NTNU
Norwegian University of
Science and Technology

Dynamic Scheduling for Autonomous Robotics

Kai Olav Ellefsen

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The goal of this thesis is to implement a dynamic scheduling system for an autonomous robot that performs a foraging task in a complex, unstable environment. The work will improve the scheduling system implemented as the author's specialization project with the ability to reschedule intelligently when an opposing robot makes changes to the playing environment. The implemented system will be used in the 2010 Eurobot competition, and as there will be a need to reschedule several times during a match, the rescheduling algorithm will have to be very time efficient.

Assignment given: 22. January 2010
Supervisor: Keith Downing, IDI

Abstract

This project report describes a hybrid genetic algorithm that works as a schedule generator for a complex robotic harvesting task. The task is set to a dynamic environment with a robotic opponent, making responsiveness of the planning algorithm particularly important.

To solve this task, many previous scheduling algorithms were studied. Genetic algorithms have successfully been used in many dynamic scheduling tasks, due to their ability to incrementally adapt and optimize solutions when changes are made to the environment. Many of the previous approaches also used a separate heuristic to quickly adapt solutions to the new environment, making the algorithm more responsive. In addition, the study of previous work revealed the importance of population diversity when making a responsive genetic algorithm.

Implementation was based on a genetic algorithm made as the author's fifth year specialization project for solving a static version of the same task. This algorithm was hybridized with a powerful local search technique that proved essential in generating good solutions for the complex harvesting task. When extending the algorithm to also work in a dynamically changing environment, several adaptations and extensions needed to be made, to make it more responsive. The extensions and adaptations included a fast-response heuristic for immediate adaptation to environmental changes, a decrease in genotype size to speed up local searches and a contingency planning module intending to solve problems before they arise.

Experiments proved that the implemented dynamic planner successfully adapted its plans to a changing environment, clearly showing improvements compared to running a static plan. Further experiments also proved that the dynamic planner was able to deal with erroneous time estimates in its simulator module in a good way.

Experiments on contingency planning gave no clear results, but indicated that using computational resources for planning ahead may be a good choice, if the contingencies to plan for are carefully selected. As no unequivocal results were obtained, further studies of combining genetic algorithms and contingency planning may be an interesting task for future efforts.

Preface

This report was written as the author's master's thesis at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU).

The project is part of NTNU's 2010 contribution to the annual Eurobot competition. The Eurobot competition gathers teams from all over the world to engage in a robotics challenge. The author became part of the Eurobot-team in the spring of 2009, when building NTNU's 2009 Eurobot-contribution as part of the multidisciplinary subject "Experts in Teamwork" (EiT). Three of the students from the EiT-group, including this author, wanted to continue their participation in the Eurobot-team, enabling transfer of experience to next year's participants. The Eurobot-team is a multidisciplinary team, currently made up of five fifth year master students, ten fourth year EiT-students and a mechanic apprentice.

The author would like to thank supervisor Keith L. Downing for guidance during the work with this project and for, together with associate professor Sverre Hendseth at the Department of Engineering Cybernetics, enabling this project to be a cooperative effort across departmental boundaries at NTNU.

Thanks also go out to all the members of the Eurobot-team for support and good cooperation, to the team's sponsor, KONGSBERG, for financial support, and to last year's team leaders, Christian W. Kjølseth and Øystein Wergeland for sharing their experience and giving this year's team a head start.

Trondheim, June 17, 2010
Kai Olav Ellefsen

Contents

List of Figures

List of Tables

List of Abbreviations

CVRP Capacitated Vehicle Routing Problem

DTSP Dynamic Traveling Salesman Problem

DVRP Dynamic Vehicle Routing Problem

EO Extremal Optimization

ER Edge Recombination Crossover

GA Genetic Algorithm

GOP Generalized Orienteering Problem

ILP Integer Logic Programming

OP Orienteering Problem

OX Order Crossover

PMX Partially-Mapped Crossover

TDVRP Time Dependent Vehicle Routing Problem

TSP Traveling Salesman Problem

VRP Vehicle Routing Problem

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 The Eurobot competition

This project is part of NTNU's contribution to the Eurobot competition in May 2010. The competition is an annual event gathering teams from all over the world to compete in a robotics challenge. The competition has existed since 1998, and it aims to "favour the public interest in robotics and encourage hands-on practice of science by young people"[21]. In 2010, the event takes place in the city of Rapperswil-Jona in Switzerland from May 26th to May 30th.

1.1.2 The team

NTNU has participated in the Eurobot competition since 2000. For some years, the contribution from NTNU has consisted of many fourth-grade students building and programming parts of the robot through the subject *Experts in Teamwork* (EiT), while fifth-grade students studying Engineering Cybernetics have implemented the main strategy of the robot, as well as building some of the more advanced parts of the robot as part of their fifth grade project and master's thesis. Normally, some of the students from EiT have continued to work on the project in the fifth grade, enabling transfer of

experience to next year's team. Also, the teams have tried to reuse parts of previous robots that have proven to work successfully.

This year, we are a team of five students working on the robot as part of our master's thesis, and additional ten students from EiT are helping us build the robot's system for gathering and delivering objects. Three of us (Bård, Kai Hugo and I) also worked on the project as part of EiT last year. For the reader to understand where the work presented in this report fits into the work on the robot, a brief presentation of what the other team members will focus on in *their* master's theses is given below.

Bård Jonas Wigestrand studies Product Development, and will work on upgrading the robot's current translation system, in order to increase the robot's speed, give more accurate positioning data and enable the robot to drive up a slope. He will also work as the leader of the team.

Kristin Holst Haaland studies Engineering Cybernetics, and will work on implementing a motor regulator for the robot's translation system.

Ole Lillevik studies Engineering Cybernetics, and will work on implementing a laser module that enables the robot to more accurately determine its own and the opposing robot's position.

Kai Hugo Hustoft Endresen studies Computer Science, and will work on implementing a stereoscopic vision module. This will allow the robot to more accurately determine the distance to objects on the playing table, and notice when they are removed by the opponent.

1.1.3 This year's task

The task this year is titled "Feed the world", and is thoroughly described in the official Eurobot rules [21]. The setup of the playing table is shown in Figure 1.1.

The tournament goes through several qualification rounds for all teams, and the sixteen teams that have gathered the most points during qualification move on to the final rounds, where the losing robot in each match is out of the tournament.

