Ant Colony Optimization for Model Checking

van de Put, Elbert J.

Award date:
2020

Master's Thesis

Ant Colony Optimization for Model Checking

E.J. van de Put

Supervisor: E.P. de Vink

February 2020

Eindhoven University of Technology
Department of Mathematics and Computer Science
Formal System Analysis

Contents

Ga naar de mier, luiaard, zie zijn wegen en word wijs.

Spreuken 6:6.

ABSTRACT
In this paper we explore the application of ant colony optimization to model checking. We analyze how ant colony optimization can be applied to the so-called model checking problem. More specifically to Boolean equation systems and parity games. We look at the challenges that arise when applying ant colony optimization to ensuing decision problems and at possible solutions to those challenges.

1. Introduction

For safety-critical systems we want to be able to prove the presence or absence of certain behavior. Some systems can be modeled as a set of states with transitions between them, where every state has a number of propositions that hold in that state. On these models we want to check the presence of 'good' and absence of 'bad' behavior. The model checking problem is the problem of validating whether a model has certain behavior. In our particular case the model is a mixed-Kripke structure, (Kripke, 1963), and the behavior is described by first order modal μ-calculus, as described in Groote and Mousavi (2014). We are interested to see if Ant Colony Optimization can be applied to this problem.

Ant Colony Optimization (ACO) (Dorigo, Maniezzo, & Colorni, 1996) is a stochastic search algorithm that is inspired by the foraging behavior of ants. When an ant finds food it leaves a trail of pheromone that other ants will follow to find the food. In the ACO algorithm artificial ants will generate a candidate solution to the problem and based on the quality of that solution an amount of pheromone is deposited on the components they used to construct the solution, which other ants can smell.

The model checking problem can be solved directly by, for example, the algorithm proposed in Emerson and Lei (1986). The problem can also be transformed to other problems with a simpler structure. It can be transformed into a Boolean equation system, a parity game or the v-parity loop problem in a switching graph. We try to apply Ant Colony Optimization to Boolean equation systems and to parity games. We do not try to solve the model checking problem directly because the model checking problem has a structure that does not lend itself well to applying ant colony optimization.

We start this report with preliminaries. In section 2 we begin with explaining the ACO paradigm. Then we give examples for how ACO can be applied. In section 2.4 we look how it can be applied to the Travelling Salesman Problem, and in section 2.5 we look how it can be applied to boolean satisfiability. Then, in section 2.6 we explain the model checking problem, to see how we can apply ACO. We conclude that we should apply it to problems that are generated from the model checking problem. After that we will present our research. In section 3.2 we will explain the framework we have built to implement ACO. In section 4 we will explain the problem of solving a Boolean equation system, and applying ACO to it. Then, in section 5 we explain the application of ACO on parity games. Finally, in section 6 we explain how we used ACO in a strategy improvement algorithm to solve parity games. We discuss our results in section 7. Implementation of the algorithms can be found in the appendices.

2. Understanding Ant Colony Optimization

Ants live and work in a colony. Ants, and other social insects, communicate using pheromones (Vander Meer, Breed, Winston, & Espelie, 2019). Pheromones are smelly substances which evaporate over time. When an ant has found food it deposits a pheromone. When other ants smell the pheromone they will follow the trail of pheromone. If, by following this trail, they find the food, they will also leave pheromone while walking back to the colony. In this way a trail is maintained if there is food at the end of the trial. If there is no food at the end of the trial no extra pheromone is deposited and the existing pheromone will evaporate.

The ant colony optimization paradigm, as proposed in (Dorigo et al., 1996), is a stochastic optimization algorithm. It is based on the foraging behaviour of ants. In the ant colony optimization paradigm ants are divided in generations. Every ant in a generation constructs a solution candidate for the problem in a stochastic way based on pheromone levels and possibly on heuristics.

Next, we explain ACO in more detail. A good explanation of ACO is given in (Blum & Dorigo, 2004). We follow their approach and notation. Ants generate a solution candidate S by selecting solution components from the set of all solution components $\mathcal{C}$ until they have constructed a complete solution candidate. Ants use a function $g : 2^{\mathcal{C}} \rightarrow 2^{\mathcal{C}}$ to determine which solutions components they are allowed to add to their partial solution candidates. This function prevents ants from creating impossible solutions. Ants pick solution components from the set of potential solution candidates until they cannot add any more solution candidates, i.e. $g(S) = \emptyset$. To pick a solution component from the set of potential solution components a conditional probability is used. By default, this probability only depends on the amount of pheromone on the solution components. The function $\tau : C \rightarrow \mathbb{R}$ gives the amount of pheromone on a solution component. The probability of picking a solution component o_i from a set of solution candidates, given by the set of current solution components S, is $p(o_i|S)$ and is defined as:

$$p(o_i|S) = \begin{cases} \dfrac{\tau(o_i)}{\sum\{\tau(o_j) \mid o_j \in g(S)\}} & \text{if } o_i \in g(S) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Ants can also base their choice on heuristic information about the quality of solution components. This is information that can help guide ants to construct solution candidates which are more likely to have a good quality. For example when we try to find the shortest road from city A to city B we prefer roads that go in the direction of city B, the direction of the road could be used as heuristic information. We use the function $\eta : C \rightarrow \mathbb{R}$ to denote the heuristic value corresponding to a solution component. Using two parameters, α and β, the relative importance of pheromone and heuristic information can be configured. The probability of selecting a solution component becomes:

$$p(o_i|S) = \begin{cases} \dfrac{\eta(o_i)^\alpha \tau(o_i)^\beta}{\sum\{\eta(o_j)^\alpha \tau(o_j)^\beta \mid o_j \in g(S)\}} & \text{if } o_i \in g(S) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

This provides us with enough information on how ants create solution candidates.

We continue by explaining how pheromone is deposited. In ant colony optimization there are generations. Every generation has three phases. In the first phase all ants generate a solution candidate. In the second phase the amount of pheromone from previous generations is reduced to model evaporation. In the third phase pheromone is deposited on solution components for solution candidates produced by ants of the current generation. The amount of pheromone deposited on a solution component is based on the quality of the solution candidates they were used in. The function $f : 2^{\mathcal{C}} \rightarrow \mathbb{R}$ determines the quality of a solution. If a solution is better, its quality value should be higher because then more pheromone is deposited on the solution components. The amount of evaporation is controlled by the parameter ρ, which can take values between 0 and 1. Equation 3 formally describes the process of evaporation and depositing pheromone for a generation that we described above. There are k ants in a generation. A solution candidate for ant i is denoted S_i.

$$\tau(o_j) \leftarrow \rho \cdot \tau(o_j) + \sum_{i=1}^{k} \Delta\tau_j^i \quad \text{where} \quad \Delta\tau_j^i = \begin{cases} f(S_i) & \text{if } o_j \in S_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

In ant colony optimization generations are run until a stopping condition is met. This could be an amount of generations or a specific solution quality is reached. The stopping condition depends on the problem.

There is a difference in quality among the solution candidates generated in a generation. In order to prevent depositing pheromone on 'bad' solution components we deposit pheromones for only a fraction of the best solution candidates.

2.1. Combinatorial Optimization

Ant colony optimization is easily applied to combinatorial optimization problems. These are problems where a solution is created by selecting a set of solution components out of a finite set of solution components. For example, linear programming is not combinatorial optimization because solutions cannot be constructed by picking components from a finite set (the size of $\mathbb{R}$ is infinite). Boolean satisfiability is a combinatorial optimization problem because there are a finite number of components from which a solution can be constructed. In the case of combinatorial optimization pheromone can be applied to every solution component and ants will select a number of solution components to build a solution based on the amount of pheromone on the solution components.

2.2. Local- and Global Optima

Ant colony optimization is a stochastic optimization algorithm. An important capability of stochastic optimization algorithms is the ability to explore solutions that are worse than the currently known best solution. This is important in order not to get stuck at a local optimum. We consider two optimization algorithms that do this exploring of worse solutions explicitly. We explain how they work and after that we discuss how ACO can escape a local optima.

Tabu search (Glover, 1989) is a local search algorithm. In every step of tabu search the direct neighbours of the current solution are explored. The algorithm then moves to the best solution among the neighbours. This process continues until a certain stopping condition is met. A list of solutions that are already explored and are therefore

forbidden (tabu) are maintained in order to not evaluate the same solution twice. By moving to a neighbour even if it is worse than the current position, the algorithm is able to escape local optima.

Simulated annealing (Kirkpatrick, Gelatt, & Vecchi, 1983) is a search algorithm. At every step of this algorithm the current solution is slightly changed and its quality is measured. Based on the quality of the new solution it decides to go to the new solution or to stay at the current solution. The chance of accepting worse solutions is gradually decreased over the run of the algorithm. In this way the algorithm can escape local optima at the start and find the best solution in the optimum it is in at the end.

In ACO solutions that are worse than the best solution can be explored because ants do not always choose the solution components with most pheromone on it. At the start of ACO almost all solutions can be explored because there is almost no pheromone present. When ACO has been running for some generations there are pheromone trails and only solutions that are slight modifications of this trail will be explored.

2.3. Complexity of Ant Colony Optimization

The computational complexity of Ant Colony Optimization algorithms is determined by three variables. The first variable limits in how many steps an ant can construct a solution. This is linear in the number of solution components. The computational complexity of selecting a solution component is at least linear to the amount of solution candidates. The second variable is how many ants there are per generation. From literature it seems that ACO algorithms work best if this is a fraction of the problem size (Dorigo et al., 1996). The third variable is how many generations are required to find an acceptable solution. We do not know how this is related to the problem size.

The conclusion that we can draw from this is that ACO algorithms have a computational complexity of at least $\mathcal{O}(n^3)$. This is if the amount of solution components is linear to the problem size, if the complexity of selecting a solution component is linear to the amount of solution components and if the amount of ants required is linear to the problem size. Because the problems we evaluate are computationally hard, exact algorithms typically have a non-polynomial computational complexity. ACO algorithms can typically outperform exact algorithms on sufficiently large problems.

2.4. Travelling Salesman Problem

In order to gain a better understanding of ant colony optimization and how to apply the paradigm we will apply it to a number of standard NP-COMPLETE problems, starting with the Travelling Salesman Problem. ACO was applied to the Travelling Salesman Problem by (Dorigo et al., 1996). The Travelling Salesman Problem is the following problem: Given a set of n towns that are all connected, find a minimal length closed tour. A closed tour is a route that visits every town exactly once and that ends at the same town where it started. We call d_{ij} the length of the path between towns i and j. An instance of the TSP is given by a graph (V, E). Where V is the set of towns and $E : V \times V \to \mathbb{R}$ gives the length of the edges between the towns.

When applying ACO to the Travelling Salesman Problem every edge connecting two towns is a solution component. An ant starts at a random town. Then it chooses an edge to travel out of the set of edges to towns that it has not visited yet. This choice is made by a weighted probability based on the inverse of the length of the edges and the amount of pheromone on them. The ant will do this until it has visited all towns. For this problem the inverse of the edge length is used as heuristic information. This is because it is more likely that the shortest tour consists of short edges than that it consists of long edges.

After all ants have created a tour an amount of pheromone is evaporated. After that an amount of pheromone is deposited on edges based on the inverse of the tour length for all tours created by ants. This is because we want to find the shortest tour. We have written a simple implementation of ACO for TSP (Put, 2019) in order to run experiments. More explanation about the implementation can be found in section 3.1.

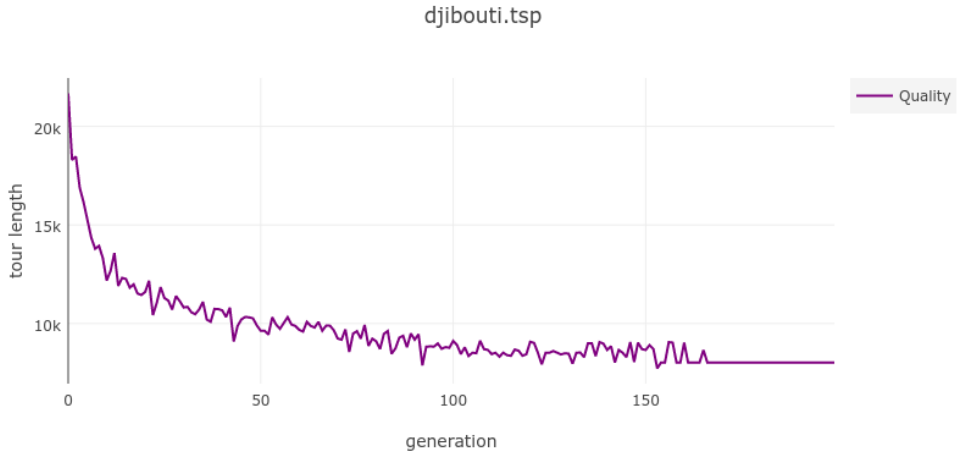


Figure 1.: Ant Colony Optimization applied to the TSP Djibouti dataset

A result can be found in figure 1, which is a set of 38 towns in Djibouti. The graph shows the shortest tour per generation of ACO. This benchmark comes from (uwaterloo tsp dataset). The best known length for this problem is 6656. As can be seen our algorithm does not find this global optimum, but it does find a reasonable approximation.

Hyperparameters

There are a number of parameters that influence the performance of the ant colony optimization algorithm. They can be found in section 3.1. We try to determine how to

tune the hyperparameters in order to improve performance. That is, we want to make a good tradeoff between finding the global optimum and convergence speed.

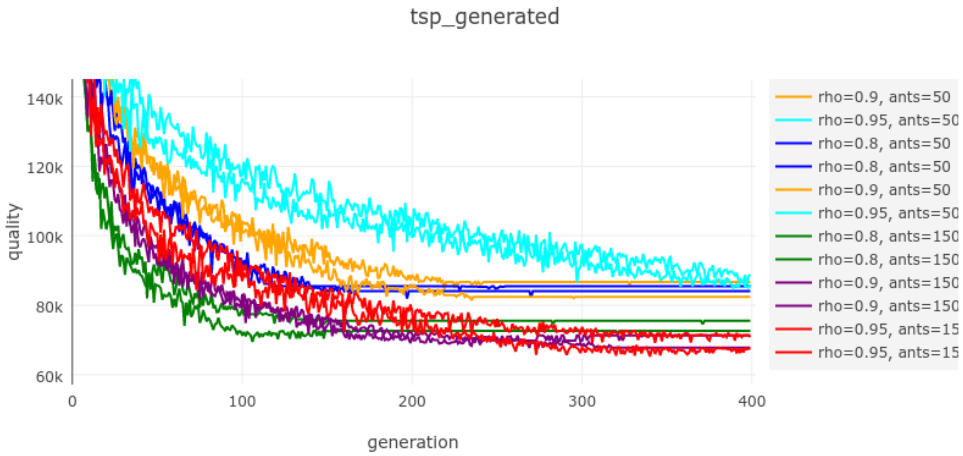


Figure 2.: Results with different hyperparameters on a benchmark with 100 towns.

We ran experiments to see how these parameters affected the algorithm. An overview of how the number of ants and the evaporation coefficient effect performance can be found in figure 2. We see that with a higher evaporation coefficient convergence is slower, but better results are achieved. We also see that with more ants per generation better results are achieved, but running times go up. We ran an experiment to see

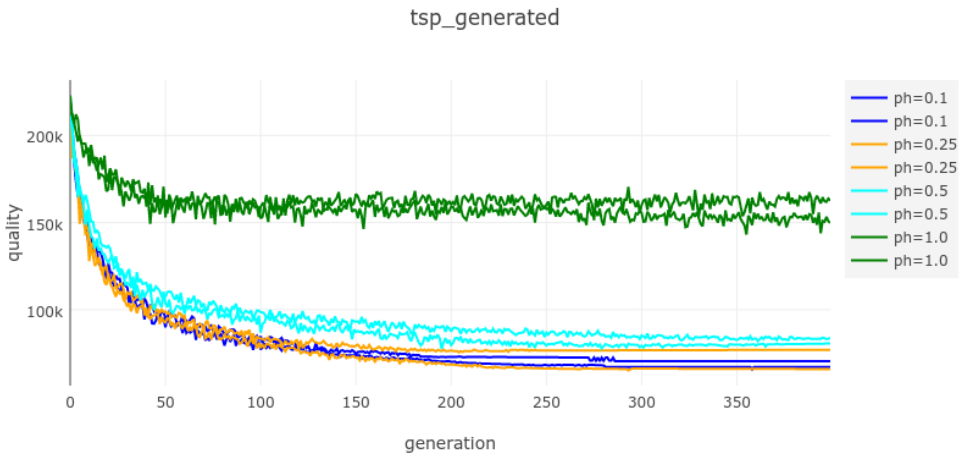


Figure 3.: Results with different pheromone fractions on a benchmark with 100 towns.

the influence of the amount of ants that were allowed to deposit pheromone. This experiment was run with 150 ants and $\rho = 0.8$. Results from this epxeriment can be found in figure 3. We see that if less ants are allowed to deposit pheromone, the solution quality increases.

2.5. Boolean Satisfiability

The next problem considered here is satisfiability of a Boolean formula. We want to apply ACO to this problem in order to explore how to map ACO to problems that are not directly relatable to graphs. Every Boolean formula can be rewritten to 3SAT. This is the format that we will use. Given a formula in the following form:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge ... \wedge (a_k \vee b_k \vee c_k)$$

The goal is to find an assignment for all Boolean literals a_i, b_i and c_i, where $i \in \{1, ..., k\}$, such that ϕ is *true* or report that such an assignment does not exist. This problem is NP-COMPLETE (Garey & Johnson, 1979).

ACO for satisfiability

In (Moritz & Springer, 2010) ant colony optimization is applied to the Boolean satisfiability problem. A solution candidate for a formula consists of a truth value for every Boolean variable in the formula. For a problem with k variables, pheromone values $\tau : \{1, ..., k\} \to \mathbb{R}$ and $\bar{\tau} : \{1, ..., k\} \to \mathbb{R}$ represent the pheromone levels of a variable and its negation respectively. For a solution to be valid every variable should be assigned a truth value. If a variable is not assigned *true*, it is assigned *false*. An ant assigns a variable l *true* with probability:

$$p(\ell) = \frac{\tau(\ell)}{\tau(\ell) + \bar{\tau}(\ell)}$$

The amount of clauses that evaluate to *true* is used as the quality of a solution. For every Boolean variable ℓ an amount of pheromone is added to $\tau(\ell)$ if ℓ was assigned *true* and to $\bar{\tau}(\ell)$ otherwise.

In (Moritz & Springer, 2010) two heuristics are considered for improving the creation of solution candidates. The first heuristic is that variables that occur more often are more important. In the function that selects a truth value, pheromone values are weighted by the number of clauses that the variable occurs in. The second heuristic is to make clauses that are not solved by previous solutions more important. To do this the algorithm keeps track of how long a clause has not been solved by the best solutions. Then the quality of a solution is the sum of the importance of all satisfied clauses. We have not implemented these heuristics.

We ran an experiment to verify that ACO for SAT works. The result can be found in figure 4. In this example ACO gets stuck in a local optima.

Incompleteness

For boolean satisfiability an assignment of variables that will make a formula *true*, proves that the formula is satisfiable. Proving unsatisfiability is much harder and there is no simple check to do it. The basic ant algorithm applied to satisfiability has no way to prove unsatisfiability, therefore it is incomplete. This means that if a formula is found to be satisfiable by the algorithm, then it is indeed satisfiable, but if it is found to be unsatisfiable, we do not know, it might really be unsatisfiable, but there might be a solution that the algorithm was just not able to find.

