;
122             }
123             // Do not make a changed event, if no candidates has been eliminated
124             return false;
125         }
126 
127         /// <summary>
128         /// Compare a PuzzleCell to this cell and determine if they contain the same candidates
129         /// </summary>
130         public bool CandidatesEquals(PuzzleCell cell)
131         {
132             bool bEqual = this.candidates.Count == cell.candidates.Count;
133             if (bEqual)
134             {
135                 for (int i = 0, nCount = this.candidates.Count; nCount > i; i++)
136                 {
137                     if (!this.candidates[i].Equals(cell.candidates[i]))
138                     {
139                         bEqual = false;
140                         break;
141                     }
142                 }
143             }
144             return bEqual;
145         }
146 
147         /// <summary>
148         /// Determine the difference in candidates
149         /// </summary>
150         public Collection<int> CandidatesDifferent(Collection<int> candidates)
151         {
152             List<int> different = new List<int>();
153             for (int i = 0, nCount = this.candidates.Count; nCount > i; i++)
154             {
155                 if (!candidates.Contains(this.candidates[i]))
156                 {
157                     different.Add(this.candidates[i]);
158                 }
159             }
160             return new Collection<int>(different);
161         }
162 
163         public override string ToString()
164         {
165             StringBuilder result = new StringBuilder();
166             result.Append("Cell␣value:␣" + this.CellValue + "\n");
167             result.Append("Candidates:␣");
168             for (int i = 0; i < Candidates.Count; i++)
169             {
170                 result.Append(Candidates[i] + "␣");
171             }
172 
173             return result.ToString();
174         }
175 
176         #region ValueChanged event
177 
178         public event EventHandler<EventArgs<Nullable<int>>> ValueChanged;
179 
180         public void OnValueChanged(Nullable<int> internalChange)
181         {
182             if (ValueChanged != null)
183             {
184                 ValueChanged(this, new EventArgs<Nullable<int>>(internalChange));
185             }
186         }
187         #endregion
188 
```

```
189         #region CandidatesChanged event
190
191         public event EventHandler<EventArgs<int>> CandidatesChanged;
192
193         public void OnCandidatesChanged(int candidate)
194         {
195             if (CandidatesChanged != null)
196             {
197                 CandidatesChanged(this, new EventArgs<int>(candidate));
198             }
199         }
200         #endregion
201
202         #region CandidatesAdd event
203
204         public event EventHandler<EventArgs<int>> CandidatesAdd;
205
206         public void OnCandidatesAdd(int candidate)
207         {
208             if (CandidatesAdd != null)
209             {
210                 CandidatesAdd(this, new EventArgs<int>(candidate));
211             }
212         }
213         #endregion
214
215
216         #region IComparable<List<int>> Members
217
218         public int CompareTo(PuzzleCell other)
219         {
220             return this.candidates.Count - other.Candidates.Count;
221         }
222
223         #endregion
224
225     }
226 }
```

SolutionStep.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Threading;
5
6  namespace MultiAgentSudokuSolver.Data
7  {
8      public abstract class SolutionStep
9      {
10         private PuzzleCell cell;
11         protected readonly int value;
12
13         public PuzzleCell Cell
14         {
15             get { return cell; }
16         }
17
18         public int Value
19         {
20             get { return value; }
21         }
22
23         protected SolutionStep(PuzzleCell cell, int value)
24         {
25             this.cell = cell;
26             this.value = value;
27         }
28
29         public abstract bool Execute();
30
31         public abstract void Undo();
32
33         public override bool Equals(object obj)
34         {
35             return (this.cell == (obj as SolutionStep).cell) & (this.value == (obj as SolutionStep).
                   value);
36         }
37
38         public override int GetHashCode()
```

```
39          {
40              return base.GetHashCode();
41          }
42      }
43  }
```