In each match, two robots compete for gathering the most weight within 90 seconds. The objects to gather are the red, white and orange objects in

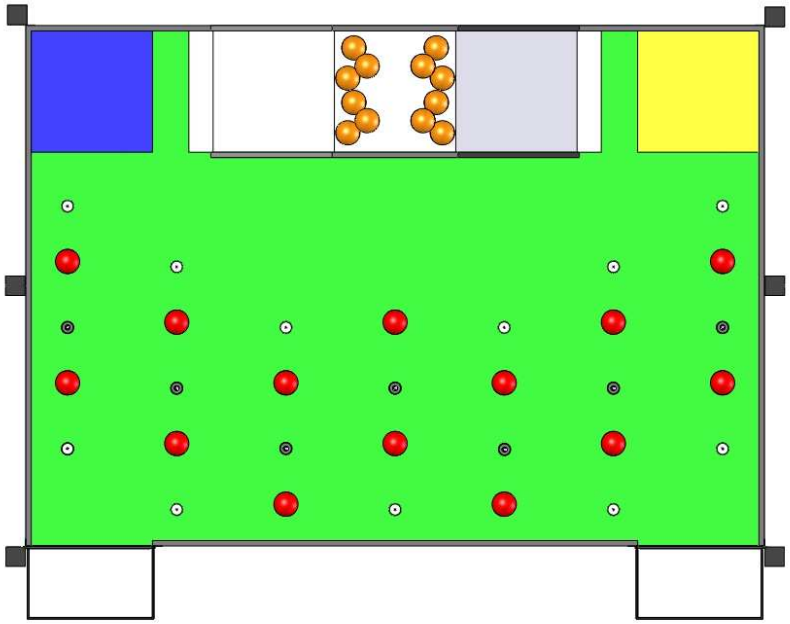


Figure 1.1: A possible configuration of the playing table

Figure 1.1. The weight varies between the different object types, and so does the difficulty associated with retrieving the objects. Gathered elements have to be placed in the robot's goal container to score points, the score being equal to the weight of the object gathered. The robot starts at either the blue or yellow field, and its goal container is the container placed at the diagonally opposite corner of the table.

The orange objects are juggling balls representing oranges, weighing 300 g. These have diameters of 10 cm and are placed on cylinders of different heights called "trees", that are placed on top of a hill. Obviously, these are difficult to obtain, as the robot both has to drive up the hill, and collect objects from trees to get them. The advantage, however, is that they are all located close to each other, and that they are the heaviest objects on the table.

The white objects are cylinders representing ears of corn, that are stuck in the ground. To obtain them, the robot has to pull them upwards or push them to make them fall. The ears of corn weigh 250 g, are 15 cm high and have a diameter of 5 cm. These are more accessible than the oranges, as they are simply spread around the flat area of the playing table, but the fact that they are stuck in the ground could make picking them up time consuming.

The red objects are juggling balls representing tomatoes. These balls weigh 150 g, have a diameter of 10 cm, and are placed on the ground. The tomatoes are probably the easiest elements to pick up, but they are also the lightest objects, meaning they give the fewest points.

The black objects are fake ears of corn. These cannot be picked – instead the robot needs to avoid them, to prevent a collision. Figure 1.1 actually only displays a *sample* configuration of the fake ears of corn. There are 36 different configurations of these elements, and one of these configurations is randomly chosen as the game begins. Each configuration has seven fake elements, and ears of corn are always in the same place, but real and fake ears may swap places, compared with the setup in Figure 1.1.

Note that the playing table is always entirely symmetrical, so a solution for a given starting position, can easily be translated to a solution for the *other* starting position.

The number of points scored by a robot during a match, is determined in the following way:

- Each robot gets one point for each gram collected and placed in the scoring container.
- The winning robot gets 200 additional points.
- The losing robot gets 50 additional points, unless it is disqualified.
- In case of a draw, each robot gets 100 additional points.
- Robots can lose points due to penalties given for reasons like colliding with the other robot or intentionally blocking its path.

It is obvious that finding the path giving the robot as many points as possible will depend greatly on the robot's design. For instance, if the robot is designed to pick oranges super-fast, picking many oranges would be a good strategy. However, if it is designed to be an excellent tomato-picker, then picking only tomatoes may yield a better result. If the robot is quite good at picking all three types of elements, the final solution will depend on factors like the robot's speed, the time taken to pick each type of element and the number of elements the robot can carry at once.

Unfortunately, the robot's design will not be completely determined until a short time before the competition, as this is when the EiT-groups working

on the robot finish their project. Therefore, the system generating strategies has to be parameterized. It should be able to take parameters such as the robot's translation speed, capacity and pick-up speed, and find a good route based on these input values. Shortly before the competition, the team can measure these parameters, enter them into the system, and good strategies based on the robot's actual functioning can be generated.

As soon as the opponent starts interacting with the playing table, changes may have to be made to the robot's plan. For instance, it may have to avoid an area to prevent a collision, or one of the objects it had planned to pick up may end up being picked by the opponent. This calls for dynamic replanning, taking place *during* the match. Last autumn, as the author's specialization project, a *static* planning system for generating plans *before* the match was made. The focus of this master's thesis will be to extend this system to enable it to modify these plans in light of changing circumstances.

1.2 Problem Definition

1.2.1 Team vision and mission

This project is part of a team effort, and as a team, we have chosen the following *vision* for our work:

> *To learn interdisciplinary technology development through a challenging and motivating project, and to spread interest in technology – particularly among young people.*

A more precise description of what our work will be, is presented in our *mission statement*:

> *We will, through our multi-disciplinary skills develop and improve NTNU's Eurobot-contribution, so that we can achieve a good position in the competition in 2010. We will develop well-functioning modules that are to be assembled into a robust, efficient and precise robot.*

In light of this statement, the project presented in this report mainly consists in making one of the robot modules, namely the strategy module. The goal of the project is stated more precisely in the next section. In addition to developing the robot, the whole team has participated in displaying and demonstrating the robot being constructed at various events, fulfilling the second part of our vision to "spread interest in technology". For more information about this, see our homepage at `http://www.eurobot-ntnu.no/`.

1.2.2 Project goal

> *The goal of this project is to implement an algorithm that is able to use the previously made strategies for the Eurobot-competition and adapt them to the actual state of the match, in particular to avoid a collision with the opponent and to generate new, efficient plans as the opponent makes changes to the playing table. Previous path optimization and scheduling problems solved by genetic algorithms and other techniques will be thoroughly researched, in order to determine how to design and implement the algorithm.*

The algorithm should also fulfill the following requirements:

- The algorithm should be fast enough to be able to efficiently handle rapid changes to the robot's playing environment.
- As the robot is not yet built when the planning system is being implemented, the algorithm has to base its calculations on parameters that can easily be changed when the robot is ready. These parameters should at least include the robot's translation speed, capacity for carrying objects, the time taken to pick up the various types of objects, the time taken to deliver objects and the time it takes to navigate around fake ears of corn and drive up and down the hill.
- The algorithm should be able to show a simple match simulation, to enable testing before the actual robot is finished. This simulation should be in real-time, showing the robot, the opponent, the playing elements and the robot's plan as it adapts to the changing environment.

1.3 The Problem Area

The problem presented in the previous section, is clearly within the domain of optimization. For given robot parameters and a state of the playing area, there is one or more paths that will yield the optimal amount of weight gathered. As the amount of points the robot might visit around the table is so big, enumerating all possible paths is unfeasible. The number of pickable playing elements at the beginning of a match is 37, and the number of possible ways to visit all these elements is 37!, which is higher than 10^{43}.

Luckily, the problem does not require an optimal solution. It requires a *good* one, but not necessarily the optimal one. The probability that the opponent interferes with the robot's plan is quite big, so most likely, changes will have to be made to the generated plan throughout the match, anyway. In addition, a good (but not optimal) plan will still score many points, so a balance needs to be found between the time taken to generate a solution and the quality of the generated solution.

Whereas the task of finding good solutions before the match lies in the domain of *static* optimization, recalculating and adapting the plan to changing circumstances lies in the domain of *dynamic* optimization. The latter is harder, due to the strict time constraints faced in a changing environment.

