Danish Technical University (DTU)

Institute for Informatics & Mathematical Modelling (IMM)

02125 Bachelor Project in Software Technology

IMPLEMENTING FAST PLANNING WITH REGARDS TO GAME-AI

Date for deliverance: 01/02/2012

Dan True s082924 ______________________

Jakob Holmelund s070476 ______________________

1. RESUME

In this project we develop and present a formal planning domain with key features alike to a classic action game. Areas covered are dynamic environments, partial-observability, hierarchical move actions, execution monitoring and (re)planning.

We analyse a few well-known game genres and establish a set of properties they share with regards to planning. We present existing and propose new theoretical solutions to the problems inherent in these types of domains. We analyse three well-known planning techniques: Forward-State Space (FSP), GraphPlan & Partial-Order Planning(POP) and their application to our specified domain. We then implement an FSP planner and a POP planner and measure their performance in the environment.

We develop an agent taking advantage of a number of the proposed solutions. This agent iterates through a queue of goals, receives percepts, maintains a belief base representing the world, formulates and monitors plans and executes actions.

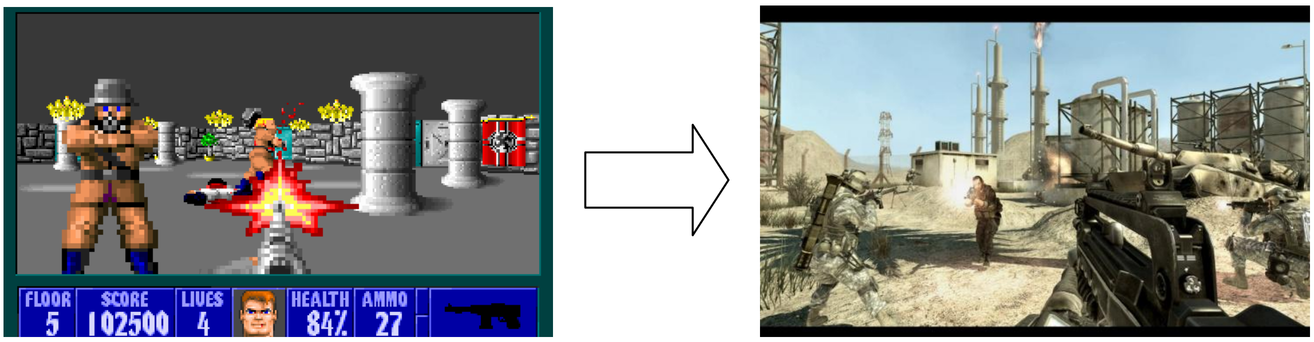
To test and demonstrate the agent method we have designed a simple graphical user interface, in which objects can be placed and moved dynamically during runtime. The agent is added to the environment and given goals to fulfil, overcoming problems caused by the dynamic environment, partial-observability etc. during run-time.

The project ran from September 2011 to and including January 2012.

2. Problem Description

The motivation for this project is dissatisfaction with the AI presented in modern computer games and the way it is applied. Furthermore it is our experience that this is an area of computer science where the industry seems to be significantly behind what is achieved in research projects[1] and at planning competitions. Therefore we wish to explore the use of state-of-the-art research results in modern computer games.

It has been 20 years since the game Wolfenstein 3D came out as one of the first fast-paced action-games, a genre that became and still is among the most popular genres of games, with modern titles beating Hollywood blockbusters in terms of revenue[2]. These games differ significantly from many earlier games, which were primarily boardgame-like in style, in that they offer a dynamic (often 3D) world in which the player agent acts.



Wolfenstein 3D, 1992 *Call of Duty: Modern Warfare 3, 2011*

Nearly 20 years of advances in sound, graphics, network and controls; yet the computer-controlled agents on the images still fire without taking cover.

Since then there has been a massive advancement in games, especially graphics, controls, interface and sound. The behaviour of computer-controlled agents has comparatively not advanced nearly as much; they still often stand in the open and shoot at you (unable to formulate a proper plan) and if they do take cover they often ignore even blatantly obvious flanking moves (unable to change its plan effectively due to changes in the environment). Difficulty of the game is often controlled by making opponents tougher, more numerous or simply by allowing them to “cheat”[3] (spawn behind the player, ignore line of sight etc.) instead of making the opponents fight smarter. Usually computer game agents are scripted to react in specific ways to specific input via Finite State Machines(FSMs) and/or behaviour trees [4,5,6,7].

We believe that many games could be much better if more resources were given over to implementing planning techniques in agent behaviour and we believe it is possible if the Game Studios’ used the results from research achieved in the last twenty years. In our view this is true for many areas of computer games:

- Action games could become more tactical as opponents behave more intelligently and coherently
- Simulations (such as The Sims or SimCity) can become more realistic and complex with sub elements modelled as agents instead of simple deterministic reactors

- Any type of story-driven game can be modelled with participants as agents with goals and varying initial states, such that the details of the story change within certain constraints each time it is played

2.1.1. *Arguments against AI in games*

There can be numerous reasons for not wishing to implement planning in computer games[8], including:

- Computational complexity – the game needs the CPU cycles for other areas
- Agents taking own actions more liberally can ruin story-driven events
- Hard to develop/debug – As the AI can itself take decisions, the debugging procedure becomes more cumbersome

However, some game developers who have introduced planning in their games find that they can be easier to develop/alter/debug as you simply change the action schemas, goals etc instead of a large FSM[9]. So that planning AIs are harder to develop or maintain is not a final conclusion.

We have no statistical indication that better AI *in general* makes a game more fun to play however and the popularity of simple games for handheld devices such as smartphones could indicate that the average computer-game consumer is not interested in better AI.

However, we believe that modern planning techniques could have a lot to bring to computer games, especially genres where AI seems to be important such as strategy games - and that is why we find it relevant to explore techniques and their application for this area.

2.2. Type and Complexity of Problems

The type of problems needed to be solved in computer games can vary significantly from game to game - it depends a lot on the genre of game in question. We will here briefly analyse a few different genres of computer games, and attempt to procure a set of common properties on which to base a planning domain.

Many genres have a completely different set of problems and this project will not be relevant for these, or even for all aspects of the covered genres. For instance, the AI we focus on in this project does not deal with the problem of resource-gathering and scheduling of unit production, which are common areas of Strategy games, and are in themselves complex planning problems[10]. We deal only with the agent problems that we find are most common across a multitude of genres: navigating and surviving/succeeding in a world, such as opponents in shooter games, troops in strategy games, population in simulation games, etc.

We only focus on single-agent planning, leaving the complexities of multi-agent planning in these environments to future projects.

It is assumed that an AI agent in one of these domains will try to either hinder the player in attaining his goals, or to attain goals itself of a nature similar to the ones given to a human player. Therefore the properties of the computer game genres are not dependent on viewing it from a human- or AI agent perspective.

In the following property determination of genre properties, we use the following terms and definitions. Note that these might differ from the usual domain defining properties of the same name, as we are in the following analysing computer games – not planning domains.

Pace	By the **pace** of a game is meant the overall speed of gameplay. i.e. games where events have short time in between or where quick reflexes are necessary to avoid death are termed as **fast paced**, where slower games are **slowly paced**.
Dynamism	The level of dynamism is tied to origin of changes in the game world. A game where change only happens when the agent acts is called **static**, whereas other games are termed as **dynamic**.
Complexity	The level of complexity of a game is determined upon the complexity of goals a player or AI agent is normally given to solve. A goal is termed as complex if it has a lot of interactions / couplings to other goals– i.e. the solution to one goal can block the solution to another goal. For instance, placing a set of buildings in a city-building game in limited building space can be a complex problem, as they might have to be placed in a very specific pattern to avoid blocking each other. Games where the player usually receives non-complex goals are termed as having **low complexity**, games where complex goals are ordinary are termed as having **high complexity**.
Observability	Observability is keyed to the available data in a given instance of the game. Worlds where no data is hidden from the player or an AI agent are termed as **fully observable**, whereas worlds where walls block line of sight, there exists invisible objects etc. are **partially observable**.
Determinism	Determinism here reflects whether results of actions taken in the game are deterministic or not. In a **deterministic** environment the next state of the world is completely determined by the current state of the world and vice versa. For instance games using pseudorandom numbers for weapon damage are here termed as **non-deterministic**.

We will now analyse three genres of computer games with respect to the above defined properties:

First Person Shooters (FPS)	
Fast paced	Indecision often means death for the agent, so handling of changes will have to be fast and efficient.
Dynamic	Game worlds of this genre are generally dynamic, and plans tend to often break, as changes in the world occurs quickly and often - for instance another agent blocking a path or a grenade making an area threatened would happen often.
Low complexity	Goals tend to be simple (Kill player, Plant a bomb etc.). This is a consequence both of the genre (emulating the perplexity of combat) and the fact that the dynamic and fast-paced nature of the genre rarely leaves time to come up with solutions to complex plans
Partially observable	There are often object which agents are not aware of. This can be both in the direct sense, that the world description does not include them - a mine for instance, or in a more consequential way because other objects block line of sight: even if the domain does not include object which cannot be perceived, buildings, tall grass etc will often limit the field of vision and hide potential threats.
Deterministic/ Effectively Non-Deterministic	Whether a specific FSP game is non-deterministic or not, is hard to say. Many games do not include non-determinism, because that puts a strain on the bandwidth when playing multiplayer. However, the nature of the environment is that even though everything might be deterministic, just like in the real-world the effects might be so fine-grained that to all intents and purposes the domain can be considered non-deterministic. For instance, the procedure to determine where a grenade falls might

	be deterministic, but movement, small objects etc make it impossible for a human to determine the precise location. An agent would probably be given the same limitation, for instance by making the throw of a grenade coarse-grained (aiming for an area instead of a specific position in the game world).
Real-Time Strategy (RTS)[i]	
Averagely paced	Real-time strategy games can be very fast-paced or very slow-paced, depending on specific game or player settings. For instant StarCraft II is very fast paced and can potentially require reactions on par with the ones needed for an FPS. On the other hand games like Hearts of Iron I-III can be slowly paced, to provide ample time to attention to details.
Dynamic	The worlds of this genre tend to be dynamic, as units can be created, buildings destroyed etc. also there are potentially a lot of agents, which can affect the world around them.
Low- To Average complexity of plans	This varies greatly from game to game. In some games like StarCraft II or Warcraft the complexity of goals are rather low – they tend to be about destroying a specific (or all) unit(s) and/or building(s). However, these may be more complex because of unit abilities or bonuses acquired from a combination of units or locations. Other games of the genre, like Hearts of Iron or the Total War series can have a high complexity of goals.
Partially observable	Almost all real-time strategy games provide only a limited view of the world, be it from a fog-of-war effect, ability for units to hide in terrain or because of limited line-of-sight.
Deterministic / Effectively Non-Deterministic	Some strategy games are deterministic due to multiplayer issues, for these see the deterministic property of FSP games above. Other games include non-deterministic factors such as randomised damage for some weapons or weather effects.
Simulation/Economic Games (SIM)	
Slowly paced	Simulations are usually slowly paced. An agent in The Sims or The Anno series usually have ample time to react to changes and there are not many lethal elements to terminate indecisive planners.
Dynamic	The worlds of this genre are usually dynamic, as else the simulation would stagnate and become rather boring. Neighbours might come to visit or the player might construct new buildings. The changes in the environment are however often more subtle and non-lethal than the ones found in FSP or RTS.
High complexity	Goals in these kinds of games can be very complex, with many negative interactions between sub-goals. For instance the placement of building A, might require demolishing of another building B - which will have to be constructed anew elsewhere so the population does not get unhappy. The placement of this new building may be constricted by range or effect or the like, which may again interfere with the placement of building A.
Fully- or Partially observable	There are examples of games of this genre with full-observability. It is however also common that games include a fog-of-war or other limiting properties.
Non-Deterministic	It is not uncommon for these types of games to have non-deterministic elements, such as weather effects or randomised effects of actions.

[i] Note, RTS is an ambiguous abbreviation usually ascribed to a sub-genre of strategy, containing games such as StarCraft, Command & Conquer and Warcraft. Here it is used instead to denote all strategy games that take place in real-time, as contrasted to turn-based strategy games.

As mentioned earlier many modern games use Finite State Machines (FSMs) or behaviour trees for decision-making. With the above determined properties of various genres of computer games, it is clear that this approach cannot be sufficient if the aim is to make better game AI: The computational triviality of FSMs makes them able to make decisions fast, to cope with dynamic nature. However, FSMs and behaviour trees become large and cumbersome to create, maintain and change[11]. In addition, they cannot inherently handle long-term goals, and complex decisions are very hard to specify. The problems of maintaining FSMs can, and probably already is, handled through better frameworks and one can introduce some long-term goal awareness by building on the FSMs framework.

However, that FSMs are the best candidate for game AI has been a paradigm for the last twenty years of game development, a time when computational resources were much more limited than today. With modern processing power and multi-core CPUs, perhaps a more computationally heavy method for decision-making would be possible, while still be real-time. If nothing else, it is relevant to question the paradigm.