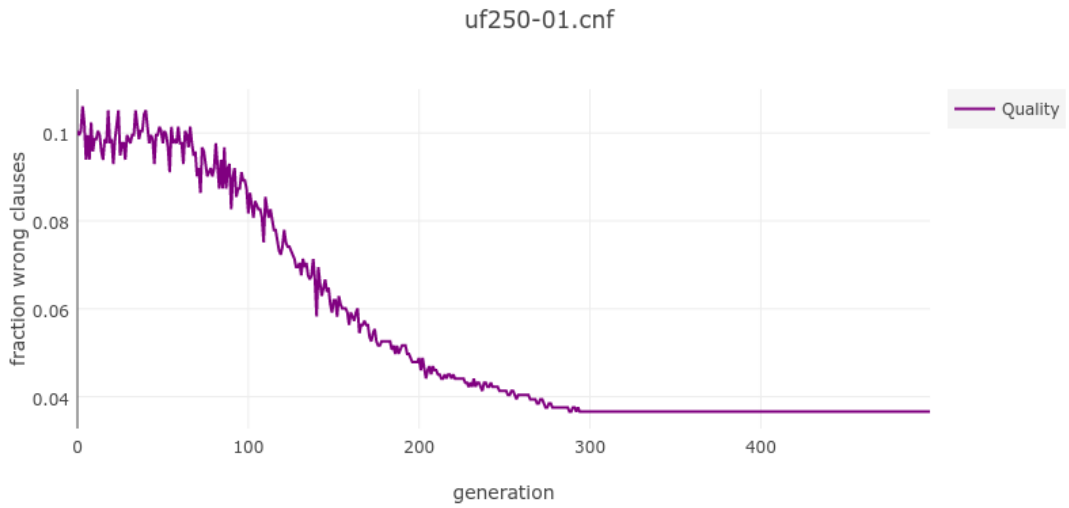


Figure 4.: ACO for SAT on a problem with 250 variables and 1065 clauses.

2.6. The Model Checking Problem

We now turn to the main topic of this thesis, concerning an ACO approach to the model checking problem. We need to understand the problem in order to determine how to apply ACO. We start with an explanation of what the model checking problem is.

There are systems for which it is crucial that they have or do not have certain behavior. Example of these systems are medical systems or controllers in an aircraft or car. We want to prove the absence or presence of certain behaviour mathematically. The problem to prove that a system has a certain behavior is called model checking. In this section we will explain how this problem can be formalized and solved.

The system we wish to check is represented by a model of its behavior. The behavior we want to check is represented by a formula. We wish to check in which states of the model this formula holds and especially if it holds in the initial state. We need a model of the system we want to check. The systems that we deal with can be abstracted to systems with the following properties: A system has a set of states that it can be in. Every state has a set of atomic propositions that hold in that state. There are transitions from a state to another. Systems with these properties can be represented by a mixed Kripke-structure. This is the model we will use for the model checking problem.

Definition 2.1 (Mixed Kripke-structure)**.** Formally, this model is a six tuple.

$$M = \langle S, s_0, Act, AP, R, L \rangle$$

where:

S is a set of *states*.

s_0 is an *initial state*.

Act is a set of action labels.

AP is a set of atomic propositions.

$R \subseteq S \times Act \times S$ is a transition relation.

$L : S \to 2^{AP}$ is a labeling function.

Behavior can be specified by the modal μ-calculus (definition 2.2). We follow the exposition of (Kozen, 1982).

Definition 2.2 (Syntax of modal μ-calculus)**.**

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X.\varphi \mid \nu X.\varphi \mid X$$

In this definition a is an action label and X is a variable from the set of variables Var.

This calculus is used to specify which atomic propositions hold in a state and what actions are possible from a state. We give a very brief overview of two important parts of this calculus, they are the modalities and the fixed-points [1]. There are two modal operators in this calculus, $[a]\varphi$ and $\langle a \rangle \varphi$. The first modality $[a]\varphi$ says that from the current state, φ should hold in all states that are reachable by a single a action. The other modal operator $\langle a \rangle \varphi$ says that there is a state which is reachable from the current state with an a action, in which φ holds. The fixed point classifiers μ and ν can be used to specify finite or infinite behaviour. The least fixed point $\mu X.\varphi$ indicates that the path where φ holds in every state is finite. The greatest fixed point $\nu X.\varphi$ indicates that the path where φ holds in every state is infinite.

We have a definition of a model and a definition of behavior. We continue by defining in which states a formula holds. The formal definition can be found in 2.3. In order to understand the meaning of this definition we present an example below.

Definition 2.3 (Semantics of modal μ-calculus)**.** $[\![\varphi]\!]_e$ denotes the set of states where φ holds given context $e : Var \to 2^S$.

$$\begin{aligned}
[\![\mathit{true}]\!]_e &= S & [\![\langle a \rangle \varphi]\!]_e &= \{s \mid \exists t. s \to^a t \implies t \in [\![\varphi]\!]_e\} \\
[\![\mathit{false}]\!]_e &= \emptyset & [\![[a]\varphi]\!]_e &= \{s \mid \forall t. s \to^a t \implies t \in [\![\varphi]\!]_e\} \\
[\![p]\!]_e &= \{s \mid p \in L(s)\} & [\![X]\!]_e &= e(X) \\
[\![\neg\varphi]\!]_e &= S \setminus [\![\varphi]\!]_e & [\![\nu X.\varphi]\!]_e &= \nu(Z \mapsto [\![\varphi]\!]_{e[X:=Z]}) \\
[\![\varphi_1 \wedge \varphi_2]\!]_e &= [\![\varphi_1]\!]_e \cap [\![\varphi_2]\!]_e & [\![\mu X.\varphi]\!]_e &= \mu(Z \mapsto [\![\varphi]\!]_{e[X:=Z]}) \\
[\![\varphi_1 \vee \varphi_2]\!]_e &= [\![\varphi_1]\!]_e \cup [\![\varphi_2]\!]_e
\end{aligned}$$

In this definition $s \to^a t$ has the meaning that there is a transition with the label a from state s to state t.

[1] A fixed point for a function $f : X \to X$ is a value $x \in X$ for which $f(x) = x$

For the fixed-point classifiers we need monotonic functions. With a non-monotonic function it is not possible to compute a fixed-point. Monotonicity is guaranteed by only allowing an even number of negation for a variable.

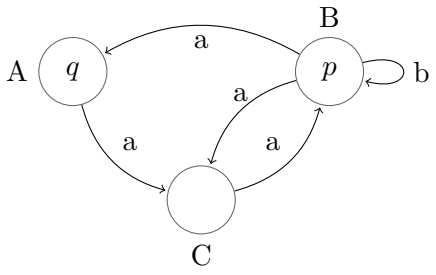


Figure 5.: A model represented by a mixed Kripke structure.

Example 2.4 (Solving a model checking problem)**.** Consider the model in figure 5 and the formula $\nu X.p \wedge [b]X$. This formula expresses that there is an infinite path of b actions where in every state the proposition p holds. We want to check in which states of the model this formula holds. We do this by using definition 2.3. In order to solve $\nu X.$ we need to find the greatest fixed-point of $X \mapsto p \wedge [b]X$. It follows from the Knaster-Tarski theorem that we can find the greatest fixed-point by setting the initial value of X to the set of all states. This gives the following iterations:

$$
\begin{aligned}
X_0 &= \{A, B, C\} \\
X_1 &= p \wedge [b]X_0 = \{B\} \wedge [b]\{A, B, C\} = \{B\} \\
X_2 &= p \wedge [b]X_1 = \{B\} \wedge [b]\{B\} = \{B\}
\end{aligned}
$$

We have found a fixed-point because X_1 is the same as X_2. We conclude that the formula holds only in state B. This makes sense because from state B a b action can be done infinitely often and the proposition p is always *true*.

The structure of the model checking problem does not lend itself to the application of ACO. A solution for a model checking problem is the set of states in which a formula holds. We think that we cannot efficiently compute the quality of a solution, which is required for ACO. Therefore, we convert the problem into problems with a simpler structure and try to apply ACO to those problems.

2.7. Complexity of the Model-Checking Problem

The model checking can be converted in linear time to the problem of solving a parity game. Parity games can be solved in quasi-polynomial time, as shown by (Calude, Jain, Khoussainov, Li, & Stephan, 2020). Solving the model checking problem is computationally hard and therefore applying ACO to it might be a good idea.

3. Ant Colony Optimization Technically

In this section we discuss how we implemented ACO and what challenges there are when implementing ACO.

3.1. Hyperparameters

There are a number of parameters that influence the performance of the ant colony optimization algorithm. The first one is the number of generations for which the algorithm is run. This parameter depends on the other parameters as they determine the convergence speed. We want to limit the maximum number of generations in order to limit computation time. The second one is the number of ants that are used. The third one is the number of ants that are allowed to lay down pheromone. The fourth one is the amount of evaporation that occurs. The amount of evaporation is determined by the evaporation coefficient ρ. Another hyperparameter is the relative importance of pheromone and heuristic values, controlled by α and β. We use grid search to select values for hyperparameters.

3.2. Experimentation Framework

To apply Ant Colony Optimization to multiple problems in a clean and efficient way we built a framework (Put, 2019). We make a division between components that are problem specific and components that are generic. Generic components are components that are the same for every implementation of ant colony optimization. Specific components differ between implementations of ant colony optimization. In the framework we provide interfaces for the problem specific components. While the generic components are implemented in the framework.

In our framework we call the class that stores pheromone amounts for the solution candidates the terrain. For every application of ant colony optimization there are ants that generate solution candidates based on the terrain. There is an evaluation function that will update the terrain based on the quality of the solution. Notice that the evaluation function has two tasks. One is to calculate the quality of a solution candidate. The other is to update pheromone levels of the terrain based on the solution quality. If we build proper interfaces for these components we can make a generic implementation of the ant colony optimization algorithm.

We define the following problem specific components: the `Ant`, the `Evaluator`, the `Terrain` and the `SolutionCandidate`. An implementation of `Ant` needs to implement the function `search` : `Terrain` $\rightarrow$ `SolutionCandidate`. An implementation of `Evaluator` needs to implement the function `evaluate` : `SolutionCandidate` $\rightarrow$ $\mathbb{R} \times$ `Terrain`. The first return value is the quality of the solutions candidate, better solution candidates should get a lower value for quality. A `Terrain` needs to implement an `evaporate` function. This function should reduce the effect of previous pheromone. How this is implemented is problem specific but usually multiplying the pheromone levels with the evaporation coefficient is enough. A `SolutionCandidate` has no required functions or data, it is completely problem specific. Using the interfaces specified above `AntColonyOptimizer` implements the algorithm. In figure 6 the components and their relation are listed.

The implementation of `AntColonyOptimizer` is generic for all problems. Pseudocode for `AntColonyOptimizer` can be found in listing 1. The main function of this

Listing 1: AntColonyOptimizer

```
class AntColonyOptimizer(
    ant: Ant,
    evaluator: Evaluator
    pheromoneFraction: Double,
    evaporationCoef: Double
) {
  var terrain: Terrain
  var bestSolution: SolutionCandidate
  var bestQuality: Double

  fun execute(initialTerrain: Terrain): SolutionCandidate {
    terrain = initialTerrain

    while (!stoppingCondition) {
      // result: Pair<quality, bestSolution>
      var result = runGeneration(generation)
      if (result.quality < bestQuality) {
        bestSolution = result.solution
        bestQuality = result.quality
      }
    }

    return bestSolution.second
  }

  fun runGeneration(): Pair<Double, SolutionCandidate> {
    var evaluations = (0..numberOfAnts).map {
      var solution = ant.search(terrain)
      Pair(solution, evaluator.evaluate(solution))
    }

    terrain.evaporate(evaporationCoef)

    val sorted = evaluations.sortedBy { x -> x.second.first }
    sorted
      .take(numberOfAnts*pheromoneFraction)
      .forEach { e -> terrain.addPheromoneFrom(e.second.second) }

    return Pair(sorted.first().second.first, sorted.first().first)
  }
}
```

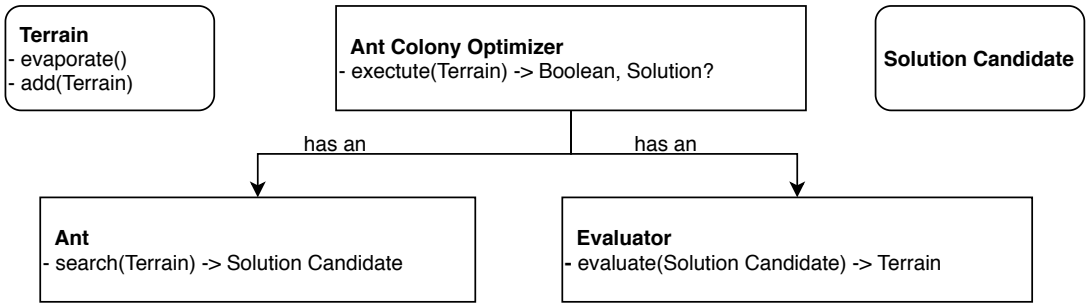


Figure 6.: Framework for implementing Ant Colony Optimization

class is `execute`. It will run generations of ant colony optimization until a specific stopping condition is met. This stopping condition could be after number of generations or if a certain solution quality is achieved.

The other function in this class is `runGeneration`. It will let a number of ants create candidate solutions, after that it will evaluate the solutions. Then it will evaporate pheromone and add pheromone for the evaluated solutions. Notice that only a fraction of solution candidates with the highest quality are allowed to deposit pheromone.

We built the framework in kotlin, because it is multi-platform (can run on the JVM), has good built-in support for paralellization and has a nice concise syntax. The implementation of this framework is only 120 lines of code, it can be found in appendix A.

Example 3.1 (Implementation of ACO for TSP)**.** In order to get a better feeling for how this framework can be used we provide an example implementation for the Traveling Salesman Problem. For the Traveling Salesman Problem a solution candidate is a closed tour. A closed tour consists of edges between towns. So the terrain consists of a pheromone value for the edges. The terrain is stored as a matrix where the indexes represent the start and end town of the edge.

Listing 2: Ant implementation for TSP

```
class TspAnt(val graph: CompleteGraph): Ant {
  var lastCity

  fun search(terrain: Terrain): SolutionCandidate {
    var tour = Tour(graph.numberOfNodes)

    lastCity = 0
    tour.visit(lastCity, 0.0);

    for (i in 0..graph.numberOfNodes - 2) {
      lastCity = step(graph, terrain, tour)
    }

    tour.visit(
      tour.getFirstCity(),
      graph.getValue(tour.getFirstCity(), lastCity)
    );

    return tour
  }
```

```
  fun step(graph: CompleteGraph, terrain: TspTerrain, tour: Tour) {
    // Get the cities that the ant can move to.
    val candidates = tour.getNotVisited()

    val attract = candidates.map {cd ->
      // Attractiveness is based on pheromone and edge length.
      var a =
          terrain.getPheromone(lastCity, cd).pow(ALPHA) +
          (1.0 / graph.getValue(lastCity, cd)).pow(BETA)
      Pair(cd, a)
    }

    var total = attract.fold(0.0) { acc, att -> acc+att.second}

    // Weighted choice between all candidates
    var rnd = Random.nextDouble(total)
    var choice = attract.fold(Pair(null, rnd)) { acc, att ->
      if (acc.second < 0.0)
        acc
      else
        Pair(att.first, acc.second - att.second)
    }

    tour.visit(choice.first, graph.getValue(lastCity, choice.first));

    lastCity = choice.first
  }
}
```

Selecting a next town is based on pheromone and edge length. In the code this is called the attractiveness of a town. The parameters `ALPHA` and `BETA` are used to control the influence of pheromone levels and edge length.

Listing 3: Evaluator implementation for TSP

```
class TspEvaluator(val graph: CompleteGraph): Evaluator {
  fun evaluate(candidate: SolutionCandidate): Pair<Double, Terrain> {
    // Quality is the length of the tour.
    val quality = candidate.length
    val qualityMatrix = candidate.matrix.mapIndexed { x,y,v ->
      if (0.0 == v)
        v
      else
        // If the tour is longer less pheromone should be added.
        1.0 / quality
    }

    return Pair(quality, TspTerrain(qualityMatrix))
  }
}
```

As can be seen in listing 3 the evaluation is very simple. This is because the edges used in the tour are stored in a matrix that has the same shape as the terrain. On this edges an amount of pheromone relative to the tour length is deposited. The implementation of ACO for TSP is around 330 lines of code, this includes all required datastructure.

4. Boolean Equation Systems

The model checking problem can be converted to the problem of solving a Boolean Equation System (BES). More specific, to the problem of finding the solution of the first equation of a BES. We are interested if we can apply ACO to the problem of finding a solution to a BES. The structure of a BES is similar to the structure of Boolean satisfiability. Therefore we expect that we can apply ACO in a similar way. This turns out not to be the case, as this chapter will make clear.

A BES is an ordered sequence of Boolean equations with fixed-point classifiers. The advantage of a BES over a modal μ-formula, definition 2.2, is that a variable in a BES is a Boolean, whereas a variable in a modal μ-formula is a set of states. The complexity of solving a BES is the same as the complexity of solving the modal μ-calculus but the problem has a simpler structure. If we can solve a BES, we can solve the model checking problem.

We start this section by defining the structure of a BES and showing how it can be solved in order to understand the problem. Then we will show how a modal μ-formula and a mixed Kripke-structure can be converted to a BES. After that we will show how we applied ACO to BES and discuss our findings.

We start with the formal definition of a Boolean equation system. The definition consists of the syntax and the semantics of this problem.

Definition 4.1 (Syntax of Boolean Equation Systems)**.** A Boolean Equation System $\mathcal{E}$ has the following structure.

$$f ::= X \mid \mathit{true} \mid \mathit{false} \mid f \vee f \mid f \wedge f$$
$$\mathcal{E} ::= \varepsilon \mid (\mu X = f)\mathcal{E} \mid (\nu X = f)\mathcal{E}$$

In this definition, X is a Boolean variable. We introduce an order on truth values where $\mathit{true} > \mathit{false}$ in order to be able to define the greatest and least fixed points classifiers. The least fixed point classifier μ is the value of the variable which yields the minimum fixed point for the equation, see section 2.6. The greatest fixed point classifier ν is the value of the variable which yields the maximum fixed point for the equation. The empty Boolean equation system is denoted by ε.

The syntax of a BES can be found in definition 4.1. There are a couple of things worth mentioning about this definition. There are no negations, if there where, an unsolvable BES could be constructed. In the context of a BES, by *equation* we mean the equation for one variable in the Boolean equation system. So $\nu X_1 = X_2 \vee X_3$ is an equation. Another point to note is that we will write equations in a column for clarity.

Definition 4.2 (Semantics of Boolean Equation Systems)**.**

$$\begin{aligned}
&[\![\varepsilon]\!](\eta) = \eta \\
&[\![(\mu X = f)\mathcal{E}]\!](\eta) = [\![\mathcal{E}]\!](\eta[X := [f](\eta_\mu)]) && \text{where } \eta_\mu = [\![\mathcal{E}]\!](\eta[X := \mathit{false}]) \\
&[\![(\nu X = f)\mathcal{E}]\!](\eta) = [\![\mathcal{E}]\!](\eta[X := [f](\eta_\nu)]) && \text{where } \eta_\nu = [\![\mathcal{E}]\!](\eta[X := \mathit{true}])
\end{aligned}$$

In this definition, η is the environment $\eta : X \to \mathbb{B}$. It assigns a truth value to every variable in the BES. The evaluation function $[\![\ ]\!] : \mathcal{E} \times \eta \to \eta$ creates an environment from a BES.

The problem of checking a modal μ-formula on a mixed Kripke structure can be converted to the problem of solving a Boolean Equation System. This transformation is explained in Mader (1997). Given a model M, a state t an a modal μ formula φ, we define a BES with the following property:

$$([\![\mathcal{E}]\!](\eta))(X_t) = \mathit{true} \iff M, t \models \sigma X.\varphi$$

Where σ denotes a fixed-point classifier (either μ or ν). This equation states that the formula holds for the initial state if the first equation in the corresponding BES is *true*. This BES can be found using the following procedure: For every sub-formula $\sigma X.\varphi$ for each state $s \in S$ we create an equation. The order of the equations in the BES should be the same as the order of the fixed-points in the modal μ-formula. For every sub-formula $\sigma X.\varphi$ for each state $s \in S$ we create the following equation:

$$\sigma X_s = \mathrm{RHS}(s, \varphi)$$

We want that $\mathrm{RHS}(s, \varphi)$ iff $s \models \varphi$. This is achieved using the following definition of RHS.

