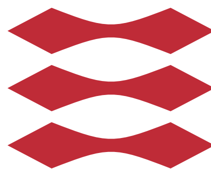
Implementation and Evaluation of Estimation-of-Distribution Algorithms

Nikolaj Nøkkentved Larsen

Supervisor: Carsten Witt, Associate Professor



Kongens Lyngby 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
DK-2800 Kgs. Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

In this master thesis I examine a new type of computer algorithms called Estimation-of-Distribution Algorithms (EDA for short). These algorithms are designed to find the optimal solution to a given problem, and are a type of metaheuristic. They use probabilities in order to determine the optimal solution. The algorithms can be configured in different ways by changing their input parameters which alters their behavior slightly.

I examine the performance of these algorithms by developing a benchmark program designed to test these algorithms on well-known benchmark problems. The program will report the running time for a configured algorithm on a given problem. The different configurations are then analyzed in order to determine which traits the best configuration(s) possess.

The algorithms examined in this thesis are the Univariate Marginal Distribution Algorithm (UMDA), the Compact Genetic Algorithm (cGA), the Bayesian Optimization Algorithm (BOA) and the Max-Min Ant System (MMAS). These algorithms are tested on the well-known benchmark problems OneMax, LeadingOnes, Jump and Trap.

Scientific literature has been published regarding the running time of these algorithms and it is taken into account when the algorithms are examined and are used as initial guidelines for the experiments. This literature does not cover all algorithms and problems, and I then use previous experiments and already collected data as a starting point for the experiments.

The different algorithms have different strengths which causes the best algorithm to vary for the different problems. For OneMax the variance in performance is

relatively low with UMDA and cGA coming out on top. LeadingOnes is a much more difficult problem to solve for the algorithms, requiring many more evaluations on average. BOA uses far less evaluations than the other algorithms because its bayesian network allows it to model the problem perfectly. BOA is also the best performing algorithm in Jump, being able to complete a larger jump than the other algorithms and also requiring fewer evaluations on average. Trap is a difficult problem to asses because its design causes the algorithms to either terminate quickly or possibly not at all, causing a high variance in the running times. BOA and MMAS did seem to come out on top, being able to complete larger problem sizes than UMDA and cGA.

Finally, the boundaries of the project are explored with extra suggested work that could expand the understanding of the algorithms and their performance.

Resumé

I dette speciale undersøger jeg en ny type algoritmer, som hedder Estimering-af-Fordeling Algoritmer (forkortet EDA). Disse algoritmer er designet til at finde den optimale løsning til et givent problem, og er en type af metaheuristikker. De bruger sandsynligheder til at finde den optimale løsning. Disse algoritmer kan blive konfigureret på forskellig vis ved at ændre på deres inputparametre, som ændrer deres opførsel.

Jeg undersøger algoritmernes præstation ved at udvikle et benchmark program. Programmets formål er at teste disse algoritmer på velkendte benchmark problemer. Programmet vil indsamle data om køretiden for den konfigurerede algoritme på problemet. De forskellige konfigurationer bliver senere analyseret for at bestemme, hvilke træk de bedste konfigurationer besidder.

Algoritmerne, der undersøges i dette speciale, er the Univariate Marginal Distribution Algorithm (UMDA), the Compact Genetic Algorithm (cGA), the Bayesian Optimization Algorithm (BOA) og the Max-Min Ant System (MMAS). Disse algoritmer bliver testet på de velkendte benchmark problemer OneMax, LeadingOnes, Jump og Trap.

Der er udgivet videnskabelig litteratur, som beskriver køretiden af nogle af disse algoritmer. Disse artikler bliver inddraget, hvor det er relevant og bliver brugt som udgangspunkt for mine eksperimenter. Denne litteratur dækker ikke alle algoritmer og problemer, og jeg vil da i stedet bruge mine egne eksperimenter og allerede indsamlet data som udgangspunkt for mine undersøgelser.

De forskellige algoritmer har forskellige styrker, hvilket betyder, at én algoritme ikke altid er bedst. For OneMax er variansen i præstation relativt lav med

UMDA og cGA som værende de bedste. LeadingOnes er et meget vanskeligere problem at løse. Der kræves gennemsnitligt ofte mange flere evalueringer. Dog viser det sig, at BOA bruger mange færre evalueringer end de andre. Dette skyldes dens "Bayes-netværk", som tillader at modellere problemet præcist. BOA klarer sig også bedst i Jump. Den kan klare større hop end de andre algoritmer og kræver også færre evalueringer. Trap er et svært problem at beskrive pga. dens design. Alle algoritmer finder enten den optimale løsning i første generation eller næsten aldrig. Dette giver meget stor varians i køretiderne. BOA og MMAS ser dog ud til at være de bedste, fordi de kan klare større problemstørrelse end UMDA og cGA.

Contents

Abstract i

Resumé iii

1 Introduction 1

2 Objectives 3

2.1 Scope . . . 3

3 Background 5

3.1 Evolutionary algorithms . . . 5

3.2 Estimation of distribution Algorithms . . . 6

3.2.1 Univariate Marginal Distribution Algorithm . . . 8

3.2.2 Compact Genetic Algorithm . . . 9

3.2.3 Bayesian Optimization Algorithm . . . 9

3.2.4 Max-Min Ant System - Iteration Best . . . 12

3.3 Problems . . . 14

4 Benchmark program 17

4.1 Modules and overall structure . . . 17

4.1.1 EDA Module . . . 17

4.1.2 Problem Module . . . 18

4.1.3 Graphics Module . . . 18

4.1.4 Calculation Module . . . 20

4.2 Algorithms . . . 20

4.2.1 UMDA . . . 21

4.2.2 cGA . . . 21

4.2.3 BOA . . . 21

4.2.4 MMAS_ib . . . 22

4.3 Problems - Jump 23
4.4 Operating the program 23

5 Hypotheses, experiments and analyses of runtimes 25
5.1 UMDA 26
5.1.1 OneMax 26
5.1.2 LeadingOnes 32
5.1.3 Jump 33
5.1.4 Trap 37
5.2 cGA 38
5.2.1 OneMax 38
5.2.2 LeadingOnes 40
5.2.3 Jump 40
5.2.4 Trap 42
5.2.5 Similarity with UMDA 42
5.3 BOA 43
5.3.1 Networks 43
5.3.2 OneMax 45
5.3.3 LeadingOnes 47
5.3.4 Jump 48
5.3.5 Trap 50
5.4 MMAS 52
5.4.1 OneMax 53
5.4.2 LeadingOnes 56
5.4.3 Jump 57
5.4.4 Trap 60
5.5 Summing up 61
5.5.1 OneMax 61
5.5.2 LeadingOnes 62
5.5.3 Jump 62
5.5.4 Trap 63

6 Further Work 65
6.1 Program 66

7 Conclusion 67

Bibliography 69

Chapter 1

Introduction

Computers are widely used because they, among other things, can solve problems much faster than humans. An algorithm is a specification of how the computer solves such problems.

A common type of problem that is often encountered are optimization problems. The defining factor for these problems is that there are many solutions for each of them. The goal is the to optimize the solution until no better solution can be found (ideally).

Many problems have been studied and specific algorithms have been developed that can solve them efficiently and find the optimal solution. However, there are also many problems that have not been studied extensively and thus do not have specialized algorithms designed for them. Or it could be that the information about the problem is incomplete. In those cases, a metaheuristic can be used. A metaheuristic is a method of guiding the search for an optimal solution. Many types of metaheuristics have been tried throughout time and many have been studied.

Finding increasingly better algorithms to solve these problems plays a large role is the efficiency of today's systems. Many new problems are discovered and having efficient metaheuristics will allow computers to tackle almost any situation. A new topic of research is Estimation-of-Distribution Algorithms.

They are interesting because they employ probability distributions to estimate the location of the best solution(s).

Theoretical research have been done to examine the efficiency of Estimation-of-Distribuion Algorithms, but not much empirical data has been collected. This thesis will thus focus on the topic of data collection and analysis of this data. First, Estimation-of-Distribution Algorithms will be introduced alongside the optimization problems used in this paper. Then the implemented benchmark program will be described along with possible decisions about the specifics of the implementation. The next chapter will detail the hypotheses, analyses and experiments of different algorithms and problems. Data is provided and analyzed in order to figure out what general guidelines can be created with regards to choice of algorithm and input parameters for the different problems. The efficiency of the algorithms are examined. The chapter is thus based on data collected from runs of the algorithms in the implemented benchmark program. Lastly additional possible further work will be suggested.

Chapter 2

Objectives

This master thesis will implement four different Estimation-of-Distribution Algorithms in a benchmark program designed to test and visualize their runtime. The thesis will examine these algorithms and their input parameters to find possible guidelines for choice of parameters to optimize runtime. Proofs of running time for these algorithms will be taken into account when relevant and additional independent research will be conducted to construct hypotheses about these algorithms. These hypotheses will be tested in order to explore the possible optimal input parameters for certain problems and sizes.

2.1 Scope

I will create a benchmark program that will measure the runtime of the algorithms but will not measure other metrics, though e.g. memory usage could be of potential interest.

The benchmark program is designed to:

- Run EDAs on different problems
- Gather data about their runtime and visualize it in a graph
- Test the EDAs on well-known benchmark problems: OneMax, LeadingOnes, Jump and Trap

This thesis is focused on empirical tests done via the benchmark program and the conclusions that can be drawn from the data. Theoretical proofs are thus outside of the scope of this thesis. This thesis will therefore be a guideline for choice of parameters for these algorithms for the different problems.

Some of the theoretical proofs use functions in the input of the EDAs, but the program is limited to constant inputs. Building a math-interpreter for this purpose is considered out of scope for this thesis.

Chapter 3

Background

3.1 Evolutionary algorithms

Evolutionary Algorithms use the general idea of iterating upon solutions to increase their quality. An evolutionary algorithm maintains a set of solutions to a given problem and iterates over this set to create better solutions. An evolutionary algorithm roughly follows these steps

1. Generate initial set of solutions
2. Select promising solutions from set
3. Generate new candidate solutions from selected solutions
4. Incorporate new solutions into the set of solutions, possibly overwriting old ones
5. If termination criteria is not met, go to step 2

This iterative process allows the algorithm to increase the quality of solutions over time until a certain termination criteria is met. This criteria could for example be the creation of a solution of acceptable quality.

3.2 Estimation of distribution Algorithms

Estimation-of-Distribution Algorithms (will henceforth be referred to as an EDA) are based on Evolutionary algorithms in that they share the same general approach to solving problems. The difference between the two is that an EDA maintains a probability distribution instead of a set of explicit solutions. This changes the way solutions are represented and how the set of solutions is updated. An EDA consists of three general steps:

1. Generate *samples* from probability distribution
2. *Evaluate* samples, giving each a corresponding fitness-value
3. Use samples to update probability distribution

The algorithm will perform these three steps until some termination criteria is met. One execution of these three steps is referred to as a *generation*. A *run* is the execution of an algorithm from start to finish. The runtime of the algorithm is thus based on the evaluation of the samples and the number of generations used in a run. Evaluating the fitness values of the samples is considered the most expensive of these operation. This is because the EDAs are often run on problems with no known optimum. Evaluating the solution to a problem with no known optimum (and perhaps with millions of other solutions) is often a result of a time-consuming experiment.[1]

Updating the probability of the algorithm can be done in a myriad of ways, and because this operation can also be complex, it is also incorporated into the result of the runtime. The running time of an EDA with g generations and e evaluations per generation is then:

$$Runtime = g * e + g$$

This runtime may not be a fair comparison for all of the algorithms since some of them uses more complex operations, but since evaluating the solutions is considered the most expensive operation in a real-world scenario, the difference between operations in the algorithm is considered negligible.

[1] Source: Carsten Witt, explained during oral exchange

While the overall structure of EDAs remain the same, they can differ in a few ways:

- Sample process.
- Process of updating probability
- Dependability of variables

The sample process can differ on how many samples are generated and possibly how they are generated. The sampling process may also include choice of samples, that is, the algorithm will sample x solutions and keep the $y \leq x$ samples for further use. How (and if) this is done could potentially vary, but this selection process is often done using the fitness values of the solutions.

Updating the probability is likely where they largest difference occurs. The algorithms can use different parameters for the step-size when updating as well as how they use the generated solutions (and how they are weighted if there are more). A common method is to give the generated samples a weight that can be adjusted or calculated in a myriad of ways. The dependability of the variables shows how an algorithm perceives the connections between components of the problem. This is not an explicit step in the algorithm, but will instead influence the sampling and updating in some way. Some algorithms even naively assume that there is no connection between the different components.

Technically, EDAs can also differ in termination criteria, but for the purpose of this thesis, the termination criteria will be kept the same for all of them, namely sampling optimum. This criteria has a large impact on running time, so keeping the same for all algorithms is crucial for comparison. This criteria is chosen because it helps showcase when the algorithms reach their goal, finding the best solution to a problem.

Some EDAs keep their probability within a limit, that is, they never allow the probabilities to reach either 1 or 0. The pseudocode of some EDAs do not originally implement this limit, but this severely handicaps the algorithms. If the algorithm allows the probability of a bit to reach 1 or 0, the outcome is deterministic. That is, the outcome will always be the same, which will prevent the probability of said bit to ever change. If the bit is then "locked" in the wrong position, then the algorithm will never be able to sample optimum and is thus stuck. Having a limit means that the algorithm can recuperate from a mistake.

The algorithms considered in this project are restricted to a limited search space, namely $\{0, 1\}^n$.

3.2.1 Univariate Marginal Distribution Algorithm

The Univariate Marginal Distribution Algorithm[Müh97] (or UMDA for short) is an EDA that takes two parameters, both concerning the sampling process. First, UMDA takes λ samples from its probability distribution and selected the $\mu \leq \lambda$ best solution based on the fitness values of said solutions. The remaining solutions are discarded. UMDA will then update its probability according to the μ solutions in order to increase the probability of sampling those solutions. By choosing the solutions with best fitness, UMDA will slowly converge towards optimum. The pseudocode for UMDA given in [KW17] is detailed here:

$t \leftarrow 0$;
$p_{t,1} \leftarrow p_{t,2} \leftarrow ... \leftarrow p_{t,n} \leftarrow 1/2$;
while *Termination criterion not met* **do**
  $P_t \leftarrow \emptyset$;
  for $j \in \{1, ..., \lambda\}$ **do**
    for $i \in \{1, ..., n\}$ **do**
      $x_{t,i}^{(j)} \leftarrow 1$ with prob. $p_{t,i}, x_{t,i}^{(j)} \leftarrow 0$ with prob. $1 - p_{t,i}$;
    end
    $P_t \leftarrow P_t \cup \{x_t^{(j)}\}$;
  end
  Sort individuals in P descending by fitness, breaking ties uniformly at random;
  for $i \in \{1, ..., n\}$ **do**
    $p_{t+1,i} \leftarrow (\sum_{j=1}^{\mu} x_{t,i}^{(j)})/\mu$;
    Restrict $p_{t+1,i}$ to be within $[1/n, 1 - 1/n]$;
  end
  $t \leftarrow t + 1$;
end

Algorithm 1: Univariate Marginal Distribution Algorithm (UMDA)

As can be seen, the probability is kept within a bound of $[1/n, 1 - 1/n]$. These limits are introduce "[...] such that the algorithms always show a finite expected optimization time, as otherwise certain bits can be irreversibly fixed to 0 or 1" [SW16]. Additionally, each variable is weighted equally when updating the probability.

Choosing the input parameters will influence the behavior of the algorithm. The ratio between λ and μ will have an impact on the speed of convergence as well as how it converges. Keeping few solutions will increase the average quality of the fitness of those solutions, but then the structure of those solutions also dictate how UMDA converges, i.e. if the one solution that is kept has a 0, then

that zero alone will dictate how the probability for that bit is updated. Keeping more solutions will decrease the average quality of the solutions, but UMDA will then converge towards the common structure of those solutions, i.e. even if a solution has a 0-bit other solutions will hopefully have a 1 that will, on average, pull the probability in the correct direction.