A study of static and dynamic optimization techniques used to solve problems similar to this, both optimally and heuristically, is presented in Chapter 2.

1.4 The Robot Design

The building of the robot was completed shortly before the competition, after quite a few redesigns and modifications had to be made late in the building process.

At the top of the robot, a rotating laser module communicating wirelessly with three receptor beacons placed around the playing area, and one on top of the opposing robot, ensures that the robot can always determine its own and its opponent's position accurately. Below this module, the computer running the robot's AI, computer vision and navigation system is located. Attached to the computer are two cameras, using stereoscopic vision to determine the positions of playing elements in front of the robot.

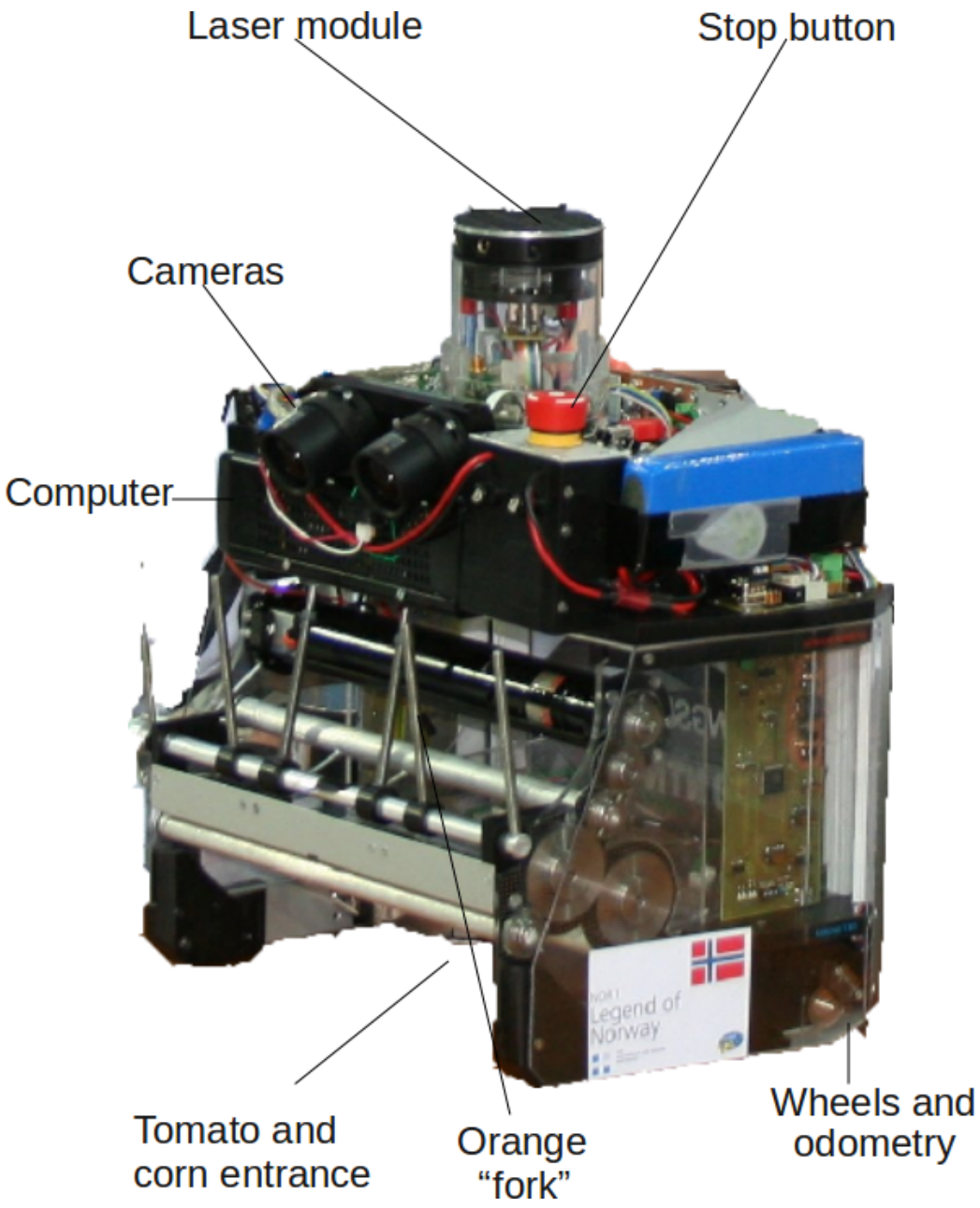


Figure 1.2: The robot

Below the computer, the picking- and delivery system of the robot is found. It begins with a "fork" that allows the robot to pick, deliver and carry three oranges gathered from either side of the hill. Below these, are a slot where both tomatoes and corns enter into the robot. Tomatoes are squeezed against the floor, and as they enter, against wheels on the inside of the robot, lifting them from the ground and into a storage area. Corns are tipped as the robot drives over them, and enter into the robot, but keep rolling against the table until they are delivered.

Delivering objects is done by a servo lowering the orange fork, and six servos pushing the corns out of the robot's center. In addition, a motor reverses the rotational direction of the tomato-picking wheels, pushing the tomatoes out from the robot's storage area.

The drive wheels of the robot are placed close to its rear end. This is a change from previous years, when these wheels were located around the middle of the robot, and it was made to enable the robot to enter and exit the slope without bumping into the table. This has the unfortunate side effect of making the robot's rotational radius larger, making it more difficult to navigate on the playing table. Outside the drive wheels, are odometry wheels that count the distance traveled to update the robot's position and orientation estimates.

Chapter 2

Background

As mentioned, the problem at hand consists of finding the most efficient way of gathering different objects and delivering them to a goal area. This task has some similarities with the traveling salesman problem (TSP), so this was where the search for good solution techniques began. As the background studies proceeded, the orienteering problem (OP) and the vehicle routing problem (VRP) were found to be even more similar to the Eurobot task. Therefore the background studies were focused on approaches to solving these problems, in particular by means of genetic algorithms.

However, the problem of finding good paths becomes more complicated when the opposing robot makes changes to the playing table. Therefore, approaches to dealing with changing environments were studied, in particular approaches that could utilize the solutions already generated for the initial playing environment.

2.1 Genetic Algorithms

This section gives a brief introduction to genetic algorithms and why they have been chosen as the solution method in this project. For a more thorough introduction to GAs, the reader is referred to [9].

2.1.1 What are they?

Genetic algorithms are a way of traversing a solution landscape in a more intelligent manner than random search. When a genetic algorithm searches for a solution, it uses previous solutions it has generated, and puts more effort into modifying *the best* of them, to hopefully make them even better. In addition, the process contains enough randomness at its early stages to get a good overview of the *entire* solution landscape.

Genetic algorithms are a subclass of *evolutionary* algorithms. They model the process of natural evolution by maintaining a set of possible solutions, represented as *genotypes* in *individuals*, and letting the best individuals reproduce to form a new *generation* of solutions. To know what individuals are the best, all solutions have to be tested on the problem, and assigned a *fitness* value describing how well they did. Individuals with a high fitness are given a better chance of growing up, and reproducing to form new solutions.