ValueDependencyMap.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using System.Collections;
5
6   namespace MultiAgentSudokuSolver.Data
7   {
8       /// <summary>
9       /// Class for managing the valuedependencies in a domain.
10      /// </summary>
11      public class ValueDependencyMap
12      {
13          #region Variables
14          Dictionary<int, UniqueChain> valueDependencyMap;// = new Dictionary<int, UniqueChain>();
15          Dictionary<int, List<UniqueChain>> chainLookupTable;// = new Dictionary<int, List<UniqueChain
                >>();
16          Dictionary<UniqueChain, int> chainLengthTable;// = new Dictionary<UniqueChain, int>();
17          int maxChainLength;
18          #endregion
19
20          public ValueDependencyMap(int puzzleSize)
21          {
22              valueDependencyMap = new Dictionary<int, UniqueChain>(puzzleSize);
23              chainLookupTable = new Dictionary<int, List<UniqueChain>>(puzzleSize+1);
24              chainLengthTable = new Dictionary<UniqueChain, int>(puzzleSize);
25              maxChainLength = puzzleSize;
26          }
27
28          #region Properties
29          /// <summary>
30          /// Gets or sets the unique chain which represents all the cells that are dependent on the
31          /// value supplied.
32          /// </summary>
33          /// <param name="key">Value</param>
34          /// <returns>UniqueChain containing all the cells dependent on the value key</returns>
35          public UniqueChain this[int key]
36          {
37              get { return (UniqueChain)valueDependencyMap[key]; }
38              set { valueDependencyMap[key] = value; }
39          }
40
41          /// <summary>
42          /// Returns a collection of the keys in the ValueDependencyMap
43          /// </summary>
44          public ICollection Keys { get { return (valueDependencyMap.Keys); } }
45
46          /// <summary>
47          /// Returns a collection of the values in the ValueDependencyMap
48          /// </summary>
49          public ICollection Values { get { return (valueDependencyMap.Values); } }
50
51          #endregion
52
53          #region Public methods
54          /// <summary>
55          /// Adds a PuzzleCell to the ValueDependencyMap.
56          /// For each candidate in the cell, the cell is added to the
57          /// corresponding entry in the ValueDependencyMap.
58          ///
59          /// Runs in O(n^2).
60          /// </summary>
61          /// <param name="cell">The cell to be added to the ValueDependencyMap</param>
62          public void AddCell(PuzzleCell cell)
63          {
64              UniqueChain chain;
65              int key;
66
67              for (int i = 0; i < cell.Candidates.Count; i++)
68              {
69                  key = cell.Candidates[i];
70                  if (!valueDependencyMap.ContainsKey(key))
71                  {
```

```
72                      chain = new UniqueChain(key, maxChainLength);
73                      chain.ChainChanged += new EventHandler(chain_ChainChanged);
74                      valueDependencyMap.Add(key, chain);
75                      chainLengthTable.Add(chain, 0);
76                  }
77                  valueDependencyMap[key].AddCell(cell);
78              }
79          }
80
81          /// <summary>
82          /// Remove a value from the ValueDependencyMap, meaning that no cells in this map
83          /// are longer dependent on this value.
84          ///
85          /// Runs in O(n)
86          /// </summary>
87          /// <param name="value">The candidate value to be removed</param>
88          public void RemoveValue(int value)
89          {
90              if (valueDependencyMap.ContainsKey(value))
91              {
92              // Unsubsribe event.
93              valueDependencyMap[value].ChainChanged -= new EventHandler(chain_ChainChanged);
94              // Remove reference from lookup table
95              chainLengthTable.Remove(valueDependencyMap[value]);
96              valueDependencyMap.Remove(value);
97              }
98          }
99
100         /// <summary>
101         /// Removes all references to a given cell from this ValueDependencyMap.
102         ///
103         /// Runs in O(n^2)
104         /// </summary>
105         /// <param name="cell">The cell to be removed</param>
106         public void RemoveCell(PuzzleCell cell)
107         {
108             List<int> removeKeys = new List<int>();
109
110             for (int i = 0; i < cell.Candidates.Count; i++)
111             {
112                 if (cell.Candidates[i] != cell.CellValue.Value)
113                 {
114                     if (valueDependencyMap[cell.Candidates[i]].ContainsCell(cell))
115                     {
116                         valueDependencyMap[cell.Candidates[i]].RemoveCell(cell);
117                     }
118                 }
119             }
120         }
121
122         /// <summary>
123         /// Returns an iterator that iterates through the ValueDependencyMap
124         /// </summary>
125         public IEnumerator GetEnumerator()
126         {
127             return valueDependencyMap.GetEnumerator();
128         }
129
130         /// <summary>
131         /// Remove a cell from a given value dependency.
132         ///
133         /// Runs in O(n)
134         /// </summary>
135         /// <param name="value">The value that the cell is no longer dependent of</param>
136         /// <param name="cell">The cell that is no longer dependent of the value</param>
137         public void RemoveCellAt(int value, PuzzleCell cell)
138         {
139             if (valueDependencyMap.ContainsKey(value))
140             {
141                 UniqueChain chain = ((UniqueChain)valueDependencyMap[value]);
142                 chain.RemoveCell(cell);
143             }
144         }
145
146         /// <summary>
147         /// Add a cell to a given value dependency.
148         ///
149         /// Runs in O(n)
150         /// </summary>
151         /// <param name="value">The value that the cell should be dependent of</param>
152         /// <param name="cell">The cell that are dependent of the value</param>
153         public void AddCellAt(int value, PuzzleCell cell)
154         {
155             // O(n)
156             if (!valueDependencyMap.ContainsKey(value))
157             {
```