3.2.2 Compact Genetic Algorithm

The Compact Genetic Algorithm (cGA for short) uses much the same approach as UMDA. cGAs input parameters instead specify the update rate of the algorithm instead of the sampling process. The input for the algorithm is the denominator for the *update strength* of the algorithm. That is, for an input K, the update strength will be $1/K$. Instead of having many samples, cGA chooses specifically two. It then lets the two solutions compete based on their fitness values and keeps the winner. The structure of the winner then decides which probabilities get increased/decreased. The learning rate determines the magnitude of the change. [SW16] provides the pseudocode for cGA.

cGA was originally introduced in [HLG99] but Sudholt and Witt introduced the limits on the probability. As was mentioned earlier, it can be detrimental for the probability of a bit to reach either 1 or 0. The original version of cGA did not include this restriction which could easily lead to very bad performance.

3.2.3 Bayesian Optimization Algorithm

The Bayesian Optimization Algorithm (BOA) is a general structure for how to approach the EDA from a slightly different angle[PGCP99]. BOA creates a model from the sampled data and tries to find structure in the data. This means that, unlike UMDA and cGA, that BOA believes its variables to be *dependant* on each other in some form. The model that BOA creates is then instantiated in order to generate new (and hopefully better) solutions.

BOA is very general in that it does not give the specifics of each individual step. When sampling, BOA will retrieve "promising strings"[PGCP99] from its distribution. How these strings are gathered is left to the user to determine. Additionally, the newly generated strings from the model are incorporated into the probability distribution, but this is also left to the user to determine. This means that there are potentially many ways to go about this implementation and the different implementations is likely to have an impact on its performance.

```
t ← 0
p_{1,t} ← p_{2,t} ← ... ← p_{n,t} 1/2
while Termination criteria not met do
    for i ∈ {1, ..., n} do
        x_i ← 1 with prob. p_{i,t}, x_i ← 0 with prob. 1 − p_{i,t}
    end
    for i ∈ {1, ..., n} do
        y_i ← 1 with prob. p_{i,t}, y_i ← 0 with prob. 1 − p_{i,t}
    end
    if f(x) < f(y) then
        Swap x and y
    end
    for i ∈ {1, ..., n} do
        if x_i > y_i then
            p_{i,t+1} ← p_{i,t} + 1/K
        end
        if x_i < y_i then
            p_{i,t+1} ← p_{i,t} − 1/K
        end
        if x_i = y_i then
            p_{i,t+1} ← p_{i,t}
        end
        Restrict p_{i,t+1} to be within [1/n, 1 − 1/n]
    end
    t ← t + 1
end
```

Algorithm 2: Pseudocode for the Compact Genetic Algortihm (cGA)

The model that BOA uses to simulate the structure of the data is a Bayesian Network (BN). A BN is a structure that correlates different parts of the data to each other. The structure of a BN can either be predetermined or it can be inferred from some given data. BNs are graph-like structures where each node in the structure is a variable and edges represent correlations between these variables. If a node j has an edge to node k, then j is said to be the *parent* of k. The edges in the graph are all directed and the graphs must be acyclic. It may be possible to work with cyclic graphs, but that is considered out of scope for this thesis.

As mentioned earlier, these algorithms are limited to the search space of $\{0,1\}^n$. A node in the network corresponds to one bit in the string as well as a probability for each of its values being chosen. The probabilities in the network are either unconditional or conditioned on their parent(s). If a node does not have a parent, the probability is clean in that there is a specific chance for choosing one value and one chance for choosing a different value. If a node has one or more

parents, the probability of the node will change depending on the parents choice. If a node has one parent, the node will have two probability tables detailing its probability for choosing the different values in each case. If a node has more parents, then there exists more combinations of their outcomes, resulting in a larger probability table for the child since it needs to take all these possibilities into account.

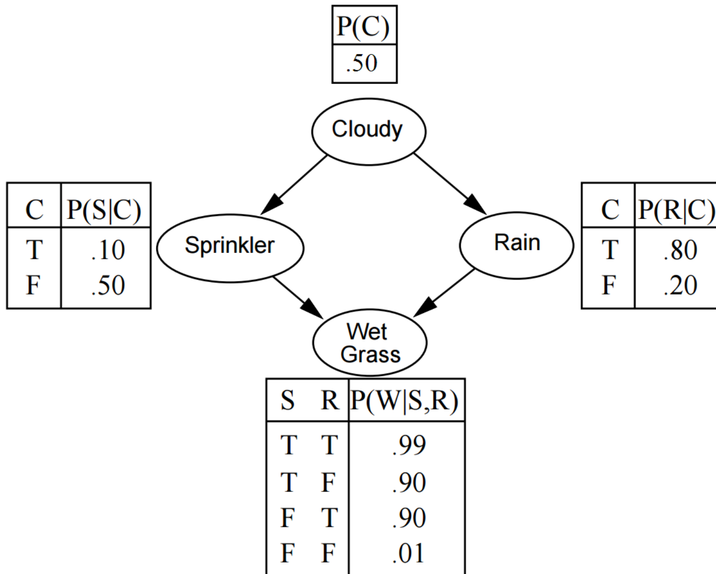


Figure 3.1: Example from MIT that shows a Bayesian Network interpreted as conditional probability tables.

The probabilities in the network are used when instantiating. Instantiating (or creating an instance of) refers to the network generating an output based on the probabilities and connections[2]. This output is an assignment of a truth value to each node in the network. Instantiating a network will generate a new bit-string that can then be used to update the probability of the algorithm.

Above in figure 3.1, each node has a conditional probability table. Each entry in the table is a probability. First it is decided if it is cloudy. This happens half the time. Say it does become cloudy. Then the probability for the sprinkler and raid based on that fact, that is, the sprinkler has 10% chance to be turned on and it has an 80% chance to rain. The outcome of these two probabilities

[2]M. Pelikan originally calls this "sampling". However, in order to not confuse this sampling with sampling from a probability distribution, it will instead be referred to as *instantiating*. Similarly, a sample that is generated by instantiating a network is referred to as an *instance*.

then determine the probability for the grass being wet. In the most probable example, the sprinkler is off and it is raining. This gives the grass a 90% chance of being wet.

A BN does thus not provide better solutions by itself, but instead tries to model its input data and generate similar outputs. To improve the quality of the data sampled from the probability distribution, BOA uses a selection process when choosing which sampled data to model. Choosing the best samples will then allow the model to instantiate new and similar solutions. Incorporating these new solutions into the probability distribution can improve it and thus improve any further generated samples. Algorithm 3 shows the (very general) psudocode for BOA:

(1) set $t \leftarrow 0$
Randomly generate initial population
(2) select a set of promising string $S(t)$ from $P(t)$
(3) construct the network B using a chosen metric and constraints
(4) generate a set of new strings $O(t)$ according to the joint distribution encoded by B
(5) create a new population $P(t+1)$ by replacing some strings from $P(t)$ with $O(t)$
set $t \leftarrow t+1$
(6) if the termination criteria are not met, go to (2)

Algorithm 3: Pseudocode for the Bayesian Optimization Algorithm (BOA) [Pel02]

The pseudocode here uses a population P, but luckily, it is easily possible to replace this population with a probability distribution by simply sampling from it and using instantiated strings to update the probability.

Constructing the network is the difficult step. Because a network models the structure of the data, creating a network requires learning this structure from the samples and their fitness value. Extrapolating the dependability of variables from randomly generated samples is considered out of scope for this thesis. Instead, pre-determined networks will be provided and explored for the problems. These networks are discussed in chapter 5 section 5.3.

3.2.4 Max-Min Ant System - Iteration Best

The Max-Min Ant System (MMAS) is a variation of the Ant System algorithm. Such algorithms attempt to model an ant colony as they appear in nature. The

ants of the colony will seek out solutions individually and communicate their findings to the colony. While the system tries to simulate an ant colony, it is strikingly similar to an EDA and will be considered as such.

A solution is said to consist of *solution components*. Solution components are the individual building blocks that make up a solution. An ant starts with an empty solution and iteratively adds components to its solution until it is complete [SH00]. In the case of this thesis, the solutions to all problems considered are represented as binary strings, and as such, each bit in these strings can be considered a solution component.

Ants communicate with each other using a chemical called *pheromone*. When the ants have found solutions, the ant with the best solution deposits pheromone on each solution component. In order to avoid bad solutions, this pheromone is set to decay over time. If an ant deposits pheromone on a solution component, but no other ant does, the pheromone will eventually decay. This allows the algorithm to "forget" bad solutions [SH00].

There exists a few different versions of the Max-Min Ant System. The version of MMAS that is considered in this project is the simplified version described by Kötzing in [KNSW10] because it is specifically designed for the problems that are the focus of this thesis. The differences between the original Ant System and Kötzings simplified MMAS are:

- The pheromone on a solution component is restricted to be within a specific interval of $[1/n, 1 - 1/n]$ with n being the problem size.
- The update of the pheromone trail uses a slightly different formula.
- The probability of choosing a specific solution component.

Kötzing describes the way an ant constructs a solution to be a path through a graph with binary choices. Each choice is between two edges leading to the same node. These edges represent the bits 0 and 1. After the solutions have been created, x is the best chosen path. The probabilities (pheromone values) of the algorithm is then updated.

"Depending on whether an edge e is contained in the path $P(x)$ of the solution x, the pheromone values τ are updated to τ' as follows:"[KNSW10]

$$\tau'(e) = \begin{cases} min\{(1-\rho) * \tau(e) + rho, 1 - 1/n\} & \text{if } e \in P(x) \\ max\{(1-\rho) * \tau(e), 1/n\} & \text{if } e \notin P(x) \end{cases}$$

This algorithm takes two inputs. The first is the number of ants, i.e. the number of solutions generated. The second is the pheromone factor. As can be seen from the pheromone update formula, it determines both the pheromone decay and the amount of pheromone that is added based on the chosen solution.

Throughout, MMAS may be refereed to as "λ-MMAS", e.g. 2-MMAS for $\lambda = 2$.

3.3 Problems

This project will concern itself with simple problems whose solutions can be represented as an n-bit string. Each of these problems have unique fitness functions as well as exactly one optimum. The fitness functions are pseudo-boolean functions $f : \{0,1\}^n \to \mathbb{N}$ [DN10].

These problems are carefully chosen. They are relatively simple and the optimal solution is known. They are important for initial theoretical analysis of the algorithm's efficiency. Using the same problems in this project allows for parallels to be drawn between experiments and theory.

The first problem is OneMax. OneMax uses a simple linear fitness function which counts ones. For any given string $t = (x_1, ..., x_n)$, the fitness value of OneMax is

$f(t) = \sum_{i=1}^{n} x_i$. [KW17]

LeadingOnes poses the problem of how many ones will "lead", that is, how many consecutive ones there are from the first position. The highest number is of course n which gives LeadingOnes the same optimum as OneMax. The function is given as

$f(t) = \sum_{i=1}^{n}(\prod_{j=1}^{i}(x_j))$ [DN10]

The third problem is Jump. Jump uses the same fitness function as OneMax, but with a gap. The motivation for using Jump is that the difficulty can be adjusted. That is, for a problem size of length n and a fitness value $g(t) = \sum_{i=1}^{n} x_i$, there exists a value m such that the fitness value becomes

$$f(t) = \begin{cases} m + g(t) & \text{if } g(t) \leqslant n - m \text{ or } g(t) = n \\ n - g(t) & \text{otherwise} \end{cases}$$

[DJW02]

This means that there is a "hole" in the middle of the function that the algorithms will need to overcome. With wrong parameters (or if the hole is too big) the algorithms may become stuck in the local maximum.

The last problem is Trap. Trap poses a big problem for some algorithms because it has exactly the inverse fitness function of OneMax but still has the same optimum. The fitness value for trap for a binary string of length k with μ ones is defined in [GMB08] as follows:

$$f(t) = \begin{cases} k - \mu - 1 & \text{if } \mu < k \\ k & \text{if } \mu = k \end{cases}$$

Trap can be considered an extreme version of the Jump problem. Trap is the most difficult of all of the problems. The reason is that if optimum is not sampled, then the best solution(s) used to update the probability will bring it further away from optimum. Most algorithms keep the probabilities within an upper and lower limit, and if the algorithm converges away from optimum, then it may end up in a situation where all of the probabilities are at or close to the lower bound. This means that it has a very small chance to sample a 1 on each position, and the larger the problem size, the more unlikely it is to sample optimum. This means that many algorithms will never find a solution with large problem sizes.

In this project, a bit-string that can be given a fitness value for these problems is referred to as a *solution* or a *sample*. This does not necessarily mean that it is the *optimal solution* of which there is only one.

Chapter 4

Benchmark program

4.1 Modules and overall structure

I developed the program using a black-box approach with modularity in mind. Different modules were created, each with a specific purpose. The modules are essentially one or more functions that execute the purpose of that module. The modules only communicate with a few other modules in order to keep it simple and maintainable. The modularity also allows the possible expansion of the program with additional problems or algorithms.

4.1.1 EDA Module

This module contains the definitions for an EDA. This module can be used to create and run an EDA. It takes care of the overall structure and the differences between the EDAs are given as input. To create an EDA, two functions are specified. The two functions are a sample function and an update function respectively. Each of these functions are given some inputs. The sample function is given the EDA's probability distribution and two extra inputs that are specified by the user. The update function is also given the EDAs state, two user-inputs (that are different from the samples functions user-inputs) and is

also given a list of evaluated samples. The samples are generated by the sample function, evaluated by the problem module and then given to the update function, all supervised by the calculation module.

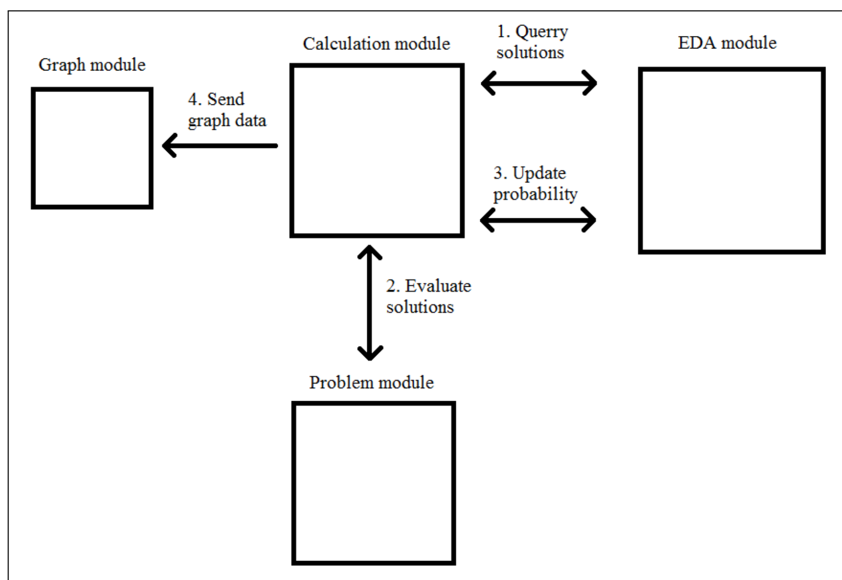


Figure 4.1: The modular structure of the program. The boxes are supposed to simulate the "blackbox" approach.

4.1.2 Problem Module

This module is relatively simple. Since the program is limited to problems that use a binary strings as its data, the only thing needed to create a problem is a fitness function and an optimum. The fitness function will evaluate any solution passed to it. In the case of OneMax, simply summing all the numbers in the solution together and dividing by the problem size will yield a fitness value.

4.1.3 Graphics Module

The graphics module acts as the supervisor of the system. The graphics module interacts only with the Calculation Module (explained below) in that it initiates calculations and processes the results from the calculations. The graphics modules task is to show the graphics on the screen as well as receive and preprocess

user inputs. Once a user has used the Graphical User Interface (GUI) to select algorithm, problem and their respective parameters, the graphics module checks the inputs to make sure that they are valid.

If the inputs are correct, the graphics module will initiate the calculations module and relay the user information to it. In order to be able to run the calculations in parallel with the graphics (so the app does not appear frozen), a thread is created. This thread is used to run the calculations while the main thread focuses on the graphics.

During the run of the algorithm, the calculation module continuously sends information about the state of the run back to the graphics module. The graphics module will then show the status of the run to the user, that is, the user will see what problem size is currently being tested and what iteration it has reached so they know how much of the current run is completed.