Definition 4.3 (RHS)**.**

$$\begin{aligned}
\mathrm{RHS}(t, \mathit{true}) &= \mathit{true} \\
\mathrm{RHS}(t, \mathit{false}) &= \mathit{false} \\
\mathrm{RHS}(t, p) &= \begin{cases} \mathit{true} \text{ if } p \in L(t) \\ \mathit{false} \text{ otherwise} \end{cases} \\
\mathrm{RHS}(t, X) &= X_t
\end{aligned}$$

$$\begin{aligned}
\mathrm{RHS}(t, \mu X.f) &= X_t \\
\mathrm{RHS}(t, \nu X.f) &= X_t \\
\mathrm{RHS}(t, f \wedge g) &= RHS(t, f) \wedge RHS(t, g) \\
\mathrm{RHS}(t, f \vee g) &= RHS(t, f) \vee RHS(t, g) \\
\mathrm{RHS}(t, [a]f) &= \textstyle\bigwedge_{v \in S} \{RHS(v, f) \,|\, t \to^a v\} \\
\mathrm{RHS}(t, \langle a \rangle f) &= \textstyle\bigvee_{v \in S} \{RHS(v, f) \,|\, t \to^a v\}
\end{aligned}$$

This results in a BES that expresses the same as the model and the formula.

Example 4.4 (Converting a model checking problem to a BES)**.** Consider the following model.

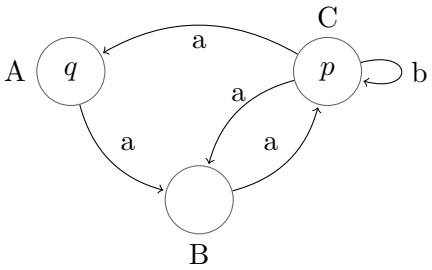


Figure 7.: A simple model

And the formula $\mu X.(\nu Y.p \wedge \langle b \rangle Y) \vee \langle a \rangle X$. This formula expresses that there is a finite path of a's that leads to an infinite path of b's where p holds. Let's look at how this

model and formula can be converted to a BES.

$$\begin{aligned}
\mu X_A &= \text{RHS}(A, (\nu Y.p \wedge \langle b \rangle\, Y) \vee \langle a \rangle\, X) = Y_A \vee X_B \\
\mu X_B &= \text{RHS}(B, (\nu Y.p \wedge \langle b \rangle\, Y) \vee \langle a \rangle\, X) = Y_B \vee X_C \\
\mu X_C &= \text{RHS}(C, (\nu Y.p \wedge \langle b \rangle\, Y) \vee \langle a \rangle\, X) = Y_C \vee (X_A \vee X_B) \\
\nu Y_A &= \text{RHS}(A, p \wedge \langle b \rangle\, Y) = \mathit{false} \\
\nu Y_B &= \text{RHS}(B, p \wedge \langle b \rangle\, Y) = \mathit{false} \\
\nu Y_C &= \text{RHS}(C, p \wedge \langle b \rangle\, Y) = \mathit{true} \wedge Y_C = \mathit{true}
\end{aligned}$$

As we can see, there is an equation for every node (A, B and C) for every fixed point (μX and νY). The order of the fixed-points is preserved, μX comes before νY. The expansion has been done by applying the rules of definition 4.3.

4.1. Algorithms for checking BES

Gauss elimination is the standard algorithm that is used to solve a BES. This algorithm resembles the Gauss elimination technique that can be used to solve systems of linear equations. In Gauss elimination, equations are evaluated starting at the last equation and working back to the first. If a solution is found for an equation, i.e. it evaluates to the value *true* or *false*, the variable is substituted by the truth value in all other equations. If no solution is found for an equation, the expression is substituted to equations at the left. When a variable occurs in its defining equation, it is assigned a truth value value due to the fixed-point. Because of this, and the substitution to the left, there will eventually be an equation that is assigned a truth value.

Example 4.5 (Solving a BES using Gauss Elimination)**.** We will show how a BES is solved using Gauss elimination. Consider the following Boolean Equation System.

$$\begin{aligned}
\nu X_1 &= X_1 \wedge X_3 \\
\mu X_2 &= X_1 \wedge X_2 \\
\nu X_3 &= X_4 \\
\mu X_4 &= X_3
\end{aligned}$$

iteration $\rightarrow$	0	1	2	3	4
$X_1 =$	$X_1 \wedge X_3$	*	X_1	*	*true*
$X_2 =$	$X_1 \wedge X_2$	*	*	*false*	*
$X_3 =$	X_4	X_3	*true*	*	*
$X_4 =$	X_3	X_3	*true*	*	*

Table 1.: Gauss elimination

Table 1 shows how the BES is solved by Gauss elimination. In iteration 1 the expression for X_4 does not evaluate to a truth value, therefore every occurrence of X_4 is substituted by the expression (X_3). In iteration 2 the variable X_3 is evaluated to *true* because of the greatest fixed-point (ν). After that, all occurrences of X_3 are eliminated by substituting them with *true*. In iteration 3 X_2 is set to *false* because of the least

fixed-point (μ), because of that $X_1 \wedge X_2$ evaluates to *false*. In iteration 4 X_1 is set to *true* because of the greatest fixed-point (ν).

The complexity of this algorithm lies in the iterative substitutions. In the worst case all equations are substituted. Because the substitutions are repeated, the size of the Boolean equations can grow exponentially. Assume that a Boolean equation system consists of n variables and equations, then the size of the Boolean expressions created by the algorithm is bound by 2^n. Therefore the worst case complexity is $\mathcal{O}(2^n)$. This result can be found in Mader (1997).

4.2. Applying ACO to solve a BES

In our implementation ants assign truth value to every variable. To implement this we need an algorithm to compute the correctness of an assignment of variables. However, there is no simple algorithm to check if a solution candidate for a BES is correct. Therefore we use a heuristic to evaluate solution candidates.

As an initial heuristic, we check how many equations are correct. An equation being correct means that the evaluation of the expression matches the value assigned to the variable. The quality of the solution candidate is the amount of equations that are correct. As a second heuristic we assign more weight to variables that have the value of their fixed-point. So *false* for least-fixed-points and *true* for greatest-fixed-points. We will build multiple heuristics for evaluating solution candidates and test which works best. We will try to build heuristics for the evaluation function that when optimized lead to the correct solutions. For the ants we wish to search more promising areas of the search space first and that reduce the search space. Reducing the search space can be done by letting an ant not generate solution candidates that are obviously wrong.

In order test how well Ant Colony Optimization can be applied to BES we create an implementation. This implementation is using our framework (see section 3.2), the implementation can be found in appendix B. To do this we need to implement the interfaces specified by the framework. They are `SolutionCandidate`, `Ant`, `Evaluator` and `Terrain`. The solution of a Boolean Equation System is the assignment of a truth value to every variable so that all equations are satisfied. A `SolutionCandidate` for a BES is an assignment of a truth value to every variable. This is implemented by an array of Booleans, one for every variable. A solution component is the assignment of a variable to *true* or *false*. We will have pheromone on the assignment of a variable to *true* and pheromone on the assignment of a variable to *false*. For all variables pheromone is deposited on the truth values that these variables are assigned by the solution candidate. A `Terrain` for a BES consists of two arrays of pheromone levels. One is for the assignment of variables to *true*. The other is for the assignment of variables to *false*. Obviously, every variable can be assigned only one truth value at a time. The implementation of `Ant` and `Evaluator` are specific per heuristic. Their implementation will be discussed in the corresponding sections.

In order to verify the performance of our algorithm we compare the assignment of variables to a known correct algorithm. With this information we have an absolute measure of correctness. The percentage of incorrect variables is used as a measure of performance. We have obtained this information by modifying the *bessolve* tool of the *mcrl2* toolset.

4.3. Initial version

The first version uses an ant that chooses purely based on pheromone levels. The same weighted probability is used as for boolean satisfiability, see section 2.5. With the exception that it assigns variables where the equation is a truth value to that truth value. So for $\nu X_1 = \mathit{false}$, the ant will always choose $X_1 = \mathit{false}$. The heuristic that is used to rate solution candidates only checks if the individual equations are correct. This means that evaluating the Boolean expression yields the same value as was assigned to that variable.

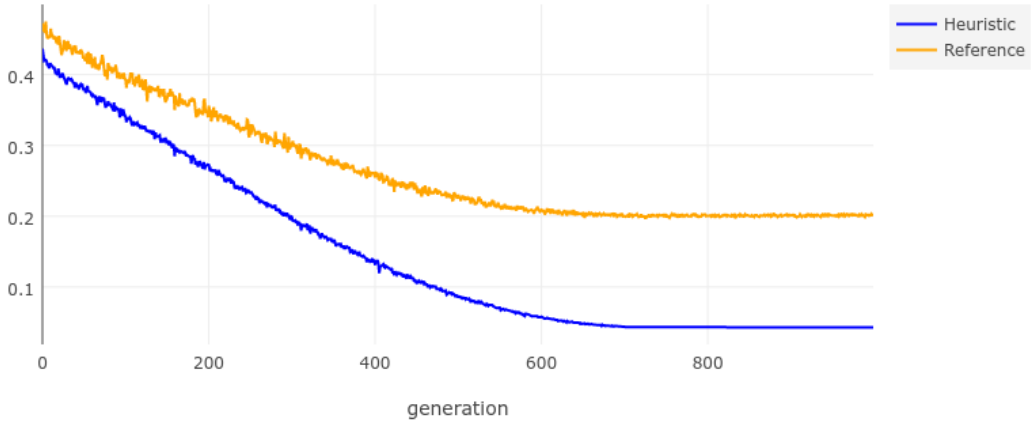


Figure 8.: Initial optimization (domineering/player_2_can_win)

Figure 8 shows how this algorithm performs on an example test. The reference is the fraction of variables that are chosen incorrectly. The value of the heuristic is the fraction of variables that do not match with their expressions. From this figure we can see that the algorithm does indeed optimize, and that minimizing the heuristic leads to an objectively better solution. We also see two problems with this result, the first being that the algorithm does not get all equations correct, the second is that there is a gap between the heuristic and the actual solution, so that even if the algorithm gets all equations correct, it still does not have the correct solution for the BES. The first problem can be reduced by reducing the search space. This is described in section 4.5. It is also possible that this behaviour can be fixed by implementing a so-called catastrophe. This means to reset a random number of variables if it is detected that the algorithm is stuck in a local minimum.

4.4. Hyperparameters

We ran an experiment to choose good hyperparameters. For a description of hyperparameters for ant colony optimization we refer the reader to section 3.1. Figure 9 shows the results of an experiment to choose the number of ants and evaporation coefficient. The experiment was run on (domineering(4x4)/player_2_can_win), which has about 2000 variables. The figure shows the quality of the best solution per generation. It is interesting to see that a higher value of ρ, which means less evaporation, causes a slower optimization process. Better results are obtained when more ants are used,

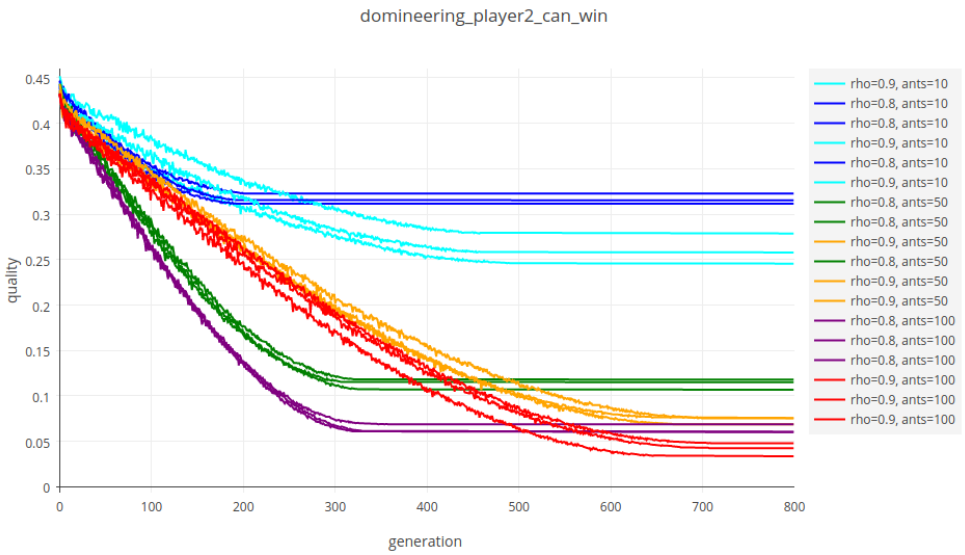


Figure 9.: Hyper parameter optimization (domineering/player_2_can_win)

but optimization is slower. The hypothesis is that better results are obtained because with more ants more different solution candidates are explored, which prevents premature optimization. On different test sets similar results were obtained, although the difference between the settings was less noticeable on tests with fewer variables. From this experiment we see that a ρ of 0.9 and 100 ants are good settings for the hyperparameters. They will be used for the rest of the experiments.

4.5. Reducing the search space

In order to improve performance we want to prevent the ants from generating solution candidates that are obviously false. To do this, we let the ant do a simple analysis. By doing this we reduce the search space. A simple analysis that can be done for BES is this. If a variable X is assigned *false* by an ant then all variables for which X occurs in a conjunction in their defining equation will evaluate to *false* as well. The same goes for variables that are assigned *true* and occur in dis-junctions. The ant that implements this behaviour is `DependencyAnalyzingBesAnt` (listing 17). For a BES with n variables it assigns variables starting at X_{n-1} and ending at X_0. When it assigns a variable X_i it checks for all equations X_0 to X_{i-1} if the equation evaluates to a truth value using the analysis described above and assigns the variable if that is the case.

To illustrate this, consider the following example.

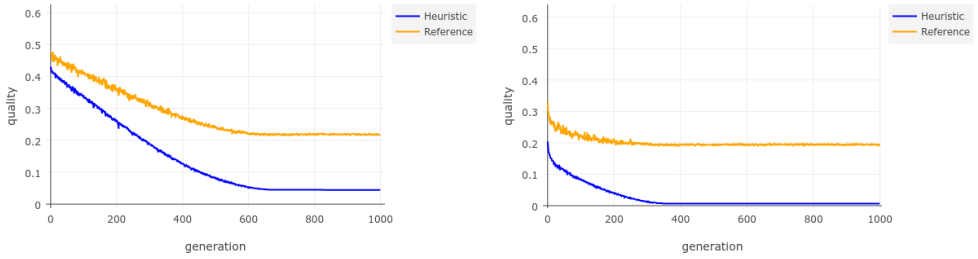
Example 4.6 (Simple dependencies in a BES)**.**

$$\sigma X_0 = X_1 \wedge \phi_1$$
$$\sigma X_1 = \phi_2$$

In this example if X_1 is assigned *false*, then it is impossible that assigning X_0 to *true* leads to a correct solution.

experiment	variables	heuristic	% assignments incorrect
dining3_seq_nodeadlock	93	0.003	0.3
dining3_seq_nostarvation	372	0.002	0.6
domineering(4x4)_player2_can_win	2443	0.008	17.1
domineering(4x4)_nodeadlock	2443	0.004	92.7
dining3_seq_nostuffing	553	0.033	84.9

Table 2.: Results of `DependencyAnalyzingBesAnt`.



(a) Initial ant

(b) Dependency analyzing ant

Figure 10.: Convergence speed of dependency analysis ant

As can be seen from table 2 this analysis caused a lower number of variable to be assigned the wrong value in the solution. The heuristic is the fraction of equations for which the evaluated expression has a different truth value than was assigned to the variable. In figure 10 we compare the performance of ants that do dependency analysis against ants that do not, the test we used is (domineering_player2_can_win). From figure 10 we see that dependency analysis increases convergence speed. This is because fewer wrong solutions could be created. This version introduced a new problem as now some variables have a large effect on the correctness of the solution, so picking them wrong initially leads to a large error.

4.6. Important fixed-point classifiers

Currently fixed-point classifiers are not taken into account when evaluating solution candidates. This prevents us from finding the correct solution in general. An idea is to analyze for which equations the fixed-points are important, and for which they are not important and assign the important fixed-points first, and check if they are assigned a value corresponding to their fixed-point and not care about other fixed-points. To explain what important fixed-points are, consider the following example:

Example 4.7. Consider the following BES.

$$\mu X_0 = X_0$$
$$\nu X_1 = X_2$$
$$\nu X_2 = X_1$$

First we introduce the rank of a variable. The rank of a variable is the position where

it is defined. Variables that are defined more to the left have a higher rank. In this example X_0 has a higher rank then X_2.

In this example X_0 and X_1 are marked important, but X_2 is not. The value of X_2 depends on the truth values of variables with a lower rank, because all variables in its expression (X_1) have a lower rank than the variable. Because of this, the fixed-point classifier of X_2 does not affect the solution.

It is always the case that if all variables of an equation for a variable have a rank lower than that variable that the expression will be substituted and therefore the fixed-point will have no effect.

For important fixed-points we reward solutions that have a truth value corresponding to the fixed-point. This is done to imitate the behaviour of the fixed-points, when a fixed-point is evaluated first a solution is sought with the truth value corresponding to the fixed-point. Finding important fixed-points is simple as we only have to compare the rank of the variable of the highest rank in the expression with the rank of the variable that is defined by the expression. Unfortunately this does not solve the problem of opposite optimization. The heuristic is implemented in `ImportantFPBesEvaluator`, listing 20.

experiment	heuristic value	% assignments incorrect
dining3_seq_nodeadlock	-0.605	90.3
dining3_seq_nostarvation	-0.543	17.1
domineering(4x4)_player2_can_win	-0.604	60.7
domineering(4x4)_nodeadlock	-0.728	12.1
dining3_seq_nostuffing	-0.262	28.3

Table 3.: Results of `ImportantFPBesEvaluator`.

As we can see from table 3 this heuristic reverses which tests are not fixed correctly. This heuristic does not solve the problem of which fixed-points are used. There are instances where it works, but in most problems there are fixed-points that are important according to the heuristic but that do not actually have an effect. To explain this, we consider the following example.

Example 4.8 (Important fixed-points are not always important)**.** Consider the following BES.

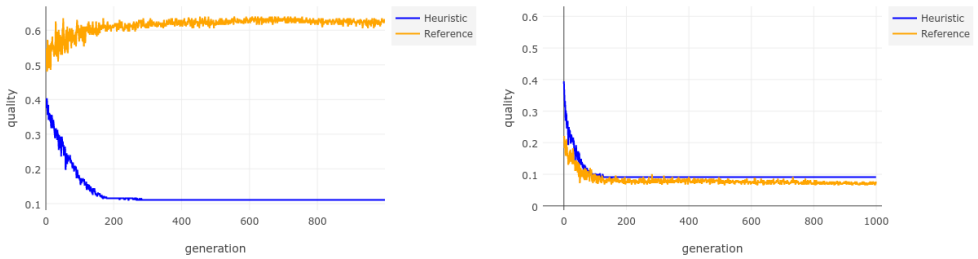
$$\mu X_0 = X_0$$
$$\nu X_1 = X_2$$
$$\nu X_2 = X_0$$

Here X_1 is considered an important fixed-point by the heuristic. When we solve this BES we see that the fixed-point classifier of X_1 does not have an effect (X_2 gets substituted by X_0).

A better estimation of which fixed-points are important can be made by doing all substitutions and keeping track of the rank of the highest variable. The approach of finding important fixed-points is not enough to take the behavior of the fixed-point classifiers into account.