```
158                 valueDependencyMap.Add(value, new UniqueChain(value, maxChainLength));
159                 valueDependencyMap[value].ChainChanged += new EventHandler(chain_ChainChanged);
160                 chainLengthTable.Add(valueDependencyMap[value], 0);
161             }
162 
163             UniqueChain chain = valueDependencyMap[value];
164             // O(n)
165             if (!chain.ContainsCell(cell))
166                 chain.AddCell(cell);
167         }
168         #endregion
169 
170         #region Eventhandlers
171         /// <summary>
172         /// Eventhandler for the ChainChanged event. Updates the lookuptables, so
173         /// the chain can be located by length.
174         ///
175         /// Runs in O(n)
176         /// </summary>
177         /// <param name="sender">The chain that has been changed</param>
178         /// <param name="e">Not used</param>
179         private void chain_ChainChanged(object sender, EventArgs e)
180         {
181             UniqueChain chain = sender as UniqueChain;
182 
183             int previousLength = chainLengthTable[chain];
184             int newLength = chain.Count;
185 
186             // O(n)
187             if (chainLookupTable.ContainsKey(previousLength))
188             {
189                 chainLookupTable[previousLength].Remove(chain);
190             }
191             // O(n)
192             if (!chainLookupTable.ContainsKey(newLength))
193             {
194                 chainLookupTable.Add(newLength, new List<UniqueChain>());
195             }
196             // O(n)
197             chainLookupTable[newLength].Add(chain);
198             chainLengthTable[chain] = chain.Count;
199 
200             // If chain is decremented and chain length is 1
201             // we have a unique value
202             if ((previousLength > newLength) && chain.Count == 1)
203             {
204                 // Raise event
205                 OnUniqueValue(new ValueSolutionStep(chain[0], chain.Value));
206             }
207         }
208         #endregion
209 
210         #region Events
211         public event EventHandler<EventArgs<SolutionStep>> UniqueValue;
212 
213         private void OnUniqueValue(SolutionStep step)
214         {
215             if (UniqueValue != null)
216             {
217                 this.UniqueValue(this, new EventArgs<SolutionStep>(step));
218             }
219         }
220         #endregion
221     }
222 }
```

B.5 Cache

Solution.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using System.Collections.ObjectModel;
5 
6 namespace MultiAgentSudokuSolver.Cache
```

```
 7  {
 8      public class Solution
 9      {
10          //The steps along the way to the solution
11          private Collection<CacheSolutionStep> steps;
12          public Collection<CacheSolutionStep> Steps
13          {
14              get { return steps; }
15          }
16          private int nakedCount;
17          public int NakedCount
18          {
19              get { return nakedCount; }
20          }
21          private int hiddenCount;
22          public int HiddenCount
23          {
24              get { return hiddenCount; }
25          }
26          private int intersectionCount;
27          public int IntersectionCount
28          {
29              get { return intersectionCount; }
30          }
31
32          private int guesses;
33          public int Guesses
34          {
35              get { return guesses; }
36          }
37
38          private bool isSearched;
39          public bool IsSearched
40          {
41              get { return isSearched; }
42          }
43
44
45          public Solution(Collection<CacheSolutionStep> steps, int guesses, bool isSearched, int
                 nakedCount, int hiddenCount, int intersectionCount)
46          {
47              this.steps = steps;
48              this.guesses = guesses;
49              this.isSearched = isSearched;
50              this.nakedCount = nakedCount;
51              this.hiddenCount = hiddenCount;
52              this.intersectionCount = intersectionCount;
53          }
54      }
55  }
```