Once the calculation module has run its course, the graphics module receives information about the algorithms running time for each problem size. The graphics module then creates a graph with the received data and shows it to the user. The user can click the "Save Data" button to save the graph for the run as well as the raw data for the run. The data can be used in other programs (e.g. Matlab) for various purposes.

To show graphics on the screen, a python library, Kivy [VP], is used. Kivy structures the graphics into so-called Layouts. A Layout is a part of the screen that can contain objects that the user can interact with or more layouts (or both). There are a few different layouts that each will change the structure of the screen in different ways. To separate the different functions of the program, several layouts were used. First, the screen is separated vertically into two. The left half is used for control and user-inputs, while the right side is used for showing the graph. The left side is then further divided into two, the upper half which controls the algorithm and its parameters and the bottom half which controls the problem, its parameters, the start button and status text.

Using this structure makes it easier to change the position and size of the objects on screen. These objects are called Widgets. When a Widget is added to a layout, its position and size relative to that layout can be specified. This makes it very easy to display the graph as its size is simply set to fill out the entire layout it is in. It is possible to just specify the raw position and size relative to the entire window, but that gets tedious to maintain.

The graph is not drawn by hand since Kivy handily includes library for drawing those as well. Since it is a part of Kivy, it perfectly integrates with the layouts.

4.1.4 Calculation Module

The calculation module does all the calculations for the algorithms. The module supervises the algorithms and ensures that they run according to the pre-defined rules. These rules include the termination criteria for each algorithm and the amount of runs per size of the problem. The module supervises the run in the sense that the algorithm and problems are slaves. They do not operate on their own and only execute the jobs given to them by the calculation module. There are three tasks. The module can ask the EDA module to generate samples from its probability, ask the problem to evaluate these samples and then finally ask the EDA again to update its probability based on these evaluated samples. As mentioned above, the calculation module will continuously send data about the progress of the calculation to the graphics module.

Because these algorithms are based on probability, the outcome of each test can vary. In order to create reliable (statistically reliable) data, each algorithm is run 100 times on each size of the problem. This way, we eliminate outliers in the tests and they thus give us a much clearer look into the running time of the algorithm.

Once the run is finished, the calculations module returns the data of the run to the graphics module.

The calculation module uses Python's built-in random function to sample the probability distribution.

In order to avoid having programs run forever, an upper limit of generations has been set. This limit is 100 000 generations. If an algorithm reaches this limit, it is unlikely that they will ever finish. A run that reached this limit is considered dead and is not counted in the average of runs when creating the graph. I.e. if there are two failed runs for a problem size, the result that is used on the graph will be the average of the remaining 98 runs. There may be a few situations where this limit is too small. If tests are run with a higher limit in chapter 5 it will be stated. If nothing is stated, then the normal limit of 100.000 is used.

4.2 Algorithms

As some algorithms are not explained in detail, decisions will have to be made as to how these details work. This section will explain these decisions and why they are made. These decisions will obviously have a performance impact on the

algorithm and while it might be interesting to examine them and compare different versions, considering different versions of the algorithms will be considered out of scope as they would have to be compared on each problem.

4.2.1 UMDA

UMDA uses some relatively simple steps in its calculations. [KW17] provides pseudocode for UMDA and also provides all the details. Thus there are no implementation-specific considerations to be made.

4.2.2 cGA

As with UMDA, the suggested implementation in [SW16] provides all the details necessary to implement cGA with no implementation-specifics left out except for the possible removal of the limits on probability, but that will not be considered in this project.

4.2.3 BOA

Since [PGCP99] only gives a general algorithm, many specifics of the implementation is left to the user. Therefore I have had to make many choices in regard to how BOA is implemented. I will later briefly touch on the structure of the networks and attempt another network structure for Trap to see what impact it might have.

The original description of BOA uses a population instead of a probability distribution. The selection process is described as follows: "From the current population, the better strings are selected. Any selection method can be used." [PGCP99] The only limitation of the selection process is that "the better strings are selected". In addition, the pseudocode of BOA describes the selection process as "select a set of promising strings $S(t)$ from $P(t)$". So the selection process must sample the probability distribution and then select the better strings from those samples. An algorithm that already employs a selection process like this is UMDA. Since the only metric that exists for determining whether or not a string is "promising" is the fitness value of that string, the UMDA method is the most straight forward.

After the network has been instantiated, the instance must be incorporated

into the probability distribution. How this is done is also left entirely to the user. Many algorithms like to use different weights for variables in order to control their impact. This simple solution is also chosen here. The instance is given a weight and is then combined with the probability to generate the new probability. This method is exactly the same as MMAS uses and will therefore be referred to as ρ.

Given probability P and weight w
$I \leftarrow instantiate(BayesianNetwork)$
for $i \in P$ **do**
  $P_i^{'} \leftarrow I_i * \rho + P_i * (1 - \rho)$
end
$P \leftarrow P^{'}$

Algorithm 4: Probability-update algorithm of the implementation of BOA

To model Bayesian Networks, I use a python library [JMS]. Sadly, the library is incomplete and did not include a function to sample the network and generate solutions, so I had to implement it myself. The library uses probability tables for showing its probability based on parent(s). Instantiating the network is done by using the networks probabilities and conditional probabilities. First, all the roots values (the nodes with unconditional probabilities) are determined by their probability. Then their children's value are calculated based on their parents' value and their conditional probability. Then their children are determined and so on. This process gives way for a priority list which denotes which nodes needs to be calculated in which order. Once all nodes have been determined, their values are extracted and an instance is created.

In addition to the above, limits are imposed on the probability update, meaning that the probabilities are limited to be within $[1/n, 1 - 1/n]$. This is common trait among the other algorithms. The reason is explained in the previous chapter.

4.2.4 MMAS_ib

The original version of MMAS was first introduced by Stützle in [SH00]. This version, however, was designed to be used on graphs. Kötzing instead proposes a simplified version of the algorithm in [KNSW10]. This simplified version does change the algorithm a bit, but adapts it in such a way that it fits perfectly for problems with binary strings. One can imagine a different implementation of course, but Kötzing argues in detail about why these specific changes have been made.

4.3 Problems - Jump

There are actual variations in the Jump problem just like with the algorithms. The problem has a factor k which determines the length of the Jump. This is initially set to 5 but experiments can be done with larger jumps. However, due to the problems not taking any kind of input, this will require editing a single value in the code.

4.4 Operating the program

The user will be prompted with selecting the parameters of the run and the program will take care of the rest. When the program is started, the user will see the start screen, shown in figure 4.2.

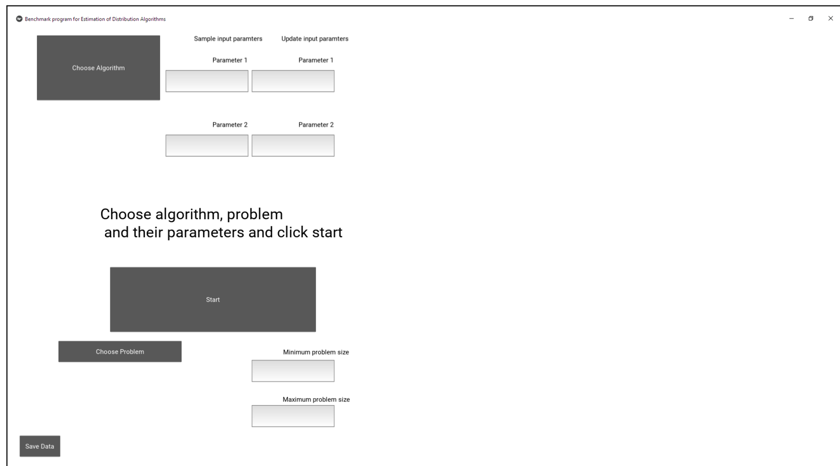


Figure 4.2: Benchmark program in idle state, ready to receive input.

The user will see three buttons. "Choose algorithm", "Choose problem", and "Start". The buttons should be self-explanatory in that the user will choose an algorithm to test and a problem to test the algorithm on. The "Start" button will run the algorithm. Additionally the user will see some text-fields. These text-fields are either the input parameters of the algorithm or the constraints of the problem. The user is able to select an minimum size problem and a maximum size. Above each text-field is some accompanying text that explains to the user what should be typed in the field. The text is always the same for the problem, but it may change depending on the algorithm chosen. As

each algorithm uses different input parameters, the text-fields have a different meaning depending on the chosen algorithm. There are four text-fields, two for inputs to the sample function and two for inputs to the update function. Since not all algorithms use all inputs, some of the text-fields may have the text "Not in use" above them, indicating that the algorithm does not use that input.

Once all parameters have been determined, the user can click the "Start" button to start the run. If everything is correctly entered, the program will run the algorithm. If there are any errors in the inputs, the program will provide an error message.

Between the two areas of input is the status text. While the program is running, this status text will serve as a progress bar, showing how far the program is in its calculations.

Once the run is completed, the status text will change to express that and a graph will appear on the right side of the program. This graph will show the running-time for the algorithm with the provided inputs for each size of the chosen problem. This will allow the user to see how the algorithm's running time evolves with respect to problem size. For large runs (problem sizes over about 40) the y-axis will not show numbers. This is because there are simply too many results to show. As the run completes, the "Start" button is re-activated and the user can change the settings and run again. Only one run will be shown on the graph at a time. If the user wants the precise output of a run, the exact values for each run is always output in a text-file if the user clicks "Save Data".

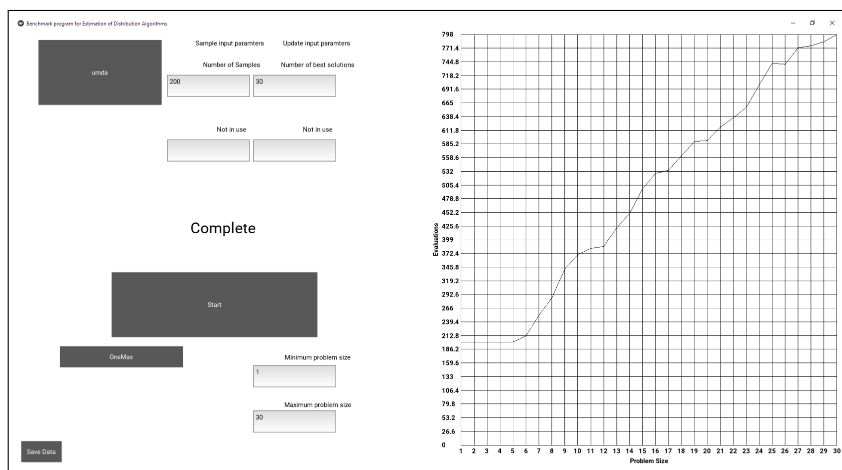


Figure 4.3: Benchmark Program post-run

Chapter 5

Hypotheses, experiments and analyses of runtimes

In this chapter I will use the benchmark program to attempt to analyze the running-time of the algorithms with different input-parameters. I will analyze their running time, and attempt to hypothesize what other configurations are interesting to analyze. If possible, I will try to find a configuration of inputs that will allow the algorithm to perform well. Additionally, I will examine the implementation-specifics of BOA and MMAS in an attempt to see if other ways of implementing will provide a better running-time.

I will examine each combination of algorithm and problem individually, but I will draw inspiration from similar problems where it is relevant.

The scientific literature may give mathematical proofs for certain algorithms and problems, and if they do, I will examine the ranges in which the proofs hold, but I will also examine the ranges outside of the proofs in which I will go into more detail to see if I can find any kind of tendency.

There is scientific literature that provide insight into the general idea of parameters for these algorithms. One of these is Sudholt and Witt who prove for cGA and 2-MMAS that small update strength are best. Using update strengths that are too large leads to *genetic drift*. Their analysis "[...] reveal a gen-

eral trade-off between the speed of learning and genetic drift.". Having a high update-strength will increase the speed at which the algorithm converges, but will also increase the risk of "adapting to samples that are locally incorrect" [SW16]. They propose an update strength of

$S \sim 1/\sqrt{n} logn$

for cGA and 2-MMAS. However, they "are optimistic to be able to extend them to other EDAs such as the UMDA" [SW16], providing guidelines for the analysis of these three algorithms.

It should also be noted that for algorithms whose input determines update strength (cGA, BOA and MMAS), I will not choose an update strength of zero for Trap, even though it might be optimal because it will cause the initial probability to never change, defeating the purpose of the algorithms.

5.1 UMDA

UMDA takes two input parameters. The amount of samples to generate λ and how many samples to keep μ. Increasing λ heightens the chance to generate optimum, but also requires more evaluations. A low value for μ will likely increase the average quality of the samples but will also allow those few samples to dictate the change in probability. If those samples are not optimal, then they will also pull a few bits in the wrong direction.

5.1.1 OneMax

Several analyses specifically of UMDA on OneMax have been released. Most notably, an upper bound [Wit17] and lower bound [KW17] have been proved for certain ranges of inputs. These bounds do not cover all inputs for the algorithm, but could still be used as a general guideline for the runtime of the algorithm. [KW17] provides an analysis of the algorithm and provides a lower bound for the algorithm in the following theorem

Theorem 5.1 *Let $\lambda = (1 + \beta)\mu$ for some constant $\beta > 0$. Then the expected optimization time of the UMDA on OneMax is $\Omega(\mu\sqrt{n} + nlogn)$.*

Within the bounds of the theorem, I will run tests for two cases of lambda, namely $\lambda = 2$ and $\lambda = 200$. In both cases, $\beta = 1$ and that gives the expected optimization time of $\Omega(\sqrt{n} + nlogn)$. Both runs uses problem size 1 to 20.

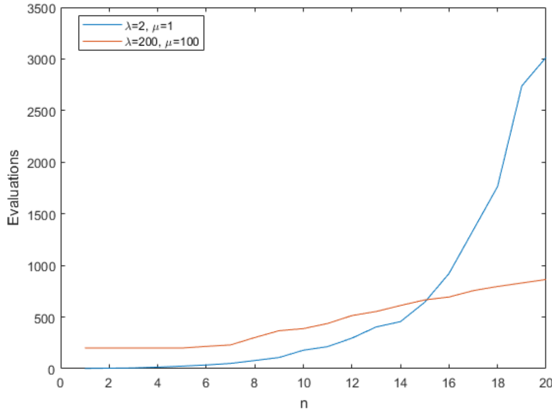


Figure 5.1: UMDA on Onemax $n = 1, ..., 20$ for two configurations

As can be seen, the small sample size seem to generate an exponential increase in evaluations while the large sample size does not. The reason for this is explained in [Wit17] where it says that values for "[...] $\mu = o(logn)$ will lead to a too coarse-grained frequency scale and exponential lower bounds on the runtime". The value for μ and thus also λ must be large enough compared to n in order to avoid this exponential running time. The problem size of 1 to 20 may, however, not show the total tendency of the runtime. It could potentially be exponential but the current problem size does not show that. To test, I run the algorithm for up to problem size 100.

The graph does not become exponential, but even for a 100-bit problem, μ is still much larger than $logn$. It is possible that running for larger problem sizes while keeping constant μ and λ will result in exponential running time. The graph initially looks linear, but the larger graph up to $n = 100$ looks radical (runtime of $\sqrt{n}$), but the amount of evaluations is still above the lower bound.

The above theorem only states that the ratio between λ and μ must be constant, but does not pose any other requirements. [Wit17] describes two additional theorems that put additional constraints on the value of μ.