The way adults are selected from the child population, and the way individuals are selected for reproduction, may vary. Sometimes individuals are simply selected with a probability proportionate to their fitness, other times more complicated scaling methods are utilized before selecting, typically to make the selection process more random and exploratory in the early generations, and to make for a more focused search towards the end of the algorithm's run.

Reproduction typically involves *crossover* and *mutation*. Crossover is the process of taking two individuals and mixing their genotypes (their solution representations) in some way, for instance by splitting them in the middle and taking half of each to be their child's genotype. Mutation is some way of changing the new child's genotype, and it can range from simple random swaps, to more complicated local searches optimizing the new solution.

Before the evolution can begin, an initial population needs to be formed. This can be done in several ways, from simply making random individuals to using some heuristic for making good initial guesses. Of course, heuristics may constrain the GA and lead it towards local optima, but they will also enable the algorithm to find good solutions faster, and may be necessary in complex tasks.

In addition to implementing the concepts described above, a lot of time goes into tuning parameters when solving a problem with a GA. Some of the most

important parameters to tune are the number of generations, the population size, the mutation probability and the selection mechanisms.

2.1.2 Why were they chosen as the solution technique?

The robot's task is easily represented as finding the optimal permutation of integers, where integers represent the various playing objects and the delivery area. One way to find this permutation would be to test all possibilities. But this would be enormously time consuming, as discussed in Section 1.3. Indeed, some more intelligent approach is called for.

A GA will make it possible to test a lot of possible solutions, while not wasting too much time on the bad ones. The algorithm gives no guarantee of finding the *best* solution, but it probably finds a relatively good one, if implemented correctly. Previous literature shows many examples of similar problems solved successfully by GAs, and some of these implementations are summarized in Section 2.4.

In addition, genetic algorithms are very good at adapting to changing environments. This will be important as the opposing robot may change the playing environment during the competition. Dealing with changing environments is discussed further in Section 2.2.

2.1.3 What are the alternatives?

This problem is from the domain of optimization, and a lot of similar problems have been documented and solved in this domain. The TSP is probably the first problem class that comes to mind, but even more similar problems have been extensively researched. These problems and common ways of solving them are described in Section 2.3.

2.2 Changing Environments

The fact that the opposing robot is able to make changes to the state of the playing table, complicates this problem. Therefore, an effort was made to determine what would be the best way of dealing with this kind of change.

Techniques for handling environments that vary over time are presented in this section. But first, a few words on the terminology used for describing environments.

2.2.1 Terminology

[25] classifies environments along six dimensions. They are briefly presented here, along with the Eurobot-problem's classification.

Fully observable vs. partially observable

As the robot at no time has a full view of the playing area, this environment is clearly partially observable.

Deterministic vs. stochastic

Because of both the other agent's actions, and inaccuracies in the robot's own actuators and sensors, the next state of the playing environment will never be certain. Therefore, this environment is classified as a stochastic.

Episodic vs. sequential

An episodic environment is an environment where the current action made by the agent will not affect future decisions. This is clearly not the case here: If the robot for instance decides to pick up an object, it will have to deliver it in the future to score points. Hence, the environment is sequential.

Static vs. dynamic

A dynamic environment is one that can change while the agent is deliberating. This environment is dynamic, both because of the other agent's actions, and because the match has a 90 second time limit, meaning that the agent's maximum score will get lower as it deliberates its next action.

Discrete vs. continuous

This environment is continuous, as both time and the agent's position span a range of continuous values.

Single agent vs. multi agent

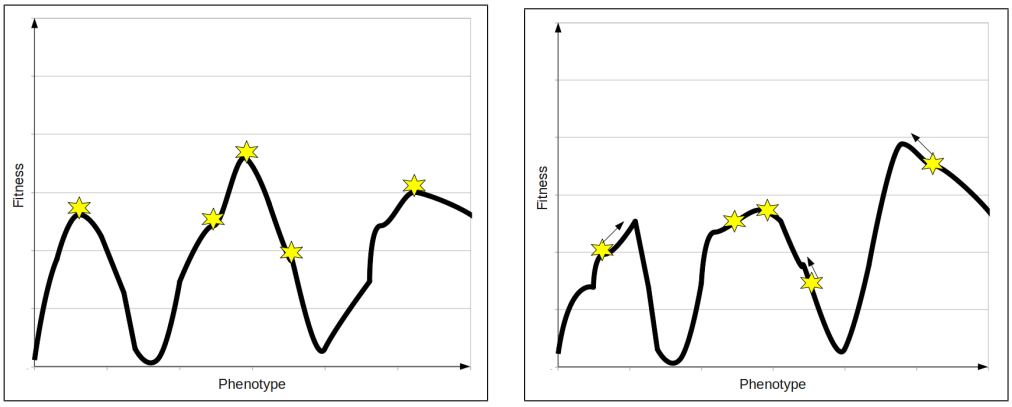
The environment is obviously multi agent, and more specifically, a *competitive* multi agent environment.

The two most interesting dimensions in the current task, and the two dimensions that will be the most challenging to handle, are the **deterministic vs. stochastic** and **static vs. dynamic** dimensions. Because the environment is both stochastic and dynamic, there will most likely be a need for the robot to adjust its plan *during* the match, and this adjustment must happen *fast*, to avoid losing points.

2.2.2 Dealing with a changing environment

According to [18] classical, mathematically founded planning methods, like dynamic and linear programming, do not adapt well to changing environments. The reason why, is that they only compute a single solution, and will have to start from scratch if any of the conditions the solution was made under change. On the other hand, planning methods like genetic algorithms, that *search* for solutions instead of calculating them, are better at dealing with change, because the optimal solution under the new environmental conditions is likely to be close to the previously calculated optimal solution. Therefore, starting a search from this previous solution will generate a new optimal solution much faster than if the algorithm is started from scratch, as long as the environmental changes are not too severe.

[18] mentions another reason why genetic algorithms are good at adapting to environmental changes: They maintain a whole *population* of solutions instead of just the optimal one. For this reason, large environmental changes that make the optimal solution infeasible, may still be handled by other solutions in the population. And even if none of these solutions can be used *directly* in the new environment, this diversity of solutions means that there



(a) (b)

Figure 2.1: GA solutions in a changing environment

will be many promising starting points when searching for a good solution after the environmental change.

Figure 2.1 shows how this works. The figures show the phenotypes of various solutions plotted according to their fitness values. Phenotypes are plotted along the x-axis, with similar phenotypes as neighbors. Fitness is plotted on the y-axis. The stars represent the population of individuals managed by the genetic algorithm. Figure 2.1a shows the situation after the algorithm has been run in a stable environment for a while. The solutions have found the peaks in the fitness landscape, and some technique has been employed to maintain solution diversity, to avoid all individuals converging towards the highest peak in the middle.

Now consider a sudden change in the environment that changes the fitness landscape into that of Figure 2.1b. The environmental change did not change the profile of the solution landscape completely (the three peaks are still there), but it was severe enough to move the optimal solution to the third peak, far away from its original position. Now, due to the fact that the genetic algorithm maintains several solutions, it will be able to find the new global and local peaks of the fitness landscape by using its *previous* population as starting points, as indicated by the arrows.