CacheSolutionStep.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Text;
 4  using System.Drawing;
 5  using MultiAgentSudokuSolver.Messaging;
 6  using MultiAgentSudokuSolver.Data;
 7
 8  namespace MultiAgentSudokuSolver.Cache
 9  {
10      //Status for the entire puzzle grid
11      public class CacheSolutionStep : ICloneable
12      {
13          //The cells in the grid
14          private CacheCell[,] cells;
15          public CacheCell[,] Cells { get { return cells; } }
16
17          //The latest cell assigned a value
18          private CacheCell newestCell;
19          public CacheCell NewestCell { get { return newestCell; } }
20
21          //The size of the puzzle
22          private int puzzleSize;
23
24          //Times the naked strategy is used
25          private int nakedCount;
26          public int NakedCount
27          {
```

```
28              get { return nakedCount; }
29          }
30
31          //Times the hidden strategy is used
32          private int hiddenCount;
33          public int HiddenCount
34          {
35              get { return hiddenCount; }
36          }
37
38          //Times the intersection strategy is used
39          private int intersectionCount;
40          public int IntersectionCount
41          {
42              get { return intersectionCount; }
43          }
44
45          public CacheSolutionStep(int puzzleSize)
46          {
47              //Initialize the grid and assigned all cells to be empty
48              this.puzzleSize = puzzleSize;
49              cells = new CacheCell[puzzleSize, puzzleSize];
50              for (int column = 0; column < puzzleSize; column++)
51              {
52                  for (int row = 0; row < puzzleSize; row++)
53                  {
54                      cells[column, row] = new CacheCell(column, row);
55                      cells[column, row].Type = CacheCell.StateChange.Empty;
56                  }
57              }
58          }
59
60          //Clone the current CacheSolutionStep
61          internal CacheSolutionStep(CacheSolutionStep step)
62          {
63              //Save a copy of the grid by cloning the current state
64              this.cells = new CacheCell[step.puzzleSize, step.puzzleSize];
65              this.puzzleSize = step.puzzleSize;
66              this.nakedCount = step.nakedCount;
67              this.hiddenCount = step.hiddenCount;
68              this.intersectionCount = step.intersectionCount;
69
70              for (int column = 0; column < puzzleSize; column++)
71              {
72                  for (int row = 0; row < puzzleSize; row++)
73                  {
74                      this.cells[column, row] = (CacheCell)step.cells[column, row].Clone();
75                  }
76              }
77              //Save the latest value placed on the Sudoku grid
78              if (step.newestCell != null)
79                  this.newestCell = step.newestCell.Clone() as CacheCell;
80          }
81
82          //Add the current value to the CacheCell
83          public void AddValueStep(PuzzleCell cell)
84          {
85              cells[cell.Column, cell.Row].Value = cell.CellValue;
86              newestCell = cells[cell.Column, cell.Row];
87              cells[cell.Column, cell.Row].Type = CacheCell.StateChange.Value;
88          }
89
90          //Add the current candidates to the CacheCell
91          public void AddCandidateStep(PuzzleCell cell)
92          {
93              cells[cell.Column, cell.Row].Candidates.Clear();
94              cells[cell.Column, cell.Row].Candidates.AddRange(cell.Candidates);
95              cells[cell.Column, cell.Row].Type = CacheCell.StateChange.Candidate;
96          }
97
98          //Remove flag that indicates that a hidden, naked or intersection strategy has been used to
99          //elimate candidate(s). Remove the flag indicating that the cell is affected by a strategy.
100         public void RemoveStrategies()
101         {
102             for (int column = 0; column < puzzleSize; column++)
103             {
104                 for (int row = 0; row < puzzleSize; row++)
105                 {
106                     cells[column, row].NakedCand = false;
107                     cells[column, row].HiddenCand = false;
108                     cells[column, row].IntersectionCand = false;
109                     cells[column, row].AffectedByStrategy = false;
110                 }
111             }
112         }
113
```