4.7. Fundamental problem

While running the experiment *dining3_ns_nostuffing* we noticed that our initial algorithm optimizes to the opposite value. This behaviour can be seen in Figure 11a. This is a more extreme case of the problem already noted in the previous section. This happens in — but is not limited to — cases where the Boolean expressions contain mainly conjunctions. When the ants start searching, solutions with more variables *false* are considered more correct.



(a) Opposite optimization (b) Fixed with bias

Figure 11.: Results from the initial version

In order to solve the problem, we tried to start searching at a specific place, based on the problem structure. For example if the problem contains a lot of conjunctions, we start assigning variables to *true*. This is done by using a bias in the ants when selecting truth values for the variables. The updated probability of setting variable ℓ to *true* becomes:

$$p(\ell) = \frac{\tau(\ell) + b}{\tau(\ell) + \bar{\tau}(\ell)}$$

Where b is the amount of bias. This is a parameter that changes how likely an ant is to set l to *true*. This can solve the problem for specific instances, the result with a positive bias can be found in Figure 11b. However, this introduces a new problem as it will also set variables to *true* when the dominating fixed points are μ, in which case variables should be assigned *false* first. We conclude that the bias should be based on the fixed point classifier. The ant that implements this behaviour is `FPHeuristicBesAnt`, listing 18.

experiment	heuristic	% variables incorrect
dining3_seq_nodeadlock	0.121	89.2
dining3_seq_nostarvation	0.075	9.5
domineering(4x4)_player2_can_win	0.086	24.9
domineering(4x4)_nodeadlock	0.048	14.2
dining3_seq_nostuffing	0.160	39.3

Table 4.: Results of `FPHeuristicBesAnt`.

We ran this version on a number of test cases, the result can be found in table 4. The problem with this solution, and all previous solutions is that the heuristic cannot reason

which fixed-points will be used. To illustrate this, consider the following example.

Example 4.9 (Determining which fixed-points are used)**.**

$$\begin{aligned}\nu X_1 &= X_1 \vee X_3\\ \mu X_2 &= X_1 \wedge X_2\\ \nu X_3 &= X_4\\ \mu X_4 &= X_3\end{aligned}$$

There are multiple possible assignments for which the equations are correct if the fixed-point classifiers are not taken into account. For example $X_1 = \mathit{false}, X_2 = \mathit{false}, X_3 = \mathit{false}, X_4 = \mathit{false}$ and $X_1 = \mathit{true}, X_2 = \mathit{true}, X_3 = \mathit{false}, X_4 = \mathit{false}$ are solutions.

In example 4.9 the fixed-points for X_3 and X_2 have an effect but the heuristics cannot detect that the solutions are wrong because of the fixed-points.

Consider the following Boolean equation system:

$$\begin{aligned}\nu X_1 &= X_2 \wedge \varphi_1\\ \mu X_2 &= X_3 \wedge \varphi_2\\ \nu X_3 &= X_1\end{aligned} \tag{4}$$

Where φ_1 and φ_2 are Boolean expressions, what they are exactly is not relevant for this example.

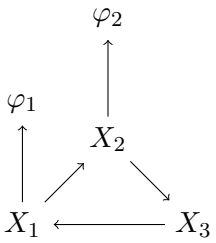


Figure 12.: Dependency graph for 4

The dependency graph for this BES can be found in figure 12. We see that there is a circular dependency, in this case the fixed-point of the variable with the lowest rank, X_1, will be used. However, due to φ_1 and φ_2 we cannot check if the fixed-point is satisfied or not. It could be that they also include circular dependencies. A heuristic that only checks circular dependencies and does not take other dependencies into account does not solve the problem of knowing which fixed-points have an effect.

This could be solved if the heuristic could have a dependency graph of which variables depend on each other. In that way it is possible to detect if an assignment satisfies the fixed-points. The problem with such a heuristic is that it has to solve the BES, because of that it makes more sense to solve the BES directly once instead of solving it many times in order to check solution candidates.

4.8. Final version

The heuristic that awards variables that are assigned according to their fixed-point in combination with the dependency analyzing ant is the best algorithm we have come up with. It is fundamentally flawed as the previous section made clear, this can be seen in the results. We will now give a brief overview of its performance, discuss problems and possible solutions.

problem	correct solution	ACO solution	steady state
dining_cs_nodeadlock	*true*	*true*	20 cycles, 14% error
dining_cs_nostarvation	*false*	*false*	25 cycles, 2% error
dining_cs_nostuffing	*true*	*true*	200 cycles, 11% error

Table 5.: Benchmark on dining philosophers testset (50-200 variables)

problem	correct solution	ACO solution	steady state
player2_can_win	*true*	*true*	200 cycles, 18% error
nodeadlock	*true*	*false*	200 cycles, 76% error

Table 6.: Benchmark on 4x4 domineering (2000 variables)

As can be seen in table 5 and table 6 the performance of this algorithm is not good, it does get a correct solution, but with a high error rate. Which means that it is not correctly solving the problem.

problem	local fixed-point (s)	ACO (s)
dining_cs_nodeadlock	0.15	0.1 - 1.8
dining_cs_nostarvation	0.04	0.1 - 1.3
dining_cs_nostuffing	1.8	19.3
domineering(4x4)_player2	0.69	336
domineering(4x4)_nodeadlock	0.5	337

Table 7.: Running time for the benchmarks

For this final version we will list the performance. We compare against the local fixed-point algorithm, which is an optimized version of gauss elimination. As can be seen in table 7 the local fixed-point algorithm is much faster than ant colony optimization on these benchmarks.

4.9. Conclusion

As we have seen in the previous section, also the final algorithm performs poorly. This is a fundamental problem because the ant colony optimization algorithm does not know which fixed-point classifiers have an effect. Without knowing this it is not possible to construct a correct solution. There was no heuristic that we could come up with that targeted this specifically. As far as we know the only method to determine which fixed-point classifiers are used is to solve the BES. Checking Boolean equations using a dependency analysis does not solve the problem.

We must conclude that we have not been able to create an algorithm that correctly solves a BES based on ACO. The fundamental problem is that the complexity of checking if a solution candidate is indeed a solution is equal to the complexity of solving the problem and therefore infeasible. And we have not found a reliable heuristic that can be used to find a correct solution. In general the complexity of checking a solution candidate should be low, which is not possible for this problem. This indicates that this problem fundamentally does not lend itself to the application of ACO.

5. Parity Games

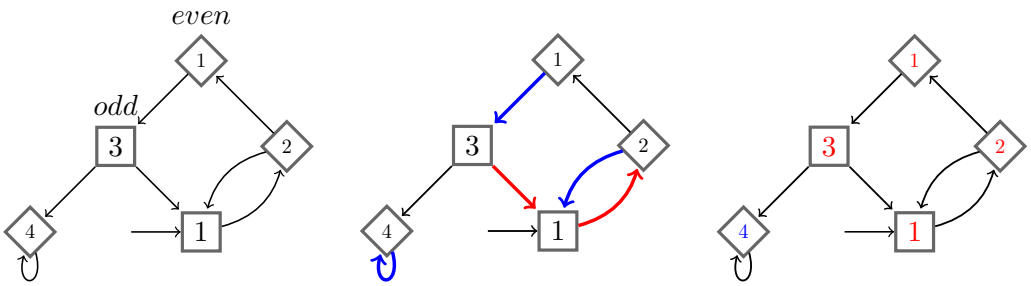
We are interested to see if ACO can be applied to solve a parity game, that is to find the winner of the game. This section starts with an explanation of parity games. After that we will explain how we applied ACO and our results.

The model checking problem can be converted to the problem of solving a parity game, as shown by (Mader, 1997). A parity game is a two player game that is played on a graph. In this graph vertices are connected using directed edges. Every vertex has an owner and a priority. The game is played by two players, odd and even. A token is placed on the initial vertex of the play and the owner of the vertex moves the token to a next vertex that is connected by an edge. The goal of the players is to let the parity of the lowest priority that occurs infinitely often be of same parity as they have, so player odd wins if the lowest priority that occurs infinitely often is odd and vice versa for even. The winner of a vertex is who wins the play starting at that vertex if both players play optimally. For the model checking problem we want to know who wins a vertex, which is called the initial vertex.

Definition 5.1 (Parity game)**.** A parity game $G = (V_0, V_1, E, p)$ consists of a set of vertices owned by player even V_0, a set of vertices owned by player odd V_1, disjoint of V_0, an edge relation $E : V \times V$, where $V = V_0 \cup V_1$ and a priority function $p : V \to \mathbb{N}$ that assigns a priority to every vertex.

The winner is determined by the parity of the *lowest* priority occurring infinitely often in a play induced by optimal strategies of odd and even. A winning strategy is independent of the history of the game (where the token has been), this is called memoryless determinacy and was proved to hold for parity games by (Emerson & Jutla, 1991) and in parallel by (Mostowski, 1991). This means that a strategy picks an edge to play to independent of where the token came from or what happened before. Because we can assume that strategies are independent of history when playing the game this play will always end up in a loop that is travelled infinitely often.

We will show an example to make the concepts of a *strategy* and *winning* more clear. In figure 13 the even player is denoted by a diamond and the odd player by a



(a) Parity Game (b) Strategies (c) Winners

Figure 13.: Example of a parity game

square. The even player is blue and the odd player is red. The strategies are denoted in figure 13b by colored lines which indicate that the player will always choose to play to that edge. Notice that there are two strategies, one for odd (in red) and one for even (in blue). In figure 13c the vertices that are won by odd and even are denoted. Winning a vertex means winning the game that is played when the token is initially

placed at that vertex.

5.1. Parity Game from a BES

The model checking problem for the modal μ-calculus is equivalent to the problem of solving a parity game, see (Stirling, 1995). A Boolean equation system can be converted to a parity game, see (Mader, 1997). This transformation can be done in linear time. The transformation works on a BES in normal form. In normal form each equation is purely conjunctive or disjunctive and does not contain truth values. Converting a BES to normal form is straightforward. To transform a BES to a parity game a BES is first divided into blocks with the same fixed-point operator. Every block is assigned a priority. The priority of the first block is 1 if the first block is μ and 0 if it is ν. Every next block has a priority one higher than the previous block. For every equation in a block, a vertex is created in the parity game with the priority of the block. If the equation is a conjunction odd owns the vertex, otherwise even owns the vertex. For every variable that occurs in the expression for an equation, create an outgoing edge to the vertex of the equation that defines that variable.

5.2. Ant Colony Optimization for Parity Games

The implementation of Ant Colony Optimization for parity games differs from the approach we used for BES. A solution for a parity game is the answer who wins the game. Who wins can be supported by providing an optimal strategy for both players, in this case the proof should be that these strategies are indeed optimal (the best possible strategy). Who wins can also be supported by a division of all vertices into a set where even wins and a set where odd wins. Both approaches are computationally hard to validate, so letting ants generate strategies or winning sets will not work.

Instead we let the ants play the game. An ant plays for both players at the same time. An ant starts at the initial vertex and moves from vertex to vertex while it remembers the choice it made for every vertex. After a finite number of moves the ant will encounter a vertex that it has already visited. When this happens there is a loop. Notice that the loop the ants find will be repeated because the ant always makes the same choice for every vertex. This loop means that a winner can be determined (by determining the parity of the lowest priority in the loop). The movement of the token is called a *play*, so an ant makes a play. To which vertex an ant moves is determined by a weigted random choices based on the amount of pheromones on the edges.

When a winner is determined for a play, every 'good' choice will be rewarded and every 'bad' choice penalized. This means that if even wins, all choices even made are rewarded, because they led to a win. Odds choices will be penalized because they led to a loss for odd. Reversed if odd wins. After a number of generations the solution is retrieved. For determining the winner, the edge with most pheromone on it is chosen for every vertex. The winner of the play that is generated in this manner is reported as the winner of the game. The idea of the algorithm is to let the ants try to find an optimal strategy for the players. We will illustrate this algorithm with an example.

Example 5.2 (ACO for parity games)**.** Consider the parity game in figure 14. In this game edge labels represent the pheromone levels on the edges. The higher the amount of pheromone is, the more likely it is that an ant will choose that edge. The initial vertex is indicated by the incoming arrow.

There is an ant that makes the play displayed in figure 15a. In this figure solid

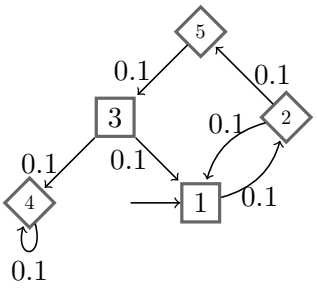
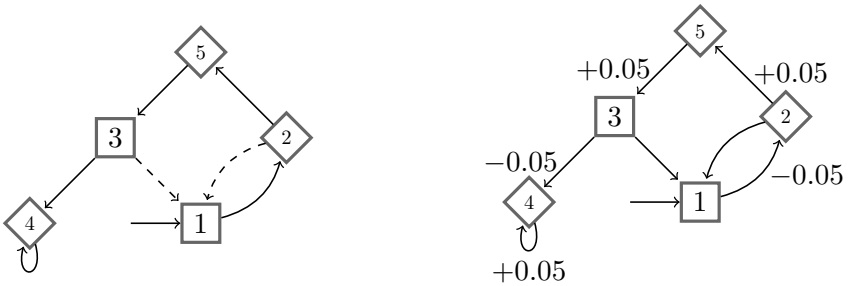


Figure 14.: A parity game with pheromone



(a) A random play

(b) Updated pheromone levels

Figure 15.: A play and the resulting pheromone updates

edges are in part of the play, the dashed edges are not in the play. This play ends with a loop in the vertex with priority 4, which means that even will win this play. When evaluating this play pheromone is deposited or removed. Every choice that even made will be rewarded and every choice that odd made will be penalized. The updated pheromones on this game can be found in figure 15b.

The final solution is retrieved by creating a strategy by always picking the edge with the highest pheromone level. A play is created using this strategy. The winner of this play is the winner of the game. The idea behind this is that the pheromone levels mark the optimal strategies. When we determine the winner of the terrain in figure 15b we start at the initial vertex (priority 1), then move to the vertex with most pheromone on the edge, the vertex with priority 2, from there we move to the vertex with priority 5, then to the vertex with priority 3 and then back to the initial vertex. The lowest priority in this loop is 1, which is odd, so odd wins this game.

We have implemented this algorithm. The implementation can be found in appendix C.

5.3. Hyperparameters

If an ant makes a good choice for a player, that choice is rewarded a certain amount of pheromone. If an ant makes a bad choice (leading to a loss) the amount of pheromone is decreased. If too much pheromone is removed, then an edge on a good path could be marked bad on only one example, which would lead to instability. If not enough pheromone is removed a bad solution would not be marked as a bad solution and ants will continue to create that solution. We are interested in what amount of pheromone to remove and how that affects the stability of the algorithm. We use a grid search to

determine the best amount of pheromone decrease. We ran experiments on a games with different sizes in amount of vertices and amount of edges per vertex. We found that removing about a third of the pheromone on a loss gives the best result. This is quite a big penalty.

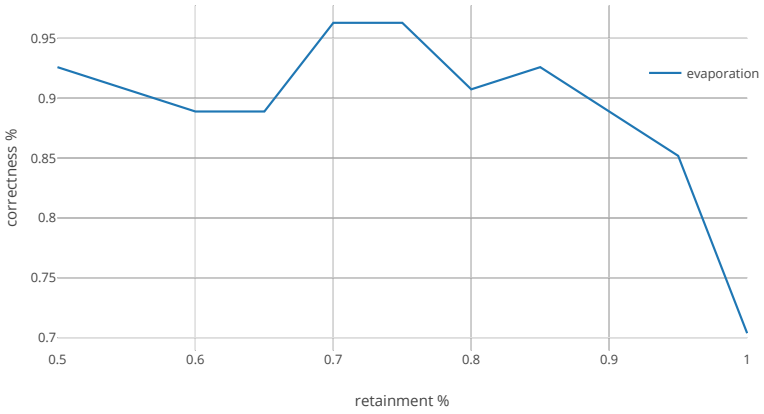


Figure 16.: Evaporation levels

We ran an experiment to determine the amount of pheromone evaporation. The result of this experiment can be found in figure 16. We see that there is a peak after the penalty for losing a game (which is 0.65). We evaporate 25% of the pheromone from previous generations. This relative high amount indicates that there is a large movement in the generated solutions and pheromone levels per generation.

We want to know how much ants we need to get a good exploration of the game. For a game with a lot of vertices and edges more ants are required than for a small game. Intuitively it makes sense to make the amount of ants linear to the amount of edges in the game. This is because the amount of edges has a big influence on the number of possible plays. We use a relation between the number of edges and the number of ants as a rule of thumb. In order to test the choice of hyper parameter, we test on random games with varying size and varying edge density. We find that using $\frac{\text{edge count}}{9}$ ants the algorithm gives the most correct answers on tests.

5.4. Random Against Best

To improve strategies faster we let the ants play a random strategy for one player against the current best strategy of the other player. Instead of letting ants use random choices for odd and even. The random strategy is created by a weighted random choice based on pheromone values on the edges. The best strategy is created by choosing the vertex with most pheromone on its edge. This is regarded as the best strategy because the edges with the most pheromone on them were part of the best plays.

If the player that made random choices wins, its strategy is better and we should reward this strategy. If the random strategy loses we penalize this strategy in order to not try it again. For a play created by an ant in this way only the choices of the random strategy are rewarded or penalized. The rationale behind this is that mediocre

strategies will not be enforced and exploring better strategies is enforced.

It solves a problem that occurs when ants make random choices for both players. There it is possible that the choices of one player get rewarded because it won a play, while that win was not due to its beneficial choices but due to the bad choices of the opponent. If this happens, other ants will follow this bad path which prevents them from following a better path that they would have followed otherwise.

We ran the same experiments we used in section 5.3 to find hyperparameters for this version. We found that a 25% penalty for a bad strategy works best combined with 30% evaporation per generation. The amount of ants required is equal to the amount of ants for the original version.

We are interested how this biased version compares to the original version. We have a benchmark of randomly generated parity games. We run the algorithm 6 times on every benchmark and average the result. We use the recursive algorithm (Zielonka, 1998) as reference. Results from this experiment can be found in table 8. Because

experiment	aco	random vs best	reference
steadygame(10,2)	0.0 (0.03s)	0.0 (0.03s)	false (0.06s)
steadygame(100,2)	0.0 (0.53s)	0.33 (0.29s)	false (0.01s)
steadygame(1000,2)	0.5 (5.93s)	0.33 (5.77s)	false (0.02s)
steadygame(10,10)	1.0 (0.01s)	1.0 (0.01s)	true (0.01s)
steadygame(100,10)	1.0 (0.20s)	1.0 (0.51s)	true (0.02s)
steadygame(1000,10)	0.0 (75.56s)	0.33 (109.48s)	false (0.02s)

Table 8.: Results of random against best

the results from ACO are averaged the result is a number between 0 and 1, with 0 indicating that all runs returned *false* and 1 indicating that all runs returned *true*. The version with biased ants performed slightly worse than the original version.

A problem with random against best is the start of the algorithm. Then the best strategy is random for both players and we have the problem that we wanted to solve, namely that we do not know if a win is due to good strategy of the winner or a bad strategy of the loser. A drawback of this approach is that ants only deposit or remove pheromone on edges for one player, this makes the ants a little less effective. In conclusion, this way of generating strategies is not much better than the original approach. It is very interesting to apply this approach to strategy improvement. In strategy improvement the best strategy for the opponent player is known. Instead of playing against a best strategy that might not be a good strategy there the best strategy is an optimal strategy. We will research this in section 6.4.