2.2.3 Maintaining diversity

A genetic algorithm typically converges to a single solution, or a group of very similar solutions when run for a long time in the same environment. This is good if we do not expect the environment to change, but if we *do* expect changes, diversity is essential, as illustrated by Figure 2.1. Without it, the new global optimum would be very hard to discover. [18] suggests the techniques of *niching* and *crowding* to deal with this challenge.

Niching means that all solutions that occupy the same region of the search space *share* a total fitness value. This means that as more solutions are generated within the same region, each individual gets a smaller fitness value. Thus, the poorest individuals are forced to another region, where there are fewer other individuals.

Crowding is a different method for ensuring diversity, in which old solutions that are similar to new ones are simply replaced by the new solutions. This prevents too many similar solutions from building up in the population.

[18] also mentions some drawbacks associated with the two techniques. Firstly, employing them can result in a quite big computational expense as all solutions have to be compared to one another, which may be a resource demanding operation. Secondly, the choice of *how* solutions should be compared will most likely not be straightforward. For instance, how should it be determined if two TSP-solutions are similar? Based on the visiting order of the cities, or perhaps on the number of identical edges in the solution?

[11] surveys the use of evolutionary optimization techniques in uncertain environments, including environments where the fitness function and, hence, the optimal solution changes over time. The survey classifies four groups of approaches to addressing the problem of solution diversity.

Generating diversity after a change: In this approach, the evolutionary algorithm is run as normal until a change is detected. As soon as this happens, some action is taken to increase diversity in the population. *Hypermutation* is one such action, which drastically increases the mutation rate in the first generations after the environmental change, resulting in a rapid increase in diversity.

Maintaining diversity throughout the run: This approach includes niching, crowding and other techniques that keep the population from converging at any time. The *random immigrants approach* inserts random

individuals into the population in each generation, making sure convergence towards a single solution is avoided.

Memory-based approaches: These approaches extend the process of evolution with a memory of previous solutions. Such techniques are useful when the environment *cycles* between different states, meaning the optima repeatedly return to previous locations in the fitness landscape.

Multipopulation approaches: These approaches divide the population into several smaller subpopulations, that each inhabit their own region of the search space. This allows the algorithm to track several promising regions of the search space, and ensures a diverse population. Examples of this approach are the "forking GA" [4] and the Shifting Balance GA [19].

2.3 Related Problems

To find previous work on the type of algorithm that was to be implemented, it was necessary to find *similar* problems that have been successfully solved by genetic algorithms. Below are descriptions of the three most relevant similar problems, and typical solution techniques. Although none of these match the task in this project exactly, ideas and solution techniques previously used for solving them may be applicable also in this project.

This section focuses on the *static* versions of the problems, which were useful for designing the pre-match planner of the robot, but which will also be relevant for the dynamic planner, because the dynamic planning problem can easily be modeled as a *series* of static planning problems over time. There may, however, be a more efficient way to approach the problem of dynamic planning, and some approaches to solving the dynamic TSP and the dynamic VRP by use of genetic algorithms are presented in sections 2.4.7 and 2.4.8.

2.3.1 The traveling salesman problem

Definition

In [3] the TSP is defined in its most general form as the problem of selecting an ordering of the numbers from 1 to n (this ordering is called a "tour") which

minimizes the *cost* of the tour. The cost is given by an n-dimensional square matrix that defines the penalty for having any two numbers be neighbors in the tour. Total cost is obtained by summing up all penalties for the tour.

Normally, the integers from 1 to n are considered to be cities, and the ordering is a way to visit all the cities. The cost matrix then represents the distance between all pairs of cities, and the task is to find the tour that minimizes the total distance traveled. The problem is NP-Complete, as proved in [20].

Common solutions

[14] gives an overview of common solution techniques for the TSP. The problem can either be solved exactly or approximately, depending on the size of the problem instance and the resources available.

Exact solution techniques often represent the TSP as an Integer Linear Programming (ILP) problem. In this representation, the function to be optimized is linear and subject to equality and inequality constraints that are also linear. In the TSP, the linear function to be optimized, is the sum of the costs of traversing the arcs in the solution. Minimizing this sum means finding the shortest path through all nodes. The additional linear constraints ensure that the tour considered is a *valid* TSP-tour.

For finding the exact solution, branch-and-bound algorithms are commonly used. The idea is to find better and better bounds for the cost of an optimal solution, and then focusing the search for solutions to those that lie within these limits. To be able to do so, the solutions are structured in a tree, and tree nodes that fall outside the bounds are not expanded further. An initial lower bound on the solution cost, can for instance be obtained by relaxing some of the constraints given in the ILP.

As the TSP is NP-hard, a lot of work has been done on finding *approximate* solutions for the problem. These fall into two categories: Those with a *guaranteed* worst-case performance, and those with a *good* empirical performance. A method for finding an approximate solution is calculating the minimum spanning tree of the problem, and traversing all the edges of this tree in a structured fashion. For a symmetrical TSP (TSP on an undirected graph), this solution is guaranteed to come within two times the optimal solution length. The method has a runtime of $\mathcal{O}(n^2)$.

There are many heuristics that have been shown to have good empirical

performance in solving the TSP. These include:

- The nearest-neighbor algorithm: At each step, simply add the nearest neighbor of the current node to the tour. Complexity $\mathcal{O}(n^2)$.
- Insertion algorithms, starting with a two-node tour, and iteratively inserting the best node, based on some criterion, like the one yielding the least increase in tour length, or the one closest to the current tour. Complexity varies between $\mathcal{O}(n^2)$ and $\mathcal{O}(n * logn)$ depending on the insertion criterion.
- The patching algorithm for asymmetrical TSPs: This algorithm first constructs a TSP allowing for subtours (loops not covering all nodes) to form, and then patches the subtours together in the way that minimizes the *cost of merging* (that is, the difference in cost between the unmerged and the merged tour).

In addition to the algorithms mentioned, that generate TSP solutions from scratch, there are also many algorithms for *improving* TSP solutions. The most important ones are described below.

The *r-opt* method considers an initial tour and removes r of the edges before reconnecting them in *all possible ways* between the nodes they previously connected. If any improvement is made, the improved tour is used as the starting point of a new round of r-opt. If no improvement is made in a round, the algorithm stops. Normally, an r-value of 2 or 3 is selected.

Simulated annealing is a method based on material annealing, the method of heating up a material until its particles are randomly distributed, and then gradually cooling it down until it reaches a stable state. In solving an optimization problem, this means starting with a high *temperature factor*, T, and iteratively reducing it as the algorithm progresses. Like the r-opt method, simulated annealing considers a neighborhood of the given solution and tries to find a better solution in this neighborhood. Unlike the r-opt method, also *worse* solutions are considered for becoming the new solution. This helps avoiding local minima. The probability of adopting a worse solution increases with the value of the T-parameter. This way, a high degree of exploration is obtained initially, and the search is more focused during the final stages.

The method of *tabu search* also considers the neighborhood of the current solution. Like simulated annealing, it allows for the solution to deteriorate.

In each step, the lowest costing neighbor of the current solution is chosen, unless this neighbor is on the *tabu list*. This is a list containing all previously tried solutions, and the use of such a list allows the algorithm to avoid loops in the solution process. The method runs until all neighbors are on the tabu list, or for a predefined number of iterations.