We will therefore aim to implement a planner for an arbitrary AI-controlled game agent. This planner does not constitute a thorough game AI, since we will not implement game tree, enemy intentions or resource management. For now we are trying to design a planning agent that is as generic and plug-able as possible, to allow for a multitude of genres and domains, while still being fast enough to handle the demands of the domain. In addition, while speed is of course an important factor, the development of good heuristics is an often time-consuming task, and as such we will accept that an implementation could run faster with use of state-of-the-art heuristics.

As the size of computer game worlds vary a great deal, we will aim to provide a planner whose performance scales well with the size of the world. However, we will most likely see a significant performance hit in receiving percepts and logical reasoning for large worlds. As knowledge representation and effective handling of perceptions is not a focus of this project, we accept bad performance on these operations.

Specifically the world-pace requirement can be hard to overcome as planning is a computationally hard problem, with the exact computational complexity depending on the search technique in use. Even state-of-the-art research planners often spent at least a few seconds on formulating plans[12] for complex goals. However, we can from the above properties note that there seems to be a correlation between pace of the game, and complexity of plans, i.e. informally: *It seems that fast-paced games, require less complex problems to be solved - while slow-paced games can often provide problems with a higher level of complexity.*

This makes sense on an intuitive level: imagine a slow-paced game where the player should perform non-complex actions such as clicking a specific point on the screen (similar to shooting an enemy in an FPS): sounds rather boring. Now imagine a fast-paced game, where you in the span of seconds should solve complex problems involving many parameters and possibly conflicting goals (similar to making a plan in a SIM): this seems too hard to be enjoyable.

This relaxes the problem of the complexity of various planning techniques, as situations where the planner will need to come up with plans quickly or suffer consequences (such as death), it will *usually* be rather simple plans it needs to come up with – and thus computationally easier to generate.

Likewise, when faced with the need for solving complex problems it will *usually* also have more time to do so. Of course neither of these are always true.

Based on the properties of the above game genres, we can state that our planner should be able to handle a planning domain with the following properties[13]. Note that these properties are now used to describe the planning domain, not the more loosely-defined terms above where we described computer game genres.

- **Partially observable:** The domain is partially observable, because the agent in the domain maintains its own representation of the world, which can potentially be incorrect. Explicitly, there may exist objects in the world domain which the agent cannot perceive, either from their nature of their position and such only a part of the surrounding world will be represented in the agents belief base.
- **Non-deterministic:** The domain can include objects with pseudo-randomised behaviour in reaction to agent actions. The effect of this is that the state of the world is not fully dependent on the earlier state of the world. As such, the domain is non-deterministic. As the agents action schemas cannot fully describe these preconditions and/or effects, it is <u>unbounded</u> non-deterministic. As such, an agent must necessarily be able to make new plans when actions have unexpected effects that affects the completeness of a plan.
- **Finite:** The world is always of finite size.
- **Dynamic:** The domain is dynamic as changes in the environment can occur from other events than agent actions. This also points to the necessity of being able to make new plans when unexpected (because the agent did not itself initiate them) actions change the world in a way that ruins the completeness of a plan.
- **Navigational:** The domain contains inherent dimensions, in this case two with each object having a set position <u>and</u> the agent acting between and on said objects. Thus routefinding will be a component of all but the simplest plans. This is in contrast to planning domains such as 'Blocks World' where there are no inherent routefinding present.

2.3. THE DOMAIN

Based on the criteria of the problems we wish to explore, we state a domain which represents a relaxed version of a typical action-shooter or RTS game world (without unit production, resources etc). It provides a domain with the properties stated in the earlier section. It also gives the user power to change the environment at run-time, to give dynamic and unpredictable behaviour. In this section this domain will be defined formally. It is the intention that this planning domain is so like a relaxed computer game domain, that our finds here could, with extension in other areas, be used to base a real computer game AI on.

Note that since the domain resembles a relaxed shooter or RTS game world (in contrast to a SIM), it is not overly complex. Due to the low complexity, we choose instead to focus on the fast-pace requirement for these games, which means that the agent must react to changes in close to real time to be considered satisfactory. Therefore our goal is that the agent will usually spend less than 1 second planning, monitoring plans and executing actions.

Note that this section concerns itself only with the planning domain and problem of the computer game. It is assumed that an implementation of such a planning domain will be a self-contained implementation running in conjunction with some framework, a game engine for instance.

2.3.1. OBJECTS

A world in this domain consists of a two-dimensional grid of initially free cells. A cell must be free before an agent can move over it. On each cell can be placed one or more object(s). Unless otherwise noted an object does not make a cell not-free and is fully visible to the agent. Certain types of objects are called items, which mean they are possible to pick up and place. The available objects and their graphical presentation are given here:

⬭ **The Agent**: There is a single agent persisting in worlds of this domain. Since it does not occupy a cell, so a theoretical second agent could move through it. The agent is described in more detail in section 0 The Agent.

🟨 **Wall**: Walls are non-movable objects, which make the cell they occupy not free. Thus they limit the free space for the agent to move over and are the main obstacle of a world of this domain.

⬬ **Bomb**: Bombs are classified as items. Generally agent missions will be to place a number of bombs on a number of goals.

🟩 **Box**: Boxes are non-movable objects, which make the cell they occupy not free. However, they can be destroyed by an agent, using the smash action. Destroyed boxes are removed.

🟦 **Goal-objects**: Goal-objects represent cells on which bombs must be placed as part of an agent's mission.

⬛ **Oil puddle**: Oil puddles cannot be perceived by the agent in any way. When an agent enters a cell with an oil puddle, it will "slip" and be placed on any adjacent square.

2.3.2. GOALS

Goals will usually be to place bombs on the goal-objects, as mentioned above. However, we do not limit ourselves to this. A goal can be any single ground logical fact to be obtained (see section 2.3.4 Beliefs for definition of facts) for instance: *free([0,0])* meaning "free position 0,0", *!box(b)* meaning "destroy box 'b'" or *at(a, [1,1])* meaning "place object 'a' at 1,1".

The belief base (see above for reference) we use, do support that we can formulate goals with conjunctions as the STRIPS[13] definition includes. However, we have implemented it via a goal queue instead. Each goal literal is handled separately and the conjunctions are then implied between each member of the goal queue. Effectively, the agent plans for each goal literal independently. For instance, a game where agent 007 had the goal of planting bomb a,b & c at specified positions and then move to the escape tunnel at 20,20 would look like this:

at(a, [5,5])
at(b, [10,10])
at(c, [15,15])
agentAt(007, [20,20])

In effect this means that there is a specified priority or ordering in which to fulfil the goal literals in.

2.3.3. ACTIONS

The domain provides the following action schematics, described in a notation similar to STRIPS[14]. However, we have informally extended STRIPS with support for equality and basic arithmetic. This has some logical repercussions, which will be handled in 2.3.4 Beliefs.

All these action schematics are considered to be primitive[15]. When executed, the framework in which the agent persists - i.e. the game engine, will return whether the action succeeded or not and update any affected parts of the world.

```
Action(moveAtomic(Agent, MoveDirection),
        PRECOND: agentAt(Agent, CurrentPosition) ∧ free(MovePosition)
        ∧ neighbour(CurrentPosition, MovePosition, MoveDirection),
        EFFECT: agentAt(Agent, MovePosition) ∧ ¬agentAt(Agent, CurrentPosition) )
Action(pickUp(Agent, Item)
        PRECOND: ¬carries(Agent, Any) ∧ agentAt(Agent, ItemPosition) ∧ at(Item,
        ItemPosition) ∧ item(Item)),
        EFFECT: ¬at(Object, Item) ∧ carries(Agent, Item))
Action(place(Agent, Item)
        PRECOND: agentAt(Agent, AgentPosition) ∧ carries(Agent, Item),
        EFFECT: at(Object, AgentPosition) ∧ ¬carries(Agent, Item))
Action(smash(Agent, Box)
        PRECOND: box(Box) ∧ agentAt(Agent, AgentPosition) ∧ f(AgentPosition) ∧
        neighbour(AgentPosition, BoxPosition, AnyDirection) ∧ at(Box, BoxPosition),
        EFFECT: f(BoxPosition) ∧ ¬at(Box, BoxPosition) ∧ ¬box(Box))
```

Note that in the report text, we will usually omit the agent id when referring to an action, as examples will contain only one agent. So for instance *moveAtomic(1, s)* would be referred to simply as *moveAtomic(s)*.

The Agent

To better analyse what planning techniques works best for planning in the specified domain, we will briefly describe our approach in agent design as this can have massive effect on the performance of various search techniques.

The agent maintains a belief base, which will be described in the next section, and on start-up the agent receives a queue of goals it will aim to fulfil. The agent is then built from the following loop, described in pseudocode, where getPercepts(), findPlan and monitorPlan() refer to some appropriate perception method, planning technique and plan monitoring operation respectively. We will further delve into the planning- and plan monitoring technique in the rest of the project.

Agent Loop

```
done = false
goals = ($G_0$, ... $G_n$) //a queue of goals
plan = empty
currentGoal = null
while !done do
        beliefs = getPercepts()
        // if we have no goal, get a new one
        if(currentGoal == null)
                currentGoal = goals.pop()

        if(plan.isEmpty()) // if the plan has been completed, get a new goal
                if(goals.isEmpty())
                        done = true
                        break
                else
                        currentGoal = goals.pop()
                        plan = empty

        // continue with executing the plan
        // check if the plan is broken or invalid
        valid = monitorPlan(currentGoal, beliefs, plan)
        if(plan == empty OR !valid)
                // replan
                plan = findPlan(currentGoal, beliefs)

                if(plan == empty) // if no plan could be found to solve the goal
                        // then skip that goal
                        currentGoal = null
        else
                // execute next action
                succeeded = takeAction(plan.next())
                if(!succeeded) // if execution of the action failed
                        plan = empty // replan next iteration
```

This loop handles the basic functionality that an agent in our domain needs be able to handle, mentioned in order of appearance in the pseudocode:

- **Perception**: The agent can percept its surroundings and maintains a base of current believed facts.
- **Goal iteration**: The agent can iterate through the queue of goals, effectively treating it as a priority queue describing the order to fulfil goals in. Goals are thus handled completely independently, though each goal can contain subgoals which might have couplings. For instance, placing a bomb requires picking it up and moving – picking up the bomb might have a negative interaction on earlier completed goals or future goals.
  Note that in this project, there is no earlier deliberation attempting to find the optimal order of goals. When all goals have been either fulfilled or abandoned, the agent leaves the loop and terminates.
- **Replanning**: The agent can replan when a plan broken or invalid, or when performing an action fails (for instance, trying to pick up a bomb that isn't there).
- **Goal abandonment**: The agent can abandon a goal if it is unsolvable in the current state. Note that the agent never retries earlier abandoned goals – but this could be simply implemented by changing the goals from a queue to a list where a goal is only removed when it is fulfilled and the agent loops over the goals till the list is empty.

2.3.4. BELIEFS

The beliefs of the agent will be represented by a Belief Base, containing believed facts of the world. It is important to note that the beliefs of the agent may be incomplete, as will be the case when the world contains oil puddles, which are invisible to the agent. Together with the dynamic property of the domain, this is the main reason the agent needs execution monitoring and ability to reform plans when something unexpected happens.

Beliefs are stated in a subset of first-order logic, by one of the following:

- Axioms: Axioms can be either action-axioms stating when actions are applicable or axioms governing general rules of the world. An axiom is represented by a Horn Clause.
- Facts: Facts are beliefs about the world, for instance the position of the agent. Facts are represented by ground horn clauses with empty (always true) bodies.

The subset of first-order logic used here is extended with equality and arithmetic operators. As we use a logic with function symbols with an arity of at least 2, logical inference in the domain is made undecidable[16] and hence the planning problem is also made undecidable. To handle logical inference we use a Prolog implementation called jTrolog[17] a performance-oriented fork of tuProlog[18] (tuProlog is much better documented and therefore referenced). We rely on prologs inbuilt handling of forcing termination, such as memory-limit on resolution procedures, as can be found in the documentation of tuProlog.

The use of Prolog for belief representation means that length of clauses can have a performance effect on logical inference, and thus we have made some abbreviations from the formal definition in the implementation, these should be self-explanatory.