5.5. Game Exploration

There are games in which one player can always force a play to a certain vertex. The algorithm does not create plays that reach this vertex because the chance to reach it randomly is very small. We constructed some games to test the algorithms ability to explore games. The games we constructed are hard to solve via random plays because there is a large chance to create wrong plays. An example of a constructed game can be found in Figure 17. In this game every vertex has an edge to every previous vertex and only one option to go to a next vertex. Consider what happens if ants try to create plays. If an ant has already visited a vertex, a loop is created when it visits that vertex again. Almost all edges create a loop, which reduces the chance to create a play that

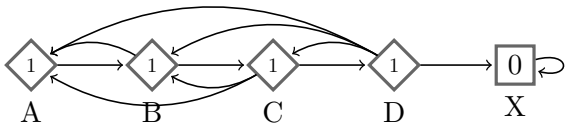


Figure 17.: Constructed game

visits vertex X. In this example the chance of playing to vertex X is:

$$1 \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4}$$

On versions of this game with more vertices it is virtually impossible for the ants to find the correct solution (vertex X).

If a loop is constructed the ant knows which player will win. This information can be used to solve the problem on the testcase mentioned above. This is done by restricting the choices of the ant on a given vertex. If the ant can go to vertices where the current player (owner of the current vertex) wins, it will choose between these vertices. If the ant can go to vertices where the current player loses, it will not consider these vertices, except if all options are losing.

In order to implement this an ant needs to keep track of the priorities of the vertices that it has already visited. When an ant keeps track of the priorities of all visited vertices, the winner can be calculated by iterating over the priorities that are in the loop (priorities between the vertex that is visited again and the end of the path). This is inefficient as it requires a calculation at every potential loop. We want to be able to lookup if a loop is winning or losing in $\mathcal{O}(n)$ because we do this lookup for every vertex we can go to at every step. This can be done by keeping track of the *lowest loop value* of the visited vertices instead of the priorities of the visited vertices. When we visit a vertex we store its priority as the lowest loop value. (Because if at that point a loop to itself is found, that is the only, and thus lowest, priority in the loop.) For all previous vertices we update their lowest loop value to the minimum of their current value and the priority of the new vertex. This is valid because if a loop is formed by visiting a vertex that is already visited again. Then its priority is the minimum of all vertices in between. We can optimize this algorithm by the lowest priorities from back to front and stopping the update if we find a lowest priority lower or equal to our priority. This is because when that occurs all vertices before that vertex necessarily have an equal or lower value. Pseudocode for this algorithm can be found in listing 4.

This behaviour is implemented in `ExploringPgAnt`, listing 25. It finds the correct solution on our artificial test sets. Table 9 shows results from test on versions of the

experiment	number of edges	aco solution	exploring aco solution
explorationgame(10)	46	false, 0.04s	true, 0.08s
explorationgame(50)	1226	false, 1.31s	true, 8.36s
explorationgame(100)	4951	false, 33.84s	true, 65.86s

Table 9.: Results of ACO on steady games.

testcase of figure 17. In this table we see that our modification works and leads the ants to find the correct solution, it also requires a lot more computation time as it has to check which edges lead to a loss.

Listing 4: Keeping track of lowest priority

```
vertices = List()
priorities = List()

fun visit_vertex(v) {
    vertices.add(v)
    priorities.add(v.priority)
    for (index, priority in priorities.reversed) {
        if (priority > v.priority) {
            priorities[index] = v.priority
        } else {
            // All values before have a lower value.
            break;
        }
    }
}

// Get the winning priority if v is visited.
fun get_winning_priority(v) {
    if (vertices.contains(v)) {
        return priorities[vertices.indexOf(v)]
    } else { // No loop, no winning priority
        return null
    }
}
```

The question is if this behaviour leads to better results on other games. This depends on how hard it is to find the solution by making random choices. In the example we created the chance to find the solution by making random choices is almost zero when more vertices are added to the chain. We suspected that this approach would improve performance. We have not found games in practice in which exploring ants performed significantly better than the standard ants.

5.6. Conclusion

We have created an implementation of ACO to solve parity games that gives a correct solution in most instances. The biggest challenge in this implementation was exploration, forcing that ants would explore more options. This algorithm is moderately useful, using Strategy Improvement (see next section) the correctness of a solution can be checked in a reasonable time. The algorithm is much slower then the recursive algorithm or small progress measures, which we used to benchmark against. This can partially be explained by our inefficient implementation. It indicates that ACO applied to parity games in this manner does not scale better then other algorithms. This is because the number of possible strategies increases exponentially with the game size (in number of vertices and edges). ACO is slowed down because it has to create a lot more plays and because the plays are generally longer. Other algorithms that do not create strategies but instead search for dominions do not suffer from this blowup in amount of possible strategies.

5.7. Future Work

Currently the reward is binary, either even wins and its choices are rewarded or odd wins and its choices are rewarded. It would be interesting to see if making the reward continuous and based on more properties of the winning play can increase the convergence speed. It would also be interesting to research what heuristics the ants could use when generating plays. Currently plays are constructed purely based on pheromone levels. Therefore it would be interesting to research if for example preferring playing to a vertex with parity of the current player, or owned by the current player improves the strategy.

6. Strategy Improvement

Strategy improvement (SI), (Vöge & Jurdziński, 2000), is an algorithm for solving parity games. In this algorithm one player is improving its strategy. She does this based on an optimal response strategy created by her opponent. From the strategy of the optimizing player and the response strategy of the opponent an order on vertices is created. This order represents the value of that vertex as seen by the optimizing player, i.e. how good playing to that vertex is. The optimizing player will improve its strategy by playing to vertices with a higher value then the vertices it currently plays to, with respect to this order. After the optimizing player has updated her strategy, her opponent will respond with a new strategy that is optimal for the opponent. This will update the order on the vertices. This process is repeated until a point where no more changes can be made that improve with respect to the vertex order (a fixed point). When this happens the strategy of the optimizing player is optimal. With these two optimal strategies we can subsequently compute the winner of the parity game, thus solving the parity game.

We want to know if ACO can help to speed up strategy improvement. We think it can be applied to decide how to improve a strategy. In the previous chapter we have applied ACO to parity games. The problem of that approach was that it was uncertain if the solution was correct. When helping SI with ACO this problem does not occur because SI will always find the correct solution. In the next part of this section we will explain the Strategy Improvement algorithm in more detail. After that we explain why applying ACO in this algorithm is a good idea. After that we will show how we applied ACO and the results thereof.

6.1. Strategy Improvement Algorithm

Let us start with explaining what a *strategy* is, and how we can use it. Recall that the vertices of a parity game are divided between two players, odd (1) and even (0) (see section 5). The set of vertices owned by even is V_0 and the set of vertices owned by odd is V_1. We define a strategy for player $i \in \{0, 1\}$ as a function $\sigma_i : V_i \to V$. For every vertex owned by player i the strategy will give the vertex she will play to. If we have a strategy for both players we can determine who wins which vertices. We can do this by starting in a vertex and playing according to the strategies until a loop is found. The parity of lowest priority in that loop determines who wins the play. A strategy is a winning strategy for a vertex v if it is winning all possible plays that start in v against all possible opponent strategies.

We introduce a generic strategy improvement algorithm. After that we show the implementation of this generic algorithm that we have used. The ideas explained below are from (Vöge & Jurdziński, 2000). In order for a player to optimize its strategy we use a pre-order $>$ on strategies for this player. This pre-order should satisfy the following two postulates:

Postulate 6.1. Given a finite set of strategies. There is a maximum element in this set with regard to the pre-order $>$.

Postulate 6.2. If σ_i is the maximum element with regard to $>$, it is winning for every vertex where player i can win.

The maximum winning strategy can be found using an order that satisfies the postu-

lates listed above. Now we define a function $improve : (V_i \to V) \to (V_i \to V)$. It takes a strategy and it returns a strategy. This function intends to improve a strategy, it should satisfy:

Postulate 6.3. If σ_i is not a maximum element given the preorder $>$, then $improve(\sigma_i) > \sigma_i$.

Postulate 6.4. If σ_i is a maximum element given the preorder $>$, then $improve(\sigma_i) = \sigma_i$.

Listing 5: Strategy improvement algorithm

```
strategy_improvement() {
  sigma = some_strategy_for(some player)
  do {
    sigma' = sigma
    sigma = improve(sigma)
  } while (sigma' != sigma)
}
```

We can use the algorithm in listing 5 to find the solution to a parity game. We will refer to an iteration of the do-while loop as an iteration of the strategy improvement algorithm. In an iteration an ordering on vertices is created and the strategy of the optimizing player is improved with respect to this valuation.

Theorem 6.5. *If a pre-order $>$ on strategies satisfies postulates 6.1 and 6.2, and a function improve satisfies postulates 6.3 and 6.4, then the algorithm in listing 5 is a correct algorithm to find the solution for a parity game.*

It is easy to see, based on the monotonicity of the improve function, that applying improve recursively will result in a fixed point. The value of this fixed-point is maximum for the ordering. An element that is maximum is an optimal strategy because of postulate 6.2. A proof for the correctness of this algorithm can be found in (Vöge & Jurdziński, 2000).

We will now describe how we can create a pre-order on strategies and an *improve* function. We start with defining a pre-order on strategies. This pre-order is based on an order on vertices. We introduce a total order on vertices $\preceq$.

We will define a pre-order on strategies based on an order on vertices $\preceq$. To do this we will introduce some definitions. We will do this first. To define $\preceq$, some data to indicate the value of a vertex given a pair of strategies is used. This data is called a *play profile*. In order to define a play profile we introduce an order:

Definition 6.6 (Reward order)**.** The reward order $<_{rw}$ is an order on vertices based on the priority of the vertex. Recall that $p : V \to \mathbb{N}$ gives the priority for a vertex. In the equation below we use v instead of $p(v)$ for readability.

$$v <_{rw} w \iff \begin{cases} \text{even}(v) \wedge \text{even}(w) \wedge v > w \vee \\ \text{odd}(v) \wedge \text{odd}(w) \wedge v < w \vee \\ \text{odd}(v) \wedge \text{even}(w) \end{cases}$$

The reward for a vertex is the value of the vertex as seen by the optimizing player. Vertices with the highest relevance and the parity of the optimizing player have the highest reward, but players with the highest relevance and parity of the opponent

player have the lowest reward. An example of this order is:

$$1 <_{rw} 3 <_{rw} 5 <_{rw} 7 <_{rw} 6 <_{rw} 4 <_{rw} 2 <_{rw} 0$$

Definition 6.7 (Play Profile)**.** A play profile for a play π is a triple of:

(1) The vertex with the lowest priority that is visited infinitely often u_π (the winning vertex).
(2) The set of vertices P_π that have a lower priority than u_π and occur in the play.
(3) The number of vertices e_π visited before the first visit of u_π.

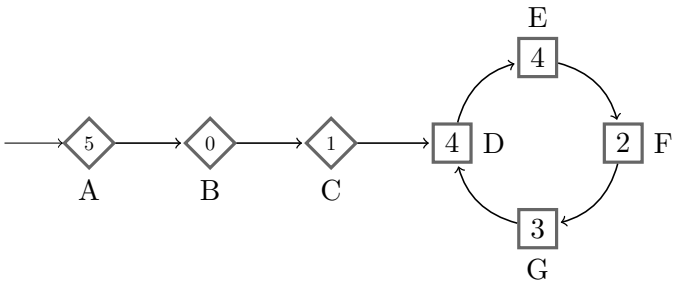


Figure 18.: Parity game

Example 6.8 (Play profiles for a simple game)**.** We create a play profile for vertex A of figure 18. The winning vertex for a play starting in vertex A is vertex F. Nodes with a lower priority than the winner (vertex F) between A and F are vertex B and vertex C. The distance from vertex A to vertex F is 5. The play profile for A becomes $(F, \{A, B\}, 5)$.

We create a play profile for vertex D of figure 18. The winning vertex for a play starting in vertex D is vertex F. There are no nodes with a lower priority than the winner between D and F. The distance from vertex D to vertex F is 2. The play profile for D becomes $(F, \{\}, 2)$.

A play profile can be constructed for a vertex by fixing the strategy for one player (the optimizing player) and using an optimal counter-strategy for the other player. This optimal counter-strategy is easy to compute if the strategy is fixed, see (Vöge & Jurdziński, 2000) for an algorithm. A valuation is a function that assigns a play profile to every vertex based on strategies for both players. It is defined as $\varphi : (V_i \to V) \times (V_{1-i} \to V) \times V \to D$, where D is the set of play profiles. For every vertex v the strategies σ_i and σ_{1-i} induce a play $\pi_{v,\sigma_i,\sigma_{1-i}}$. This play gives a play profile for the vertex v.

We examine how to compare the play profiles in order to improve the strategy for the optimizing player. To do this we first introduce the reward order on sets.

Definition 6.9 (Reward order on sets $<_{rw}$)**.**

$$min(P, Q) = \arg\min\{p(v) \mid v \in P \triangle {}^2Q\}$$

[2]symmetric difference $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$

$$P <_{rw} Q \iff \begin{cases} min(P,Q) \in P \wedge \text{odd}(min(P,Q)) \vee \\ min(P,Q) \in Q \wedge \text{even}(min(P,Q)) \end{cases}$$

This allows us to compare play profiles.

Definition 6.10 (Order on play profiles)**.**

$$(u,P,e) \preceq (v,Q,f) \iff \begin{cases} u <_{rw} v \vee \\ u = v \wedge P <_{rw} Q \vee \\ u = v \wedge P = Q \wedge (\text{odd}(u) \wedge e < f) \vee \\ u = v \wedge P = Q \wedge (\text{even}(u) \wedge e > f) \end{cases}$$

This comparison consists of three parts. A vertex is *positive* for a player if it has the same parity as that player and is negative for that player otherwise.

- The vertex u has a lower reward than v. (We prefer plays that we win, or are more likely to win.)
- The element with the highest relevance in $(P \triangle Q)$ is in P and is negative for the optimizing player or is in Q and is positive. (Maybe a loop could be found from this vertex.)
- e is higher than f if u is a positive vertex for the optimizing player (The optimizing player wins this play), if u is a negative vertex then if e is lower than f.

Example 6.11 (Comparison of play values)**.** Given play profiles $A = (3, \{0,1\}, 2)$ and $B = (3, \{0,2\}, 4)$, let us determine if $A \preceq B$. The first elements are equal $(3 = 3)$ so we compare the second elements. The symmetric difference of $\{0,1\}$ and $\{0,2\}$ is $\{1,2\}$. The minimum of this set is 1, which is in A and is odd. This means that $\{0,1\} <_{rw} \{0,2\}$ (see definition 6.9) and therefore $A \preceq B$.

Definition 6.12 (pre-order on strategies based on vertex values)**.** We define a pre-order on strategies where a strategy σ' has a higher value than σ if for all vertices $v_i \in V_i$ it holds that $\sigma(v_i) \preceq \sigma'(v_i)$, i.e every choice in the strategy has a higher or equal value with respect to $\preceq$.

The intuition behind $\preceq$ is that if the optimizing player is more likely to win from that vertex, its value is higher. The order $\preceq$ can be constructed from a given strategy σ_i for player i and an optimal strategy of its opponent $\sigma_{1-i,opt}$. This completes the pre-order on strategies that was defined in definition 6.12.

Now that we have a pre-order on strategies we define a version of *improve* that works with this pre-order. The function *improve* does two things. It creates a play profile for every vertex. This is done by creating optimal response strategy `optimalResponse(game, sigma)`. With two strategies a play profile can be created for every vertex. It improves the strategy based on the play profiles of the valuation `switch(valuation, sigma)`. At the start of this section we have seen that we improve a strategy if, for every vertex owned by the optimizing player we choose a vertex with a play profile that is higher then the current play profile. Pseudocode for the `improve` function can be found in listing 6. An optimal response strategy for a strategy σ_i can be found in the following way. First a sub-game is created by removing all edges of player i except the edges in σ_i. In this sub-game loops are searched for in increasing reward (1,3, ...2,0). The set of vertices that can reach the loop is calculated, the at-

Listing 6: Improve function

```
improve(game, sigma) {
  valuation = valuation(game, sigma)

  return switch(valuation, sigma)
}

valuation(game, sigma) {
    counterStrategy = optimalResponse(game, sigma)

    return valuation(game, sigma, counterStrategy)
}
```

tractor set. All vertices leaving the attractor or the loop are removed. When this is done all edges of player $1 - i$ form the strategy for $1 - i$.

Definition 6.13 (Switching policy)**.** The function that selects a vertex from the set of feasible vertices for every vertex in the strategy is called the switching policy. The piece of code that implements `switch(valuation, sigma)`.

The standard strategy improvement algorithm will select the node with the largest play profile, but there are often more vertices that have a higher play profile and that we can switch to. Choosing the largest play profile is not always the best choice.

6.2. A motivation for a different switching policy

There is a simple example where SI uses much more iterations than required. Notably, in this example, using ACO the correct solution can be generated immediately. The example can be found in figure 19. Other examples with a similar structure can easily be created, the pattern is that the priorities are all odd and decrease except for the last priority, which should be even. Consider the example in figure 19. This example

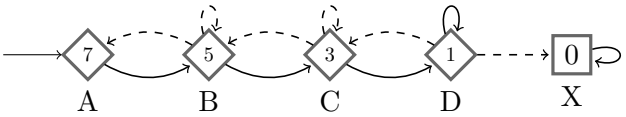


Figure 19.: Hard game for SI

consists of a game and a strategy. In this example the strategy of even is denoted by a solid line and edges that are not in even's strategy by a dotted line. Remember that a play profile is a triple containing the vertex v with the lowest priority, vertices that have a lower priority than v and are visited before v and the amount of vertices visited before v. Remember that $\varphi : (V_i \to V) \times (V_{1-i} \to V) \times V \to D$ is a function that

assigns play profiles to vertices. The play profiles under this strategy for even are:

$$\varphi(A) = (4, \{\}, 3)$$
$$\varphi(B) = (4, \{\}, 2)$$
$$\varphi(C) = (4, \{\}, 1)$$
$$\varphi(D) = (4, \{\}, 0)$$
$$\varphi(X) = (5, \{\}, 0)$$

Now let us look at how even would improve its strategy under the default switching policy (picking the vertex with the largest play profile).

- Vertex A: There is only one option, no change.
- Vertex B: The options are A, B and C. Node A has the highest value (it has the largest distance to a vertex where even loses).
- Vertex C: The options are B, C and D. Node B has the highest value so it will be selected.
- Vertex D: The options are C, D and X. Node X has the highest value (even wins) so it will be selected.

Vertex X is owned by odd and therefore the strategy of even does not decide where vertex X should play to.

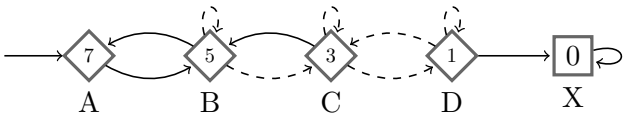


Figure 20.: Strategy after one update

The strategy after the update can be found in figure 20. We see that the new strategy is better than the previous strategy because even wins vertex D and X instead of vertex X. However, it could have been much better if a different switching policy was used. Per iteration only one node extra will play to node X.

6.3. Applying ACO

We research three approaches of applying ACO in Strategy Improvement (section 6.1). The first approach is to let ACO generate an initial strategy and use SI to correct the strategy. The second approach is to use ACO to update the strategy with the constraint that it can only increase valuations. In this approach we always find a correct solution, the question is if using ACO helps us to find the answer faster. The third approach is to use ACO to update the strategy with valuations as guidance. Because this approach does not guarantee that play values are increased, the algorithm might not terminate. We prevent this by adding a final stage where the ants do not operate anymore. The implementation of these approaches can be found in appendix D.

We expect that this problem suits ACO very well because: If the algorithm makes wrong choices the impact is limited: It can slow down the process of finding a solution, but SI will still find the correct solution. The problem of finding optimal choices is a hard problem that has to be executed many times in the run of the algorithm.