Relevance

As a lot of work has been done in representing and solving TSPs by using genetic algorithms, and many successful applications have been reported, this project could benefit from using some of the best representations and operators from this area. Section 2.4.1 gives an overview of the most common approaches to solving the TSP with a GA, and Section 2.4.7 presents a GA solution to the *dynamic* TSP.

However, this problem is more complex than a TSP: The robot needs not only to *visit* the objects on the playing table, but also pick them up and deliver them, probably needing to deliver more than once, because of capacity issues. In addition, there are different weights, and hence different *scores* for picking up the different objects, and the robot will have to prioritize, as it will not be able to pick up all objects on the table in a round.

To implement capacity constraints, different scores for different objects and the need for prioritizing, it is necessary to look towards two other optimization problems, known as the orienteering problem and the vehicle routing problem.

2.3.2 The orienteering problem

Definition

The orienteering problem (OP) is a generalization of the TSP that relaxes the requirement of visiting all nodes. As explained in [26], in an OP the cities and traveling costs are defined as usual, but in addition a starting point and an endpoint for the tour is specified, as well as a parameter specifying the *maximum allowed traveling distance* for the problem instance. It is also common that all cities have an associated score representing the *utility* of visiting this city.

A TSP is the special case of the OP where the distance available is enough to cover even the worst tour, and all cities have the same scores. As the OP has a limited distance budget, a solution to the problem needs not include all available cities. Instead, the optimal solution will be the one that collects the highest score, and at the same time stays within the distance budget. This problem is NP-hard.

Common solutions

As this problem has many similarities to the TSP, many of the heuristics mentioned for solving the TSP, can also be used for the OP, with certain modifications. As for the TSP, both exact approaches and heuristics are employed. [5] reviews some common solution techniques for the OP. Exact solution methods for the OP are often based on the branch-and-bound method described in Section 2.3.1.

A common way of finding a good solution, that is not necessarily optimal, is to first enumerate several promising tours, then optimizing the tours with a good heuristic (for instance 2-opt, a variant of the r-opt heuristic described in Section 2.3.1, can be used), and finally selecting the best tours for further improvement through some local optimization procedure.

Enumerating promising tours can be done in many ways. Typically a tour is built by rating the potential nodes to add by some *desirability measure* (often based on the score of the point and the distance to the point), and then picking one of the best rated nodes. Always picking the best node can be useful when only generating one candidate tour, but some randomization among the best nodes is needed when generating many potential tours.

The local search used for final optimization of the solution, is often based on swapping some nodes in the solution, deleting nodes or inserting new nodes. For instance, [13] uses three strategies for improving a path: “one in - zero out”, “one in - one out” and “one in - two out”, referring to the amount of nodes to delete or insert. The best result obtained by one of these strategies is saved and improved further by *cluster exchange*, replacing a cluster of nodes with another one.

Relevance

While not as commonly studied as the TSP, also the orienteering problem has been solved successfully with genetic algorithms. Studying such applications, provided good examples of possible representations and operators for the GA implemented in this project, as the OP is so similar to the robot's task.

Just like the OP, the Eurobot task has objects with different scores and a limited time budget to collect as many points as possible. Although the OP is constrained by a distance, rather than a time limit, the conversion between the two types is straightforward: Just measure the time used by the robot for traversing distances, picking up objects etc., and end it's run when the time exceeds the specified limit.

The main difference between the OP and the Eurobot task, is that the OP does not have the concept of a *capacity* and, thus, does not capture the fact that the robot needs to visit the goal area regularly to make room for more objects. This constraint, however, is captured by the class of problems known as vehicle routing problems.

2.3.3 The vehicle routing problem

Definition

The vehicle routing problem, also a generalization of the TSP, is according to [22] defined on a graph with nodes and edges, where one node plays the special role as the *depot*. There are m vehicles with a certain capacity for carrying objects from the depot to the other nodes, representing customers. There are costs associated with traveling between nodes and with unloading goods. The optimal solution is the set of routes that minimizes travel- and delivery time, while staying within the vehicle's capacity limits. This problem is also NP-hard.

Common solutions

[15] reviews some of the methods used for solving the VRP, both exactly and approximately. Exact methods are classified into three categories: Direct tree methods, dynamic programming and integer linear programming. All these

methods typically include relaxing the problem constraints to find bounds on the solution costs, and then using a branch-and-bound method to find the optimal solution.

Like the OP, VRP-solutions are often based on solutions to the TSP, with some degree of modification. For instance, always inserting the nearest neighbor of a node also makes sense in this problem, but one must be sure to only construct valid tours (i.e. not overloading the vehicle).

There are also specialized algorithms for finding good solutions to the VRP. For instance, a variant of tabu search can be used. Another possible heuristic was proposed by Clarke and Wright in 1964. Their algorithm starts with n vehicle routes containing only the depot and one other node, and iteratively merges routes, based on the largest saving in traveling distance. The complexity of this algorithm is $\mathcal{O}(n^2 * logn)$.

Relevance

For VRPs, tabu search metaheuristics have yielded the best results, easily outperforming standard genetic algorithms. However, in [22] Christian Prins demonstrated that a *hybrid* GA, where mutation is replaced by a local search procedure, is able to compete with these powerful heuristics.

The VRP also has a lot in common with the problem faced in this project: The *depot* corresponds to the goal-area where objects are dropped, and there is a single vehicle trying to minimize the time spent traveling from the depot to the various points around the playing table, without exceeding its capacity. The main difference here, is that the VRP is concerned with finding a solution visiting *all* points in the problem definition, while the time limit of the Eurobot contest means that priorities will have to be made, so the most valuable points are visited first.

2.4 Previous Work

This section presents some of the previous work that has been done in solving path optimization and scheduling problems with the help of evolutionary algorithms. Implemented systems will briefly be described, along with thoughts on how the experiences from them can benefit this project.

The first sections describe systems designed for static planning problems. They provide useful ideas and solutions for both the static pre-match planner and the dynamic in-match planner, as both essentially perform the same task, but with different time constraints. The last sections describe systems for dynamic planning problems, and will provide a further insight into important considerations when facing a problem that changes over time.

2.4.1 Solving the TSP with evolutionary algorithms

[16] presents a review of previously documented attempts to solve the TSP with evolutionary algorithms. Below follows a summary of the most important ideas from the work reviewed.

Representations

The representation of a TSP tour can take several different forms. A simple binary or integer *path representation* can represent a permutation of the cities to visit, with one number representing each city and a tour consisting of visiting all the cities in the specified order. For instance, the genotype (4, 3, 2, 1) would represent a tour beginning at node 4, and ending at node 1.

An alternative approach is the *adjacency representation*. Here, integers once again represent nodes, but the position of the integer determines what node *leads* to that integer's node. For instance, if the number 8 is at the third spot in the genotype, this would mean that there is a path from node 3 to node 8 in the tour. Unlike the path representation, this one may result in illegal tours even if all integers are only used once. During a regular crossover, both this and the path representation can give invalid tours, due to duplicates.