```
114         //Save what have been eliminated so far to the current solution step.
115         public void AddEliminationStep(EliminationStrategyMessage elimination)
116         {
117             CacheCell cell;
118
119             //Count the times the different strategies are used
120             switch (elimination.StrategyType)
121             {
122                 case "NakedAgent":
123                     if (elimination.Eliminations.Count > 0)
124                         nakedCount++;
125                     break;
126                 case "HiddenAgent":
127                     if (elimination.Eliminations.Count > 0)
128                         hiddenCount++;
129                     break;
130                 case "IntersectionAgent":
131                     if (elimination.Eliminations.Count > 0)
132                         intersectionCount++;
133                     break;
134                 default:
135                     break;
136             }
137
138             //Save each elimination performed by the strategies
139             if (elimination.Eliminations.Count > 0)
140             {
141                 foreach (PuzzleCell c in elimination.StrategyCellSet)
142                 {
143                     cell = cells[c.Column, c.Row];
144                     //Save the candidates used as part of the strategy
145                     foreach (int candidate in elimination.StrategyCandidateSet)
146                     {
147                         if (!cell.StrategyCand.Contains(candidate))
148                         {
149                             cell.StrategyCand.Add(candidate);
150                         }
151                     }
152
153                     //Determine the strategy which has performed the elimination
154                     switch (elimination.StrategyType)
155                     {
156                         case "NakedAgent":
157                             cell.NakedCand = true;
158                             //Run through all the eliminations
159                             foreach (EliminationSolutionStep eliminationStep in elimination.
                                    Eliminations)
160                             {
161                                 cell = cells[eliminationStep.Cell.Column, eliminationStep.Cell.Row];
162                                 //Set the flag AffectedByStrategy to indicate that the elimination in
                                        the cell
163                                 //is influenced by the strategy
164                                 cell.AffectedByStrategy = true;
165                                 //Add to the cell the candidate that has been eliminated
166                                 if (!cell.EliminationByNaked.Contains(eliminationStep.Value))
167                                 {
168                                     cell.EliminationByNaked.Add(eliminationStep.Value);
169                                 }
170                             }
171                             break;
172                         case "HiddenAgent":
173                             cell.HiddenCand = true;
174                             //Run through all the eliminations
175                             foreach (EliminationSolutionStep eliminationStep in elimination.
                                    Eliminations)
176                             {
177                                 cell = cells[eliminationStep.Cell.Column, eliminationStep.Cell.Row];
178                                 //Set the flag AffectedByStrategy to indicate that the elimination in
                                        the cell
179                                 //is influenced by the strategy
180                                 cell.AffectedByStrategy = true;
181                                 //Add to the cell the candidate that has been eliminated
182                                 if (!cell.EliminationByHidden.Contains(eliminationStep.Value))
183                                 {
184                                     cell.EliminationByHidden.Add(eliminationStep.Value);
185                                 }
186                             }
187                             break;
188                         case "IntersectionAgent":
189                             cell.IntersectionCand = true;
190                             //Run through all the eliminations
191                             foreach (EliminationSolutionStep eliminationStep in elimination.
                                    Eliminations)
192                             {
193                                 cell = cells[eliminationStep.Cell.Column, eliminationStep.Cell.Row];
```