Theorem 5.2 *Let $\lambda = (1 + \beta)\mu$ for an arbitrary constant $\beta > 0$ and let $\mu \geq c\sqrt{n}logn$ for some sufficiently large constant $c > 0$. Then with probability $\Omega(1)$, the optimization time of UMDA on OneMax is bounded from above by $O(\lambda\sqrt{n})$. The expected optimization time is also bounded in this way.*

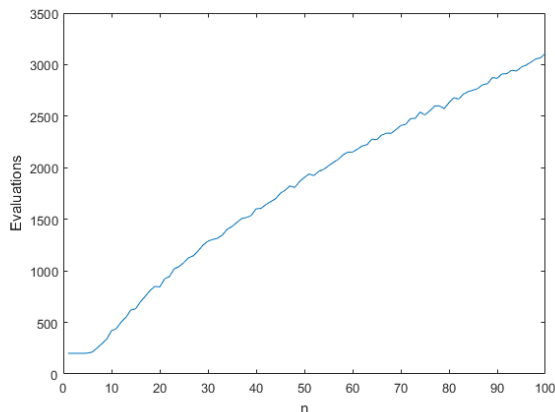


Figure 5.2: UMDA with $\lambda = 200, \mu = 100$ on OneMax for $n = 1, ..., 100$

The ratio between λ and μ is the same as the earlier theorem, but now, μ must have a minimum size according to n. Sticking with the previous choice of β, we can now choose a "sufficiently large constant $c > 0$". Since λ scales with μ we want to avoid choosing a too large c if possible. This will not change its asymptotic running time, but will affect its real running time. Choosing a values of $c = 3$ should prove sufficiently large. Since the benchmark program cannot take functions as input, a constant μ for the largest values of $n = 100$ will have to be sufficient. This gives us input parameters $\mu = 3\sqrt{100}log(100) = 138, \lambda = 2\mu = 276$. Making a similar test for an even larger $c = 5$ gives us $\mu = 230, \lambda = 460$.

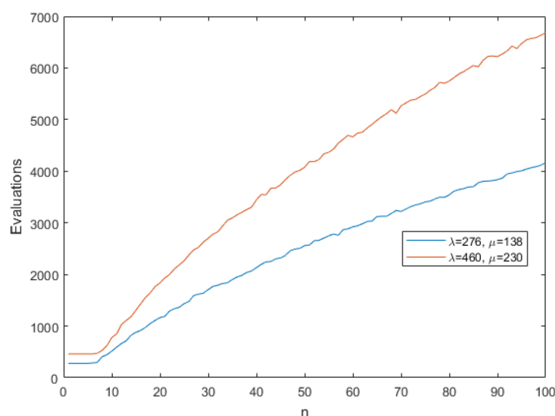


Figure 5.3: UMDA on OneMax $n = 1, ..., 100$ for two configurations with sufficiently large values of c

The graphs have much the same structure, showcasing the same asymptotic running time, but since the running time is dependent on λ, the lower value will of course have a faster running time.

Theorem 5.3 *Let $\lambda = (1+\beta)\mu$ for an arbitrary constant $\beta > 0$ and $\mu \geq clogn$ for a sufficiently large constant $c > 0$ as well as $\mu = o(n)$. Then the expected optimization time of UMDA on OneMax is $O(\lambda n)$.*

This theorem gives a smaller bound for μ than the previous theorem and also sets an upper bound on μ. The reason is that there is what Witt calls a "phase transition" between the two running times around $\mu = \sqrt{n}logn$. This transition describes the transition between the behavior of the algorithm and thus the transition between the two bounds on the running time. A configuration of UMDA that is "in" the phase transition does not fit within either theorem and will then possibly be unstable.

The first theorem defines the upper bound when μ is above this transition and this theorem defines the upper bound when μ is below this transition. Choosing a c much larger than 2 will break the upper bound on μ. Thus, choosing a value of $c = 1$ and $c = 2$ seems appropriate. Keeping the previous value of $\beta = 1$ gives us the following runs: $\{\mu = 46, \lambda = 92\}, \{\mu = 92, \lambda = 184\}$.

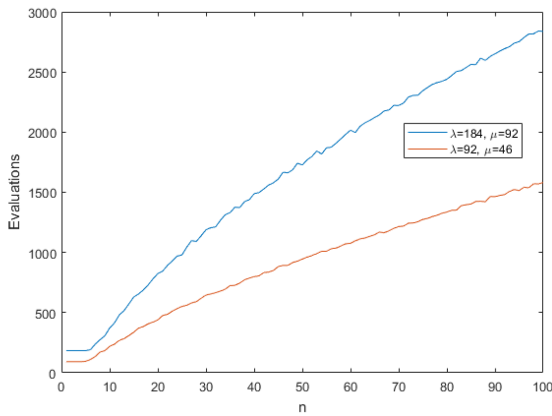


Figure 5.4: UMDA on OneMax $n = 1, ..., 100$ for two configurations for smaller values of c

This is the "same" result as the above theorem. They have somewhat similar structure except that the lower λ looks almost linear. In this case the running time is bounded by $O(\lambda n)$ which could explain the almost linear look of the lower graph. This is likely caused by the fact that both values for μ still satisfy

$\mu \geq c\sqrt{n}logn$ for $c = 1$. This means that c will have to be less than 1 in order to avoid this. Testing with $c = 0.5, c = 0.2$ gives the following results:

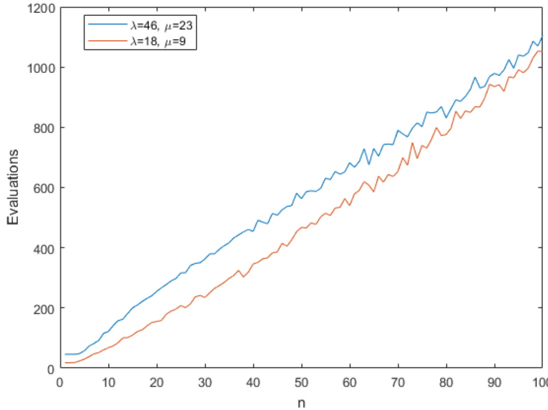


Figure 5.5: UMDA on OneMax $n = 1, ..., 100$ for two configurations for very small values of c

These runs look more linear, suggesting that they are now within the second theorem. However, these runs also appear much more unstable than the previous runs. The lowest value of $c = 0.2$ seems to be on the verge of slipping into polynomial territory, but this is not certain as it could just seem that way due to probabilistic variance.

All of the above tests have had a constraint $\lambda = 2\mu$. Choosing a different value might yield interesting results. Choosing a value of $\beta = 0$ will lead to genetic drift because it will no longer discard the bad solutions and will thus adapt to samples that are locally incorrect. Choosing a different assortment of values of $\beta = 199, 19, 0.15$ gives very different results as seen in figure 5.6.

This graph shows that a balanced β seems to work best. Keeping only one sample can lead to mistakes because if a solution is not optimum it will decrease the probability of some bits by quite a bit whereas a small "council" of solutions will remedy the mistakes of each other. Keeping too many will likely lead to an average fitness values that is too low.

There are quite a few configurations that perform pretty well. The best tested configuration is $\lambda = 18, \mu = 9$ for problem sizes up to $n = 100$. However, the ratio between λ and μ in this configuration is $\beta = 2$. For a high $\lambda = 200$ this ratio gives a running time that is fairly bad, so it is interesting that the same ratio for a low λ gives the best running time. Testing for a similar λ but with the ratio or approximately $\beta = 8$ in figure 5.7 gives results that are much

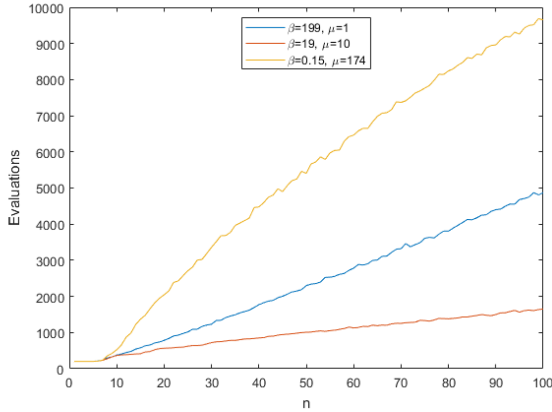


Figure 5.6: UMDA on OneMax with three values of β. The values of μ are chosen such that $\lambda = 200$ in all three cases

similar. The only notable difference is that the low $\mu = 2$ seems to be a tad more unstable for the larger problem sizes whereas $\mu = 9$ is a little more stable. However, the variance is only small and so this conclusion might be wrong and it is purely related to probability.

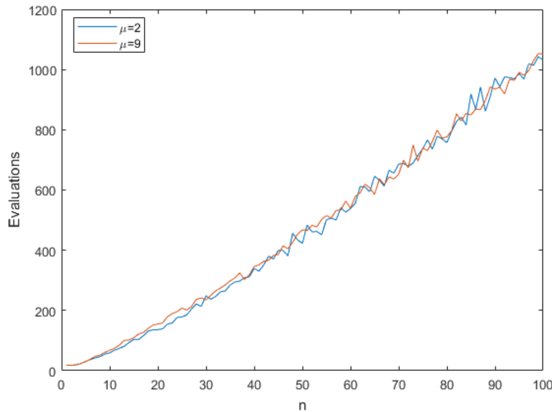


Figure 5.7: UMDA on OneMax with constant $\lambda = 18$ for two values of μ

The best configuration for UMDA on OneMax seems to be a relatively low λ with a $\beta \in 2, ..., 19$. A high λ leads to too many evaluations per iteration, but can still perform relatively well if μ is chosen carefully. If it is too high the quality of the average solution is too bad. If it is too low the few solutions dominate too much, possibly pulling some probabilities in the wrong direction. A balance with $\lambda = 200, \mu = 10$ works well as it seems to avoid the two extremes.

However, a low λ still performs better. The two best tested configurations are $\lambda = 18, \mu = 9$ and $\lambda = 18, \mu = 2$ who performed very similarly.

5.1.2 LeadingOnes

The literature mentioned above focus only on UMDA for OneMax. LeadingOnes have the same optimum as OneMax, but its fitness values are quite different. Since LeadingOnes and OneMax are so similar, I could try to use the above parameters for the best output as a starting point. However, Dang et al. show in [DL15] an expected optimization time for UMDA. They provide the following theorem:

Theorem 5.4 *The Univariate Marginal Distribution Algorithm (UMDA) with offspring population size $\lambda >= bln(n)$ for some constant $b > 0$, parent population size $\mu = \gamma_0 \lambda$ for any constants $\gamma_0 \leq 1/((1+\delta)13e)$ and $\delta > 0$ and margin $m = \mu/n$ has expected optimization time $O(n\lambda ln(\lambda) + n^2)$ on LeadingOnes.*

The theorem mentions a margin $m = \mu/n$, but the limit for the probability is calculated as m/μ which, with the given margin, becomes $(\mu/n)/\mu = 1/n$ which is exactly the limit that my implementation uses.

The parent population size μ must be a factor γ_0 smaller than λ. The largest ratio is obtained with $\delta = 1$ which gives a ratio of $1/26e = 0.014$ suggesting that only the best 1.4% of solutions should be kept. An initial test for $\lambda = 100, \mu = 1$ is seen in figure 5.8 and reveals that the runtime is indeed polynomial. An additional test with $\lambda = 200$ and $\mu = 0.014 * \lambda = 3$ is also shown.

I will now test what happens when the limit $\mu = \gamma_0 \lambda$ is broken. This happens in any situation where $\mu > 0.014 * \lambda$. I made two tests, one with slightly larger μ and one with much larger μ as seen in figure 5.9.

Going slightly outside the boundaries of the limit does not have a huge impact. The runtime is increased a bit, but the graph still has the same structure. The much larger μ, however, has become exponential. There is thus some kind of phase transition between polynomial and exponential terrain for the ratio between λ and μ.

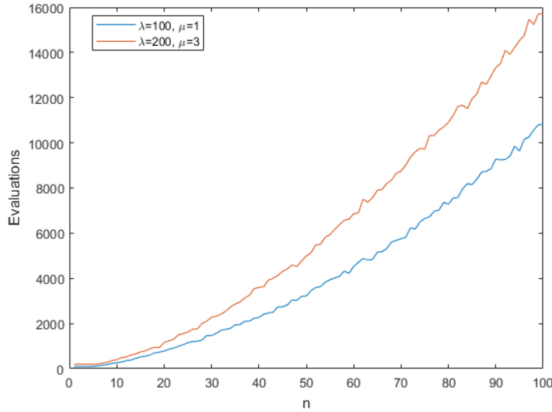


Figure 5.8: UMDA with $\lambda = 200, \mu = 3$ and $\lambda = 100, \mu = 1$ on LeadingOnes for $n = 1, ..., 100$

5.1.3 Jump

In this section I will test the Jump problem. I will test with an initial jump size of $k = 5$ and experiment with the best solutions so see how they fare for larger sizes of k. A smaller k is less interesting as it will start to mimic the OneMax problem.

Since the Jump problem mimics the OneMax function outside of the gap, an initial assumption is that a configuration that worked well on OneMax will also work on the Jump function. Testing with different configurations for UMDA on the initial jump problem reveals that it is almost too difficult. Configurations with low λ simply cannot complete it. This does not come as a big surprise since a too low λ also had trouble completing OneMax. For configurations with higher λ, the algorithm can complete the problem with only few failures. However, the running time of the algorithm is off the charts. For a problem size 10 with gap size 5, it is already more than a million evaluations and it only requires more evaluations for larger gap sizes.

Lowering k to 3 should allow the algorithm to perform better as it is then more akin to OneMax. A test with the same configuration as above shows incredible results.

The blue graph in figure 5.11 looks as if it is fairly unstable, but looking at the y-axis, the difference in evaluations is actually rather small. Because it takes 200 samples per generation, the difference between the runs are likely no more than 1 or 2 generations. (Of course, because this is average, there could be

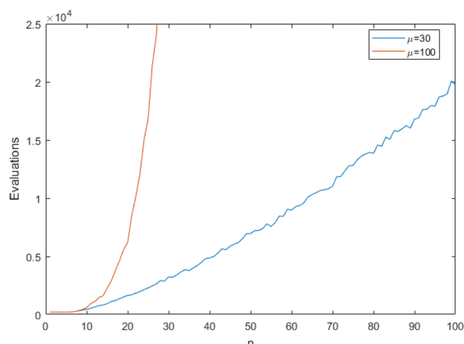


Figure 5.9: UMDA with on LeadingOnes for $n = 1, ..., 100$ with two values of μ with constant $\lambda = 200$

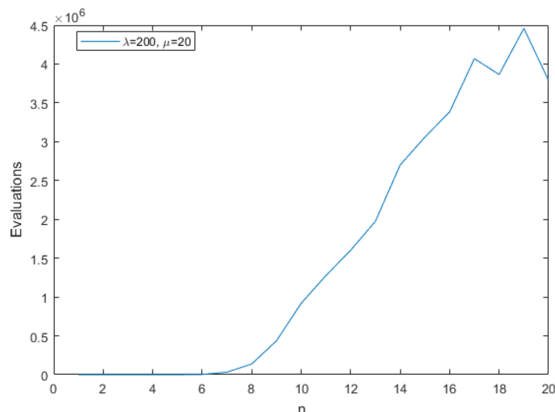


Figure 5.10: UMDA on Jump with $k = 5$ with configuration $\lambda = 200, \mu = 20$

outliers). The instability of the graph is likely because k is constant. For the evolutionary algorithm (1+1) EA, the expected optimization time for Jump is $\Theta(n^{k+1} + nlogn)$ [DJW02]. Here k is the dominating factor for the running time. Because an EDA is based on the theory of evolutionary algorithms, it is possible that the bound of UMDA is similar and could explain the behavior on Jump.

It is interesting to note that the runtime actually decreases a lot at the higher problem sizes. This could possibly just be variance in the probabilities, but it is also possible that the higher problem size is the culprit. If a bit-string has a fitness value that is in the gap, then there will only be a few zeroes if the problem size is large enough. If the problem size is small, then there are only

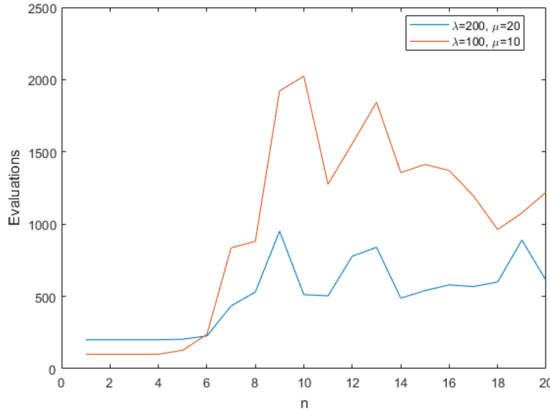


Figure 5.11: UMDA on Jump with $k = 3$ with two sets of parameters.

a few fitness-values that are outside the gap. Recall that if $k = n - 1$ then the problem is essentially Trap. This could explain why the algorithm has this spike, as it is essentially trying to solve the Trap problem in those few instances. And it is much more difficult trying to flip all bits at once instead of being able to flip them one by one. Of course, having a larger problem size means that there are more bits to flip, but this actually helps the problem. If $k = n-1$ then the algorithm might be stuck with low probabilities on all bits, whereas with $n > k$ there will be many permutations for the strings to fall within the gap. That is, the zeros that are in the generated solutions are possibly spread out in the bit-string compared to the situation where $k = n-1$ where the zeros occupy almost all positions. This allows UMDA to keep a generally higher probability for all bits which allows it to more easily sample optimum despite the larger amount of bits.