The third representation reviewed is the *ordinal representation*. This representation maintains an ordered list of nodes from 1 to n, and each spot in the genotype explains what *position* in the ordered list the next node in the tour should be selected from. Every time a node is selected in the list, this node is removed, causing the list to shrink. So, the item at a specific position will vary during the construction of the tour. For instance, the genotype (1, 1, 1, 1) will always select (and remove) the first element from the ordered list, giving the tour (1, 2, 3, 4). The reason this representation is sometimes applied, is that it allows normal crossover to be used without risking generation of illegal tours.

Crossover

The simplest kind of crossover selects a random crossover point in the parent genotypes, and recombines them at this point to generate the child genotype. This kind of crossover will work fine with the ordinal representation, but will potentially insert duplicate nodes in tours when using the other two representations. It can still be used, but further actions are needed to remove these duplicates, replacing them with the cities that are not part of the new genotype.

For the adjacency representation, a specialized crossover operator is the *alternating edge crossover*. This alternates with selecting edges from each parent, starting at one parent and then adding the appropriate edge from the second parent. For instance, if the edge (1,4) is selected from the first parent, the edge starting at node 4 is added from the second parent. If an insertion produces a cycle, a random node is inserted instead.

The most natural and most used representation is the path representation, and many crossover methods have been suggested for it. One common crossover technique is the partially-mapped crossover (PMX). This operator maps a portion of one parent genotype onto the other parent genotype, and exchanges any duplicate nodes. For instance, if the parent genotypes are (1, 2, 3, 4, 5) and (4, 2, 1 ,5, 3),the three center integers could be mapped from parent 2 to parent 1, generating the genotype (1, 2, 1, 5, 5). This generates two duplicates outside the mapping area, so they are exchanged with their corresponding integers *inside* the mapping area, in this case 3 and 4, generating the final genotype (3, 2, 1, 5, 4). This kind of crossover preserves some ordering information from the parents, while still generating novel combinations.

Another popular crossover technique for the path representation is order crossover (OX). This operator also starts by mapping one portion of a parent onto the other parent, but after the mapping point nodes are simply inserted in the order they appear in the other parent, skipping the nodes inserted in the mapping. For instance, if the parent genotypes were (1, 2, 3, 4, 5) and (4, 1, 5, 2, 3) and the mapping happens in the second and third integer, the result after mapping parent 2 into parent 1 would be (-, 1, 5, -, -). Then, integers from parent 1 would be inserted from the fourth position as they appear, wrapping around when reaching the end of the genotype, but skipping 1 and 5. The result would be (3, 1, 5, 4, 2). A theoretical analysis has proven this to be more suitable for the TSP than the PMX operator.

The final crossover operator presented here, edge recombination crossover (ER). This operator tries to preserve the edges in the parent solutions, as they carry information important to the success of an individual. The operator makes sure *only* edges from parents are present in children, except for the last edge from the final to the initial city, which may be new. The operator works by maintaining an *edge map*, which for each city shows what cities it is connected to in any of the parents. This edge map is used when generating child genotypes, by always linking a city to one of its connected cities in the edge map. When multiple cities are available, the one which has the *fewest* entries in its own edge map is selected, so that no city "runs out" of neighbors until the final edge is to be inserted. [16] indicates that this operator is better for the TSP than the other operators presented.

Mutation

[16] also reviews many mutation operators that have been used for the path representation of the TSP. *Displacement mutation* removes a sub-tour from the solution, and inserts it in a random place. *Exchange mutation* swaps two randomly selected cities in a tour. *Insertion mutation* removes a city from the tour, and inserts it in a random place. *Simple inversion mutation* selects two cut points in the genotype, and reverses the order of nodes between these points.

An alternative to these kind of mutation operators, is employing some local search, which makes some *intelligent* mutation to the genotype, optimizing it. This should be done with care, of course, to avoid getting stuck at *local* optima. One possible solution is to randomly mutate with some probability p, and employ local searches with probability (1-p).

2.4.2 Production scheduling with a hybrid genetic algorithm

[6] looks at a specific class of production scheduling problems in manufacturing. The problems are characterized by having a set of orders, and not enough capacity to process all the orders. The task therefore becomes to:

1. Select the orders that should be processed.

2. Determine the sequence these orders should be processed in, based on constraints in the problem definition.

The system

The proposed system uses a genetic algorithm combined with extremal optimization (EO) to solve the production scheduling problem. The GA ensures that the algorithm doesn't get stuck at a local optimum, while EO works as a fine-grained local search and is used to improve single solutions efficiently.

Extremal optimization plays the role as the mutation-operator in this algorithm. The EO takes a sub-optimal solution and iteratively replaces the most undesirable components of this solution with other randomly selected components. In this algorithm, a solution *component* is the execution of a single order, and the *desirability* of the order is measured by the transition cost incurred by handling that order at its current position. This cost equals the transition from the previous order to the current order, and from the current order to the next order. Thus, the EO replaces the *least efficiently placed* order.

The representation of solutions employed in this algorithm is a simple ordering of task IDs, sequencing the production tasks that have been chosen. Initialization of the first solutions is done partly at random, partly by a heuristic algorithm, and partly by using the *nearest neighbor* search method, which iteratively picks the order with the lowest transition cost, from a random starting order.

Mating selection is implemented as rank based selection, in which the probability of selecting an individual does not depend on its actual fitness, but on its *ranking* compared to other solutions' fitnesses. The algorithm also employs a technique for ensuring that the individuals involved in mating are sufficiently different, in order to maintain genetic diversity. The actual crossover is handled by variants of order crossover (OX) and partially matched crossover (PMX) that enable parents containing different subsets of tasks to mate without generating illegal solutions.

Usefulness

The task solved here resembles the Eurobot task in many ways, in particular in the way there is a need to select a subset of tasks and sequence this subset in a good way, relating to transition costs. The impact of extremal optimization has given up to 30% reduction in cost of the optimal solution, and also a much more rapid convergence, in terms of generations, indicating that *hybridizing* the genetic algorithm is reasonable for these complex optimization problems.

2.4.3 Capacitated vehicle routing with a cellular genetic algorithm

The capacitated vehicle routing problem (CVRP) is a special case of the vehicle routing problem described in Chapter 2.3.3, where all vehicles involved have the *same capacity*. [2] and [1] (both describing the same system) propose a cellular GA for solving this problem, with the additional constraint that all routes have a specified time limit.

The system

The genetic algorithm used for solving the CVRP is *cellular*, which means its population is structured in a certain topology, only allowing neighboring individuals to interact in the mating process. The topology in this case is a grid. In addition, the GA is *hybridized* with a local search method used to optimize individuals.

Solutions to the CVRP are represented as permutations of integers, some integers representing the customers to be visited, and the rest working as *delimiters*, indicating that one delivery round is done, and the next one should be initiated. This way, the number of total routes, is controlled by determining the range of integers. For instance, numbers 1 to 5 could be customers, while 6 is a delimiter, and the tour (1, 2, 3, 6, 4, 5) would consist of the two sub-tours (1, 2, 3) and (4, 5).

The population is structured in a 2D-grid, each individual having five neighbors: The ones directly next to it in the grid, and itself. All individuals

are iteratively selected for mating, the other mate being chosen through *binary* tournament selection among the neighbors, meaning that two randomly selected neighbors compete for the open mating spot.