```
194                                 //Set the flag AffectedByStrategy to indicate that the elimination in
                                         the cell
195                                 //is influenced by the strategy
196                                 cell.AffectedByStrategy = true;
197                                 //Add to the cell the candidate that has been eliminated
198                                 if (!cell.EliminationByIntersection.Contains(eliminationStep.Value))
199                                 {
200                                     cell.EliminationByIntersection.Add(eliminationStep.Value);
201                                 }
202                             }
203                             break;
204                         default:
205                             break;
206                     }
207                 }
208             }
209 
210 
211             //Save each elimination performed by the domain agents
212             if (elimination.StrategyType == "DomainAgent")
213             {
214                 //Run through all the eliminations
215                 foreach (EliminationSolutionStep step in elimination.Eliminations)
216                 {
217                     cell = cells[step.Cell.Column, step.Cell.Row];
218                     //Add to the cell the candidate that has been eliminated
219                     if (!cell.EliminationByDomain.Contains(step.Value))
220                     {
221                         cell.EliminationByDomain.Add(step.Value);
222                     }
223                 }
224             }
225 
226             //Save all the eliminated candidate(s) to the CacheCell.
227             foreach (EliminationSolutionStep eliminationStep in elimination.Eliminations)
228             {
229                 cell = cells[eliminationStep.Cell.Column, eliminationStep.Cell.Row];
230                 if (!cell.EliminatedCand.Contains(eliminationStep.Value))
231                 {
232                     cell.EliminatedCand.Add(eliminationStep.Value);
233                 }
234             }
235 
236         }
237 
238         #region ICloneable Members
239 
240         public object Clone()
241         {
242             return new CacheSolutionStep(this); ;
243         }
244 
245         #endregion
246     }
247 }
```

CacheCell.cs

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Text;
 4 using System.Drawing;
 5 
 6 namespace MultiAgentSudokuSolver.Cache
 7 {
 8     public class CacheCell :ICloneable
 9     {
10         //The cell can be either empty, assigned to a value or
11         //contain candidate(s)
12         public enum StateChange { Empty, Value, Candidate}
13 
14         //The state of the cell
15         private StateChange type;
16         public StateChange Type
17         {
18             get { return type; }
19             set { type = value; }
20         }
21 
22         //The candidates in the cell
```

```
23          private List<int> candidates;
24          public List<int> Candidates
25          {
26              get { return candidates; }
27              set { candidates = value; }
28          }
29
30          //The eliminated candidates in the cell
31          private List<int> eliminatedCand;
32          public List<int> EliminatedCand
33          {
34              get { return eliminatedCand; }
35              set { eliminatedCand = value; }
36          }
37
38          //The candidates eliminated by the hidden strategy
39          private List<int> eliminationByHidden;
40          public List<int> EliminationByHidden
41          {
42              get { return eliminationByHidden; }
43              set { eliminationByHidden = value; }
44          }
45
46          //The candidates eliminated by the naked strategy
47          private List<int> eliminationByNaked;
48          public List<int> EliminationByNaked
49          {
50              get { return eliminationByNaked; }
51              set { eliminationByNaked = value; }
52          }
53
54          //The candidates eliminated by the intersection strategy
55          private List<int> eliminationByIntersection;
56          public List<int> EliminationByIntersection
57          {
58              get { return eliminationByIntersection; }
59              set { eliminationByIntersection = value; }
60          }
61
62          //The candidates eliminated by the domain agents
63          private List<int> eliminationByDomain;