Since for OneMax, keeping a low λ gave better results, I contrast it with the red graph in figure 5.11 which has a lower λ and roughly same proportion between λ and μ. This configuration also performs rather well, all things considered, but it is still slower. Interestingly it seems as it is roughly a factor 2 slower. Perhaps having an even higher λ will provide even better results.

Figure 5.12 shows the configuration for an even higher λ. Interestingly, the graph is fairly stable and seems to scale with n quite a bit. The reason is possibly that gap is small enough so that it does not act like a "wall" that $k = 5$ does. With a smaller k, the probabilities can get closer to optimum before the jump is encountered. This, combined with the high $\lambda = 400$, possibly results in that the algorithm samples optimum as soon as it reaches the gap. This essentially "eliminates" the jump and the problem is thus reduced to OneMax.

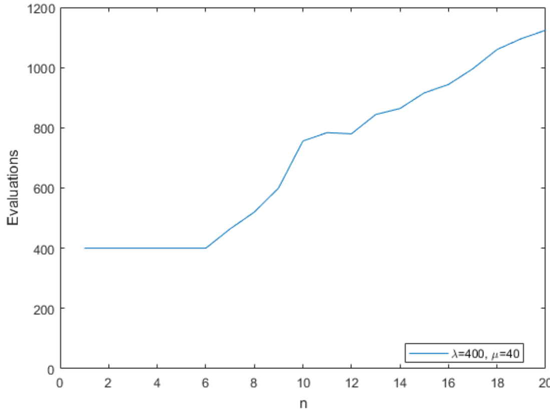


Figure 5.12: UMDA on Jump with $k = 3$ with configuration $\lambda = 400, \mu = 40$

This coincides with figure 5.11 where the higher λ generally has a faster runtime, requiring fewer generations on average to sample optimum once the jump is reached.

In order to get a better understanding of the relation between different configurations of UMDA and k, tests were run for different values of k and a constant n of 20.

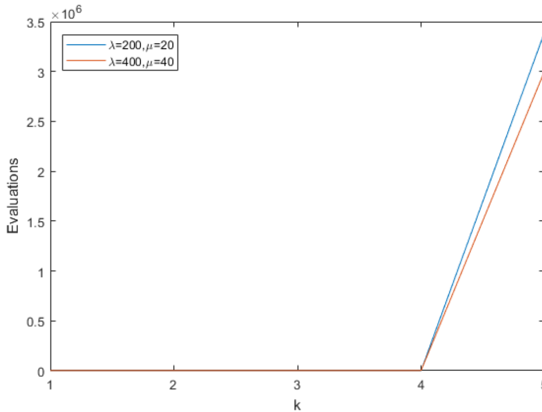


Figure 5.13: Two configurations of UMDA on Jump for different values of k

Finally it may be possible that an even higher λ could allow the algorithm to perform on problems with higher k. Figure 5.14 shows a test for this configuration for $k = 6$. The algorithm can perform decently on this, but still has several failures. Going any higher causes it to never terminate. Having a higher λ did

allow the algorithm to perform on a higher k, technically, but if sampling 10 times more solutions only allow an increase of k of 1, the amount of solutions required to go higher would likely cause the minimum required evaluations to be too high.

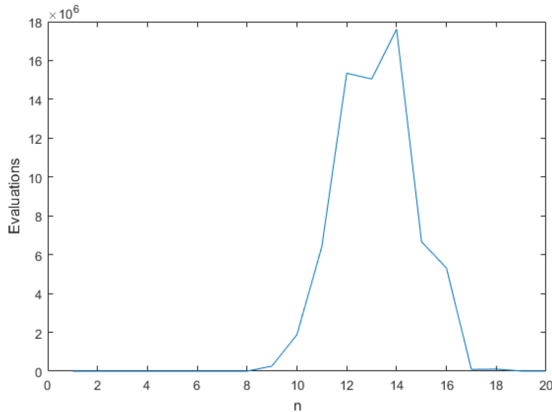


Figure 5.14: UMDA on Jump with $k = 6$ with configuration $\lambda = 2000, \mu = 200$

5.1.4 Trap

Trap can only be done for very small problem sizes. Any larger than 10 and the chance for the algorithm to terminate becomes very small. A test run for $\lambda = 200, \mu = 30$ which has worked decently well for the other problems fails to terminate for these small problem sizes. For a run from $n = 1, ..., 10$, there was a total of 158 failed attempts. This test run fails zero times up to $n = 6$ and only once for $n = 7$. For $n = 8$ it fails so many times that it only managed to complete very few times. That is why figure 5.15 has a spike. For $n = 8$ the algorithm either terminated after one generation or did not terminate at all.

To test if this limit can be raised, I tried with $\lambda = 2000, \mu = 10$ and it does indeed raise the limit to $n = 9$. It does not even fail once on $n = 8$. However, the graph is flat, which means that it samples optimum in the first generation. This makes sense since it generates 10 times more samples than the previous test. As soon as it does not sample optimum in the first generation, it converges away from optimum and fails to terminate. However, since UMDA keeps the probability within a limit, there is a chance (albeit small) to sample optimum even if it converged away from it.

The reason that UMDA fails zero times and then suddenly starts failing all

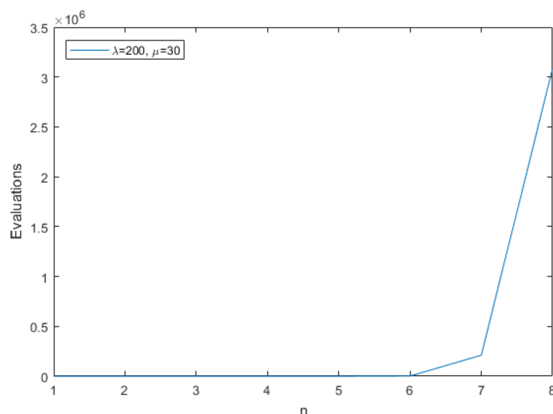


Figure 5.15: UMDA on Trap with configuration $\lambda = 200, \mu = 30$

the time is due to the probability limit. The limit scales with the problem size, becoming smaller as the problem size increases. This means that if the algorithm converges away from optimum, it not only has to generate longer solutions but also that the probability for each bit is smaller, making it even harder to sample optimum. With n bits, the probability can be as low as $(1/n)^n$.

5.2 cGA

5.2.1 OneMax

For cGA, the only parameter that can be tuned is the update strength. Tuning it is a trade-off between guarantee of reaching optimum and speed. A high update strength will allow cGA to reach optimum quickly but could also lead to genetic drift. A small update strength will slow the algorithm down, but will also avoid genetic drift. Sudholt and Witt suggest using an update strength of $S \sim 1/\sqrt{n}logn$ with n being the problem size. With a problem size of 100 this gives an update strength of 1/46. This update strength should lead to an expected runtimme of $\Theta(nlogn)$.

The graph has a mostly linear look which asymptotically sounds better than the expected runtime, however, the graph is still steeper than the expected time. It is possible that it may be closer to the expected runtime for larger problem sizes.

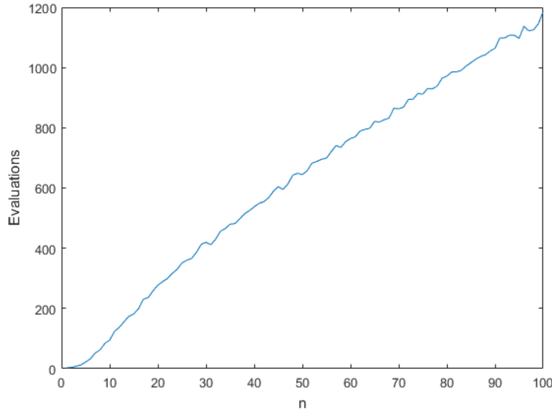


Figure 5.16: cGA on OneMax with an update strength of 1/46 for $n = 1, ..., 100$

Update Strength	Evaluations
1/10	1.099
1/100	2.109
1/500	9.564
1/1000	18.703

Table 5.1: The runtime for cGA on a 100-bit OneMax problem with different update strengths

Despite Sudholt and Witt's claim that high update strengths lead to genetic drift, an update strength higher than the previously suggested seem to yield promising results. An update strength as high as 1/10 works but leads to a slight genetic drift. The run is more unstable than with a small update strength and increasing the update strength to 1/5 leads to a very unstable run. Surprisingly, the algorithm does not start to fail (i.e. it always finds a solution after less than 10.000 generations) before an update strength of 1/3. However, the large variance between the runs does increase the average amount of evaluations required. This seems to be in accordance with [HLG99] who also finds that a high update strength provides the fastest convergence. However, they also suggest that an update strength lower than 1/100 is required to provide enough consistency to avoid genetic drift. Table 5.1 shows the running time for cga on OneMax for different update strengths, including update strengths smaller than 1/100.

For update strengths lower than 1/100, the graph is still mostly linear similar to figure 5.16, but requires more evaluations. They are, however, "cleaner" in

the sense that the probabilistic variance is smaller.

Since the amount of evaluations for cGA does not change with any input parameters, the best choice will probably depend on the situation. If the fastest convergence is wanted, then a high update strength is a good choice, but if genetic drift is very undesirable, then a lower update strength will work better.

5.2.2 LeadingOnes

This problem is much harder for cGA than OneMax. A preliminary run with update strengt 1/100 gives a polynomial running time. Testing with a high update strength, which worked well for OneMax does not work as well for LeadingOnes. It also has a polynomial running time and starts failing even before 50.

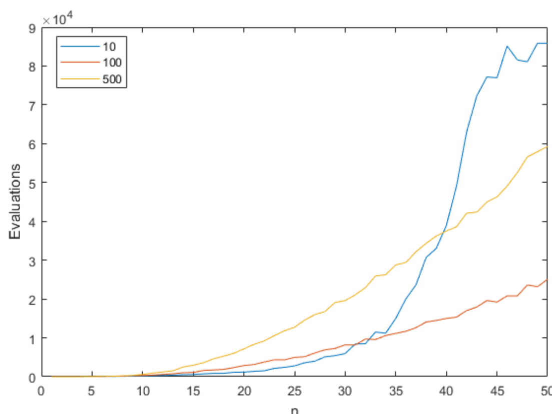


Figure 5.17: cGA on LeadingOnes for $n = 1, ..., 100$ for three different update strengths

The graph shows that the running times for the different update strengths are all polynomial of varying degree. A balanced choice of 1/100 seems to works best.

5.2.3 Jump

The differences in parameters that leads to good solutions for OneMax and LeadingOnes are *very* different. Thus it may be difficult to know what to use

as a starting point for Jump. Therefore, I performed three tests with varying parameters to get an overview of the solutions.

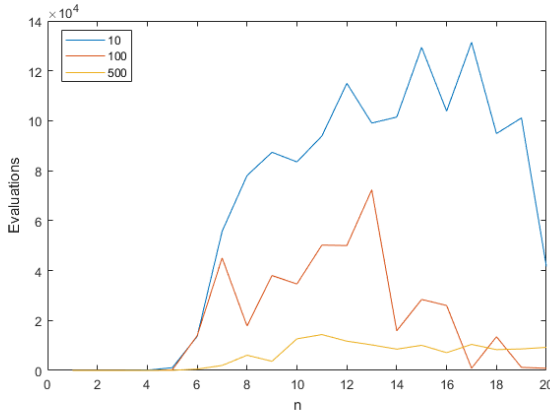


Figure 5.18: cGA on Jump with gap size 5 for varying update strengths

The difference in running time between these graphs is quite large. In addition, it seems that the larger problem sizes are easier to solve. This would be similar to the observation made for UMDA about how Jump with a problem size of $n = k+1$ essentially is Trap. This, however, may not be true. All three variants of the algorithm fail many times. 10 and 100 both fails over 1000 times (out of 2000 total runs) which very likely skews the graph to a large degree. The smallest amount of failures is for 500 which clocks in at "only" 600 failures. This is still far too many.

In order to find a configuration that will allow only few (or ideally, zero) failures, lowering the jump gap is an obvious choice. Lowering the gap size to 3 already provides massive improvements. For the same three update strengths used above, the algorithm does not fail. What is interesting though, is that a low update strength provide much better results than a high update strength.

Because cGA has such poor performance on Jump with gap size 5, running experiments on higher jump sizes would provide no valuable data and is thus left out. It is assumed that the limit of cGA is 5.

It is interesting to see that for small values of k, the high update strength is best, presumably because it bridges the gap easily, whereas large gaps are best solved using a low update strength.

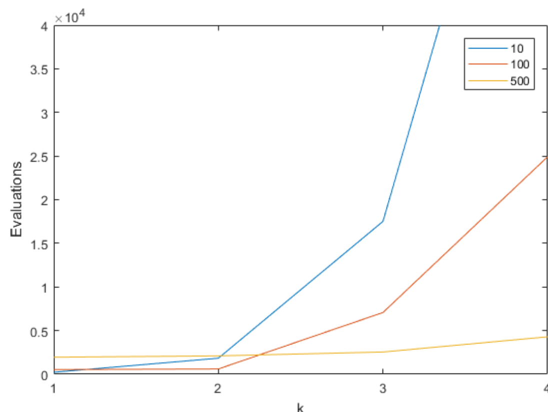


Figure 5.19: cGA as a function of k for varying update strengths

5.2.4 Trap

On Trap, a low update strength seems to be the best choice. Because of the design of Trap, if an algorithm does not sample optimum in a generation, it is very likely to converge away from optimum, making it harder to sample optimum in later generations. Because of this, low update strengths work best since they converge away slower. Ideally an update strength of 0 is chosen, but as explained in the beginning of the chapter, that choice would defeat the purpose of the algorithms.

The variance between runs are extreme as some may find optimum in the first generation and some may find it in generation 50.000 despite the low odds. It seems to work with some consistency up until problem size 6. In size 7 (and definitely 8) it fails a lot.

5.2.5 Similarity with UMDA

cGA and UMDa are fairly similar in their implementation, but varies in the input configurations. The results for OneMax also have some similarity. For UMDA a lower λ with correct choice of μ game somewhat the same results as cGA with high update strengths. The graphs are also both radical/linear, showing that they have roughly the same running time. However, for small enough λ and μ for UMDA and with appropriately high update strength for cGA, there is a small difference. UMDA can keep a linear graph while cGA tends towards a more exponential graph. Choosing a lower update strength for

cGA can put it into linear territory, however, it is still steeper than UMDA

For LeadingOnes they are both polynomial of varying degree. There are specific configurations where they perform very similarly, e.g. $\lambda = 200, \mu = 30$ for UMDA and and update strength of 1/100 for cGA has almost the same running time.

In Jump, the previously discussed Trap-scenario occurs for both cGA and UMDA, though it is more prevalent in cGA as seen in figure 5.18. UMDA does seem to perform better, having an overall lower running time, but the similarity in the graph show that these algorithms are, indeed, similar.

When it comes to Trap, UMDA has the higher limit of $n = 8$ whereas cGA is limited to $n = 5$. This is likely due to the fact that UMDA can have a larger sample size, which increases its chances of sampling optimum in the first generation, while cGA uses only two samples. Of course, having a lower update strength helps cGA to have a higher probability in later generations, but the decrease in probability is still large enough that problem sizes larger than $n = 5$ prove to be difficult.