The belief base itself consists of a conjunction of beliefs. For instance a small world could look like this with conjunction between beliefs omitted, to follow the style of prolog syntax:

```
// ACTION-AXIOMS
MOVEATOMIC(AGENT, MOVEDIRECTION)←
                (AGENTAT(AGENT, CURRENTPOSITION)∧ NEIGHBOUR(CURRENTPOSITION, MOVEPOSITION, MOVEDIRECTION)
                ∧ FREE(MOVEPOSITION))
PICKUP(AGENT, ITEMPOSITION, ITEM) ←
                (¬CARRIES(AGENT, _)∧ AGENTAT(AGENT, ITEMPOSITION) ∧ AT(ITEM, ITEMPOSITION) ∧ ITEM(ITEM))
PLACE(AGENT, AGENTPOSITION, ITEM) ←
                (AGENTAT(AGENT, AGENTPOSITION) ∧ CARRIES(AGENT, ITEM))
SMASH(AGENT, BOX) ←
                BOX(BOX) ∧AGENTAT(AGENT, AGENTPOSITION) ∧ F(AGENTPOSITION) ∧NEIGHBOUR(AGENTPOSITION,
                BOXPOSITION, ANYDIRECTION) ∧ AT(BOX, BOXPOSITION)

// GENERAL AXIOMS, HERE DESCRIBING WHEN TWO CELLS ARE NEIGHBOURS.
NEIGHBOUR([X1, Y1], [X2, Y2], N)←
                (Y1 = Y2 + 1  ∧ X1 = X2) ∧ ¬WALL([X1, Y1]) ∧ ¬WALL([X2, Y2])
NEIGHBOUR([X1, Y1], [X2, Y2], S)←
                (Y1 = Y2  - 1  ∧ X2 = X1) ∧ ¬WALL([X1, Y1]) ∧ ¬WALL([X2, Y2])
NEIGHBOUR([X1, Y1], [X2, Y2], E)←
                (X1 = X2  - 1 ∧ Y1 = Y2) ∧ ¬WALL([X1, Y1]) ∧ ¬WALL([X2, Y2])
NEIGHBOUR([X1, Y1], [X2, Y2], W)←
                (X1 = X2 + 1  ∧ Y1 = Y2) ∧ ¬WALL([X1, Y1]) ∧ ¬WALL([X2, Y2])
// FACTS ABOUT THE WORLD
WALL([0, 0])        // THERE IS A WALL AT [0,0]
WALL ([1, 0])
WALL ([2, 0])
WALL ([0, 1])
WALL ([2, 1])
WALL ([0, 2])
WALL ([2, 2])
WALL ([1, 3])
FREE([1, 1])        // POSITION [1,1] IS FREE
FREE([1, 2])
```

The prolog implementation we use has no notion of types. As such, we need to let the 'game engine' ensure that each non-wall object has a unique id – as prolog cannot express this logically. As an example we have the facts item(ID) & box(ID), if any of these were assigned the same id the logical inference could not be guaranteed to be sound – an agent given the goal of placing a bomb with id 'a', encountering a box with id 'a' risk planning to place the box instead – which will fail and that goal will be abandoned.

To solve this, we need to make the game engine handle assignments of IDs to objects so that no two objects will ever have the same ID. In our interface we give the user the option of specifying id's manually for test purposes, but this gives the user the opportunity to crash the planner and this functionality should be handled in a true implementation.

Because everything is expressed in Horn Clauses we cannot take full use of the expressiveness of first-order logic. This gives us some complications, which we will need to handle separately. Chiefly we cannot express that an agent can only be at one position at a time – formally:

$$agentAt(1, A) \land agentAt(1, B) \land (\mathrm{A} = \mathrm{B}) \ \vdash \ \emptyset$$

Conflicts of this type will have to be handled separately where needed by introducing special cases directly when used, instead of relying solely on the logical reasoning power of Prolog. As the focus of this project is not the logical properties and expressiveness of knowledge/belief representation, we will not delve further into this.

The prolog we use, is based on SLD resolution and as SLD resolutions time complexity is polynomial in the number of literals / facts in the world[19], it is clear that the performance of logical inference in this belief-base will depend on the size of the world.

3. Problem Theory

In this chapter we will provide some theoretical solution proposals to the problems present in our planning domain. The solutions we choose to implement will be benchmarked in section 6 & 7. In the following we define a few concepts in advance:

Plan	A general term, comprising both totally-ordered and partially-ordered plans. Generally a goal and a set of actions - usually some ordering constraints applied to actions.
Plan - Broken	A plan is not broken for a given state S, if for all actions *i* in the plan, the preconditions hold in S with effects of all earlier actions applied. Formally: $\left(S \wedge Action_0.effects \wedge \dots \wedge Action_{(i-1)}.effects\right) \vdash Action_i.preconditions$ If for the action *i* the above do not hold, the plan is said to be **broken at *i***.
Plan - Valid	A plan is valid for at given state if its sequence of actions fulfils the goal of the plan when performed from the given state. Vice versa it is **invalid** if not.
Plan - Complete	A plan is complete if it valid and not broken.
Action	An action is an operation supplied by the environment which the agent has some awareness of. It has **preconditions** and **effects**, stating when the action can be performed and the effects of that action respectively.
Action - Primitive	A primitive action is that does not need to be further decomposed, and it can thus be performed directly in the world environment.
Action – Hierarchical	A hierarchical action is one that cannot itself be performed in the environment, but can be decomposed to a set of other hierarchical action or primitive actions through some **decomposition scheme**. These actions are also called **non-primitive** or **high-level** actions.

3.1. Planning in Dynamic And Nondeterministic Environments

It is clear that a classical planner will not be able to solve the problems of our specified domain, due to the inherent dynamism and non-determinism. To handle this we introduce **replanning**, which signifies the procedure of either repairing a plan or plan anew from scratch.

This section will concern itself with the theoretical solutions to the problems of planning in dynamic and nondeterministic environments, specifically realising when something goes wrong – the search techniques of finding or repairing plans will be handled later on, in chapter 4 .

3.1.1. *Plan- and Execution Monitoring*

An agent in a dynamic and non-deterministic domain can be subject to a number of events, which ruins a plan:

1. **An action fails**
   Execution of an action may fail, for instance picking up an item that is not available. Situations like this can arise due to unbounded nondeterminism (if there is a chance the item slips), incorrect or incomplete beliefs about the world (if the agent or the item is not where it believed) or the dynamic nature of the world (if a wall suddenly appears in front of the agent, after the world has been perceived).
2. **The plan is no longer complete**
   The plan is broken or invalid. Perhaps an object appeared and broke the plan, or the agent is not where it believed it was - for instance, because an oil puddle made it slip.
3. **The plan is no longer the best**
   There might exists alternative plans that fulfil the goal in less steps than the current plan – for instance because a wall was moved and opened a shorter route to the goal.

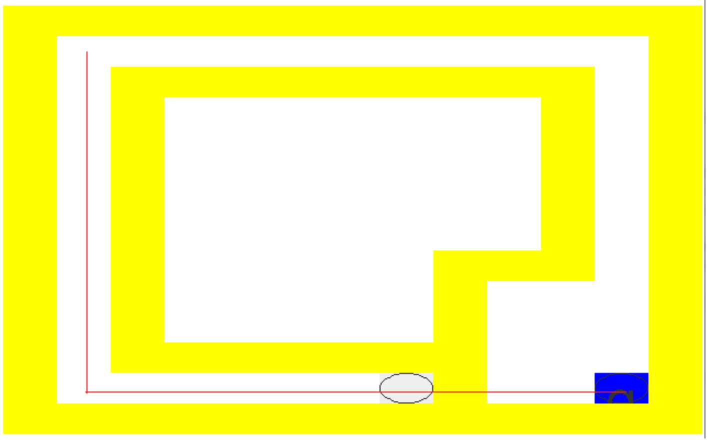
An agent that plans from scratch each time it has taken an action will be able to handle these problems. However, as planning can be a potentially time-consuming, this approach is obviously not desirable. A more efficient approach is to perform execution monitoring, and only replan when the situation validates it. There are two main approaches: action monitoring which handles the first case and plan monitoring[20] which can handle the next two. We will not attempt to cover situations where the plan is no longer the best.

3.1.1.1. Action monitoring

Action monitoring introduces a check before performing an action. This check examines the agent's percepts, to see if all preconditions of the action to be performed are still satisfied. If the preconditions are no longer satisfied, a replan should be performed.

In itself action monitoring is a weak form of execution monitoring. It has the limitation that a fault in the plan will only be discovered just before the now-invalid action will be performed, which can potentially be very ineffective, as per the example to the right.

However, action monitoring is much cheaper to perform than plan monitoring and can thus be useful for domains where situations like the one to the right will rarely occur. Due to the navigational nature of our domain, we can expect that problems like this would arise and pure action monitoring alone would thus not be efficient.



Action Monitoring*: A wall appears and blocks the plan (red). The agent ends up in a dead end because it only checks action validity for the next action to be performed.*

In addition this form of action monitoring does not handle partial observability, as the belief base of the agent might wrongly state that the action is applicable. However, another form of action monitoring can be useful. Instead of performing a check before performing an action, the agent can perform a check after the action was performed to see if it was successful. In our domain we can take advantage of the surrounding framework which will return a value indicating if the action succeeded or not, rather than checking the expected effects against the actual effects of the action.

Action monitoring of this kind can be useful also when used together with plan monitoring, which should otherwise make certain that actions are only taken as part of a valid and complete plan. The reason for this is because it is computationally fast to check whether an action succeeded, at least in our framework[ii]. In our case, an agent can save running time by checking whether an action succeeded and trigger a replan if it did not, rather than check the whole (potentially long) plan for validity on the next iteration of the agent control loop. For instance, an agent checking realising that a *moveAtomic* action failed, can replan then and there, rather than check plan to realise that it is not in the correct position and the plans thus invalid.

3.1.1.2. Plan Monitoring

Plan monitoring is a procedure for checking whether a given total-order plan is still valid, in the perceived environment. The procedure, as we have derived it from theory, can be expressed as pseudocode:

```
                    Plan Monitoring Procedure
plan = a list of Actions
goals = a conjunction of goal predicates
beliefs = the current beliefs of the agent
for i = 0 to plan.length()
        valid = true
        openPreconditions = empty set
        action = plan.get(i)
        for each precondition of action
                valid = valid && Beliefs.isTrue(precondition)
                if(!Beliefs.isTrue(precondition)) then
                        openPreconditions.add(precondition)
        if(!valid)
                return (PLAN_BROKEN_AT + i, openPreconditions)
        for each effect of action
                beliefs.apply(effect)

if(beliefs.isTrue(goals))
        return (PLAN_VALID,empty set)
else
        return (PLAN_NOT_VALID,empty set)
```

[ii]But perhaps not in all instances: a real-world robot or an agent in a computer game with strict limits on how to perceive the world. In these cases checking whether an action had the desired effect can potentially require (sensing) actions to be taken, which can again involve planning.

As can be seen from the pseudocode, the procedure in this form returns a pair (Info, Open Preconditions) representing:

- **Plan broken at i**: This means that the plan is broken at action number i in the plan, with the returned set of unfulfilled preconditions. It might be possible to replan.
- **Plan Valid**: This means that the plan is executable and fulfils the goals. There is no reason to replan.
- **Plan invalid**: This means that the plan is executable, but the goal is not fulfilled by it. For instance, because a goal was moved or because the agent is not where it expected itself to be. It might be possible to replan.

Note that the plan monitoring procedure only returns the first broken action, if any, on a plan and it will thus ignore subsequent broken actions. The reason for this is the navigational nature of our domain, which means that often plan reparation can include a new route to be taken. By only taking the first broken action into account, we can spare our planner from potentially repairing broken areas of the plan which might be circumvented anyway by a new route.

3.1.1.3. Limited plan Monitoring

As running plan monitoring on long plans can be potentially slow (see chapter 7 for a performance example) it might be worthwhile to limit plan monitoring in some way. One could limit it to check only the next 10 actions or only plan monitor every few steps instead of each step – these values could even be dynamic and based on the number of changes the agent perceives in its environment. We have not implemented any of these limitations, but we believe that an implementation for a game should take this into account. The real issue is to strike a balance between simple action monitoring and plan monitoring. The balance chosen and the ways to enforce that balance will depend greatly on the specific game in question.

3.1.2. PLAN REPAIR

When a plan becomes broken or is rendered invalid, the agent can attempt to repair the plan. The objective of plan repair is to change a plan so it again becomes complete, and we would like to do this more effectively than simply replanning from scratch. This last part is important as sometimes replanning from scratch is more effective, as shall be shown in the next section.

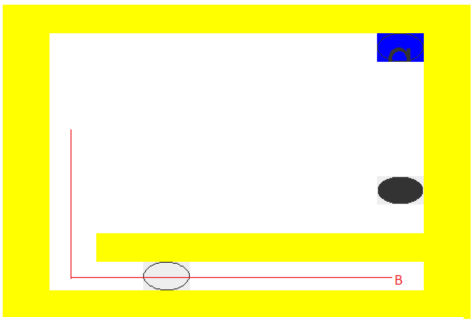
Plan repair can be expressed in general terms as a combination of two operations:

- Introducing new actions to make the plan complete again (similar to planning)
- Removing actions that obstruct the completeness of the plan

The specific method of doing plan repair varies widely on the planning domain and the search technique used for planning. Some approaches try to repair broken 'areas' of the plan by treating the entry and the exit actions/states as initial and goal in a local planning problem. Other approaches introduce some domain-specific plan-modification operators to dynamically repair the plan[21]. In chapter 4 we delve into three specific planning techniques and for two of these describe an explicit method of plan repair.

3.1.3. *Replanning From Scratch versus Plan repair*

It is intuitively clear that in some situations plan repair can potentially be very effective and save the agent from a lot of computation. But in fact it has been shown[22] that it is not possible to achieve *in general* a provable efficiency gain of plan repair over replanning from scratch. This makes sense, because as can be seen from the situation to the right there exists situations where planning from scratch would have been more effective than repairing a plan.



Ineffective Replanning*: The bomb has just moved from the position marked b to a new one. In this situation, plan reparation would require abandoning all the moveAtomic action leading into the tunnel, and then introducing new actions to reach the bomb.*

A replanning from scratch would introduce the same actions as the plan repair, but without the need to abandon actions. As such replanning from scratch would be most effective in this situation.