64          public List<int> EliminationByDomain
65          {
66              get { return eliminationByDomain; }
67              set { eliminationByDomain = value; }
68          }
69
70          //The candidates used in a strategy
71          private List<int> strategyCand;
72          public List<int> StrategyCand
73          {
74              get { return strategyCand; }
75              set { strategyCand = value; }
76          }
77
78          //The value in the cell
79          private Nullable<int> _value;
80          public Nullable<int> Value
81          {
82              get { return _value; }
83              set { _value = value; }
84          }
85
86          //Is it used in a naked strategy
87          private bool nakedCand;
88          public bool NakedCand
89          {
90              get { return nakedCand; }
91              set { nakedCand = value; }
92          }
93
94          //Is it used in a hidden strategy
95          private bool hiddenCand;
96          public bool HiddenCand
97          {
98              get { return hiddenCand; }
99              set { hiddenCand = value; }
100         }
101
102         //Is it used in a intersection strategy
103         private bool intersectionCand;
104         public bool IntersectionCand
105         {
106             get { return intersectionCand; }
107             set { intersectionCand = value; }
108         }
```

```
109
110         //Is it affected by a strategy
111         private bool affectedByStrategy;
112         public bool AffectedByStrategy
113         {
114             get { return affectedByStrategy; }
115             set { affectedByStrategy = value; }
116         }
117
118         //Column location
119         private int column;
120         public int Column
121         {
122             get { return column; }
123         }
124
125         //Row location
126         private int row;
127         public int Row
128         {
129             get { return row; }
130         }
131
132         public CacheCell(int column, int row)
133         {
134             this.column = column;
135             this.row = row;
136             this.nakedCand = false;
137             this.hiddenCand = false;
138             this.intersectionCand = false;
139             this.affectedByStrategy = false;
140             this.candidates = new List<int>();
141             this.eliminatedCand = new List<int>();
142             this.strategyCand = new List<int>();
143             this.eliminationByHidden = new List<int>();
144             this.eliminationByNaked = new List<int>();
145             this.eliminationByIntersection = new List<int>();
146             this.eliminationByDomain = new List<int>();
147
148         }
149
150         //Clone the current cell
151         internal CacheCell(CacheCell cell)
152         {
153             this.row = cell.row;
154             this.column = cell.column;
155             this.type = cell.type;
156             this._value = cell._value;
157             this.hiddenCand = cell.hiddenCand;
158             this.nakedCand = cell.nakedCand;
159             this.intersectionCand = cell.intersectionCand;
160             this.affectedByStrategy = cell.affectedByStrategy;
161             this.candidates = new List<int>();
162             this.candidates.AddRange(cell.candidates.ToArray());
163             this.eliminatedCand = new List<int>();
164             this.eliminatedCand.AddRange(cell.eliminatedCand.ToArray());
165             this.strategyCand = new List<int>();
166             this.strategyCand.AddRange(cell.strategyCand.ToArray());
167             this.eliminationByHidden = new List<int>();
168             this.eliminationByHidden.AddRange(cell.eliminationByHidden.ToArray());
169             this.eliminationByNaked = new List<int>();
170             this.eliminationByNaked.AddRange(cell.eliminationByNaked.ToArray());
171             this.eliminationByIntersection = new List<int>();
172             this.eliminationByIntersection.AddRange(cell.eliminationByIntersection.ToArray());
173             this.eliminationByDomain = new List<int>();
174             this.eliminationByDomain.AddRange(cell.eliminationByDomain.ToArray());
175
176         }
177
178         #region ICloneable Members
179
180         public object Clone()
181         {
182             return new CacheCell(this);
183         }
184
185         #endregion
186     }
187 }
```