5.3 BOA

BOA actually uses very complex operations which causes it to take much longer to complete than other algorithms. In order for the runs to be complete within a reasonable time frame, I will have to cut down on the size of the runs, using a problem size of at most 50 and in many cases only 20. This may lead to skewed results since it will be difficult to see patterns in the algorithm for larger problem sizes, but the overall structure of the graph should still show itself. The growth of running time for up to $n = 20$ is often enough to draw conclusion about the running time for $n = 100$ when comparing to other algorithms.

5.3.1 Networks

This section will introduce the networks for the different functions and suggest possible alternatives. The networks are considered simple probability tables, such that if b depends on a, then b will have specific probabilities for each possible value of a. In our case, because they are binary, this will lead to b having a probability for choosing 1 in case of $a = 0$ and a similar probability in the case of $a = 1$. This means the size of the table will scale with the amount

of incoming edges to a node. Therefore, I will attempt to keep the networks relatively simple.

5.3.1.1 OneMax

For OneMax, all the variables are independent because each bit influences the fitness value independently of the others.

5.3.1.2 LeadingOnes

The structure of LeadingOnes is fairly straight forward. The earlier bits will have an influence over the later bits in that the values of the early bits will have an influence over how the later bits can influence the fitness value. A model where each bit influences all the later bits in the string would be a precise model. However this network would also be overly complex, especially for larger problem sizes. A network that models a very similar structure would a network where each bit influences only the next bit. This means that the first bit still indirectly influences the last bit and thus retains the same structure.

5.3.1.3 Jump

Since Jump is so similar to OneMax, the easy approach is to use the same network. However, the difference between the problems might warrant some structure. The problem is that there is not one situation that leads to being in "the hole". Because it is only connected to the number of ones and not any ones in particular, setting up a precise model for this problem might be detrimental. Because this model is so hard to create accurately, Jump will be given the same model as OneMax.

5.3.1.4 Trap

Trap is an interesting problem. On the one hand, the bits could be considered independent because they can influence the fitness values equally. However, that does not hold true for the special case, optimum. With LeadingOnes, flipping a bit will either change the fitness value, or it will not. For Trap, it will always influence the fitness value, but if flipping that bit moves to or away from the special case, then the influence on the fitness value will be large. Because of that,

Network weight	Evaluations
1/5	1.762
1/10	3.192
1/100	18.760

Table 5.2: Number of evaluations by BOA on a OneMax problem with 20 bits for different values of network weight for constant $\lambda = 200, \mu = 30$

that bit will be dependant on all the others since because the bit can either be the bridge between the special case, or it cannot. However, a complete network is not feasible. Figuring out which connections to drop is difficult since it will bias the network in some fashion. Therefore, in order to not introduce such bias, I will use the empty network initially. However, since there *is* a correlation between the bits and the problem size of Trap usually is not very large, I will make tests with some network configurations. The complete network will be the obvious choice, however, Pelikan only considers directed acyclic graphs in [PGCP99]. In addition, the used python library for the implementation also has this restriction.

5.3.2 OneMax

BOA has three inputs that can be configured, however, two of them are the same as UMDA, namely how many samples to generate and how many to keep. The third parameter is the weight of the solution generated by the network. For UMDA, input values of $\lambda = 200, \mu = 30$ gave the best results. Using these values with different weights for the network yields the results as seen in table 5.2.

If we take a look at the graphs for the above runs in figure 5.20, it is clear that a high network weight decreases the number of evaluations. The graph for $w = 1/5, 1/10$ look similar while $w = 1/100$ seem to be exponential. This is an interesting parallel to cGA in which a high update strength also gave the best results.

It is also interesting examine what happens when μ is decreased, that is, a smaller number of solutions is kept. In order to compare the solutions, a constant network weight of $w = 0.1$ is chosen.

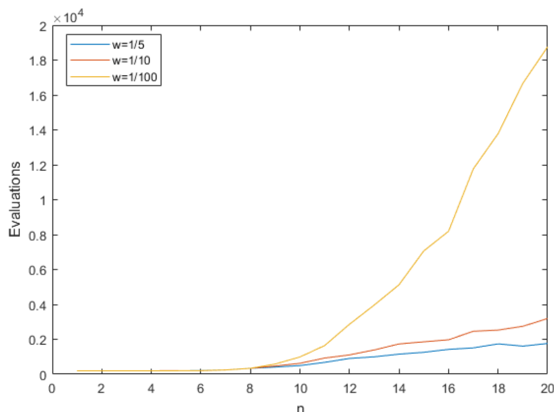


Figure 5.20: Graphs for BOA on OneMax with different weights with constant $\lambda = 200, \mu = 30$

λ	Evaluations
100	1.129
200	1.994

Table 5.3: Evaluations on a 20-bit OneMax problem for different values of λ with $\mu = 1, w = 0.1$

Table 5.3 shows that for a $\mu = 1, w = 0.1$, it is actually better to have a lower λ. While it may converge more consistently for $\lambda = 200$, simply having fewer evaluations turns out to be better. Making the same test, but with different values for the weight might change the outcome.

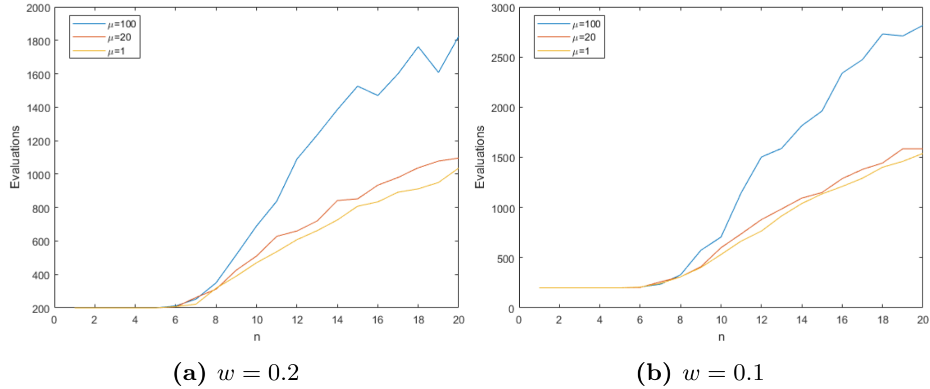
The low λ and high w, as seen in table 5.4 works much better than the other choices, reinforcing the idea that a high update strength is a good choice.

weight	λ	Evaluations
0.01	100	6.849
0.01	200	11.872
0.2	100	661
0.2	200	1.244

Table 5.4: Evaluations on a 20-bit OneMax problem for different values of λ and weight with constant $\mu = 1$

5.3.3 LeadingOnes

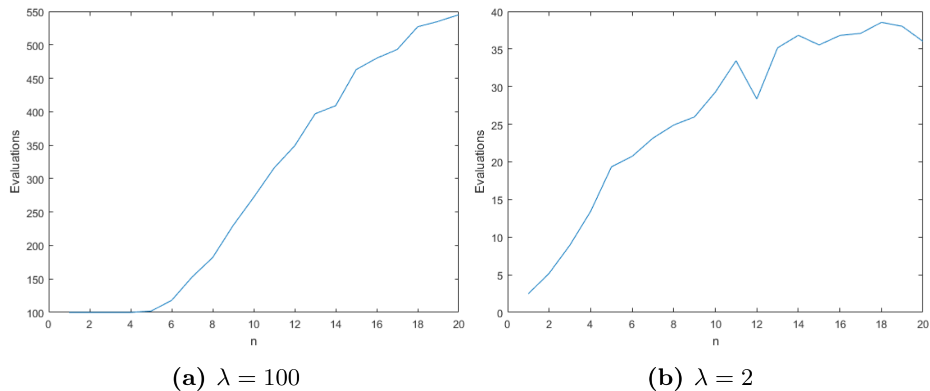
The network for LeadingOnes allows the algorithm to fairly efficiently solve this problem, in contrast to cGA and UMDA. As a first test, using a configuration that worked well for OneMax might perform well for LeadingOnes as well.



(a) $w = 0.2$ **(b)** $w = 0.1$

Figure 5.21: BOA on LeadingOnes with three different μ with constant network weight and $\lambda = 200$

It seems to perform very well, peaking only at 545 evaluations. To further explore the running time of similar configurations, tests with $\lambda = 200$ are performed in figure 5.21a.



(a) $\lambda = 100$ **(b)** $\lambda = 2$

Figure 5.22: BOA on LeadingOnes with $\mu = 1, w = 0.2$ for two different λ

As can be seen in figure 5.21a the low μ outperforms the other configurations. Compared to cGA and UMDA, all configurations perform much better, showing that the Bayesian network model helps out a lot.

The above tests are run with a high network weight of $w = 0.2$. These high weights have proved to be bad for cGA, so it is interesting to see that a more aggressive approach works best when combined with a more accurate representation of the problem. To see if a lower update strength will lead to better runtimes, the above tests are run with the same configuration with only one difference: a lowered weight of $w = 0.1$.

It was initially tempting to test with a higher λ, however, since BOA already terminates after a few generations, a higher λ will only increase the number of evaluations per generation and might not decrease the required number of generations by much. On the other hand, a lower λ seems to have the opposite effect as was seen in figure 5.22a. An even lower λ might have better results. 5.22b depicts BOA with $\lambda = 2, \mu = 1, w = 0.2$

So this is very interesting. These results show that using the lowest possible λ result in the absolute best performance. It does use a few more generations compared to the other runs, but since it uses only a fraction of the evaluations per generation, it results in a staggering performance.

5.3.4 Jump

For Jump, BOA has the empty network like for OneMax. For OneMax, a good choice of parameters was a decently high λ, a low μ and a network strength of 0.2. Testing with similar configurations for Jump with initial gap size $k = 5$ yields fairly promising results.

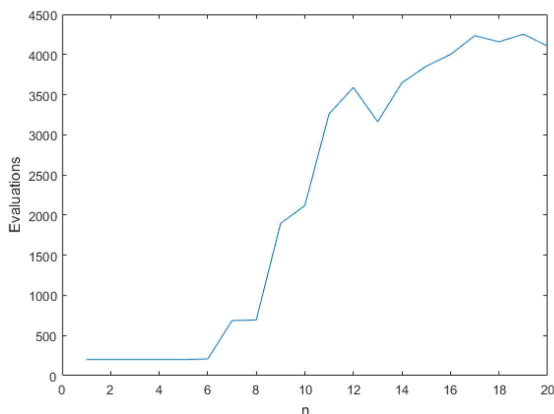


Figure 5.23: BOA on Jump5 with $\lambda = 200, \mu = 20, w = 0.2$

For a low network strength with otherwise similar configurations as above, the run requires many more evaluations. However, there are still no failed runs.

Even a very small samples size of $\lambda = 2, \mu = 1$ gives fairly good results with a high network strength of 0.2. The graph does have a spike, but as was mentioned with UMDA, a higher problem size allows more permutations for bit-strings to be in the gap meaning that it is easier to bridge the gap. Running with higher $\lambda = 100$ but still with $\mu = 1, w = 0.2$ also gives this spike, however with much larger values. This graph also has multiple spikes. However, this is likely due to pure probability since the problem size is not large and it uses a decent sample size.

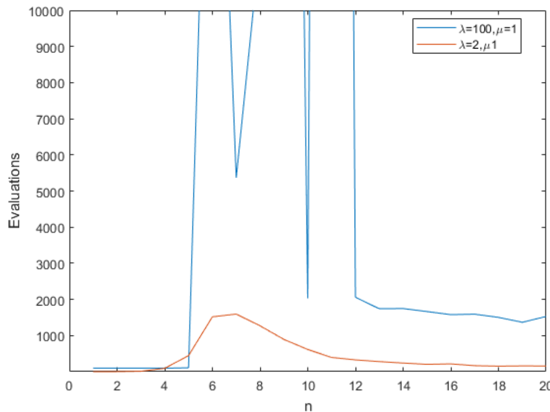


Figure 5.24: BOA on Jump with $k = 5$ for two configurations: $\lambda = 2, \mu = 1$, and $\lambda = 100, \mu = 1$ both with $w = 0.2$

Testing for even higher network strengths shows that the algorithm still performs very well in terms of runtime for $n = 20$, however, in the spike, it performs much worse. Figure 5.25 shows two such examples. They perform very similarly and with the probabilistic variance it is difficult to ascertain which configuration is superior.

To contrast the rather large jump, tests with smaller jumps are conducted.

As was the case for the higher jumps, the low λ is also the best for smaller jumps. Testing with an even higher network strengths for low jump sizes reveal that (at least for a high λ) the higher weight might be better. As explained earlier, a too high update strength will lead to genetic drift and might therefore be undesirable in some scenarios.

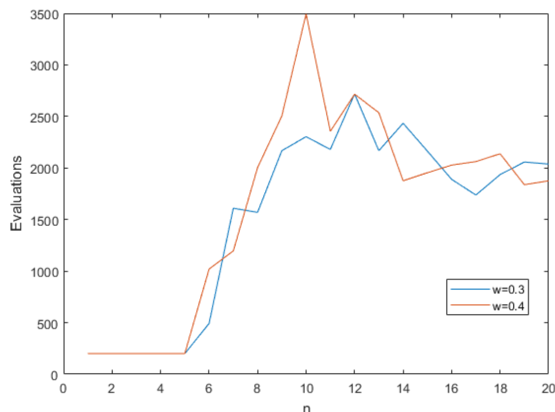


Figure 5.25: BOA on Jump with $k = 5$ with two weights, both with constant $\lambda = 200, \mu = 20$

The best configuration of BOA for Jump has been $\lambda = 2, \mu = 1, w = 0.2$. This configuration is tested for different jump sizes as shown in 5.28. The algorithm performs very well even for large $k = 7$.

5.3.5 Trap

Since the initial network for Jump and Trap is the same, the solutions that worked well for Jump are likely also good solutions for Trap.

The low λ performs best on Trap for BOA. Figure 5.29a shows its comparison to a high λ. The high λ also fails its run a few times for $n = 7$ and many times for $n = 8$.

However, considering the previously discussed point that low update strengths work best since Trap always converges away from optimum, having a high network weight of $w = 0.2$ might be a detriment. Instead, the same configurations are tested, but this time with a weight of $w = 0.01$ instead.

Figure 5.29b shows the same two runs as previously, but instead with a much lower weight of the network. The graph shows the expected results, that it is opposite of the first graph. More samples work better because it has a larger chance to sample optimum before converging away from it.

Since the constraint of BOA is that it must use acyclic graphs, a complete graph

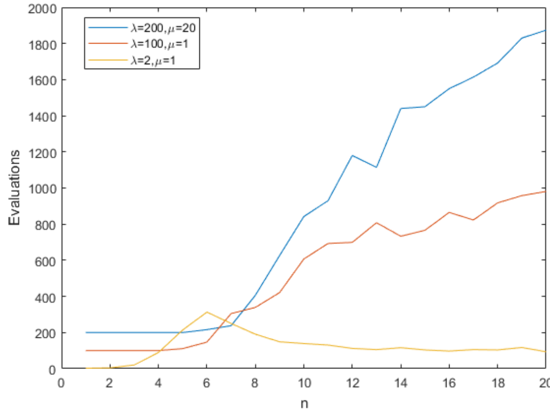


Figure 5.26: BOA on Jump with $k = 3$ with three configurations $\lambda = 200, \mu = 20$, $\lambda = 100, \mu = 1$ and $\lambda = 2, \mu = 1$. All with constant $w = 0.2$

is not possible. However, a somewhat complete graph can be constructed so that each node directly influences all the nodes ahead of it. This means that the earlier nodes will have a larger amount of control than the later nodes, but it is impossible to create an acyclic connected network without this imbalance. This means that it might not provide accurate results but it can be used as a guideline for the way that such a network might perform. A test of this network with a similar configuration as described above ($lambda = 200, \mu = 30, w = 0.01$) shows some interesting results. For all problem sizes $n \leq 8$ it samples optimum in the first generation. However, it also had a lot of failures, especially for $n = 7$ and $n = 8$. This shows that the network might actually make the algorithm worse since it *never* samples optimum after the first generation. Testing with larger a larger $n = 9$, the algorithm never terminates and fails 100% of the time.