To use these tools effectively we would need a method to gauge if a given situation validates replanning from scratch or to try to repair the plan. It is obvious that a complete answer can only be given by computing both solution and checking which is fastest or returns the shortest plan – but it is obvious that this approach is not satisfactory, as the goal was to gauge *beforehand* which method was most efficient.

It seems probable that if a plan is *not* broken but invalid, it will be most effective to plan from scratch as we have no immediate way of knowing in what way to repair the plan. However, when a plan is broken or a performed action fails, we have specific information about the action on the plan that cannot be performed, and thus know what to repair. This hints that a good guideline could be to replan from scratch when the plan is invalid and attempt to repair a plan when it is broken. However, we have not been able to conclusively prove this theory.

It may be possible to design heuristics to guide this choice. Inspiration could be had from the area of heuristics gauging the length of a plan for a given goal - but this is outside the scope of this project. As such we can only conclude that without decision-guiding heuristics it is most prudent to analyse the domain in question and conclude whether to replan from scratch, use plan repair or a combination of both.

3.2. Routefinding With Hierarchical actions

As we consider only navigational domains, navigation will be a part of all but the simplest plans. As routefinding, requires a lot of steps – and therefore expansions and search[iii], it can be prudent to disengage routefinding from the planning process and tackle this problem in a hierarchical manner. Particularly we can use our knowledge of A* routefinding to implement an efficient solution outside the planner, which can then be called by the planner using a hierarchical action.

This is of course a step away from generic planning solutions, since it involves use of specific actions and belief facts. But since we concern ourselves only with navigational domains and stand to receive great gain in running time, we believe this is an acceptable step away from generic solutions.

[iii] For some empirical data on the performance of our prolog-based belief base, see section 6 Performance of the FSP planner

To do this, we introduce a non-primitive *move* action:

```
Action(move(Agent, CurrentPosition, MovePosition),
                PRECOND: agentAt(Agent, CurrentPosition) ∧ free(MovePosition),
                EFFECT: agentAt(Agent, MovePosition) ∧ ¬agentAt(Agent, CurrentPosition) )
```

This action does not include any precondition stating that the agent must be a neighbour to the cell it wishes to move to. As such, during planning the agent assumes that it can move to any free cells in the world, without planning for the specifics of getting there.

We therefore also apply some extended version of the least commitment strategy[23], as we decide that the planner should *not commit to a specific route before it has to*. The reason for this is that in a dynamic environment, committing to a route many steps before it becomes relevant can easily land the agent in a situation where it needs to replan because of far-flung effects – some of which may be irrelevant when the agent actually gets to that point in the plan or be easily circumvented by another route. Therefore we choose to only decompose the *move* action when it is the next action to be performed from the current plan. We call this **execution commitment**.

The step of decomposing the move action is a planning problem in itself, which introduces *moveAtomic* and *smash* actions only. However, it is a problem that is very specific and can take advantage of the heuristics developed for routefinding, usually Euclidean or Manhattan distance, which are very efficient for this sub-problem. We can also take advantage of more specific data structures, useful for routefinding instead of relying on very generic and expressive, but slower, datastructures like Prolog clauses.

Formally we can describe the decompositions of the *move* action as follows:

Decompose(move(Agent, CurrentPosition, MovePosition), Beliefs) =

$\{A_0, A_1, \ldots A_n\}$

Where A_i *=*

(moveAtomic(Agent, MoveDirection) $\vee$ *smash(Agent, Box))*

And

$$\left(\sum_{i=0}^{n} = Beliefs.apply(A_i.effects)\right) \vdash agentAt(MovePosition)$$

$$\wedge$$

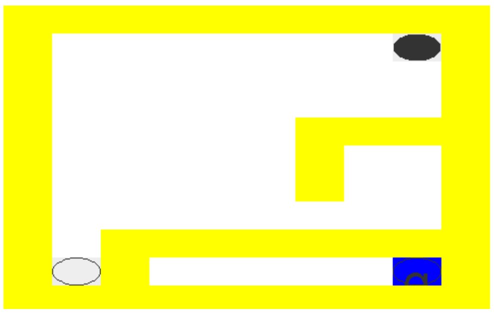
$$\left(\sum_{i=0}^{n} = Beliefs.apply(A_i.effects)\right) \nvdash agentAt(CurrentPosition)$$

And

$$CurrentPosition \neq MovePosition$$

)

Introducing execution commitment does however introduce some complications, namely with correctness of plans. In fact as long as a plan includes a hierarchical action we cannot guarantee plan completeness for a given goal – there may be obstacles later in the plan which the hierarchical action cannot overcome, as shown by counterexample on the example to the right. We call this **hierarchical incompleteness**.


Hierarchical incompleteness 1

In the shown example the agent will find the following plan:

MOVE([10,6]) $\rightarrow$ *PICKUP(Q)* $\rightarrow$ *MOVE([10,14])* $\rightarrow$ *PLACE(Q)*

The agent will only realise that the goal is not within reach when attempting the last move action. Only then will it realise that the goal is unsolvable. In the domain we have specified, this can only be because the agent has no means of reaching the goal – because only walls can block a cell and be indestructible. So in our domain it is acceptable to abandon the goal, as the agent has been given a (currently) impossible goal. But, this might not be true for all domains as there might be some way for the agent to force its way to the enclosed goal which is not handled by the decomposition procedure of the high-level *move* action.

Also note that the other case might be true. Perhaps the situation started as shown on the example to the right, but during execution of *move([10,6])* or *pickUp(q)* a wall object could be moved and allow access to the goal. In this situation using a strategy of stronger commitment, would have arrived at the conclusion that the goal was impossible to fulfil, even though it would become possible during execution.

It could also be the case that the area with the goal is opened/closed with some specific or pseudorandom intervals without the agent knowing about it. In this case the execution monitoring strategy would worry about not being able to get in when it became relevant, while the stronger commitment would check if the plan was complete now and assume that it would not change. This could lead to both 1) executing a plan which will fail later on or 2) abandoning a goal which would be possible later on.

We can therefore conclude that the strategy of execution commitment has the downside that the agent might execute plans before it realises them to be incomplete. In our domain this is not great downside and as can be seen from the examples above, since sometimes this strategy will turn out to fulfil goals which were initially impossible. In dynamic environments it might indeed be best to use action monitoring, as the agent cannot necessarily infer about the future state of the world with any degree of certainty. Of course, in domains where plan completeness is very important, this strategy will not be sufficient. In the domain we have specified we believe that partly executing impossible plans will, per the examples above, loosely even out in terms of effective versus ineffective behaviour.

3.3. REFLEXES

In dynamic environments, especially ones where there is danger for the agent, implementing reflexes[24] can be a viable way of ensuring survivability of the agent. Reflexes are essentially the ability to insert actions immediately without any planning when the situation critically validates one type of action. For instance, in a shooter game it would make sense to implement a reflex which makes the agent immediately throw itself to the side if it perceives that a grenade has landed near it.

The tricky part of mixing reflexive agents with planning agents it how to get back on the plan, after using a reflex which changes the world in such a way that a plan is no longer valid. For instance, the agent has just thrown itself to the side to avoid the grenade: therefore it is now in another position than specified by the plan. So, how does the agent get back to executing the plan without replanning from scratch? One solution could be to have 'counter reflexes', i.e. implement an action which negates the effect of the reflex (such as moving back to the former position after the grenade has blown). Another approach is to treat it as similar to repairing the first action in a given plan, and as such we believe finds in this area will be relevant for implementing reflexes also.

We have decided not to implement reflexes in this project, as we wish to focus on planning agents. Reflexes would however be a key ingredient in a game AI to ensure survival – letting the planner handle the more advanced decisions.

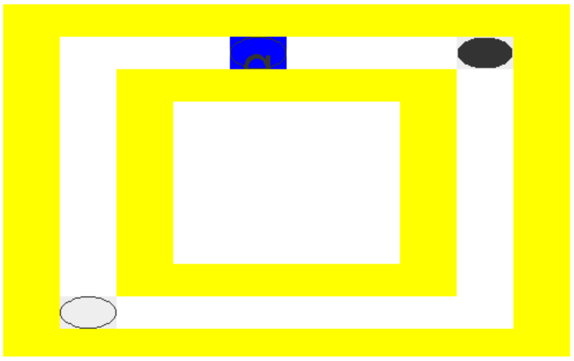
4. Planning Techniques

In this chapter we analyse a few different planning techniques and their applicability with regard to planning and replanning in our specified domain. The hope is to make an informed choice about which planning techniques to look further into. For each planning technique we will describe the various ways we would need to handle them to implement them for our domain.

4.1. Forward-State Space Planning

Forward-state space planning (FSP), also called progression planning, is in its simplest form the most straight forward way of handling planning, and it has algorithmic ties to many other problems such as graph traversal and search. The search builds a search tree, where nodes represent complete states and edges represent actions. The search traverses the state-space from the initial node until it discovers a node which fulfils the goal of the search. A thorough explanation of FSP can be found in literature[25,26].

Without using the hierarchical move action introduced in 3.2 Routefinding With Hierarchical actions, the branching factor for the search tree of a forward state-space search in our domain will be maximally 5 (4 moveAtomic actions and 1 smash, pickup or place action). However, as walls can easily be plentiful and other objects very rare, the average branching factor is more likely ~ 3, since there will rarely be boxes to smash, bombs to pick up or place, but often a wall blocking at least one direction for movement.


State Space Search example: *A rather simple planning problem, yet even here uninformed state-space search breaks down completely.*

Consider the example to the right, where the agent must find the bomb, pick it up and place the bomb on the goal. This result in a plan with a length of 20, which would mean that the agent would potentially need to examine of 3^{20} states[27], which is obviously impossible within a short timeframe to fulfil the wish for real-time reactions in our domain.

If we use hierarchical move actions, we have a much larger branching-factor (as the agent believes it can effectively 'teleport' to any free cell) but much shorter plans. The time spent decomposing the hierarchical move actions, is for simplicity assumed to be negligible, as decomposition takes place during execution. The branching factor would be equal to the number of free cells in the world, in the given example ~50. However, the plan needed is only 4 long (move, pickup, move, place). This would mean the agent could potentially need to examine 50^4 states, which is significantly lower than 3^{20}. However, it is still too much to analyse within a short timeframe.

4.1.1. *Heuristics*

There have been developed techniques for guiding the search effectively and forward-state space planning is therefore currently considered one of the potentially most efficient planning techniques[28,29]. With good heuristics both of the example searches above could potentially become fast enough to be used. Some forward-state space planners take advantage of the construction of a Planning Graph[30] for heuristic estimation. The use of a planning graph does however have some limitations, which will be described in section 4.2.

Other planners use a heuristic estimate called the empty-delete-list[31,32] which relaxes the problem by removing negative effects of actions and take advantage of the subgoal independence assumption[26], which assumes that subgoals does not have interactions. Computing the empty-delete-list heuristic for a given state necessitates running a simple planning algorithm in this relaxed problem. It returns an estimate of the number of actions left to fulfil the goal, but this should be worthwhile when comparing with the gain in lessened search space size[26].

If using hierarchical move actions then the branching factors becomes equal to the number of free cells, as established above. As the branching factor for the example above was 50, it means the planner would need to solve 50 relaxed planning problems to expand a node, and this is for a relatively simple problem. This branching factor might kill the search simply from calculating the heuristic estimate for each state – but on the other hand the heuristic and hierarchical move action might result in only needing to expand a few nodes before finding the goal. We have not been able to prove whether or not this would in practice be worthwhile.

If not using hierarchical routefinding the empty-delete-list heuristic estimate is very unsatisfactory. This is because the empty-delete-list heuristic has no concept that when moving from A to D, B and C are closer to D than A is. The result is that *moveAtomic* actions will have no effect on the heuristic value of a state unless they result in the agent being at the goal position. In effect, unless the planner generating the heuristic value is built with a concept of position, the heuristic will be useless for guiding the navigation of the agent.

However, since we only consider navigational domains it would be acceptable to extend the empty-delete-list heuristic with some concept of euclidean- or manhattan-distance from the A* search algorithm. This leaves us with the problem of how to extract the goal-position from the goal, which is not trivial. For some goals such as "place bomb at [10,10]" we could extract the position ([10,10]) directly from the goal. However, goals such as "destroy box A" require the agent to realise the goal-position should be the position of box A – but this is a planning problem in itself as it must check its action-schemas to realise that to destroy a box it must be at the same position as the box.

The problem of which goal position to use is also relevant when handling goals which have prerequisites which are sub-goals in their own right. For instance, planting a bomb requires that the agent picks up the bomb first. This means that a generic heuristic using some form of manhattan-distance as base, would need to first use the bombs position as goal and later in the same search the desired location for the bomb for calculating manhattan-distance. We could specify a domain-specific heuristic handling these complexities, but we desire a domain-independent solution and we know of no way to specify this in a domain-independent way.

Using domain-independent heuristics and some plan repair method could make forward-state space search a candidate for a planning technique in our domain. Even though we possess no such heuristic, we will make further tests to explore how a state-space search algorithm would fare, if we were in possession of a good heuristic estimate. These are presented in chapter 6 Performance of the FSP planner. The lack of a proper heuristic makes our implemented FSP useless, as it can only handle the smallest planning problems within any reasonable timeframe. As such we have implemented no plan-repair procedure for it. We will however describe a theoretical approach to plan-repair in FSP below.