SolutionBuilder.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections.ObjectModel;

namespace MultiAgentSudokuSolver.Cache
{
    //Used to save a CacheSolutionStep each time a cell is assigned a value
    class SolutionBuilder
    {
        //Size of the puzzle
        private int puzzleSize;
        //List containing the different steps
        private List<CacheSolutionStep> stepList;

        private int nakedCount;

        public int NakedCount
        {
            get { return nakedCount; }
            set { nakedCount = value; }
        }

        private int hiddenCount;

        public int HiddenCount
        {
            get { return hiddenCount; }
            set { hiddenCount = value; }
        }

        private int intersectionCount;

        public int IntersectionCount
        {
            get { return intersectionCount; }
            set { intersectionCount = value; }
        }

        private int guesses;
        public int Guesses
        {
            get { return guesses; }
            set { guesses = value; }
        }

        private bool isSearched;
        public bool IsSearched
        {
            get { return isSearched; }
            set { isSearched = value; }
        }

        //The numbers of steps so far
        public int CurrentStepIndex { get { return stepList.Count; } }

        public SolutionBuilder(int puzzleSize)
        {
            this.puzzleSize = puzzleSize;
            stepList = new List<CacheSolutionStep>(puzzleSize * puzzleSize);
        }

        //Save the current step
        public void SaveSolutionStep(CacheSolutionStep step)
        {
            stepList.Add(step);
        }

        //Get a Cell at the index given by stepIndex
        public CacheCell GetCellAtStepIndex(int stepIndex, CacheCell cell)
        {
            return stepList[stepIndex].Cells[cell.Column, cell.Row];
        }

        // Return copy of current stepList
        public Collection<CacheSolutionStep> GetSolutionSteps()
        {
            return new Collection<CacheSolutionStep>(new List<CacheSolutionStep>(stepList));
        }
    }
}
```

PuzzleValidator.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using MultiAgentSudokuSolver.Data;
5
6  namespace MultiAgentSudokuSolver.Cache
7  {
8      //Validate the puzzle
9      public class PuzzleValidator
10     {
11         public static bool Validate(PuzzleCell[,] cells, int puzzleSize)
12         {
13             bool result = false;
14             //Run trough each cell
15             for (int column = 0; column < puzzleSize; column++)
16             {
17                 for (int row = 0; row < puzzleSize; row++)
18                 {
19                     //Validate the cell
20                     if (ValidateCell(cells, cells[column, row], puzzleSize))
21                     {
22                         result = true;
23                     }
24                     //All cells must validate for the entire grid to validate
25                     else
26                     {
27                         return false;
28                     }
29                 }
30             }
31
32             return result;
33         }
34
35
36         private static bool ValidateCell(PuzzleCell[,] cells, PuzzleCell cell, int puzzleSize)
37         {
38             List<int> valueList;
39             int puzzleOrder = (int)Math.Sqrt(puzzleSize);
40             int square = (cell.Row / puzzleOrder) + (cell.Column / puzzleOrder) * puzzleOrder;
41
42             //Validate Column
43             valueList = new List<int>();
44             //Run trough all cells in the column belonging to the current cell
45             for (int column = 0; column < puzzleSize; column++)
46             {
47                 //Add all cell values to valueList except the value in the current cell
48                 if (column != cell.Column)
49                 {
50                     valueList.Add((int)cells[column, cell.Row].CellValue.Value);
51                 }
52             }
53             //If the value in the current cell is contained in valueList, the value
54             //is present in the column more than once -> constraints are violated
55             if (valueList.Contains(cell.CellValue.Value))
56             {
57                 return false;
58             }
59
60             //Validate Row
61             valueList = new List<int>();
62             //Run trough all cells in the row belonging to the current cell
63             for (int row = 0; row < puzzleSize; row++)
64             {
65                 //Add all cell values to valueList except the value in the current cell
66                 if (row != cell.Row)
67                 {
68                     valueList.Add(cells[cell.Column, row].CellValue.Value);
69                 }
70             }
71             //If the value in the current cell is contained in valueList, the value
72             //is present in the row more than once -> constraints are violated
73             if (valueList.Contains((cell.CellValue.Value)))
74             {
75                 return false;
76             }
77
78             //Validate Square
79             valueList = new List<int>();
80             int lowerGlobalX = (square % puzzleOrder) * puzzleOrder;
81             int lowerGlobalY = (square / puzzleOrder) * puzzleOrder;
82             int upperGlobalX = lowerGlobalX + puzzleOrder - 1;
83             int upperGlobalY = lowerGlobalY + puzzleOrder - 1;
```

```
84              //Run trough all cells in the sqaure belonging to the current cell
85              for (int i = lowerGlobalX; i <= upperGlobalX; i++)
86              {
87                  for (int j = lowerGlobalY; j <= upperGlobalY; j++)
88                  {
89                      //Add all cell values to valueList except the value in the current cell
90                      if (cells[j, i] != cell)
91                      {
92                          valueList.Add(cells[j, i].CellValue.Value);
93                      }
94                  }
95              }
96              //If the value in the current cell is contained in valueList, the value
97              //is present in the square more than once -> constraints are violated
98              if (valueList.Contains(cell.CellValue.Value))
99              {
100                 return false;
101             }
102
103             return true;
104         }
105     }
106 }
```