Applying an optimal algorithm to it would slacken the algorithm because of the extra computation time. Using a naive method like taking a random switch might lead to slow optimization or exponential behaviour. ACO could be a good middle ground between these two approaches, we will research if it is. In this section we will first explain how we used ACO, and after that we will show results of this research.

The first approach is to let ACO generate an initial strategy which SI uses. We let the ants generate a strategy for the parity game in the same manner as explained in section 5. Ants will create plays, deposit pheromone on choices that the winner made and remove pheromone on choices that the loser made. In the final terrain, for every vertex the successor with the highest amount of pheromone is included in the strategy. This is how an initial strategy is generated. The Strategy Improvement algorithm uses this strategy and uses the default switching policy. We want ACO to prefer strategies with a higher valuation. To do this we add a bonus for the priority of the winning node in a play, the lower the priority is, the higher the bonus. With ran experiments to compare initial strategies created by ants versus random initial strategies. The

experiment name	random (iterations, time)	ACO (iterations, time)
steadygame(10,2)	(2.8, 0.009s)	(2.0, 0.010s)
steadygame(100,2)	(6.0, 0.028s)	(6.0, 0.040s)
steadygame(1000,2)	(22.6, 1.109s)	(18.4, 1.033s)
steadygame(10,5)	(3.0, 0.001s)	(3.2, 0.002s)
steadygame(100,5)	(6.0, 0.003s)	(6.2, 0.020s)
steadygame(1000,5)	(10.6, 0.161s)	(10.0, 1.207s)
steadygame(10,10)	(3.2, 0.000s)	(2.8, 0.001s)
steadygame(100,10)	(5.6, 0.005s)	(4.6, 0.019s)
steadygame(1000,10)	(10.6, 0.246s)	(10.2, 2.444s)

Table 10.: Results of ACO generating an initial strategy.

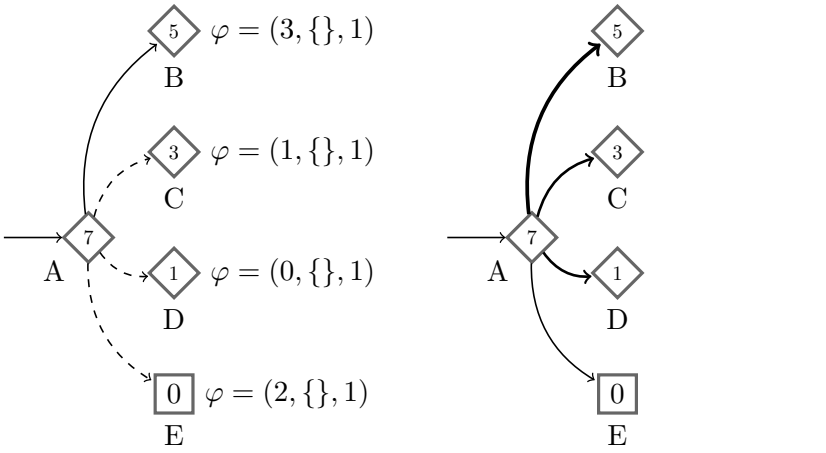
experiments found in table 10 are experiments on games with a different number vertices and average out-degree. The *steadygame* games are semi random games, they are generated using tools from *pgsolver* (Friedmann & Lange, 2019). They are in general hard to optimize for and therefore make good benchmarks. The out-degree is the average number of edges originating from a vertex. The first parameter is the number of vertices, the second is the average out-degree. We ran five instances per experiment. The running time and the amount of iterations in the table are an average of the results of the instances. We used ACO with 20 generations in order to create the initial strategy. We have verified that this is long enough to create a good solution. We measure the speed of finding a solution in the amount of iterations that are required to find a solution, and by the computation time that is used, including initialization time.

We can see that there is not much improvement in using ACO to generate an initial strategy. In cases where SI needs a lot of iterations to get a result, ACO seems to speed up the algorithm. It might be that the strategy created by the ants is not similar enough to the strategy with a maximal valuation.

Another approach that we tested is to use ACO to update the strategy. This is motivated by the problem with SI as described in section 6.2. Recall that in every iteration of the strategy improvement algorithm a valuation is created for the current strategy and an improved strategy is created based on this valuation. A strategy is improved when choices are made to vertices with higher play profiles. There is often

more than one vertex with a higher play value. We use ACO to determine which of these options to choose.

We do this by running ACO for several generations and using the terrain to make choices. We select the vertex with the highest pheromone level. We set a budget for the number of generations that ACO can run. Every iteration of SI, ACO uses half of its budget. This happens until the budget is gone, then the default switching policy is used again. This implementation makes the amount of computation time for ACO regulatable. We save and restore the terrain created by ACO between iterations to improve performance. In ACO information from previous generations is used to create better solutions in the current generation. Therefore we want to retain the information from previous generation. We illustrate how we applied ACO by an example.



(a) Valuation and current strategy (b) Terrain created by ACO

Figure 21.: Choices for a switching policy

Example 6.14 (Choices in a switching policy)**.** In figure 21a we see a vertex, vertex A, from which a choice has to be made in order to improve the strategy. The current strategy is to play to vertex B. The valid choices from this vertex are vertices with an equal or higher play profile, they are B, D and E. We use ACO to make a decision between vertices. We let ACO run and generate a terrain, after that we choose the vertex with the most pheromone on it. In figure 21b the edge thickness indicates the amount of pheromone on that edge. In this example we would choose vertex B. Notice that this is different from the default switching policy. The default switching policy selects the vertex with the highest play profile, which is vertex D in this case.

In all tests we ran, ACO performed worse than the default switching policy. We think that this might be because the ants do not create the same strategy as the strategy SI searches for. Another problem might be that the strategy that ACO generates is better but in the current valuation SI cannot see that.

This motivates us to check what happens if we allow changing the strategy to a lower valuation. We are interested if ignoring the rule to increase valuations might lead to finding a solution quicker. If this is done there is a chance that the algorithm might not terminate. However, this problem is mitigated because we set a generation budget for ACO and if that budget runs out the default switching policy will be used again. We have implemented the following rules for updating the strategy:

- If the choice created by ACO is a clear choice (pheromone is 1.5 times than the average) use that.
- Otherwise take the vertex with the highest valuation.

We ran experiments using the same benchmarks as we used when generating an initial strategy. We compare the performance of the switching policy where violations are allowed against the default switching policy.

experiment	default (iterations, time)	ant (iterations, time)
steadygame(100,5)	(6.4, 0.005s)	(12.8, 0.576s)
steadygame(1000,5)	(11.2, 0.244s)	(16.6, 49.372s)
steadygame(100,2)	(6.2, 0.011s)	(11.6, 0.567s)
steadygame(1000,2)	(18.8, 0.930s)	(28.0, 23.785s)
steadygame(10,10)	(4.0, 0.001s)	(9.4, 0.019s)
steadygame(10,2)	(2.4, 0.011s)	(6.6, 0.038s)
steadygame(100,10)	(5.6, 0.004s)	(11.2, 0.898s)
steadygame(10,5)	(3.0, 0.001s)	(8.4, 0.014s)
steadygame(1000,10)	(9.8, 0.180s)	(16.6, 86.920s)

Table 11.: Results of SI with violations

In table 11 we see that in all cases ACO makes SI perform worse. When the budget of ACO has run out SI starts to correct the strategy again using the default switching policy.

At this moment there is no interaction between the terrain created by the ants and the valuation. The advantage of linking the terrain and the valuations might be that ants do not have to explore the whole game, but only the part of interest. The weakness of a combination is that valuations are only based on the current strategy for the optimizing player. So it does not represent the "real" value of a vertex for a player. Because of this weakness we have chosen not to link the terrain and valuations.

A practical advantage of using ACO as a switching policy is that it can run in parallel to the rest of the algorithm. In particular it can be run in parallel to the creation of a valuation. So that, with proper tuning, it can be implemented in a way so that it does not make the improvement steps take more computation time.

We repeat every run three times to get a reasonable sample of the performance of the algorithm, which varies because SI is initialized with a random strategy by default and because ACO is stochastic. In all benchmarks ACO used 200 ants and ran for 160 generations to come up with an initial strategy and ran for 40 generations per strategy improvement iteration. With these values ACO is able to create a reasonable strategy for most games.

6.4. Biased ants with valuations

We tried to apply a version of random against best ants for the strategy improvement algorithm. The idea is to use valuations to create optimal plays for player odd. When an ant creates a play it will select an edge based on pheromone levels for player even. For player odd it will select the edge leading to the vertex with the lowest play profile, this is because a play profile indicates the value of a vertex for player even, player odd wants to minimize that value.

This approach was not fruitful. An hypotesis is that this is because the strategy of

odd is only optimal against the current strategy of even. It might be that it is a very bad strategy if even would make different choices.

6.5. Results on simple games

We illustrate the ACO can work as a switching policy by applying it to simple games. The first benchmark that we used are versions of the game that was shown in section 6.2.

experiment	number of vertices	default iterations	ACO iterations
hard(10)	11	11,10,11	1,1,0
hard(100)	101	101,100,101	1,0,1
hard(500)	1001	498,501,501	1,1,1

Table 12.: Results of ACO on SI hard game.

As can be seen from the results in table 12 this game is rather simple for ACO as a switching policy. This indicates that ACO could be used as a switching policy, but this test case is simple so we try a more difficult test case.

The next benchmark is a benchmark on which strategy improvement algorithms should show exponential behaviour.

experiment	number of vertices	default iterations	ACO iterations
cunningham(2)	19	4,6,5	3,1,2
cunningham(10)	91	10,6,8	9,3,3
cunningham(20)	181	20,24,23	22,23,23
cunningham(100)	901	103,99,103	101,103,103

Table 13.: Results of ACO on other SI hard games.

It is interesting to see that the advantage of using ACO decreases when the problem size increases. On this benchmark we found that using ACO did not improve performance a lot.

In our research on ACO for parity games we used parity games that were created from the model checking problem. This conversion created parity games where there were no choices for even. Because of this they are always solvable in one iteration and therefore do not make a good benchmark.

6.6. Conclusion

Thus far, when applying ACO to Strategy Improvement, there is not much improvement with respect to the standard algorithm. In simple examples we found that ACO outperformed the standard strategy improvement algorithm. Therefore we assume that using ACO might also improve performance in general. It could be that we have not done sufficient tuning so that the choices of the ants are on average worse than choosing the maximum play profile. It could also be the case that the strategy created by ACO is very different from the strategy created by SI. The only conclusion that we can make here is that an implementation of ACO for parity games can be used as a switching policy for Strategy Improvement.

The problem with a negative result on an algorithm that requires tuning is that it is hard to determine the cause of the negative result. The idea that it might have worked if some extra tuning or technique was used remains until we find a fundamental flaw. We are in this state, we have a negative result but have not found any reason why this approach is fundamentally incorrect.

6.7. Future Work

One future direction of research is to see if ACO for parity games can be adjusted to search for strategies which are similar to strategies the SI searches for (strategies with a maximum valuation). There are multiple points that made the application of ACO as a switching policy hard to debug. The first being that for large games it is hard to if what ACO does is correct, so if the advice given by ACO is valid. There are variants of ACO which require less tuning, applying one of these might help prevent the problems we found. The second is that it is unclear what happens if the strategy found by ACO uses paths of decreasing valuation. In this case the switching policy makes a choice that is neiter good according to the valuation nor good according to the ants, it might be interesting to research how this problem can be solved and if it makes a large impact on performance. Another approach might be to reset pheromone levels if the playprofile clearly indicates that a path leads to a bad play.

7. Conclusion

The research question we started with was if ACO could be used to solve the model checking problem. We have researched this by applying ACO to Boolean equation systems and parity games, which are both problems to which the model checking problem can be transformed. We start this conclusion by evaluating the results we got for ACO for BES, parity games and in strategy improvement. After that we will discuss what we learned about implementing and using ACO.

We started our research with applying ACO to solving BES. ACO applied to BES is not particularly useful as it gives a correct solution with a very low probability. While for the BES problem we want to be certain that the solution is correct. The fundamental problem is that no heuristic could be found that takes the fixed points into account correctly and is computationally easy enough to be executed very often during the optimization process. In hindsight we should have noticed this problem earlier and concluded that applying ACO to BES directly was not possible. For this part of the research we conclude that ACO cannot be applied to solve BES directly. We have learned that it is crucial to have a valuation function that does not give false positives, i.e. it should not mark a solution correct that is incorrect.

After that we applied ACO to parity games. For parity games ACO seems to find the correct solution of a parity game with a high probability. Having an algorithm that finds the correct solution of a parity game most of the time is not what we want because we cannot verify if a solution is correct. We conclude that ACO can be applied to solve parity games but that it is not reliable and therefore not useful in practice. We have learned that we can apply ACO in a creative manner. The problem of a parity game is to find optimal strategies, but we were able to apply ACO by letting ants generate plays instead of optimal strategies.

Finally we used ACO as part of a strategy improvement algorithm. When ACO is applied as a switching policy in strategy improvement it is able to prevent exponential behaviour of the strategy improvement algorithm on some test sets. However it does not improve performance on more complex problems. We have not been able to fully debug the problem so we cannot conclude whether this result indicates a fundamental flaw or a problem that is solvable.

It is interesting to see how the structure of the problems affects the effectiveness of ACO for that problem. For BES it was not possible to create a good heuristic and therefore the algorithm could not find good solutions. For parity games there was a good heuristic (the winner) that determined which plays were good and bad. In this way the correct solution was frequently found but the problem of incorrect solutions remained. In the application of ACO in strategy improvement the problem of finding a wrong solution was resolved because the valuations restricts the choices of the ants to good choices.

The answer to the main research question is that we do not know. No research has previously been done in this area. We have tried different approaches and learned how to apply ACO and gained a better understanding of the problems derived from the model checking problem. We know that applying it directly to BES does not work. That applying it to parity games can work. But we have not found an answer to the most interesting and promising question. Namely if ACO can be used to improve the strategy improvement algorithm. More research has to be done in this area.

8. Future work

As stated in the conclusion this was the first research in this area. There are promising ways to continue research. The question if ACO or other optimization algorithms can be used to improve strategy improvement remains an open question. It might be interesting to research ACO or other optimization methods for the switching policy of strategy improvement. In general, the most promising way of applying ACO for model checking is as a part of an algorithm that is correct, so that a correct solution is always found. Strategy improvement is one such example but there might be more algorithms in which ACO can be used.

References

Blum, C., & Dorigo, M. (2004, April). The hyper-cube framework for ant colony optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *34*(2), 1161-1172.

Calude, C. S., Jain, S., Khoussainov, B., Li, W., & Stephan, F. (2020). Deciding parity games in quasi-polynomial time. *SIAM Journal on Computing*.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: optimization by a colony of cooperating agents. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, *26*(1), 29–41.

Emerson, E. A., & Jutla, C. S. (1991). Tree automata, mu-calculus and determinacy (extended abstract). In *32nd annual symposium on foundations of computer science, san juan, puerto rico, 1-4 october 1991* (pp. 368–377). IEEE Computer Society.

Emerson, E. A., & Lei, C.-L. (1986). Model checking in the propositional mu-calculus.

Friedmann, O., & Lange, M. (2019). *pgsolver.* `https://github.com/tcsprojects/pgsolver`.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability* (Vol. 174). freeman San Francisco.

Glover, F. (1989). Tabu searchpart i. *ORSA Journal on computing*, *1*(3), 190–206.

Groote, J. F., & Mousavi, M. R. (2014). Modeling and analysis of communicating systems..

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, *220*(4598), 671–680.

Kozen, D. (1982). Results on the propositional mu-calculus, in ninth international colloquium on automata. *Languages, and Programming, pp*, 348–359.

Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, *16*(1963), 83–94.

Mader, A. (1997). *Verification of modal properties using boolean equation systems.* Edition versal 8.

Moritz, D., & Springer, M. (2010). Solving satisfiability with ant colony optimization and genetic algorithms.

Mostowski, A. W. (1991). *Games with forbidden positions* (Tech. Rep.). University of Gdansk.

Put, E. v. d. (2019). *Thesis implementation.* `https://gitlab.com/nouwaarom/thesis`.

Stirling, C. (1995). Local model checking games. In *International conference on concurrency theory* (pp. 1–11).

Vander Meer, R. K., Breed, M. D., Winston, M., & Espelie, K. E. (2019). *Pheromone communication in social insects: ants, wasps, bees, and termites.* CRC Press.

Vöge, J., & Jurdziński, M. (2000). A discrete strategy improvement algorithm for solving parity games. In *International conference on computer aided verification* (pp. 202–215).

Zielonka, W. (1998). Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, *200*(1-2), 135–183.

Appendix A. Framework

Listing 7: Ant Colony Optimizer

```
package framework

import arrow.core.None
import arrow.core.Some
import arrow.core.getOrElse
import kotlin.math.ceil

class AntColonyOptimizer(
    private val numberOfAnts: Int,
    private val maxNumberOfCycles: Int,
    val ant: Ant,
    val evaluator: Evaluator,
    private val evaporationCoef: Double = 0.5,
    // The fraction of ants that are allowed to deposit pheromone.
    private val pheromoneFraction: Double = 0.25
) {
    init {
        require(numberOfAnts > 0) { "Cannot␣execute␣with␣less␣than␣1␣ant" }
    }

    lateinit var terrain: Terrain
        private set

    val quality: DoubleArray = DoubleArray(maxNumberOfCycles)
    var bestQuality: Double = Double.MAX_VALUE
    var bestSolution: SolutionCandidate? = null

    var listener: GenerationDoneListener? = null
        set(deb) {
            deb?.initialize(maxNumberOfCycles)
            field = deb
        }

    // Returns (isSolutionCorrect, foundInGeneration, bestSolution)
    fun execute(initialTerrain: Terrain, _forGenerations: Int = maxNumberOfCycles):
            Triple<Boolean, Int, SolutionCandidate?> {
        terrain = initialTerrain

        for (generation in 0 until maxNumberOfCycles) {
            val result = runGeneration(generation)
            if (result.first) {
                return Triple(result.first, generation, result.second)
            }
            //println("generation: $generation")
        }

        return Triple(false, _forGenerations - 1, null)
    }

    private fun runGeneration(generation: Int): Pair<Boolean, SolutionCandidate?> {
        val evaluations = (0 until numberOfAnts).map {
            val solution = ant.search(terrain).copy()

            val evalOption = evaluator.evaluate(solution)
```

```
            if (evalOption.isDefined()) {
                val eval = evalOption.getOrElse { throw Exception("Critical error") }
                if (eval.isCorrect) {
                    return Pair(true, solution)
                }
                Some(Pair(solution, eval))
            } else {
                None
            }
        }.filter { x -> x.isDefined() }.map {
                x -> x.getOrElse { throw Exception("Critical error") }
        }

        terrain.evaporate(evaporationCoef)

        val sorted = evaluations.sortedBy { x -> x.second.quality }
        sorted.take(ceil(numberOfAnts * pheromoneFraction).toInt())
            .forEach { e -> terrain.addPheromoneFrom(e.second.terrain) }

        val qualityOfBestSolution = sorted.first().second.quality
        quality[generation] = qualityOfBestSolution

        listener?.onGenerationDone(sorted.first().first, qualityOfBestSolution, terrain)

        if (qualityOfBestSolution < bestQuality) {
            bestQuality = qualityOfBestSolution
            bestSolution = sorted.first().first
        }

        return Pair(false, sorted.first().first)
    }
}
```

Listing 8: Ant

```
package framework

interface Ant {
    fun search(terrain: Terrain): SolutionCandidate
}
```

Listing 9: Evaluator

```
package framework

import arrow.core.Option

interface Evaluator {
    fun evaluate(solutionCandidate: SolutionCandidate): Option<Evaluation>
}

data class Evaluation(val quality: Double, val terrain: Terrain, val isCorrect: Boolean)
```