4.1.2. *Plan Repair*

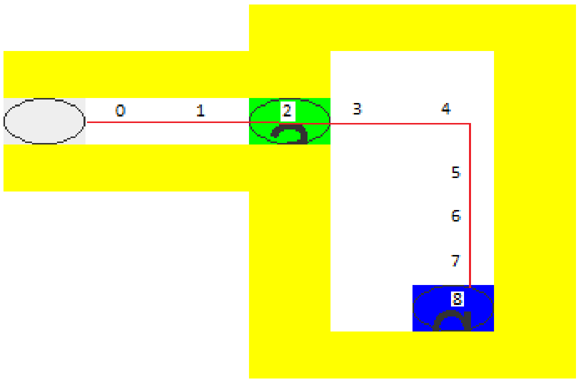
As specified in the earlier chapter, we would like to use plan repair techniques to make replanning faster. We will here describe how a plan repair procedure could be designing for FSP. We would like to have a plan-repair procedure which can use our already existing techniques, as to stick with the same basic methodology and avoid writing too much code. As such, it would be ideal if a plan-repair procedure for FSP could be formulated as a new FSP planning problem. This can be done[33] and we will give an example of reparation below. Recall from section 3.1.2 that there were two operations to use when repairing a plan: introducing actions and removing actions. This plan repair procedure uses both of these.

4.1.2.1. Example Plan-Repair

Consider the example world below. The agent, which utilises plan monitoring, receives that its plan is broken at action 2.

The agent can now repair the plan, by dropping the part of the plan before action 4[iv]. The actions 4-8 are now called the *continuation*$_{complete}$. The agent can now formulate a new planning problem with:

initial state : current state
goal state: any state that fulfils all the preconditions of any action τ on *continuation*$_{complete}$ OR the original goal state.



FSP Agent Plan Repair Example 1
A box appears and breaks the plan (red)

The plan returned by this new planning problem is called *repair*. The procedure then removes any action from *continuation*$_{complete}$ that is before τ with the resulting plan *continuation*$_{rest}$ or if the original goal state was reached, the whole *continuation*$_{complete}$ is dropped and *continuation*$_{rest}$ rendered empty.

A new complete plan is found as the concatenation of *repair* and *continuation*$_{rest}$.

4.1.2.2. Issues

There are some issues or properties about this form of plan reparation, which should be taken into account when designing such a system.

Invalid plans: The plan repair procedure referenced above is able to repair plans when a plan is broken – it cannot do anything particularly intelligent about invalid plans. When a plan is rendered invalid instead of broken, the best strategy for this procedure is simply to replan from scratch.

[iv] As action 4 is the next action which still have its preconditions fulfilled.

Goal state: Note that the plan repair procedure initiates a search with multiple goal states, formally:

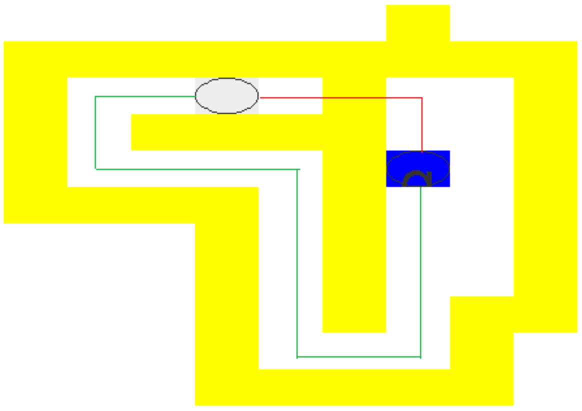
$$Goal = (A_0.precondition_0 \wedge \ldots \wedge A_0.precondition_i) \vee \ldots \vee (A_n.precondition_0 \wedge \ldots \wedge A_n.precondition_i) \vee originalGoal$$

Where A_i is an action on the plan.

This is not ordinary for a FSP search and as such an ordinary FSP will have to be extended to accept disjunctions in goals and handle accordingly. This is a rather small change in itself, but it leads to the next issue.

Heuristic Estimate: If planning for multiple possible goal states, which heuristic estimate should the search use? It might be possible to calculate the heuristic estimate for all goal states, and then selecting the lowest – to aim for the 'closest' part of the plan in terms of actions. However, intuitively this seems very expensive in terms of computations. Also, consider the example to the right.

In this example a wall has just appeared in the agent's path, breaking its plan. In this situation the original goal is actually closer to the agent in terms of actions, than any action on the old plan. The agent cannot detect this without planning, so it cannot simply drop the whole plan and start from scratch.

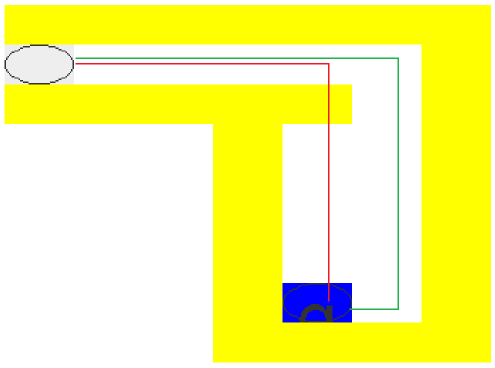


FSP Agent Plan Repair Example 2
The new plan(green) does not come into contact with the old one (red)

To handle these situations also, the best strategy seems to keep calculating the heuristic estimate on the old goal. This means that the agent must be extended with the notion of a **main goal** which it bases heuristic estimates on, and **secondary goals** which it hopes to reach simply because it knows the plan from there on.

With this strategy we hope the agent will reach the continuation-part of the old plan before it reaches the goal, but we cannot guarantee it. In fact, sadly sometimes the agent can find a new complete plan from scratch even though a possibility for hitting the continuation-part exists. Consider the example to the right.

In this example the planner might find the green plan instead, when trying to repair. However, even though this is a complete plan in itself, the planner could have saved time by moving west and hitting the old plan. This suggests that it might be suitable to let the old plan have *some* influence on heuristic. We have however not been able to determine with any certainty how and to what degree this should be introduced, especially since it should be a domain-independent solution.



FSP Agent Plan Repair Example 3
A wall has appeared on the red plan

These issues show us that plan repair can be a complicated to introduce and the gain is not always certain. Recall from section 3.1.3 that it has been shown that no gain is ensured from repairing plans – and that in some situations it can be more time consuming to repair a plan than replan from scratch.

4.2. GRAPHPLAN

GraphPlan is an algorithm constructing planning graphs and extracting plans from them. As mentioned under forward-state space search above, planning graphs can also be used for heuristic estimates and these are in fact very accurate[34]. A thorough explanation of the workings of the planning graph and GraphPlan algorithm can be found in literature[35].

GraphPlan at first glance seems tempting to use for this project, as it seems easy to replan with. When reason for replanning or repairing a plan arises, the agent can use the already-constructed planning graph to simply extract a new plan to the goal.

Planning graphs do however have some constraints on the domains and planning problems they can be applied to. First of all it requires a propositional description of the world (where we use a subset of first order logic through prolog[v]) to represent belief about the world. This means that in order to construct a planning graph, the algorithm will need to propositionalise the logical representation of the current state and action schemas. We will therefore need to analyse the size increase when propositionalising a state representation:

Given a world of length m and width n with ω non-wall or non-box objects, α axioms with A possible instantiations per cell and βactions with B possible instantiations per cell, we define the size of a world representation as the total number of facts and rules it contains. As there is either one *free*, *wall* or *box* at each cell the size of the first-order representation, Σ_p of the given world is

$$\Sigma_\mathrm{p} = m * n + \omega + \alpha + \beta$$

However, a propositional representation of the world will include the ground instances of all axioms and actions. The size depends on the content of the domain. We will cover two extremes; both examples stated in our domain and are not cases in general.

1. Worlds filled with walls will have no available actions and no instances of axioms as the neighbour relation (the only axiom) does not apply to cells with walls on. Its size will therefore be:

$$\Sigma_1 = m * n$$

2. Worlds without any walls at all will in broad terms have A relations per cell and B actions applicable per cell. Formally:

$$\Sigma_2 \approx m * n + (m * n) * \mathrm{A} + (m * n) * \mathrm{B} + \omega$$

Note that this formula does not take into account cell on the edge of the world, which will have fewer move actions and neighbour relations, hence ≈.

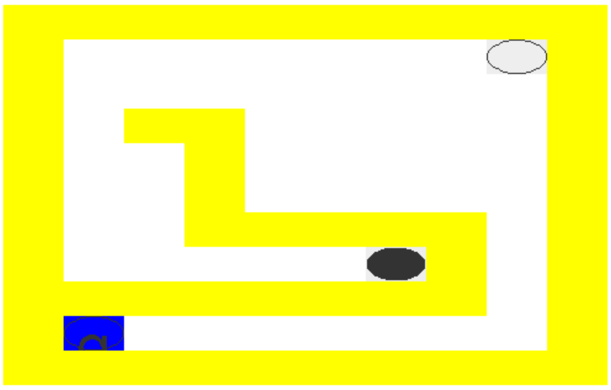
[v] See section 2.3.4 Beliefs

It is clear that example number 1 is simply a special case of 2, where A = B = ω = 0. We can then establish the difference an upper bound over the difference a world representation and its propositional representation Σ_{Diff} as:

$$\Sigma_{\text{Diff}} \leq \Sigma_2 - \Sigma_p = (m * n) * A + (m * n) * B + \omega - \alpha - \beta$$

Given the above, we see that the size of the world representation can potentially become very large indeed. A practical example is shown to the right.

This example world has half the cells with walls and the others are free[vi]. When represented in prolog, it would have a size of $\Sigma_1 = 138$[vii]. When propositionalised it would have a size of $\Sigma_2 = 511$[vii], little less than $\frac{1}{3}$ of what Σ_2 above suggests would be the maximum size[viii].


Example of explosion in size of propositional representation

So propositionalisation leads to about a tripling in the size of the representation for this example. This is a rather small world (10x11). According to the establish formulas above, a large open world, such as 100x100 would have a size of maximally 150.000 facts and rules. If this world had about the same ratio of walls/free cell as the example above, and a similar distribution of them (as location of walls have an effect on axioms and action instantiations), then the final size of the state would be $\approx \frac{1}{3} * 150.000 = 50.000$. For some understanding of the performance of the prolog implementation we use, see chapter 6 Performance of the FSP planner. This should give a rough idea of how prolog will react to handling such massive representations. We could design a specific data structure for the planning graph, but it would be beyond the scope of this project to also make it able to handle logical inference without parsing the structure to prolog anyway.

Now recall that this is simply for propositionalising the initial state of the world. The time complexity of construction a planning graph is low-order polynomial in the number of actions and literals[36,37]. It stands to reason then that the number of literals is simply too big for a polynomial-complexity algorithm to handle. One of the major known disadvantages of GraphPlan is that large worlds and/or many available actions result in a computational blowup[38] of the search.

[vi] There are 55 walls in a world of 10x11 = 110 cells. Found by counting
[vii] Found by manual counting
[viii] $10 * 11 + (10 * 11) * 4 + (10 * 11) * 10 + 3 = 1653$, as A = 4 neighbour relations and B = 4 move actions + 4 smash actions + 1 pickup + 1 place action, per cell.

4.2.1. *Hierarchical actions*

With regards to hierarchical actions, earlier work has shown that GraphPlan have problems in general with using hierarchical actions[39] and for our specific hierarchical action, introducing it would mean an massive explosion of literals, as the B value mentioned above would suddenly become equal to the amount of free cells in the world – even for small examples this would quickly mean millions of literals in the initial state of the planning graph.

4.2.2. *Dynamic Environment*

One could argue that it might be worthwhile to build this planning graph anyway when starting the agent, and then maintain it throughout the runtime of the agent – simply extracting a plan with GraphPlan when dealing with new goals. However, the dynamic nature of the domain means that walls can be moved, deleted or introduced at any time by a human player. Even though GraphPlan represents worlds possible from the initial state, it only represents states *reachable* through the actions of the agent.

This means GraphPlan will have to either: 1) build the planning graph from scratch each time a wall is moved. This is costly in time as we have just realized above that the planning graph for worlds in this domain will be large, or 2) expand our notion of planning graphs to represent all possible configurations of walls.

This would require introducing further actions (not performable by the agent) for the movement of walls – in effect making the agent aware that walls could potentially at any time move to any other square or disappear. However, this further lead to an extreme explosion of state representation size as each level of the planning graph would include every possible assignment of wall/free cells possible: there are 2^{m*n} such assignments.

As such, we can see no way of implementing GraphPlan in dynamic environments without violating our desire of a scalable solution that can handle environmental changes in close to real-time.

4.3. Partial-Order Planning

We will also cover the implications of using a partial-order planner (POP) for this project. Strictly speaking any planning procedure which can place two actions into a plan without specifying a ordering between them is a POP[40], regardless of the search space. However, POP is here identified with a more specific POP search-procedure which searches in the space of partially-instantiated *plans*, instead of states. A partial-ordered plan is similar to a classic topological ordering[41], extended with the notion of causal links. A detailed specification and relevant definitions can be found in relevant literature[42,43] which we have based our implementation of the *refine-plan* procedure on[44].

However, as mentioned under forward-state space planning, partial-order planning is not currently holding the lead in international planning competitions. Moreover, navigation is strictly total-ordered, as you cannot change the order of two non-identical primitive move actions and still hope to end up at the correct destination – so maintaining a partial-ordering can seem like overkill in this case.