With the empty network, there were no failures for up to $n = 8$. This is interesting because this network is much closer to the complete network which is believed to be the best network to use in this case, or at least the one that most accurately models the problem. Using a network that is close to complete seems to introduce some kind of bias that stops the algorithm from performing well and that using the exact opposite structure, that is the empty network, works best. Of course, it can also be argued that the empty network is the closest related network to the complete network, seeing as they are direct opposites without any kind of bias on any of the nodes, which seems to be the case.

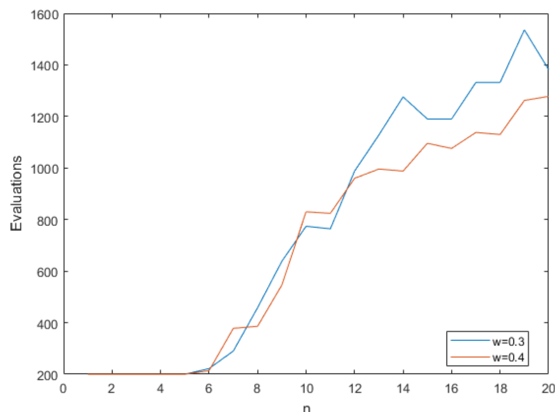


Figure 5.27: BOA on Jump with $k = 3$ with two weights, both with constant $\lambda = 200, \mu = 20$

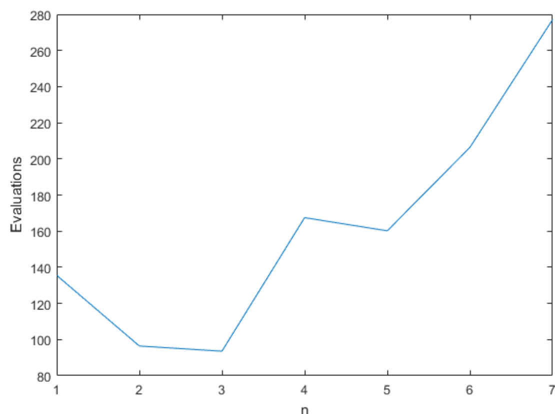
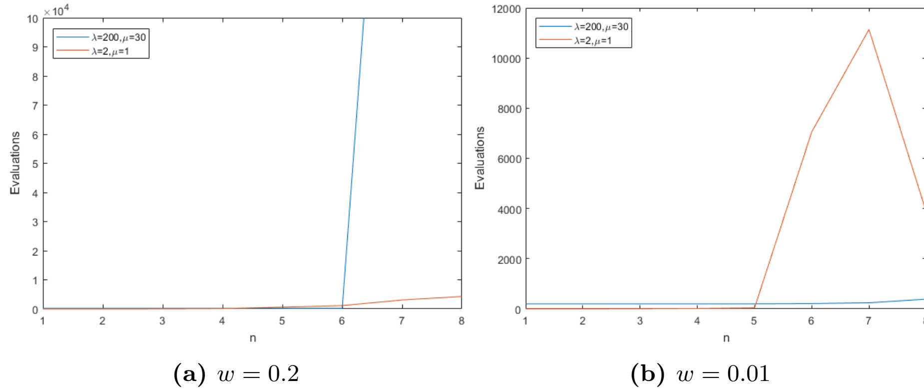


Figure 5.28: BOA on Jump for varying k with configuration $\lambda = 2, \mu = 1, w = 0.2$

5.4 MMAS

The initial probability is set to maximum in [SH00] is both good and bad. In some cases it may be seen as cheating (in the case of OneMax, since the probability of sampling optimum in the first iteration is quite high) or a super bad initial probability in the case of ZeroMax[1]. Therefore, choosing the uniform probability of $1/2$ is preferred.

[1]This problem is identical but reciprocal to OneMax



(a) $w = 0.2$ **(b)** $w = 0.01$

Figure 5.29: BOA on Trap for two configurations for different network weights

5.4.1 OneMax

Since UMDA on OneMax works best with a high number of samples, a starting point for an analysis would be to test it with a large number of ants and an appropriately chosen value for ρ. However, in [NSW10] it is proven that the algorithm can efficiently solve OneMax with just two ants. To accompany it, a low value of ρ must be chosen. Specifically, for $\rho = 1/(c\sqrt{n}logn)$ for some constant $c > 0$, the running time will be bounded by $O(nlogn)$. The tests I run are often of the size $n = 1, ..., 100$. For the maximum size $n = 100$, the value becomes $\rho = 0.02$ for $c = 1$. Running a few tests for two ants and ρ values *above* this threshold showcase that the running time becomes either unstable (in the case of $\rho = 0.2$) or will have many failed attempts (in the case of $\rho >= 0.3$). However, there seems to be a phase-transition around $\rho = 0.1$ as can be seen in figure 5.30. This is in line with [SW16] as they mention that "If ρ is chosen unreasonably large, $\rho \geq c^{'}/(logn)$ for some $c^{'} > 0$, the algorithm shows a chaotic behavior and needs exponential time even on this very simple function". Choosing $c^{'} = 1$ for a 100-bit problem gives $\rho \geq 1/log(100) = 0.22$. Therefore it is not unreasonable to see a phase-transition around $\rho = 0.1$. The graph for $\rho = 0.2$ is very unstable and looks quadratic, which is in accordance with Sudholt and Witt.

For $\rho = 0.01$ the graph looks almost radical and more stable (i.e. less variance between runs). Since $\rho = 0.01$ is theorized to have a running time of $O(nlogn)$, a radical running time sounds better on paper, but the numbers tell a different story. For $\rho = 0.1$ it reached a peak of 1.824 evaluations, while for $\rho = 0.01$ it reached a peak of 4.167 evaluations. So for OneMax problems below 100, ρ must be chosen carefully. In addition, for $\rho = 0.2$ the graph looks polynomial.

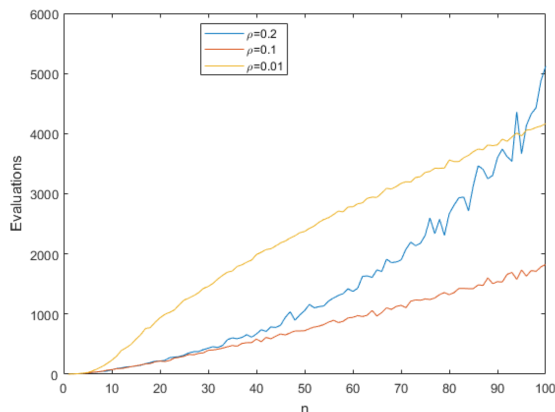


Figure 5.30: 2-MMAS with three different pheromone strengths for OneMax with problem size $n = 1, ..., 100$

ρ	Runtime for $n = 100$	Runtime for $n = 200$	Runtime for $n = 300$
0.2	.5122	70.901	133.426
0.1	1.824	4.116	6.884
0.01	4.167	6.809	9.121

Table 5.5: Runtime for 2-MMAS on OneMax for large problem sizes for three different pheromone values

This is mentioned by [NSW10] who says "For two ants, this gives an exponential lower bound if $\rho \geq 1/(c^{'} logn)$". For $c^{'} = 1$ this gives $\rho = 0.22$. The tested value does not exceed this, but it is likely that it is in the phase transition between logarithmic and exponential. Since $\rho = 0.1$ has the best running time, there must be some optimum at or around that value. Before searching for this value, I will take a look at the radical graph. The linear is still better, but perhaps the logarithmic will win for larger problem sizes.

The difference between 0.1 and 0.01 seem to be about the same for all problem sizes. So if the radical scaling of 0.01 is supposed to overtake 0.1, it either requires a much larger problem size, or it will never happen at all. This suggests that the running times for $\rho = 0.1, 0.01$ is not exactly radical or exactly $nlogn$.

Since the gap between the two is very much the same, it is unlikely that it will ever happen, meaning that a pheromone value of 0.1 for 2-MMAS is the best choice for all cases in OneMax.

Testing different values for ρ around 0.1 revealed that the optimum is truly

ρ	Runtime for $n = 100$
0.15	2.113
0.12	1.822
0.1	1.824
0.08	1.804
0.07	1.837
0.05	1.854
0.03	1.923

Table 5.6: The runtime for 2-MMAS on OneMax for $n = 100$ for different pheromone values around 0.1

around 0.1. The tests show that 0.08 is the best configuration for $n = 100$, however, due to pure randomness in the probability, it is impossible to say if this will hold true in all cases. The runtimes in table 5.6 are the average of 100 tests, so there is *some* truth to it, however, the differences are so small that they could easily be within some error-margin.

The above tests assume that "a few ants are enough", as is the title of [NSW10]. To contrast this, testing with similar pheromone values for more ants should show if two ants are the better choice.

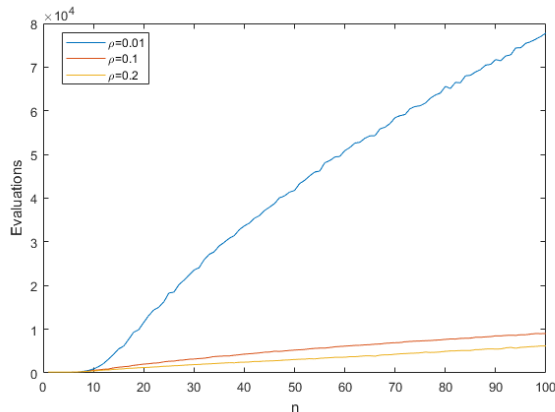


Figure 5.31: 200-MMAS with three values of ρ on OneMax with problem size $n = 1, ..., 100$

This test with 200 ants and the previously discovered optimal pheromone yields a mostly linear but steeper graph than 2-MMAS. It reaches a peak of 9.018 evaluations, about 4 to 5 times higher than for 2-MMAS.

ρ	Runtime for $n = 100$
0.2	26.494
0.1	35.514
0.01	223.680

Table 5.7: Runtime for a 100-bit LeadingOnes problem with 200 ants and different values for ρ

5.4.2 LeadingOnes

LeadingOnes requires a lot more evaluations than OneMax for the algorithm to reach optimum as was also the case with the other algorithms. The pheromone update strength of 1/10 was approximately the best update strength for OneMax. However, I suspect that more ants is a good idea since it will increase the average fitness value per generation. A preliminary test for two ants with pheromone strength $\rho = 0.1$ on a 100-bit LeadingOnes problem fails. In above 90% of the runs the algorithm does not find a solution and is cut off after 100.000 generations. However, testing instead with 200 ants (and same pheromone) allows the algorithm to terminate with 100% success rate. Perhaps the low number of ants will terminate after more than 100.000 generations, but that is already above 200.000 evaluations, which is far from optimal. The test with 200 ants requires 35.000 evaluations for a 100-bit problem. Additional tests for 200 ants can be seen in table 5.7.

Since 200 ants are better than two ants, more ants might provide an even better algorithm. However, testing with 500 ants (and 0.2 pheromone) gives 47.545 evaluations for $n = 100$, showing that more ants are not always better. Maybe the opposite is true? That two ants are simply too low a number to give a meaningful algorithm but 200 ants are a bit too much. Testing with 100 ants instead of 200 (and still 0.2 pheromone) reveals that it takes 17.760 evaluations for $n = 100$, which is better than with 200 ants. It is possible that a lower ant count will always have a better runtime, provided that it is above some possible phase-transition. Going too low, as seen with two ants, will bring it below the phase-transition.

It might be possible that there is some correlation between ρ and the number of ants. For 200 ants, the optimal ρ seems to be around 0.2 whereas this might be different for a different number of ants. Table 5.8 compares two different amounts of ants with varying values of ρ.

The low number of ants with high ρ might seem better, but as previously discussed, these runs are likely to be affected by genetic drift. However, considering

Ants	ρ	Runtime for $n = 100$
50	0.3	11.604
50	0.2	13.172
50	0.1	16.424
50	0.01	84.820
10	0.3	8.891
10	0.2	9.306
10	0.1	10.533
10	0.01	39.940

Table 5.8: Runtime for a 100-bit LeadingOnes problem with 50 ants and different values for ρ

that only the first string of ones in the solution determines the fitness value, the selection process does not care about how they are organized or the number of ones or zeroes. That is, the probabilities of the later bits in the string are only affected by pure probability until they become a part of the string of ones in which case they will, on average, be pulled towards 1. This would explain why a high update strength works best because it possibly has to work through the bits one by one. That is, it cannot determine the correct value for the later bits as long as the correct values for the first bits are determined. It will have to increase the probability of the first bits so they will be sampled as 1 the majority of the time. The high update strength might speed up this process.

5.4.3 Jump

Since the fitness function for Jump has much in common with OneMax, initial tests will be based around a good configuration for OneMax with two ants. Testing with different values of pheromone reveals that this configuration is extremely inconsistent.

The graph in figure 5.32 is of course flat before n reaches 5 because the gap is not present before. After that, however, the runtime quickly rises to high levels and becomes very inconsistent. However, it seems that it has much the same runtime for larger problem sizes. This makes sense since the gap of the jump does not change in size. Of course, because of this inconsistency it is difficult to say if this is really true. This may also be because of the cut-off implemented in the program. If a run reaches 100.000 generations, it is stopped and not counted into the final average. The run also had 922 failed attempts, suggesting that this could very much be the case.

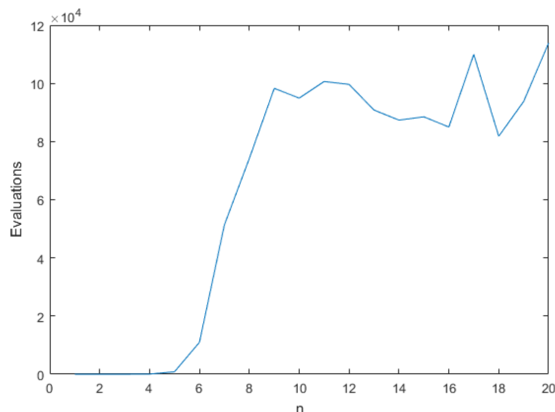


Figure 5.32: MMAS on Jump with $k = 5$ for problem sizes $n = 1, ..., 20$ with 2 ants and pheromone value 0.1

Because of the large number of failures for two ants, it may be possible that a lot more ants, e.g. 200, can provide more consistency.

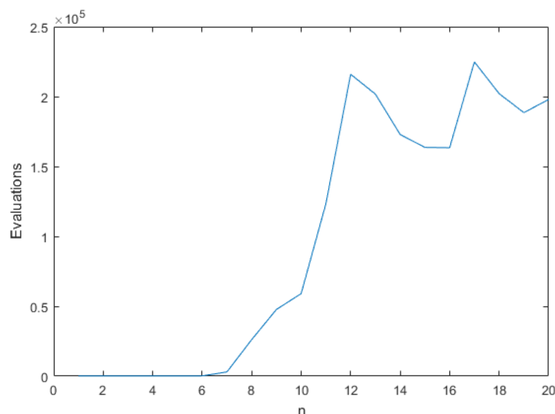


Figure 5.33: MMAS on Jump with gap size 5 for problem sizes $n = 1, ..., 20$ with 200 ants and pheromone value 0.1

A test with the same pheromone value and problem size for 200 ants reveals that it is indeed more consistent, forming a mostly linear graph that clocks in with at most 2.902 evaluations and zero failed attempts. Sampling more solutions per generation increases the changes that the algorithm samples a solution on the other side of the gap and thus pulls the probabilities in the right direction.

ρ	Runtime for $n = 20$
0.2	972.016
0.1	197.954
0.01	21.976

Table 5.9: Runtime for 2-MMAS on a 20-bit Jump problem with gap size 5 for different values of ρ

Trying to push this configuration with 200 ants with larger with jumps shows that a gap size of 6 is the limit. Any higher and the algorithm starts failing too much. A gap size of 6 can be completed with only minimal failures but the amount of evaluations quickly rises to extreme numbers. This is a possible correlation between the gap size and the number of ants required to reliably find optimum without failing. That is, for a jump size of k, there may be a number of ants n and a corresponding ρ that can complete the problem without failing. This would require a theoretical analysis, which is not part of this project.

The initial choice of pheromone for 200 ants was chosen as a balance. Neither too high nor too low. Of course, there could be other pheromone values that perform better.