Listing 10: Terrain

```
package framework

/**
 * The terrain holds the pheromone for a problem.
```

```
 */
interface Terrain {
    fun evaporate(rho: Double)

    fun addPheromoneFrom(terrain: Terrain)

    fun copy(): Terrain
}
```

Listing 11: SolutionComponent

```
package framework

interface SolutionComponent {
}
```

Listing 12: SolutionCandidate

```
package framework

interface SolutionCandidate {
    fun copy(): SolutionCandidate
}
```

Appendix B. Implementation for BES

Listing 13: BooleanEquationSystem

```
package bes

data class BooleanEquationSystem(val equations: List<BooleanEquation>) {
    override fun toString(): String =
        "BooleanEquationSystem:\n" + equations.map { e -> e.toString() }.joinToString("\n")

    val numberOfVariables = equations.size

    fun clone(): BooleanEquationSystem {
        return BooleanEquationSystem(equations.map { e -> e.clone() })
    }
}

data class BooleanEquation(
    val variable: Int,
    val fixedPointOperator: FixedPointOperator,
    var booleanExpression: BooleanExpression
) {
    fun isTruthValue(): Boolean {
        return booleanExpression is TruthValue
    }

    fun clone(): BooleanEquation {
        return BooleanEquation(variable, fixedPointOperator, booleanExpression)
    }

    override fun toString(): String {
        return fixedPointOperator.value + "␣X" + variable +
                "␣=␣" + booleanExpression.toString()
    }
}
```

```
sealed class BooleanExpression

data class TruthValue(val value: Boolean) : BooleanExpression() {
    fun negate(): TruthValue {
        return TruthValue(!value)
    }

    override fun toString(): String {
        return value.toString()
    }
}

data class Variable(val name: Int) : BooleanExpression() {
    override fun toString(): String {
        return "X%d".format(name)
    }
}

data class Not(val body: BooleanExpression) : BooleanExpression()
data class And(
    val left: BooleanExpression,
    val right: BooleanExpression
) : BooleanExpression()

data class Or(
    val left: BooleanExpression,
    val right: BooleanExpression
) : BooleanExpression()

data class Impl(
    val left: BooleanExpression,
    val right: BooleanExpression
) : BooleanExpression()

enum class FixedPointOperator(val value: String) {
    MU("mu"), NU("nu")
}
```

Listing 14: SatTerrain

```
package sat

import framework.Terrain
import java.lang.IllegalArgumentException
import kotlin.random.Random

class SatTerrain : Terrain {
    val positivePheromone: Array<Double>
    val negativePheromone: Array<Double>

    constructor(numberOfVariables: Int) {
        positivePheromone = Array(numberOfVariables) { 0.0 }
        negativePheromone = Array(numberOfVariables) { 0.0 }
    }

    constructor(positive: Array<Double>, negative: Array<Double>) {
        positivePheromone = positive
        negativePheromone = negative
```

```
    }

    override fun copy(): Terrain {
        return SatTerrain(positivePheromone.copyOf(), negativePheromone.copyOf())
    }

    fun addPheromone(variable: Int, value: Boolean, amount: Double) {
        if (true == value) {
            positivePheromone.set(variable, amount)
        } else {
            negativePheromone.set(variable, amount)
        }
    }

    companion object {
        fun createRandom(numberOfVariables: Int): SatTerrain {
            val terrain = SatTerrain(numberOfVariables)
            terrain.positivePheromone.forEachIndexed { i, _ ->
                terrain.positivePheromone.set(i, Random.nextDouble())
            }
            terrain.negativePheromone.forEachIndexed { i, _ ->
                terrain.negativePheromone.set(i, Random.nextDouble())
            }

            return terrain
        }
    }

    override fun evaporate(rho: Double) {
        positivePheromone.forEachIndexed { i, x -> positivePheromone.set(i, x * rho) }
        negativePheromone.forEachIndexed { i, x -> negativePheromone.set(i, x * rho) }
    }

    override fun addPheromoneFrom(terrain: Terrain) {
        if (terrain !is SatTerrain) {
            throw IllegalArgumentException("Can␣only␣add␣SatEnvironment")
        }

        positivePheromone.forEachIndexed { i, x ->
            positivePheromone.set(i, x + terrain.positivePheromone.get(i))
        }
        negativePheromone.forEachIndexed { i, x ->
            negativePheromone.set(i, x + terrain.negativePheromone.get(i))
        }
    }

    override fun toString(): String {
        return positivePheromone.joinToString { x ->
            "%.3f".format(x)
        } + "\n" + negativePheromone.joinToString { x ->
            "%.3f".format(x)
        }
    }
}
```

Listing 15: Assignment (Solution Candidate)

```
package bes
```

```
import framework.SolutionCandidate

data class Assignment(val assignedVariables: BooleanArray) : SolutionCandidate {
    override fun copy(): SolutionCandidate {
        return Assignment(assignedVariables)
    }

    override fun toString(): String {
        return "Assignment(" +
                assignedVariables.mapIndexed { i, l -> "X$i:$l" }.joinToString("␣") +
                ")"
    }

    fun debugCompareTo(other: Assignment, context: BooleanEquationSystem) {
        val isCorrect = (assignedVariables zip other.assignedVariables).mapIndexed { i, p ->
            if (p.first != p.second) {
                println("Variable␣X$i,␣should␣have␣been␣${p.first},␣but␣was␣${p.second}")
                println("In␣equation␣${context.equations[i]}")
            }
            p.first == p.second
        }.fold(true) { acc, b -> acc && b }
        if (isCorrect) {
            println("The␣assignments␣are␣equal.")
        }
    }
}
```

Listing 16: BesAnt

```
package bes

import framework.Ant
import framework.Terrain
import sat.SatTerrain
import kotlin.random.Random

class BesAnt(val bes: BooleanEquationSystem): Ant {
    override fun search(terrain: Terrain): Assignment {
        require(terrain is SatTerrain) { "Only␣works␣on␣Sat␣or␣Bes" }

        val pheromones = terrain.positivePheromone zip terrain.negativePheromone
        val assignment = (pheromones).mapIndexed { i, x ->
            // If the equation is a truth value, assign it.
            if (bes.equations[i].isTruthValue()) {
                (bes.equations[i].booleanExpression as TruthValue).value
            } else {
                Random.nextDouble(x.first + x.second) < x.first
            }
        }.toBooleanArray()

        return Assignment(assignment)
    }
}
```

Listing 17: DependencyAnalyzingBesAnt

```
package bes

import framework.Ant
```

```
import framework.Terrain
import sat.SatTerrain
import kotlin.random.Random

/**
 * When assigning a variable check which equations are set by that assignments.
 */
class DependencyAnalyzingBesAnt(val bes: BooleanEquationSystem) : Ant {
    private val trueDependencies: HashMap<Int, List<Int>> = HashMap()
    private val falseDependencies: HashMap<Int, List<Int>> = HashMap()

    init {
        bes.equations.reversed().forEach { e ->
            trueDependencies[e.variable] =
                findTrueDependencies(e.variable, bes.equations.take(e.variable))
            falseDependencies[e.variable] =
                findFalseDependencies(e.variable, bes.equations.take(e.variable))
        }
    }

    private fun findTrueDependencies(variable: Int, bes: List<BooleanEquation>): List<Int> {
        return bes.filter { be ->
            findTrueDependency(variable, be.booleanExpression)
        }.map { be -> be.variable }
    }

    // This is a very simple analysis, this might be made a bit more complex.
    private fun findTrueDependency(variable: Int, be: BooleanExpression): Boolean {
        return when (be) {
            is Or -> {
                findTrueDependency(variable, be.left) ||
                        findTrueDependency(variable, be.right)
            }
            is Variable -> (be.name == variable)
            is Impl -> false
            is TruthValue -> false
            is And -> false
            is Not -> false
        }
    }

    private fun findFalseDependencies(variable: Int, bes: List<BooleanEquation>): List<Int> {
        return bes.filter { be ->
            findFalseDependency(variable, be.booleanExpression)
        }.map { be -> be.variable }
    }

    // This is a very simple analysis, this might be made a bit more complex.
    private fun findFalseDependency(variable: Int, be: BooleanExpression): Boolean {
        return when (be) {
            is And -> {
                findFalseDependency(variable, be.left) ||
                        findFalseDependency(variable, be.right)
            }
            is Variable -> (be.name == variable)
            is Impl -> false
            is TruthValue -> false
            is Or -> false
            is Not -> false
```

```
        }
    }

    override fun search(terrain: Terrain): Assignment {
        require(terrain is SatTerrain) { "Only works on Sat or Bes" }

        val assignment = MutableList(bes.numberOfVariables) { Bit.Undefined }
        (0 until bes.numberOfVariables).reversed().forEach assign@{ i ->
            if (Bit.Undefined != assignment[i]) {
                return@assign
            }

            val equation = bes.equations[i]
            if (equation.isTruthValue()) {
                assignment[i] = fromBool((equation.booleanExpression as TruthValue).value)
            } else {
                val total = terrain.positivePheromone[i] + terrain.negativePheromone[i]
                val assigned = Random.nextDouble(total) < terrain.positivePheromone[i]
                if (assigned) {
                    assignment[i] = Bit.True
                    if (trueDependencies[i]?.isNotEmpty() == true) {
                        trueDependencies[i]?.forEach { d -> assignment[d] = Bit.True }
                    }
                } else {
                    assignment[i] = Bit.False
                    if (falseDependencies[i]?.isNotEmpty() == true) {
                        falseDependencies[i]?.forEach { d -> assignment[d] = Bit.False }
                    }
                }
            }
        }

        return Assignment(assignment.map { v -> Bit.True == v }.toBooleanArray())
    }
}
```

Listing 18: FPHeuristicBesAnt

```
package bes

import framework.Ant
import framework.Terrain
import sat.SatTerrain
import util.config.AntConfig
import kotlin.random.Random

class FPHeuristicBesAnt(
    val bes: BooleanEquationSystem,
    val config: AntConfig.FpHeuristicBesAntConfig
) : Ant {
    override fun search(terrain: Terrain): Assignment {
        require(terrain is SatTerrain) { "Only works on Sat or Bes" }

        val pheromone = terrain.positivePheromone zip terrain.negativePheromone
        val assignment = pheromone.mapIndexed { i, x ->
            val equation = bes.equations[i]
            // For a truth value it does not make sense to assign it a different value.
            if (equation.isTruthValue()) {
                (equation.booleanExpression as TruthValue).value
```

```
            } else {
                // Prefer the corresponding fixed point.
                val offset = if (equation.fixedPointOperator == FixedPointOperator.MU)
                    config.fpPenalty
                else
                    config.fpReward
                // Prefer setting a variable to true.
                Random.nextDouble(x.first + x.second) < x.first + offset
            }
        }.toBooleanArray()

        return Assignment(assignment)
    }
}
```

Listing 19: InitialBesEvaluator

```
package bes

import arrow.core.Option
import arrow.core.Some
import framework.Evaluation
import framework.Evaluator
import framework.SolutionCandidate
import sat.SatTerrain

class InitialBesEvaluator(val bes: BooleanEquationSystem) : Evaluator {
    override fun evaluate(solutionCandidate: SolutionCandidate): Option<Evaluation> {
        require(solutionCandidate is Assignment) { "Can only evaluate Assignment." }

        val quality = checkBooleanEquations(bes, solutionCandidate)
            .fold(0.0) { acc, b -> acc + b.second }
        val pairs = (solutionCandidate.assignedVariables).map { l ->
            if (l) Pair(quality, 0.0) else Pair(0.0, quality)
        }

        // The quality if the percentage of correct equations
        return Some(
            Evaluation(
                (bes.numberOfVariables - quality) / bes.numberOfVariables,
                SatTerrain(
                    pairs.map { p -> p.first }.toTypedArray(),
                    pairs.map { p -> p.second }.toTypedArray()
                ),
                quality >= bes.numberOfVariables
            )
        )
    }
}
```

Listing 20: ImportantFPBesEvaluator

```
package bes

import arrow.core.Option
import arrow.core.Some
import framework.Evaluation
import framework.Evaluator
import framework.SolutionCandidate
```

```
import sat.SatTerrain

class ImportantFPBesEvaluator(val bes: BooleanEquationSystem) : Evaluator {
    private val importantFPs: MutableList<Int> = mutableListOf()

    init {
        bes.equations.forEach { e ->
            if (isImportant(e.variable, e.booleanExpression)) {
                importantFPs.add(e.variable)
            }
        }
    }

    private fun isImportant(variable: Int, booleanExpression: BooleanExpression): Boolean {
        return when (booleanExpression) {
            is Variable -> booleanExpression.name >= variable
            is TruthValue -> false
            is And -> {
                isImportant(variable, booleanExpression.left) ||
                        isImportant(variable, booleanExpression.right)
            }
            is Or -> {
                isImportant(variable, booleanExpression.left) ||
                        isImportant(variable, booleanExpression.right)
            }
            is Impl -> {
                isImportant(variable, booleanExpression.left) ||
                        isImportant(variable, booleanExpression.right)
            }
            is Not -> isImportant(variable, booleanExpression.body)
        }
    }

    override fun evaluate(solutionCandidate: SolutionCandidate): Option<Evaluation> {
        require(solutionCandidate is Assignment) { "Can only evaluate Assignment." }
        val assignedLiterals = solutionCandidate.assignedVariables

        val quality = checkBooleanEquations(bes, solutionCandidate).fold(0.0) { acc, b ->
            val correctFPReward =
                if (importantFPs.contains(b.first.variable) &&
                    ((b.first.fixedPointOperator == FixedPointOperator.NU) ==
                            assignedLiterals[b.first.variable])
                )
                    1.0
                else
                    0.0
            acc + b.second + correctFPReward
        }
        val pairs = solutionCandidate.assignedVariables.map { l ->
            if (l) Pair(quality, 0.0) else Pair(0.0, quality)
        }

        return Some(
            Evaluation(
                (bes.numberOfVariables - quality) / bes.numberOfVariables,
                SatTerrain(pairs.map { p -> p.first }.toTypedArray(), pairs.map { p ->
                    p.second
                }.toTypedArray()),
                quality > 0.99 * bes.numberOfVariables
```

```
            )
        )
    }
}
```

Appendix C. Implementation for parity games

Listing 21: ParityGame

```
package pg.data

import java.lang.Integer.max

class Node(val id: Int, val even: Boolean, var priority: Int, val successors: List<Int>) {
    val parity = if (even) 0 else 1

    val predecessors = mutableListOf<Int>()

    override fun toString(): String {
        return this.id.toString() +
                (if (this.even) "␣<${this.priority}>␣" else "␣[${this.priority}]␣") +
                "␣successors␣" + successors.joinToString(",")
    }

    fun toShortString(): String {
        return this.id.toString() +
                (if (this.even) "␣<${this.priority}>" else "␣[${this.priority}]")
    }
}

class ParityGame {
    val nodes: MutableList<Node> = mutableListOf()
    var goalVertex: Int = 0
    var highestPriority: Int = 0
        private set

    val allNodes: Set<Int>
        get() {
            return nodes.indices.toSet()
        }

    fun addNode(node: Node) {
        this.nodes.add(node)

        highestPriority = max(highestPriority, node.priority)
    }

    fun getNode(nodeNumber: Int): Node {
        return this.nodes[nodeNumber]
    }

    fun getEdgeCount(): Int {
        return nodes.fold(0) { acc, x -> acc + x.successors.size }
    }

    override fun toString(): String {
        var result = ""
        for (i in this.nodes.indices) {
            result = result + "␣" + this.nodes[i].toString() + "\n"
```

```
        }

        return result
    }
}
```

Listing 22: PgTerrain

```
package pg.ant

import framework.Terrain
import pg.data.ParityGame
import kotlin.random.Random

/**
 * The terrain consists of pheromone levels for all outgoing edges per node.
 */
class PgTerrain : Terrain {
    var pheromones: List<List<Double>>

    constructor(game: ParityGame) {
        pheromones = game.nodes.map { n -> (n.successors.indices).map { 0.0 } }
    }

    constructor(pheromones: List<List<Double>>) {
        this.pheromones = pheromones
    }

    override fun evaporate(rho: Double) {
        pheromones = pheromones.map { node -> node.map { ph -> ph * rho } }
    }

    override fun addPheromoneFrom(terrain: Terrain) {
        require(terrain is PgTerrain) { "Can␣only␣add␣PgTerrain" }
        pheromones = pheromones.mapIndexed { i, node ->
            node.mapIndexed { j, ph -> ph + terrain.pheromones[i][j] }
        }
    }

    override fun copy(): Terrain {
        return PgTerrain(pheromones.map { i -> i.toList() })
    }

    override fun toString(): String {
        return pheromones.mapIndexed { i, ph ->
            i.toString() +
                    "␣(" + ph.joinToString(",") { j -> String.format("%.3f", j) } + ")"
        }.joinToString("\n")
    }

    companion object {
        fun createRandomFor(game: ParityGame): PgTerrain {
            val terrain = PgTerrain(game)
            terrain.pheromones = terrain.pheromones.map { n ->
                n.map { p -> Random.nextDouble(0.1, 0.5) }
            }
            return terrain
        }
    }
```

```
}
```

Listing 23: Play (Solution Candidate)

```
package pg.ant

import framework.SolutionCandidate
import pg.data.ParityGame

class BiasedPlay(
    steps: List<Step>,
    lassoStart: Int,
    val optimalForEven: Boolean
) : Play(steps, lassoStart) {
    override fun copy(): Play {
        return BiasedPlay(steps.toList(), lassoStart, optimalForEven)
    }
}

/**
 * A play is a series of nodes that are visited by the token during a play.
 */
open class Play(val steps: List<Step>, val lassoStart: Int) : SolutionCandidate {
    val length = steps.size

    override fun copy(): Play {
        return Play(steps.toList(), lassoStart)
    }

    fun toString(game: ParityGame): String {
        return steps.joinToString("\n") { s ->
            val to = game.nodes[s.node].successors[s.choice]
            val priority = game.nodes[s.node].priority
            "(node:${s.node} prio:$priority choice:${s.choice} to $to)"
        }
    }
}

// The choice made at the current node.
data class Step(val choice: Int, val node: Int)
```

Listing 24: PgAnt

```
package pg.ant

import framework.Ant
import framework.Terrain
import pg.data.ParityGame
import kotlin.random.Random

/**
 * Creates a play by using pheromone levels on the edges.
 * If a loop is detected the play ends because the loop is repeated.
 */
class PgAnt(private val game: ParityGame) : Ant {
    override fun search(terrain: Terrain): Play {
        require(terrain is PgTerrain) { "Can only use a PgEnvironment" }

        val strategy = game.nodes.mapIndexed { i, _ ->
```

```
            val pheromones = terrain.pheromones[i]
            val totalAttractiveness = pheromones.fold(0.0) { acc, att -> acc + att }

            // Weighted choice between all candidates
            val rnd = Random.nextDouble(totalAttractiveness)
            val choice = pheromones.foldIndexed(Pair(0, rnd)) { j, acc, att ->
                if (acc.second <= 0.0) acc else Pair(j, acc.second - att)
            }.first
            Step(choice, i)
        }

        val lassoStart: Int
        val steps = mutableListOf(strategy[game.goalVertex])
        while (true) {
            val lastNode = game.nodes[steps.last().node]
            val lastChoice = steps.last().choice
            val nextStep = strategy[lastNode.successors[lastChoice]]
            if (!steps.any { visited -> nextStep.node == visited.node }) {
                steps.add(nextStep)
            } else { // We have found a lasso.
                lassoStart = steps.indexOfFirst { visited -> nextStep.node == visited.node }
                break
            }
        }

        return Play(steps, lassoStart)
    }
}
```