Studying the POP procedure referenced above, it is easy to realise that the branching factor of the search tree when trying to close an open precondition O, is equal to the number of ground actions that contain O in their effects. For navigation there are between 0 and 4 ways of entering a cell (one from each direction). We therefore arrive at the same problem of guiding the search as we did with FSP –

navigation would be next to impossible without some heuristic to guide the search and we are not in possession of a proper domain-independent heuristic estimate that can also handle navigation. In fact, plan-space is not finite[45] which means POP navigation would further require a heuristic to be worthwhile at is may not even discover the correct route through expanding the whole search-space.

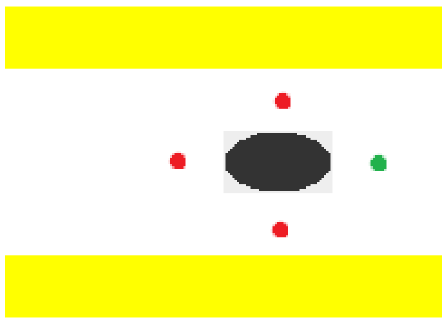
However, there are two major non-heuristic performance enhancements which we have found can be applied to POP: Action design limitation and the hierarchical move action mentioned earlier. These two could potentially make POP a viable option, as we shall see in the following two sections:

4.3.1. Action design limitation

Action design limitation is introduced, as the process of designing action schemas to limit needless branching factor in the plan-search. To illustrate, imagine a *push* action where the agent must be at a neighbouring cell to the bomb and the bomb is pushed to a neighbouring cell[46]. Consider the example below.

The agent wishes to move the bomb to the cell marked with a green spot. As it can do so from any of the cells marked with red, this means that there are three actions fulfilling the goal:

- Stand north of the bomb, push east
- Stand west of the bomb, push east
- Stand south of the bomb, push east


Action limitation example

This is three actions essentially fulfilling the same goal: moving the bomb to the east. By designing the action schematics such that only one ground action fulfils the goal, we can drastically limit the branching factor of a POP search space as well as the logical reasoning needed to instantiate ground instances of action schemas. This strategy is an extension of the idea behind the most-constrained-variable heuristic[47]: that the open preconditions with the fewest-possible actions fulfilling them are best to close first – as they ensure lower branching and thus a slimmer/more linear search tree. A formal definition:

```
Definition: limited action schemas
An action schema A that has been designed such that, for every instantiated effect ε there
is logically only one ground instance α that fulfills ε, is called a limited action
(schematic).
An action where this is not true is called an unlimited action (schematic).
```

Note that limiting an action is *per action schema*, not per possible effect. We might have both the *place* action, but also a *placeAndHide,* and a *placeAndArm* which share effects (place a bomb) but with other side-effects - such as hiding the bomb or arming it.

We have already limited the *pickUp* and *place* actions (see section 2.3.3 Actions), by specifying their preconditions such that the agent must be at the same position as the bomb/where the bomb goes, which for these two actions limits the possible instances to one grounded action per bomb per cell.

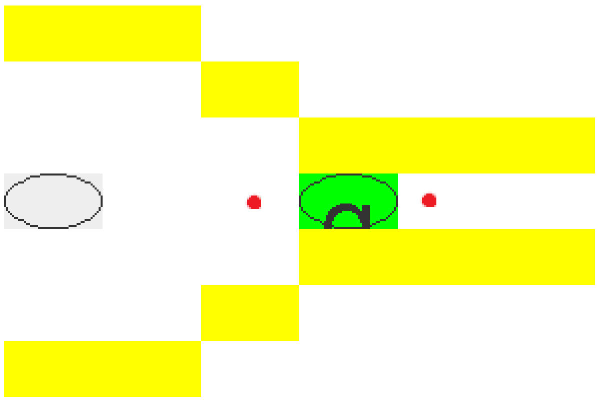
As we shall see in chapter 7 Performance of the POP Planner, using limited actions makes POP planning much more useful, as there is only one applicable grounded action per open precondition. This results in a branching factor of 1 for most open preconditions, which means that POP almost[ix] approaches a complexity that is linear in terms of the length of the plan and the number of actions.

In computer games the game designer should be able to design both the action schematics as limited actions and ensure that the game engine is able to handle this. Note that some action schematics might not be able to be limited – throwing a grenade will for instance likely always require some choice between several likely locations to throw from.

4.3.2. HIERARCHICAL MOVE ACTION

Above we mentioned that using the hierarchical move actions in POP can enhance the performance of the planner. This is because 1) the hierarchical move action is in fact a limited action: With the primitive move actions, the act of moving from A to B could be fulfilled with a several combinations of different *moveAtomic* actions, which complicates the search a lot as a lot of backtracking can occur. 2) because we have no heuristic for guiding the navigational search in the POP planner and 3) Because routefinding can be done much faster in the domain-specific implementation than the planner is able to. The performance of the routefinding algorithm is described in chapter 5.

However, recall from section 3.2 Routefinding With Hierarchical actions, that introducing hierarchical actions makes us unable to guarantee if a plan fulfils its goal. When using a backward oriented search, such as POP this introduces a second problem. An example is provided to the right.


POP and hierarchical incompleteness

In this example the goal of the agent is to make the cell occupied by the box free. The only action with effects fulfilling this precondition is to use *smash*. However, there are two possible ground *smash* actions fulfilling the precondition – one executed from each of the cells marked with red.

Now assume that the agent chooses the one on the far side of the box. In regular POP planning the agent would then try to introduce actions closing the precondition of being on that cell – if it could handle the navigational aspect. It would then realise that no route exists and therefore backtrack and select the other cell as origin of the *smash*.

However, a planner using hierarchical actions would simply introduce the hierarchical *move* action to the far cell and then terminate as there are no more open preconditions – as the move action would only be decomposed during execution. It is clear though that the plan is not complete, as the agent needs to move through a box.

Note that this problem would not arise if the action schema for *smash* was a limited action as described in section 4.3.1. However, we cannot guarantee that limiting action schemas solves this

[ix]Though not entirely, due to the computational complexity of logical reasoning, when finding open preconditions and ground instances of actions in a first-order environment

problem *in general*. Secondly, we cannot guarantee that all action schemas can be limited, so merely using limited actions is not sufficient for a domain-independent solution.

We have not been able to deduce a proper domain-independent way of handling these types of problems. One solution which could be keyed to hierarchical move actions could be to introduce a *accessible(Agent, Position)* fact, stating what positions are immediately accessible to the agent and this fact could then be a precondition for the hierarchical move action. This is similar to existing solutions where agents hold a list of nearby positions which are in cover[48]. How this list of *accessible*-facts is produced is a question of percepts and there might exist efficient[x] solutions. We have however not implemented this.

4.3.3. Plan Repair

The very flexible way of planning by refinement on closing open preconditions, make this planning technique obvious for use in replanning. Combined with action- and plan monitoring, a simple and elegant way of repairing plans appear: When action- or plan monitoring returns that a plan is broken, it receives the preconditions that were not met[xi] as specified in section 3.1.1.2. The missing preconditions could then be added to the existing plan as new open preconditions and we could refine it until the plan is again complete or POP returns that the plan is impossible. Given a POP refinement procedure *refine-plan,* we can then easily specify a basic plan-repair procedure as follows:

```
                          POP Plan Repair
pop = a POP plan
failedAction = an Action // The action that couldn’t be performed, as returned by plan
                         // monitoring
openPreconditions = (p1, ... pn) // The set of facts that were not fulfilled for
                                 // failedAction, as returned by plan monitoring

for i = 0 to openPreconditions.length()
            pop.addOpenPrecondition(openPreconditions(i))

return refine-plan(pop)
```

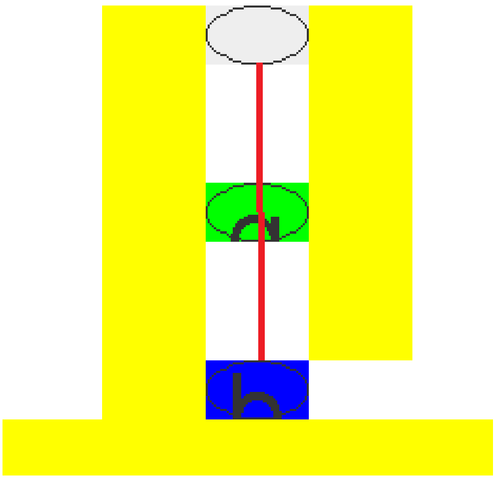
This procedure uses our already establish *refine-plan* procedure to repair the plan when new precondition opens, which is an elegant way of doing it, compared to implementing a special case for repairing plans. Using such a dynamic and elegant plan-repair procedure also makes it easier to implement other functionalities such as ignoring actions that have already failed once in a given situation[49].

However, this simple procedure does have a complication, which should be handled by extending the procedure appropriately. This will be described below.

[x] To avoid running an A* routefinding algorithm from every cell in the world to the agents position, every single time a percepts are received by the agent.
[xi] If any – due to partial observability the agent might now be able to detect the missing precondition.

4.3.3.1. State explicitness

One complication of using a POP planner is that we lose any notion of the explicit intermediate states of the search[50], so we cannot guarantee precisely how the state of the world is at any given point through the plan. During formulation of the state this is no problem as when the procedure introduce an action, it checks whether its preconditions are fulfilled by the agents current state and if not add it as an open precondition. However, when closing a newly-introduced open precondition for a broken action with a hierarchical action, we cannot guarantee that we can truly determine if a precondition is fulfilled or not. Consider the example to the right.



***POP plan repair example 1**: The plan of the agent (red) is broken as a box appears.*

Plan monitoring returns that the agent *moveAtomic(s)* is not possible as *free([2,3])* is not fulfilled. Therefore *free([2,3])* is introduced in the plan as an open precondition for the action. The planner will quickly realise that to make the cell free, it must use the *smash* action and that closes the open precondition.

However, *smash* has a precondition as well which is not fulfilled: in this case *agentAt([2,2])* as the agent must be on a neighbouring cell to the box. This precondition is not true in the current state so *move([2,2])* is added to the plan to fulfil it. We see here a situation where plan-repair conflicts with hierarchical move actions – as the *agentAt([2,2])* is actually already fulfilled in the plan by a number of primitive *moveAtomic* actions. However, we cannot immediately devise an efficient way of realising that the earlier *moveAtomic* actions is in fact the *move([2,2])* action decompositioned. We need to extend plan-repair in a way to handle this problem.

1) An obvious and simply solution could be to simply iterate through all actions that are partially-ordered earlier than the broken action and check if any contains *agentAt([2,2])* as an effect (which we will find in this case). However, this only works if the fact is a ground fact and the effect of a single action. If we imagine a precondition relying on an axiom fulfilled by several actions, this is not sufficient.

2) Another obvious way to handle this problem is to find all actions that are partially-ordered to come before the broken action and apply their effects to the current state. This will then build an explicit state representation, effectively building a notion of the explicit intermediate state and thereby solving the problem, as we can check the preconditions of the newly introduced *smash* action in this new state instead. However, as we shall see in chapter 6, with our prolog-based belief representation, copying a state and applying effects to it is computationally heavy. We would therefore like to consider an alternative if possible. However, the strategy of introducing a notion of explicit intermediate states might be a potent one given a more effective belief base.

3) A third option could be to simply remove all actions that are partially-ordered before the broken action from the plan, with all their corresponding orderings and causal links. We can then reopen all preconditions these actions closed, and refine anew. In the above example this

would mean removing the *moveAtomic* actions leading to the box, introducing the *smash* action and then refine with *agentAt([2,2])* as the only open precondition. Sometimes this will be faster than solution 2 above but other times – for instance with long plans or high-performance belief bases – it could easily be slower.

These different approaches all have their value under different circumstances. We can conclude that one should analyse ones domain and belief base performance, to make an informed choice about these approaches. We have gone with option 3 above, which sadly can make our plan-repair slower than could otherwise be achieved.

4.4. Conclusion

Above we have analysed three different search techniques in regard to planning/replanning in our domain. Of these three we have decided to further implement a Forward-State Space Planning (FSP) and a Partial Order Planning (POP).

The reason to implement FSP is because it is one of the main planning methods and currently viewed as one of the fastest planning techniques[24,25]. As such we believe it mandates further research, to understand what would be needed to make it fast enough to fulfil our runtime requirements. We will therefore perform some practical benchmarks to gain an understanding of the practical runtime of FSP with our chosen belief representation and domain. These can be found in chapter 7 Performance of the FSP planner.

We have chosen to implement POP because of the obvious use for plan repair and because our analysis showed that it could become potentially very fast with the right means, namely action design limiting and hierarchical move actions.

5. Performance of Hierarchical Routefinding

The hierarchical routefinding is implemented using the well-known A* algorithm which we assume the reader knows and the heuristic estimate used is manhattan-distance. We will in this chapter perform some benchmarks on the A* algorithm, to get a feel for how it will affect the running time of the planner.