Interestingly, the low pheromone update provides the best results. One would think that a high pheromone might be best since it will allow the algorithm to jump over the gap easier instead of being stuck in the local optimum. However, for MMAS, the pheromone strength also constitutes to the decay of the pheromone . Having a high pheromone value will also make the algorithm forget its other solution faster. Even though it updates slowly, it seems that remembering the good solution and building upon that is a better idea than trying to bridge the gap in a few generations.

Since two ants performed well on OneMax lowering the gap size to 3 (from 5) might be enough for two ants to be able to terminate without failing. Using the same parameters as before, it turns out that two ants can indeed perform well.

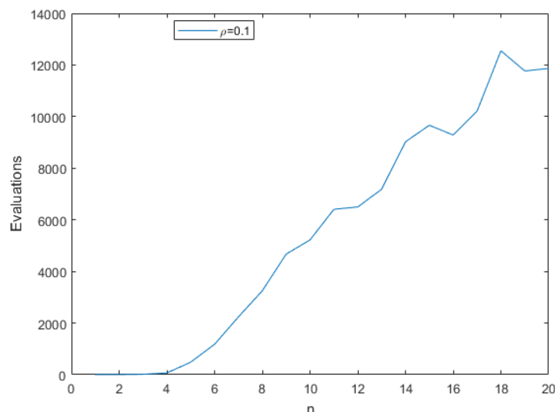


Figure 5.34: MMAS on Jump with gap size 3 for problem sizes $n = 1, ..., 20$ with 2 ants and pheromone value 0.1

The graph is still unstable, but not as much as before. It has also taken a linear shape, which makes sense since 2-MMAS with the same parameters was also linear on OneMax. It still requires a lot of evaluations, suggesting that a high number of ants might still perform better, even on a small gap size. Testing with 200 ants on the same gap size does indeed reveal that it performs much better. 2-MMAS requires above 100.000 evaluations for larger problem sizes whereas 200-MMAS only reaches a peak of almost 3.000 evaluations.

Since the running time of the algorithm scales more with k, and not with n, figure 5.35 then shows the running time for a 20-bit Jump problem for MMAS with 200 ants and 0.1 pheromone as a function of k.

5.4.4 Trap

I started by testing the initial configurations that have worked in the past. As with the other algorithms, any problem size of above 8 is likely not possible for the algorithm to solve.

Using only two ants yield some decent results. As discussed earlier, having a low update strength increases the chances of sampling optimum because it converges away from the initial probability slower. A low pheromone gives an average of

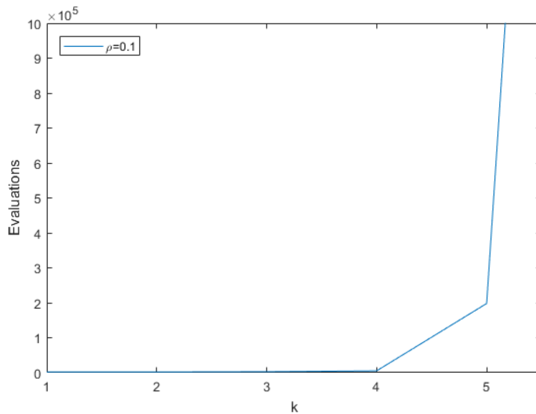


Figure 5.35: 200-MMAS on a 20-bit Jump problem for varrying k

3.000 evaluations for $n = 8$ whereas middle to high values have over 80.000 evaluations. Of course, having more ants will increase the chance of sampling optimum in each iteration.

Testing with 200 ants and low pheromone (0.01) shows staggering results. It reaches a top average of only 336 evaluations and does not fail once. A test to compare it with a similar amount of ants but instead a high pheromone (0.2) shows 2.5 million evaluations for size 8 and a few failures. Trying to push this for up to size 12 causes problems at size 10 and above.

5.5 Summing up

This section notes the conclusions about the best configurations for each of the problems considered in this thesis.

5.5.1 OneMax

The four algorithms had a wide variety of performance when it came to the input parameters. In the best-case scenario for UMDA, cGA and MMAS, they all performed fairly similarly. The exception is BOA. The runtime for BOA was too high, so tests were only conducted for $n \leq 20$. However, the other algorithms outperformed BOA at this problem size. The observed graphs of BOA also indicates that it is not likely to outperform the others for $n = 100$.

The best algorithms for OneMax have been UMDA and cGA whose performance are very close with UMDA only winning with a few evaluations (1099 for cGA vs. 1036 for UMDA). MMAS used at best 1824 evaluations, quite a bit behind the other two. One interesting aspect of MMAS, however, is that the runtime in seconds does not increase too much with higher problem sizes, easily allowing for tests with n up towards 300. This could hint towards MMAS being a good choice for large problems.

5.5.2 LeadingOnes

The difference between the algorithms for OneMax was not significant. LeadingOnes, however, has much larger variance and seems to be opposite of OneMax. For OneMax, UMDA and cGA performed best, but on LeadingOnes they perform the worst, both reaching over 100.000 evaluations for $n = 100$. MMAS is still in the middle with 17.760 evaluations and the exception is again BOA who, in this case, outperforms the other algorithms by an order of magnitude. It reaches only as few as 54 evaluations for a 100-bit LeadingOnes problem. The reason for this is probably the structure of the problem and that the Bayesian network used models the problem perfectly.

5.5.3 Jump

The algorithms share some similar trends on the Jump problem but their performance varies greatly. The algorithms generally seem to have trouble dealing with any jump larger than $k = 5$. Some of the algorithms can deal with $k = 6$ however that requires very specific (and often undesirable) configurations. The algorithm with the slowest performance seemed to be UMDA, requiring a staggering number of evaluations compared to the other algorithms that had varying degrees of success. MMAS had better performance, but still required about 22.000 evaluations. cGA and BOA had the best performances with a little less than 1.000 evaluations and 200 evaluations respectively. The difference between the algorithms is quite extreme. BOA is even able to deal with large jumps going all the way up to $k = 7$ while still maintaining a very low amount of evaluations. It might be possible for BOA to solve problem with an even higher k. Of course, decreasing the size of the jump to 3 allows all of the algorithms to terminate without much trouble as the problem is now more similar to OneMax.

Some interesting observations is the difference in update strength between the algorithms. For cGA and MMAS, a low update strength seems to provide the superior performance while for BOA a high update strength is best. This is

possibly due to the fact that BOA's network allows it to see the general trend in the generated samples, namely many ones. The network is thus likely to generate a string with many ones and since the instance created by the network is not evaluated by fitness, it can possible increase its probabilities above what the other algorithms are capable of.

Some graphs for different algorithms have a spike. Initially this spike was written off as probabilistic variance, but the same pattern kept appearing throughout the project. The reason is that for a jump size $k = 5$, having $n = 6$ will result in a trap problem. A trap problem with $n = 6$ is much more difficult to solve than the Jump problem with $n = 20, k = 5$.

5.5.4 Trap

When comparing the algorithms on the Trap problem, the difference is likely to be located mostly in how large problem sizes they can solve. The reason is that it is likely to sample optimum in the first generation due to the small problem size. If, however, it does not, it will converge away and will thus likely never sample optimum. This changes of course if very small update strengths are used since the probabilities are kept somewhat high for at least a few generations. However, comparing the number of evaluations between the algorithms can sometimes be misleading. If, say, the probabilities for two algorithms have converged to minimum, one may sample optimum in generation 50 whereas the other may sample optimum in generation 50.000. This is based purely on probability. However, the average number of evaluations can still provide information about how consistently the algorithm samples optimum within the first few generations.

That said, the difference between the algorithms can mostly be seen in how large problem sizes they can solve. The algorithms all seem to have a limit, that is, a problem size they can solve without much trouble. Increasing the problem size above this limit results in the algorithm only terminating in a fraction of the runs. cGA performs worst and can only solve trap problems of up to $n = 6$. MMAS and UMDA can both be pushed to $n = 10$. However, UMDA requires a very high λ in order to do so, which simply forces UMDA to sample optimum in the first generation. BOA can also perform on up to $n = 10$ with the empty network. With the attempted network implemented, the algorithm actually performed worse, showing that the closest emulation to a complete graph may actually be the empty graph. The best performance were reached with MMAS and BOA both sample 200 solutions per generation. Considering that they have the same limit they are considered equally good.

Naturally, UMDA, BOA and MMAS can all be "pushed" to complete larger trap problems by simply generating a lot of solutions and sampling optimum in the first generation. But due to the fact that evaluating solutions is expensive, this possibility is not considered.

CHAPTER 6

Further Work

In some experiments, there were situations where two input configurations resulted in two different types of graphs. For example there was a radical graph and a linear graph. In most of those cases, the linear graph had the superior running time. However, since the experiments was only run with a limited n, it may be possible that the radical graph would obtain a better running time than the linear for large enough problems sizes.

Through the experiments of MMAS on Jump it was discovered that it may be possible to determine a formula for the configurations of MMAS and a corresponding jump size k that the configuration will be able to complete. Such a theoretical analysis could be conducted.

The projects scope obviously limits the exploration of these algorithms to a handful of scenarios. Going outside of these boundaries could give a better insight into the algorithms and how they compare to each other. These boundaries could be the inclusion of cyclic networks for BOA, different termination criteria for the algorithms, and different benchmark problems.

The focus in this thesis was to provide guidelines for the choice of input parameters for the different algorithms in order to achieve the best running time. On the other hand, experiments that try to find the worst possible configuration could also be interesting. As this thesis has focused on a possible lower bound

for the algorithms, another thesis could potentially attempt to find an upper bound.

The algorithms often sample λ solutions each generation and keep μ of them. The μ solutions are all weighted equally, but one could imagine a system where a weight was introduced such that the solutions with higher quality would be weighted higher. The weight distribution would have to be carefully chosen as too much focus on the best solutions could lead to the algorithm finding only a local maximum and not a global maximum.

Additionally, in order to avoid possible local maximum (if a problem is e.g. known to have many of them) one could introduce a different selection algorithm that does not keep the μ best, but instead keeps every second solution until μ solutions are kept. This would reduce the average quality of the fitness value, but it might prevent the algorithm from focusing too much on a local maximum.

One interesting note about the experiments with BOA is that the running times look pretty good compared to the other algorithms. However, the running time in seconds is often longer. This is likely due to the fact that constructing and instantiating a network is a complex operation while it still only counts as one operation. Examining this in further detail has not been done in this thesis, but it could be a possible area of research in order to create a more "fair" comparison between the algorithms.

6.1 Program

The benckmark program has had a few limitations as a result of the scope of the project. To further the understanding of the algorithms, performing experiments outside of these restrictions could be interesting. One of those limitations are functions as user input. Building a math-interpreter would allow the user to specify functions and perform experiments that more accurately reflect the theory used in this thesis.

Chapter 7

Conclusion

In this thesis I have conducted experiments on Estimation-of-Distribution Algorithms. To that end, I implemented a benchmark program that would measure the running time of the algorithms on different problems for different input parameters. The program measures the running time in the number of fitness evaluations plus the number of generations used. Because the algorithms use probabilities, variance between the runs of an algorithm is expected. To counter this, each algorithm is tested 100 times per problem size and the results are then averaged. While variance is still possible (and was somewhat observed), its impact is reduced.

For each problem, I tested the performance of different configurations of the algorithms in order to find the configuration(s) that are close to optimal. I then compared the performance of the different best configurations to find the algorithm most suited for each problem.

For OneMax, the best algorithms were UMDA and cGA with very similar performance. They use only about 1.000 evaluations for a 100-bit problem. MMAS also performed well with 1.800 evaluations. BOA was the worst, using 3.000 evaluations for only a 50-bit problem. Tests were not conducted for a 100-bit problem because the running time was too large.

LeadingOnes is in stark contrast to OneMax with BOA outperforming the other algorithms by a large margin. The difference in running time between the different algorithms is also much larger, with UMDA and cGA using over 100.000 evaluations, MMAS at 17.760 evaluations and BOA using only 54. BOA is able to use its Bayesian Network to model the LeadingOnes problem perfectly, allowing BOA to learn the structure of the best solutions and reach optimum much faster than the other algorithms.

In Jump, the algorithms are all able to terminate for jumps with $k = 5$ with the right configuration. BOA is the only one able to go above and can complete jumps with $k = 7$. The performance of the algorithms varies widely. UMDA can only barely terminate with $k = 5$ while cGA and MMAS uses a high, but still usable number of evaluations. BOA again wins, using only a fraction of the evaluations the other algorithms use. This is interesting because BOA uses the empty network, assuming full independence between the bits, just like the other algorithms.

Trap was a difficult problem to work with because it is designed in such a way that the algorithms will always converge away from optimum (assuming the use of the termination criteria of sampling optimum). Because of that, the best configurations were similar for all the algorithms. Sampling many candidate solutions increases the chance of sampling optimum in the first generation. Having a low update strength decreases the speed at which the algorithm converges away from optimum. I observed that an algorithm either samples optimum in the first generation or does not at all. Sometimes there were outliers that sampled optimum after many generations, but this is just probabilistic variance. Despite the similarity in configurations, MMAS and BOA seemed to perform better, being able to terminate with $n = 8$ whereas UMDA can only reliably terminate with $n = 6$ and cGA performing worst with a limit of $n = 6$.

Finally, the boundaries of the project are explored with extra suggested work that could expand the understanding of the algorithms and their performance.

Bibliography

[DJW02] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theor. Comput. Sci.*, 276(1-2):51–81, 2002.

[DL15] Duc-Cuong Dang and Per Kristian Lehre. Simplified runtime analysis of estimation of distribution algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, pages 513–518, 2015.

[DN10] Benjamin Doerr and Frank Neumann. Optimal Fixed and Adaptive Mutation Rates for the LeadingOnes Problem(Doerr et al, PPSN XI 2010).pdf. pages 1–10, 2010.

[GMB08] Jörn Grahl, Stefan Minner, and Peter A. N. Bosman. Learning structure illuminates black boxes - an introduction to estimation of distribution algorithms. In *Advances in Metaheuristics for Hard Optimization*, pages 365–395. 2008.

[HLG99] Georges R. Harik, Fernando G. Lobo, and David E. Goldberg. The compact genetic algorithm. *IEEE Trans. Evolutionary Computation*, 3(4):287–297, 1999.

[JMS] JMSchrei. Pomegranate. https://github.com/jmschrei/pomegranate.

[KNSW10] Timo Kötzing, Frank Neumann, Dirk Sudholt, and Markus Wagner. Simple max-min ant systems and the optimization of linear pseudo-boolean functions. *CoRR*, abs/1007.4707, 2010.

[KW17] Martin S. Krejca and Carsten Witt. Lower bounds on the run time of the univariate marginal distribution algorithm on onemax. In *Proceedings of the 14th ACM/SIGEVO Conference on Foundations of Genetic Algorithms, FOGA 2017, Copenhagen, Denmark, January 12-15, 2017*, pages 65–79, 2017.

[Müh97] Heinz Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997.

[NSW10] Frank Neumann, Dirk Sudholt, and Carsten Witt. A few ants are enough: ACO with iteration-best update. In *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, pages 63–70, 2010.

[Pel02] Martin Pelikan. Bayesian Optimization Algorithm: From Single Level to Hierarchy. *Doctoral Dissertation*, 2002.

[PGCP99] M Pelikan, D E Goldberg, and E Cantú-Paz. {BOA}: The Bayesian Optimization Algorithms. *Proceedings of the 1st Genetic and Evolutionary Computation Conference, GECCO 1999*, 1:525–532, 1999.

[SH00] Thomas Stützle and Holger H. Hoos. MAX-MIN ant system. *Future Generation Comp. Syst.*, 16(8):889–914, 2000.

[SW16] Dirk Sudholt and Carsten Witt. Update strength in edas and ACO: how to avoid genetic drift. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 61–68, 2016.

[VP] Mathieu Virbel and Gabriel Pettier. Kivy: Cross-platform Python Framework for NUI Development. https://kivy.org/.

[Wit17] Carsten Witt. Upper Bounds on the Runetime of the Univariate Marginal Distribution Algorithm on OneMax. In *Proceedings of the Genetic and Evolutionary Computation Conference*, Berlin, Germany, 2017.