Listing 25: ExploringPgAnt

```
package pg.ant

import framework.Ant
import framework.Terrain
import pg.data.ParityGame
import pg.data.Node
import kotlin.random.Random

/**
 * Keeps on exploring if a loop would mean a loss for the current player.
 */
class ExploringPgAnt(private val game: ParityGame) : Ant {
    override fun search(terrain: Terrain): Play {
        require(terrain is PgTerrain) { "Can only use a PgEnvironment" }

        val lassoStart: Int
        val steps = mutableListOf<Step>()
        // The priority that would win if we jump back to this node.
        var previousPriorities = mutableListOf<Int>()
        val previousNodes = mutableListOf<Int>()
        while (true) {
            val currentNodeIndex = if (steps.isEmpty())
                game.goalVertex
            else
                game.nodes[steps.last().node].successors[steps.last().choice]
            val currentNode = game.nodes[currentNodeIndex]

            previousNodes.add(currentNodeIndex)
```

```
        // The priority for a node is
        // the priority that the loop would have if we would jump back to that position.
        previousPriorities.add(currentNode.priority)
        previousPriorities =
            previousPriorities.map { p ->
                if (p > currentNode.priority) currentNode.priority else p
            }.toMutableList()

        val loops = findLoops(currentNode, previousNodes, previousPriorities)

        // Check if we can win by jumping back.
        val winningChoices = loops.filter { choice ->
            choice.second.rem(2) == currentNode.parity
        }

        var choices = currentNode.successors.indices.toList()
        // If we have winning choices, choose them.
        if (winningChoices.isNotEmpty()) {
            choices = winningChoices.map { wc -> wc.first }
        } else {
            // Check with which edge's we lose.
            val losingChoices = loops.filter { choice ->
                choice.second.rem(2) != currentNode.parity
            }.map { c -> c.first }

            // Remove the losing choices from the choices
            if (losingChoices.size != choices.size) {
                choices = choices.subtract(losingChoices).toList()
                //println("after subtraction: $choices")
            }
        }

        val nextStep = createStep(currentNodeIndex, choices, terrain)
        if (!steps.any { visited -> nextStep.node == visited.node }) {
            steps.add(nextStep)
        } else { // We have found a lasso
            lassoStart = steps.indexOfFirst { visited -> nextStep.node == visited.node }
            break
        }
    }

    return Play(steps, lassoStart)
}

private fun createStep(node: Int, choices: List<Int>, terrain: PgTerrain): Step {
    val pheromones = terrain.pheromones[node]
    val totalAttractiveness = choices.fold(0.0) { acc, c -> acc + pheromones[c] }

    // Weighted choice between all candidates.
    val rnd = Random.nextDouble(totalAttractiveness)
    val choice = choices.fold(Pair(choices.first(), rnd)) { acc, c ->
        if (acc.second <= 0.0) acc else Pair(c, acc.second - pheromones[c])
    }.first

    return Step(choice, node)
}

// find all loops, return a list of <index, priority>
private fun findLoops(
```

```
        currentNode: Node,
        previousNodes: List<Int>,
        previousPriorities: List<Int>
    ): List<Pair<Int, Int>> {
        val loops = currentNode.successors.mapIndexedNotNull { i, trans ->
            val index = previousNodes.indexOf(trans)
            if (-1 == index) {
                null
            } else {
                Pair(i, previousPriorities[index])
            }
        }

        return loops
    }
}
```

Listing 26: PgEvaluator

```
package pg.ant

import arrow.core.Option
import arrow.core.Some
import framework.Evaluation
import framework.Evaluator
import framework.SolutionCandidate
import pg.data.ParityGame

/**
 * Checks which player wins a play.
 * Rewards the choices of the winning player.
 * Penalizes the choices of the losing player.
 */
class PgEvaluator(
    private val game: ParityGame,
    private val rewardAmount: Double,
    private val penaltyPercentage: Double
) : Evaluator {
    override fun evaluate(solutionCandidate: SolutionCandidate): Option<Evaluation> {
        assert(solutionCandidate is Play) { "Can only evaluate plays." }
        val play = solutionCandidate as Play

        // Check what the parity of the parity of the lowest priority in the lasso is.
        val lasso = play.steps.takeLast(play.steps.size - play.lassoStart)
        val winningParity =
            lasso.fold(Pair(false, Int.MAX_VALUE)) { acc, s ->
                val node = game.nodes[s.node]
                if (node.priority < acc.second) Pair(node.even, node.priority) else acc
            }.second.rem(2) == 0

        val pheromones = PgTerrain(game).pheromones.map { i ->
            i.toMutableList()
        }.toMutableList()
        play.steps.forEach { s ->
            val node = game.nodes[s.node]
            if (node.even == winningParity) {
                pheromones[s.node][s.choice] += rewardAmount
            } else {
                pheromones[s.node][s.choice] =
```

```
                    pheromones[s.node][s.choice] * penaltyPercentage
                }
            }

            return Some(Evaluation(1.0, PgTerrain(pheromones), false))
        }
    }
```

Appendix D. Implementation for strategy improvement

Listing 27: Strategy

```
package pg.strategy_improvement

/**
 * Contains a strategy from V_i to V_{1-i}.
 */
class Strategy(val strategy: Map<Int, Int>) {
    override fun equals(other: Any?): Boolean {
        if (other !is Strategy) {
            return false
        }

        return other.strategy == strategy
    }

    override fun hashCode(): Int {
        return strategy.hashCode()
    }

    fun forNode(i: Int): Boolean {
        return strategy.containsKey(i)
    }

    fun getTransition(i: Int): Int {
        return strategy[i]!!
    }

    override fun toString(): String {
        return strategy.map { e -> "${e.key}=${e.value}"}.joinToString("␣")
    }
}
```

Listing 28: SubGame

```
package pg.strategy_improvement

import pg.data.Node
import pg.data.ParityGame

/**
 * Stores a subgame.
 * A subgame is a game with fewer transitions or fewer vertices than the original game.
 */
class SubGame(
    private val game: ParityGame,
    _successors: Map<Int, List<Int>>,
    val nodes: Set<Int>
) {
```

```
private val predecessors: Map<Int, List<Int>> by lazy {
    val _predecessors = nodes.map { n ->
        n to mutableListOf<Int>()
    }.toMap().toMutableMap()
    successors.forEach { s ->
        s.value.forEach { x ->
            _predecessors[x]?.add(s.key)
        }
    }

    _predecessors.mapValues { v -> v.value.toList() }
}

private val successors: Map<Int, List<Int>> by lazy {
    _successors.filterKeys { k -> nodes.contains(k) }.map { k ->
        k.key to k.value.filter { s -> nodes.contains(s) }
    }.toMap()
}

fun getNode(id: Int): Node {
    if (nodes.contains(id)) return game.nodes[id]
    throw Exception("The node $id is not in this subgame.")
}

fun getPredecessors(id: Int): List<Int> {
    return predecessors[id]!!
}

fun getSuccessors(id: Int): List<Int> {
    return successors[id]!!
}

infix fun subtract(other: Set<Node>): SubGame {
    return SubGame(game, successors.toMap(), nodes.subtract(other.map { n -> n.id }))
}

infix fun partition(part: Set<Node>): SubGame {
    return SubGame(game, successors.toMap(), part.map { n -> n.id }.toSet())
}

infix fun subtract(edges: Map<Int, List<Int>>): SubGame {
    val newSuccessors = successors.map { entry ->
        entry.key to if (edges.contains(entry.key))
            entry.value.subtract(edges[entry.key]!!).toList()
        else
            entry.value
    }.toMap()

    return SubGame(game, newSuccessors, nodes.toSet())
}

companion object {
    fun fromStrategy(game: ParityGame, strategy: Strategy): SubGame {
        val _successors = game.nodes.map { n ->
            n.id to if (strategy.forNode(n.id))
                listOf(strategy.getTransition(n.id))
            else
                n.successors
        }.toMap()
```

```
            return SubGame(game, _successors, game.nodes.indices.toSet())
        }

        fun fromParityGame(game: ParityGame): SubGame {
            val successors = game.nodes.map { n -> n.id to n.successors }.toMap()

            return SubGame(game, successors, game.nodes.indices.toSet())
        }
    }
}
```

Listing 29: Valuation

```
package pg.strategy_improvement

import pg.data.Node
import util.isEven
import util.isOdd

class Valuation {
    private var valuations = mutableMapOf<Int, PlayProfile>()

    fun add(other: Valuation) {
        other.valuations.forEach { entry ->
            valuations[entry.key] = entry.value
        }
    }

    fun set(i: Int, playProfile: PlayProfile) {
        valuations[i] = playProfile
    }

    fun contains(node: Node): Boolean {
        return valuations.contains(node.id)
    }

    fun of(node: Node): PlayProfile {
        return of(node.id)
    }

    fun of(id: Int): PlayProfile {
        val playProfile = valuations[id]
        require(playProfile != null) { "A␣valuation␣is␣missing␣for␣node␣$id." }

        return playProfile
    }

    // Takes self loops into account correctly, use this in switching policies.
    fun ofEdge(from: Node, to: Int): PlayProfile {
        // The most important node in a self-loop is the only node there is.
        return if (from.id == to) {
            PlayProfile(from)
        } else {
            of(to)
        }
    }

    override fun toString(): String {
```

```
        return valuations.toSortedMap().map { e ->
            "phi(${e.key})=${e.value}"
        }.joinToString("\n")
    }
}

data class PlayProfile(
    val mostImportantNode: Node,
    val moreImportantNodes: MutableSet<Node> = mutableSetOf(),
    var initLength: Int = 0
) {
    fun addImportantNode(node: Node) {
        moreImportantNodes.add(node)
    }

    /**
     * -1 -> first object is less than the second.
     *  0 -> same
     *  1 -> first object is more than the second.
     */
    operator fun compareTo(other: PlayProfile): Int {
        val priorityDiff = RewardOrder.compare(mostImportantNode, other.mostImportantNode)
        if (0 != priorityDiff) {
            return priorityDiff
        }

        val moreImportantDiff = rewardCompareTo(moreImportantNodes, other.moreImportantNodes)
        if (0 != moreImportantDiff) {
            return moreImportantDiff
        }

        return if (mostImportantNode.priority.isOdd()) {
            initLength.compareTo(other.initLength)
        } else {
            other.initLength.compareTo(initLength)
        }
    }

    override fun toString(): String {
        return "PlayProfile(" +
                mostImportantNode.toShortString() + "," +
                moreImportantNodes.joinToString("␣") { n ->
                    "${n.id}(${n.priority})"
                } + "," +
                initLength + ")"
    }
}

fun rewardCompareTo(a: Set<Node>, b: Set<Node>): Int {
    val aMinB = a.minus(b)
    val bMinA = b.minus(a)

    if (aMinB.isEmpty() && bMinA.isEmpty()) {
        return 0
    }

    val highest = aMinB.plus(bMinA).maxWith(RelevanceOrder)!!
    return if (aMinB.contains(highest)) { // The highest element is only in a.
        if (highest.priority.isOdd()) -1 else 1 // If it is odd a is smaller.
```

```
    } else { // The highest element is only in b.
        if (highest.priority.isEven()) -1 else 1 // If it is even a is smaller.
    }
}
```

Listing 30: Default Switching Policy

```
package pg.strategy_improvement.switching_policy

import pg.data.ParityGame
import pg.strategy_improvement.PlayProfileRewardOrder
import pg.strategy_improvement.Strategy
import pg.strategy_improvement.SwitchingPolicy
import pg.strategy_improvement.Valuation

/**
 * Always choose the node with the highest valuation.
 */
class DefaultSwitchingPolicy(game: ParityGame) : SwitchingPolicy(game) {
    override fun generateInitialStrategy(): Strategy {
        return Strategy(game.nodes.filter { n -> n.even }.map { n ->
            n.id to n.successors.random()
        }.toMap())
    }

    override fun improve(strategy: Strategy, valuation: Valuation): Strategy {
        //println("# START switching policy")
        //println("Current strategy: $strategy")
        //println("Valuation:\n$valuation")
        var changeCount = 0;
        val improved = strategy.strategy.map { e ->
            val node = game.getNode(e.key)
            val playProfiles = node.successors.map { s ->
                s to valuation.ofEdge(node, s)
            }

            val best = playProfiles.maxWith(PlayProfileRewardOrder)
            // Only change if the best valuation is strictly higher.
            // This ensures that we find a fixed-point.
            if (valuation.of(e.value) < best!!.second) {
                changeCount++
                node.id to best.first
            } else {
                node.id to e.value
            }
        }.toMap()
        println("Default switchingpolicy made $changeCount changes.")
        //println("# END switching policy")

        return Strategy(improved)
    }
}
```

Listing 31: Ant Switching Policy

```
package pg.strategy_improvement.switching_policy

import framework.AntColonyOptimizer
import pg.ant.*
```

```
import pg.data.ParityGame
import pg.strategy_improvement.*
import kotlin.math.ceil

/**
 * Uses ACO to guide changing the switches.
 */
class AntSwitchingPolicy(
    game: ParityGame,
    private val config: AntSwitchingPolicyConfig
) : SwitchingPolicy(game) {
    val ant = ExploringPgAnt(game)

    val evaluator = ValuationPgEvaluator(game)

    private val optimizer = AntColonyOptimizer(
        config.numberOfAnts,
        config.generationsPerIteration,
        ant,
        evaluator,
        config.rho,
        1.0
    )

    var terrain = PgTerrain.createRandomFor(game)
    private var generationsLeft = config.generationsPerIteration

    private fun startOptimizer(forGeneration: Int) {
        optimizer.execute(terrain, forGeneration)
    }

    override fun generateInitialStrategy(): Strategy {
        startOptimizer(config.generationsForInit)
        // Store the environment for the next iteration.
        terrain = optimizer.terrain as PgTerrain

        // Choose the edge with the most pheromone.
        val choices = game.nodes.filter { n -> n.even }.map { n ->
            val pheromones = terrain.pheromones[n.id]
            val choice =
                pheromones.foldIndexed(Pair(0, 0.0)) { j, acc, att ->
                    if (acc.second > att) acc else Pair(j, att)
                }.first
            n.id to n.successors[choice]
        }

        return Strategy(choices.toMap())
    }

    val defaultSwitchingPolicy = DefaultSwitchingPolicy(game)

    override fun improve(strategy: Strategy, valuation: Valuation): Strategy {
        //println("# START switching policy")
        //println("Current strategy: $strategy")
        //println("Valuation:\n$valuation")
        if (0 == generationsLeft) {
            return defaultSwitchingPolicy.improve(strategy, valuation)
        }
```

```
        // Store the environment for the next iteration.
        val generations = ceil(generationsLeft / 2.0).toInt()
        startOptimizer(generations)
        generationsLeft -= generations
        println("Running for $generations")
        terrain = optimizer.terrain as PgTerrain

        var changeCount = 0
        // Use the pheromone levels on the environment to make choices.
        val improved = strategy.strategy.map stratMap@{ e ->
            val node = game.getNode(e.key)
            val playProfiles: Map<Int, PlayProfile> = node.successors.map { s ->
                s to valuation.ofEdge(node, s)
            }.toMap()

            // Only allow changes to strictly higher play profiles.
            val choices = playProfiles.filter { pp ->
                PlayProfileRewardOrder.compare(
                    pp.toPair(),
                    Pair(1, playProfiles[e.value]!!)
                ) == 1
            }.keys
            if (choices.isEmpty()) {
                return@stratMap e.key to e.value
            }
            changeCount++

            val choiceIndices = choices.map { c -> node.successors.indexOf(c) to c }
            val pheromones = terrain.pheromones[node.id]
            val pheromoneChoices = choiceIndices.map { c ->
                c.second to pheromones[c.first]
            }

            val choice = pheromoneChoices.fold(Pair(-1, -1.0)) { acc, c ->
                if (acc.second > c.second) acc else c
            }.first

            //println("Changed: ${e.key} from ${e.value} to ${choice}.")
            e.key to choice
        }.toMap()
        println("Ant switchingpolicy made $changeCount changes.")
        //println("#end switchingpolicy")

        return Strategy(improved)
    }
}
```

Listing 32: Strategy Improvement

```
package pg.strategy_improvement

import pg.data.Node
import pg.data.ParityGame
import util.isEven

/**
 * An implementation of the strategy improvement algorithm.
 * Based on
 * ‘A Discrete Strategy Improvement Algorithm for Solving Parity Games‘
```

```
 * by Jens Voge and Marcin Jurdzinski.
 */
class StrategyImprovement(private val switchingPolicy: SwitchingPolicy) {
    var lastNumberOfIterations = 0

    /**
     * Run strategy improvement, where the strategy of player 0 is optimized.
     */
    fun run(pg: ParityGame): Pair<Boolean, Strategy> {
        val initialStrategy = switchingPolicy.generateInitialStrategy()

        var lastStrategy: Strategy
        var currentStrategy = initialStrategy
        var valuation: Valuation

        val startTime = System.currentTimeMillis()
        var iterations = 0
        do {
            // Create a sub-game based on the current strategy.
            val subGame = SubGame.fromStrategy(pg, currentStrategy)
            //println("Current strategy: $currentStrategy")
            valuation = valuation(subGame)
            lastStrategy = currentStrategy
            // Optimize the strategy according to the valuation.
            currentStrategy = switchingPolicy.improve(currentStrategy, valuation)
            iterations++
        } while (currentStrategy != lastStrategy)

        val duration = System.currentTimeMillis() - startTime
        println("Solved␣in␣$iterations␣iterations␣and␣$duration␣ms.")
        lastNumberOfIterations = iterations

        val evenWins = valuation.of(pg.goalVertex).mostImportantNode.priority.isEven()
        return Pair(evenWins, lastStrategy)
    }

    private fun valuation(_game: SubGame): Valuation {
        var game = _game
        val valuation = Valuation()

        val nodes = game.nodes.map { i -> game.getNode(i) }
        nodes.sortedWith(RewardOrder).forEach { w ->
            if (!valuation.contains(w)) {
                // L contains nodes that can reach w, where w has the highest priority.
                val L = lessRelevantReach(game, w)
                val containsLoop = game.getSuccessors(w.id).fold(false) { acc, s ->
                    acc || L.any { n -> n.id == s }
                }
                if (containsLoop) {
                    val R = reach(game, w)
                    valuation.add(subValuation(game partition R, w))
                    game = game subtract vertices(R, R)
                }
            }
        }

        return valuation
    }
```

```
    private fun subValuation(_game: SubGame, w: Node): Valuation {
        var game = _game
        val valuation = Valuation()
        game.nodes.forEach { n -> valuation.set(n, PlayProfile(w)) }

        // Iterate more relevant nodes than the winner, starting with the most relevant one.
        val moreRelevantNodes = game.nodes.filter { i ->
            game.getNode(i).priority < w.priority
        }.map { i -> game.getNode(i) }
        moreRelevantNodes.sortedWith(RelevanceOrder).reversed().forEach { u ->
            if (u.priority.isEven()) { // We want to avoid nodes with an even priority.
                val U = reach(game subtract setOf(u), w)
                (game.nodes.map { n -> game.getNode(n) } subtract U).forEach { v ->
                    valuation.of(v).addImportantNode(u)
                }
                // Remove edges that go through u that are not necessary to reach w.
                game = game subtract vertices(U.plus(u), U)
            } else { // We want to visit nodes with an odd priority
                val U = reach(game subtract setOf(w), u)
                U.forEach { v -> valuation.of(v).addImportantNode(u) }
                // Remove edges that avoid u.
                game = game subtract vertices(U.minus(u), U.plus(w))
            }
        }

        if (w.priority.isEven()) {
            maximalDistances(game, w).forEach {
                require(it.value != -1) {
                    throw Exception("Too many edges have been removed!")
                }
                valuation.of(game.getNode(it.key)).initLength = it.value
            }
        } else {
            minimalDistances(game, w).forEach {
                require(it.value != Int.MAX_VALUE) {
                    throw Exception("Too many edges have been removed!")
                }
                valuation.of(game.getNode(it.key)).initLength = it.value
            }
        }

        return valuation
    }
}
```