5.1. A* PERFORMANCE

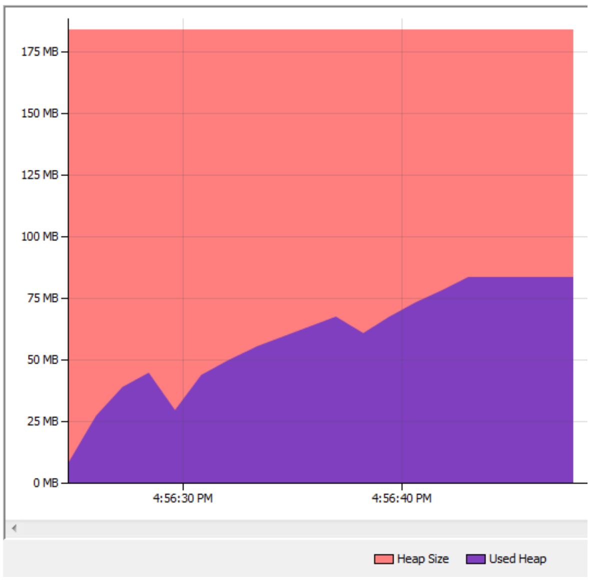
Our A* implementation is done by maintaining a HashSet of longs. We can easily convert between longs and coordinates with simple bit operations, which is a very effective way of finding the neighbours of a cell, i.e. expanding it. If you compare the speed of our A* implementation with the speed of expanding states in the Forward State Planner in the next chapter, it is easy to see the merit performance wise of handling routefinding separately.

The following table with benchmarks are for an agent standing in one corner of the world, receiving a move action to move to the diametrically opposite corner. The worlds are empty of obstacles, which of course makes the search easier. However, the worlds here used are much larger than any we have tried our planner on, so we believe the size of the large worlds here make up for the loss of complexity from 'real' world examples.

Size of World	States expanded	Route Length	Time:
10x10	52	19	1 ms
20x20	112	39	1 ms
40x40	232	79	2 ms
80x80	472	159	5 ms
160x160	952	319	13 ms
320x320	1912	639	15 ms
640x640	3832	1279	36 ms
1280x1280	7672	2559	96 ms

From the data in the table above we can see that even for the largest worlds, the routefinding search is still well within a permissible timeframe, which serves as an indication that the routefinding could handle complex routes – such as labyrinths – in smaller worlds. We can therefore conclude that even for larger worlds than we would otherwise test in, the decomposition of hierarchical move actions is well below our 1-second limit for real-time execution.

To the right is a graph showing the memory footprint of an A* search in a 1000x1000 world. As can be seen the memory use of the search (the purple area) is not very large. As these resources will quickly be released again when the search is over, we believe that the routefinding procedure will not take critical resources away from other critical areas of a game such as graphics or the planner.



Memory Usage of A*

6. Performance of the FSP planner

In this section we will benchmark the FSP planner as we have implemented it. As we have been unable to design or obtain a satisfactory heuristic, this performance benchmark will deal with the performance of the part of the planner which we can execute in an acceptable timeframe, which sadly extends to expansions of states.

As an FSP planner works by searching through all possible states, the procedure of expanding a state and compute its child-states is essential to the runtime of the planner. The procedure of doing this includes cloning the state to be expanded, and then applying the effects of the action leading to the child state. The prolog implementation we use is not designed for use as a belief base and is not designed to be easily copied / cloned, as it works by parsing strings to java-objects which is a rather costly affair computation-wise.

We have performed a test to examine how long it takes to expand a state to get its children states in a 20x20 world.

Time Spent (ms)	31	16	32	16	31	15	47	31	**219**
# Actions	3	2	2	3	2	2	3	3	**20**
ms / Action	10.33	8	16	5.33	15.5	7.5	15.66	10.33	**10.95**

So averagely the FSP planner spends **10.95 ms** expanding each action, when producing child states. This time is spent on cloning the current state and applying the effects of an action to the clone. Recalling the finds in the theory on FSP, section 4.1, we can attempt to gauge how FSP would fare in the example world presented to the right.

6.1.1. *With primitive move actions*

Distance-based generic Heuristic

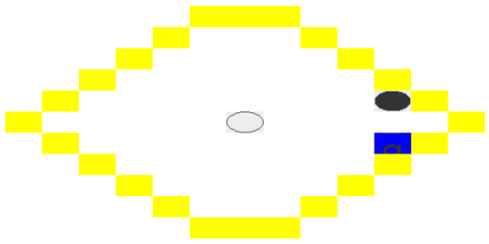
A theoretical heuristic combining the benefit from using a distance-based heuristic such as manhattan distance and a effect-based heuristic, such as empty-delete list

Steps in plan	8
Average # applicable actions / step	4
Number of states to expand	4*8 = 32
Gauged running time	$32 * 10.95\,ms = 350\,ms = \mathbf{0.35\,s}$

No heuristic

A pure non-informed

Steps in plan	8
Average # applicable actions / step	4
Number of states to expand	4^8 = 65536
Gauged running time	$65536 * 10.95\,ms = 717619.2\,ms = \mathbf{717.6\,s}$


FSP Performance Example

6.1.2. *With Hierarchical Move Actions*

Distance-based generic Heuristic	
A theoretical heuristic combining the benefit from using a distance-based heuristic such as manhattan distance and a effect-based heuristic, such as empty-delete list	
Steps in plan	4
Average # applicable actions / step	59
Number of states to expand	59*4 = 236
Gauged running time	$236 * 10.95\ ms = 2584.2\ ms = \mathbf{2.5\ s}$
No heuristic	
A pure non-informed	
Steps in plan	4
Average # applicable actions / step	59
Number of states to expand	59^4 = 12117361
Gauged running time	$4096 * 10.95\ ms = 132685102.95\ ms = \mathbf{132685.1\ s}$

As can be seen from the examples, a search without a guiding heuristic estimate is not applicable with our time demands, as was already theoretically established in section 4.1. This is true for both a search using primitive move actions and hierarchical move actions. In fact, contrary to the example shown in section 4.1 we see hierarchical move actions giving such a high branching factor, that it is even slower than uninformed search with primitive actions. However, we can now see an example of how much gain we would actually receive if we had a proper domain-independent heuristic estimate containing both elements of navigational heuristics and effect-based heuristics.

However, we also see that the greatest reason the lengthy search is not the branching factor in itself, but the time it takes to expand a state. This is for a simple example with a rather short plan, and a low number of applicable actions in each state. If we imagine a real action game, there might be many more actions – to throw grenades, use first aid kit, reload etc – it becomes clear that even with guiding heuristics the expansion time per action and the heightened number of applicable actions would render the search method unusable.

6.1.3. *Conclusion on Results*

Using a prolog-based belief representation for state-space-search is not viable for a real-time domain, as it simply doesn't scale well with the needed length of the plan and the number of applicable actions. We do not posses a heuristic estimate that is domain-independent and can handle reasoning about goals and position. Therefore we must conclude that for our domain, belief base representation and demands for real-time planning and execution, Forward-State Space search is unsuitable.

7. Performance of the POP Planner

We will in this section run some performance tests of the POP Planner. As can be seen in the former chapter, the running time for the A* search – and therefore the decomposition of the hierarchical move action - is rather fast. In this chapter we will analyse how the POP planner handles finding the plan itself– before decomposing the hierarchical move action.

The planner here uses:

- Hierarchical Move actions with Execution commitment
- Action Monitoring
- Plan monitoring
- Plan repair of broken plans, handles hierarchical move actions by dropping earlier parts of the plan. Replans from scratch on invalid plans.

We wish to know primarily how the planner scales with size of the world, and number of available actions. As such we have performed three sets of benchmarks, with varying number of actions. The original four action schemas are the *move, pickUp, place* and *smash*. The other scenarios have been created by copying these action schemas 5 times & 10 times, with new names. Each set of tests contain various world sizes.

The following performance benchmarks have been performed on a computer with the following hardware: Core i7-2675QM 4x2.20GHz, 8GB ram.

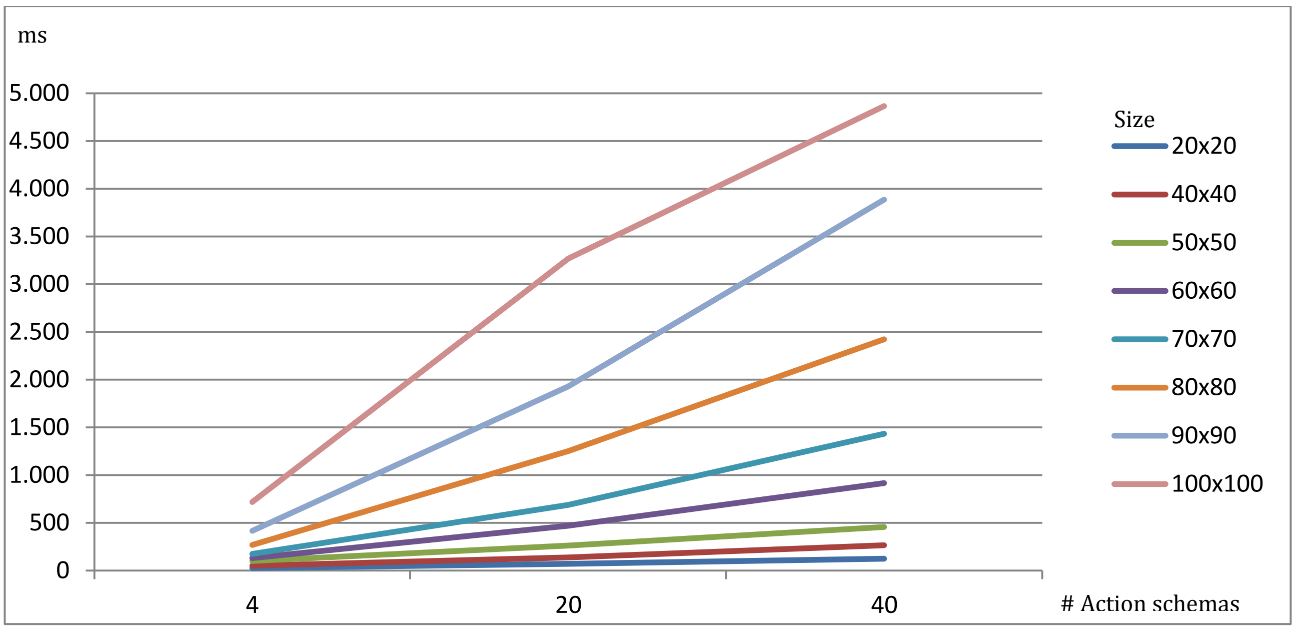
We have focussed on making the performance tests short and precise, so it will not be extensive tests shown here, such as functionality test or

7.1. POP planning

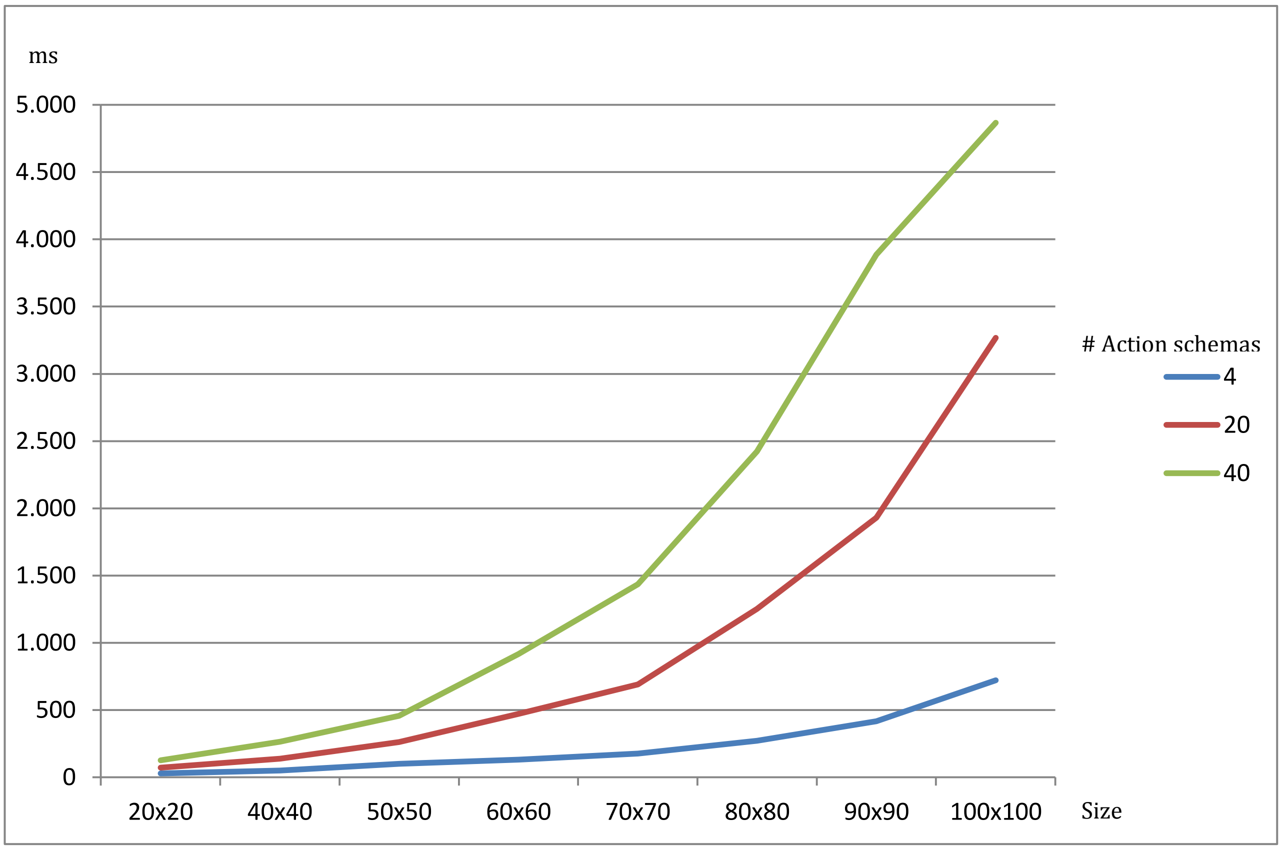
The following is the times spent on formulating a plan in the world. The plan is finding a bomb and placing it on the correct position. The agent starts in one corner of the world together with the goal and the bomb is placed in the diagonally opposite corner. All times are in milliseconds.

Size / Milliseconds	**4 actions Schemas**	**20 actions Schemas**	**40 actions Schemas**
20x20	28.0	72.2	125.0
40x40	49.9	138.8	264.5
50x50	100.2	261.7	457.3
60x60	131.6	472.0	915.6
70x70	176.0	689.8	1434.7
80x80	270.3	1252.9	2422.7
90x90	416.0	1930.8	3887.3
100x100	720.4	3268.6	4866.5

The data in the tables above, shown as a curve. Here as a function of number of actions:



The same data, as a function of size of the world:



7.1.1. RESULTS

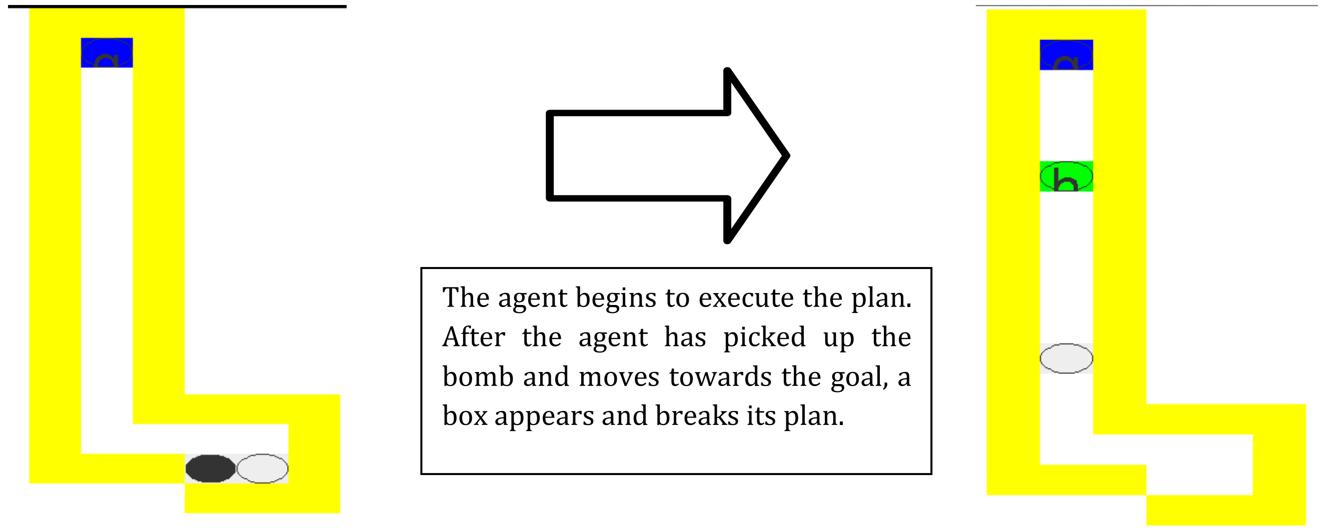
It is clear to see that the performance of the POP planner is dependent on both the size of the world and the number of action schemas. In fact the curves indicate that running time increases linearly with regards to number of actions, but as a polynomial with regards to size of the world. We believe this is due to logical inference - as we have mentioned earlier in section 2.3.4, prologs SLD-resolution gets slower when the believe-base grows. A better performing belief base could limit this problem to some degree.

Backing this conclusion is the fact that the influence of the size of the world, grows with the number of actions. This is because when the planner has more actions, it has more logical reasoning to perform. For instance, when a plan has 5 actions *move1, move 2... move5*, and attempts to close an open precondition *agentAt([0,0])* it receives 5 possible actions that can fulfil this precondition. Each of these will have to have its variables instantiated, to become a ground action, and this is done through logical reasoning. Therefore, giving the agent 5 identical actions to solve the same preconditions, means 5 times as much logical reasoning when instantiating them.

Note that this also means the results here are a worst-case example of cases with 20 or 40 available actions. The same number of actions, but with no effects in common will not slow down the planner as much as this, as the agent will still only need to instantiate one action schema for each precondition.

7.2. Plan repair vs. Replanning from Scratch

To benchmark how plan repair reacts to a typical situation we have measured the performance of the agent in the following situation. We have tested for the various number of actions, as per above. We have only performed one single test of plan repair vs. Replanning from scratch, as our analysis (see section 3.1.3) shows that it is highly situational. As such it is outside the scope of this project to perform enough tests to create a clear picture and only one test have therefore been performed.



The agent begins to execute the plan. After the agent has picked up the bomb and moves towards the goal, a box appears and breaks its plan.

Method / Milliseconds	4 actions Schemas	20 actions Schemas	40 actions Schemas
Repair plan	8.7 ms	29.2 ms	32.0 ms
Replan from scratch	4.7 ms	17.1 ms	20.7 ms

The benchmark above shows an example where replanning from scratch is more efficient than repairing the plan. The reason for this is the abandonment of actions, to fix the problem of inserting hierarchical actions in an inexplicit state (see section 4.3.3.1). However, the planning problem is very simple as the smashing of the box is handled by the hierarchical move action and therefore replanning from scratch is faster in this situation.

This does not mean however, that plan-repair of a POP plan does not have its merits in both this planning domain and more complex domains. If the domain & plan included elements like opening doors, finding correctly coloured keys for the doors etc. reusing large parts of the plan seems much more efficient – however, the simple nature of our domain makes us unable to benchmark this properly.

7.3. Plan monitoring

What cannot be seen in the benchmarks above is that the execution of the plan gets very slow between a world sizes above 40x40. This is due to plan monitoring not scaling well with plan length, as was mentioned in section 3.1.1.2. In fact it takes ~5 seconds to monitor a plan of ~ 160 steps in a 80x80 world. This is not acceptable as our goal is a real-time planner. However in a final implementation this would be tweaked and optimized via the proposals in section 3.1.1.3. Because plan monitoring uses the belief base for cloning the current state and applying effects, a better performing belief base would help relieve this performance problem.

7.4. Conclusion on results

After running these performance benchmarks, we can conclude that the amount of logical reasoning needed, is the main limitation on how fast the planner is able to formulate plans. With few actions even large worlds of 100x100 can be planned in, within our wish for 1 second runtime. With more actions, the effect of the world size becomes much more evident as the amount of logical reasoning rises. For 20 actions and a size of 80x80 the wish for 1 second runtime is violated and from there on the planner becomes increasingly slower.

However, performance of the planner is good for smaller problems and/or a small set of actions. We can therefore conclude that further work is needed, especially a more efficient belief base, before planning of this kind would be viable for running the AI of a real action-shooter game. The POP here benchmarked is introduced without heuristic estimate for guiding the search. Implementing heuristic estimates could also help the planner handle large set of action schemas much more effectively.

8. Conclusion

After having finished this project, we can draw some conclusions from the experiences we have made. We can conclude that game AI presents an interesting challenge as a planning problem, also for theoretical research. We believe that the area holds interesting problems which we have only scratched the surface of – and there is a lot of potential for future work.

We have learned that planning holds potential for use in fast-paced games, much more than we believed when we were first introduced to planning and its time-complexity. Planning in real-time environments can be made possible through action design limitation and creating domain-specific solutions for problems which arise often, such as navigation, which can be used by the agent through hierarchical actions. However, we also learned that using hierarchical actions introduce some complexity not present earlier, namely regarding completeness of plans. It will require future work to establish exactly how to handle this added complexity and incompleteness in a coherent way across multiple types of hierarchical actions and planning problems.

After analysing, implementing and benchmarking both a Forward-State Space planner (FSP) and a Partial-Order Planner (POP) we can conclude that for domains similar to the ones we used in this project, FSP is simply too cumbersome – it is too dependent on the effectiveness of the belief base and currently no (known to us) heuristic is satisfactory for handling both a domain-independent and navigational planning problem within real-time time constraints.

We have used a prolog implementation for basing the belief base on, and the performance benchmarks on both the FSP and POP indicate that the belief base is central to the runtime of the planner. Designing and implementing a more efficient belief base could yield very positive results in this area. Perhaps sacrificing expressiveness and limit the belief base to a strictly propositional representation of the world might be the key, although our analysis of GraphPlan indicates that large worlds are simply too large & cumbersome to represent in propositional logic.

9. References

1 Radha-Krishna Balla & Alan Fern : UCT for Tactical Assault Planning in Real-Time Strategy Game

2 Daily Mail: Modern Warfare 3 hits $1 billion in 16 days - beating Avatar's record by one day. http://www.dailymail.co.uk/sciencetech/article-2073201/Modern-Warfare-3-hits-1-billion-16-days--beating-Avatars-record-day.html

3 Scott, Bob (2002). "The Illusion of Intelligence". In Rabin, Steve. AI Game Programming Wisdom. Charles River Media. pp. 16–20

4 Jeff Orkin, Monolith Productions .Three States and a Plan: The A.I. of F.E.A.R. pp. 1

5Alex J. Champandard: 10 Reasons the Age of Finite State Machines is Over http://aigamedev.com/open/article/fsm-age-is-over/

6 Jarret Raim : Finite State Machines in Games

7 Damian Isla: Handling Complexity in the Halo 2 AI

8 Peter Andreasen: Top 5 reasons not to use AI in computer games, IO Interactive, Copenhagen.

9 Jeff Orkin, Monolith Productions .Three States and a Plan: The A.I. of F.E.A.R. pp. 3,6-10

10 Hei Chan, Alan Fern, Soumya Ray, Nick Wilson & Chris Ventura: Online Planning for Resource Production in Real-Time Strategy Games

11Jeff Orkin, Monolith Productions .Three States and a Plan: The A.I. of F.E.A.R. pp. 3

12 Blai Bonet & Héctor Geffne: Planning as heuristic search. Pp. 11-17

13 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 41

14 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 379

15 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 422

16 Alonzo Church, 1936 & Turing, 1937

17 http://java.net/projects/jtrolog

18 http://alice.unibo.it/xwiki/bin/view/Tuprolog/

19 Paolo Frasconi,Francesca A. Lisi: Inductive Logic Programming: pp. 217

20 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 441-444

21 Sven Koenig & David Furcy & Colin Bauer: Heuristic Search-Based Replanning

22 Bernhard Nebel & Jana Koehler: Plan reuse versus plan generation a theoretical and empirical analysis

23 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 387

24 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 46

25 Malig Ghallab, Dana Nau & Paolo Traverso. Automated Planning : Theory and Practice, 2004. pp. 69

26 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 382

27 Jon Kleingberg & Éva Tardos: Algorithm Design, 2006. pp. 79-83.

28 Malig Ghallab, Dana Nau & Paolo Traverso. Automated Planning : Theory and Practice, 200. pp. 81

29 Subbarao Kambhampati : Back to The Future of Planning, from ACAI Summer School; 10 Jun 2011, Arizona State University, http://rakaposhi.eas.asu.edu/summer-school-11-final.pdf, pp. 60-62

30 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 395

31 Blai Bonet & Héctor Geffne: Planning as heuristic search. Pp. 8-10

32 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 386-387

33 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 442

34 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 398

35 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 395-402

36 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 397

37 Malig Ghallab, Dana Nau & Paolo Traverso. Automated Planning : Theory and Practice, 200. pp. 131

38 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 395

39 Anders Rasmussen & Bo Kanstrup Hansen: Planning in a dynamic environment (in Danish) (Master Thesis at Danish Technical University), pp 94

40 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 388

41 Jon Kleingberg & Éva Tardos: Algorithm Design, 2006. pp. 99-113.

42 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 387-395

43 Malig Ghallab, Dana Nau & Paolo Traverso. Automated Planning : Theory and Practice, 200. pp. 99-100

44 Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 390

45 Malig Ghallab, Dana Nau & Paolo Traverso. Automated Planning : Theory and Practice, 200. pp. 99, 101

46 Thomas Bolander, Andreas Garnæs and Martin Holm Jensen : 02285 AI and MAS - Assignment 3: Mandatory Programming Project (Assignment given at Danish Technical University, spring 2011) pp. 4-5

[47] Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 395
[48] Jeff Orkin, Monolith Productions .Three States and a Plan: The A.I. of F.E.A.R. pp. 14
[49] Jeff Orkin, Monolith Productions .Three States and a Plan: The A.I. of F.E.A.R. pp. 10: 'Dynamic Problem Solving'
[50] Stuart Russell & Peter Norvig. Artificial Intelligence : A Modern Approach, 2nd Edition. pp. 388