Crossover is edge recombination crossover, and mutation is a combination of insertion (moving a gene), swap (swapping two genes) and inversion (inverting a sequence of genes). Mutation can happen within a single route, or involving several routes. After mutation, the algorithm attempts to optimize the individuals through two optimization methods: 2-opt and 1-interchange. The best individual (among the original mutated one and the results from the two optimization algorithms) is selected, and inserted into the next generation if it is better than the parent at the current place in the grid.

2-opt is a special case of the r-opt algorithm described in Section 2.3.1, and works by randomly breaking two non-adjacent edges, splitting the solution in two halves, and then recombining the halves by inserting edges between them in the opposite way from how they were previously connected. For instance, if the previous edges were (1,2) and (3,4), the new edges are (1,3) and (2,4). The effect of this is to reverse the sequence of nodes between the breakpoints.

1-interchange optimization exchanges single customers between two routes in every way possible, trying to find a better route organization.

The fitness of an individual decreases with the total time delivery takes, and with penalties incurred for spending too much time, or carrying more than the vehicle's capacity on a single route.

The algorithm was tested on several large instances of CVRP, and was able to *improve* the best known solution to nine of them, and coming very close (within 3%) to the best known solutions of the rest.

Usefulness

As previously mentioned, the Eurobot problem has many similarities with the VRP, so solution techniques from this algorithm are also relevant for the Eurobot task. The fact that all vehicles share the same capacity, makes it even more similar than the general VRP. A particularly interesting feature of this system, is the route delimiters employed in the solution representation, which give the GA freedom to experiment with generating new routes, combining routes and so on.

2.4.4 Solving the generalized orienteering problem with a genetic algorithm

[27] used a genetic algorithm to solve the generalized orienteering problem (GOP). GOP is a generalization of the OP, where the points to be visited have *several* scores, with respect to various attributes, and where the overall objective function is a nonlinear combination of these attribute scores. As a motivating example, the article mentions taking a trip in China, where places to visit can be ranged by their beauty, historic relevance and several other attributes. The traveler wishes to maximize all scores, but this is probably not possible, so he faces an optimization problem with *multiple* goals.

The system

The solutions to the problem are represented as permutations of integers, representing the possible cities to visit. Producing the initial population is done by starting at the first city, and then adding the closest city with a probability of 0.4, and the second and third closest with probabilities of 0.3. Doing this 50 times, and saving the tours in both forward and backward order, generates 100 individuals. To get the tours to stay within the time limit given, the algorithm finally removes as few as possible consecutive cities from the end of the tours. Finally, the 51 best tours are chosen to serve as the initial population.

Crossover is based on queen-bee selection and uses edge recombination crossover. Queen-bee selection means that the current best path (the queen) gets to participate in all the matings, and the other mate is chosen by proportionate fitness selection. The edge recombination crossover used here, is a variant in which the node to be added is not the one with the fewest neighbors in the edge table, but the one *closest* to the current node. This ensures *the best* edges are transferred to the next generation, at the cost of more edges being broken, compared to normal ER crossover.

Mutation is implemented as the 2-opt procedure explained in Section 2.4.3. If a tour is selected for mutation, 2-opt is run on *all pairs* of nodes, so the best possible mutated sequence is found.

After mutation and crossover, the tour may have become longer than the distance constraint, so the feasibility of all tours has to be checked, and cities may have to be removed from the end of the tour. During crossover and

mutation, more individuals than required are generated, and intermediate individuals (for instance the individuals that are done with crossover but not with mutation) are also saved, so to generate the next generation, a selection has to be made. This selection is done by *always* keeping the very best path (this is known as elitism), and selecting others by proportional fitness among the remaining individuals.

Usefulness

The Eurobot task has only one type of score for each point: The weight of the object there. So, the fact that this is a *generalized* version of the OP is not really useful. However, many of the ideas here are quite interesting. Ideas like queen-bee selection and elitism are efficient in preserving information about the best-so-far solutions throughout generations. This is often necessary in these complex problems, as small changes to good solutions may decrease their fitness severely.

Another interesting idea presented, is the idea of letting the GA produce too long tours, and then simply adjusting them to the distance constraint later. This gives the GA more freedom to experiment and combine good solutions, even though they may result in invalid tours.

2.4.5 Solving the orienteering problem using a genetic algorithm with an adaptive penalty function

[26] looks at another way of solving the OP with a GA. Quite a few interesting ideas are presented, among others an adaptive penalty function, allowing infeasible solutions to survive, in hope that they can produce useful offspring.

The system

This algorithm uses the same solution representation as most of the other algorithms considered here: A permutation of integers. Initial solutions are randomly generated, but it is made sure that they follow the distance constraint of the OP. Crossover is handled by *injection*, to facilitate genotypes of different lengths. Injection means that a sub list from the first parent is inserted at an injection point in the second parent to generate a child. Any

duplicate points injected are deleted from the child. Finally, the child is fitted to the size of the first parent.

Mutation is enriched by a local search, and this proved to significantly improve the performance of the GA. The four local search methods employed are add, omit, replace and swap of integers. Each of the four operators is tried ten times at random places on the offspring that is to be mutated, saving the best offspring created.

The most interesting idea of the article is the *adaptive penalty function*, which allows also infeasible tours (i.e. tours longer than the distance constraint) to survive. This function decreases the fitness of solutions according to their *distance from feasibility*. This way, solutions with high scores, but that are a little too long, may survive and perhaps produce offspring within the feasibility limit.

Usefulness

The problem considered here has a lot of similarities with the Eurobot task, the most noteworthy *difference* being the robot's need to deliver objects to the scoring area several times during a run. So, the ideas here may prove valuable in this project – in particular the adaptive penalty function seems interesting.

2.4.6 Solving the vehicle routing problem with a genetic algorithm

[22] presents an evolutionary algorithm for solving the VRP that is able to compete with the powerful tabu search (TS) algorithms in terms of solution costs.

The system

The system represents a solution as a permutation of the n customer nodes. Unlike the VRP-solver presented in Section 2.4.3, this system has no trip delimiters. Instead, a procedure finds the optimal places to split the tour, based on capacity constraints, after the tour has been generated.

The individuals of the population are kept in an array sorted by the cost of the solution. To avoid premature convergence, the algorithm has a limit of how similar two individuals are allowed to be. To avoid having to check genotypes in detail, this limit is based on cost: the costs of two solutions cannot be closer than a given constant.

Three initial solutions are generated by three different heuristics known to be efficient. The rest of the initial population is initialized as a random permutation of customers, making sure to avoid similar costs. For crossover, the order crossover operator is used.

Instead of random mutation, the algorithm uses local search, with a fixed probability of mutating offspring. The local search considers all pairs of nodes in the problem and tries to optimize the tour by nine different simple swapping rules changing the position of one or both nodes in the genotype. Among these rules are the 2-opt and 1-interchange moves described in Section 2.4.3. The local search has a runtime of $\mathcal{O}(n^2)$, where n is the genotype size.

Parent selection is done by the binary tournament method. Two tournaments yield the two parents. The OX-crossover generates two children, and a *random* child is selected for entering the population. Selecting a random child instead of the best child gave a better average GA performance. To make room for the child, a mediocre chromosome (drawn above the median) is removed from the population. In each iteration of the GA, only one child is inserted into the population. The algorithm is run for a given number of iterations, until convergence, or until reaching a predefined optimal cost.

Usefulness

The fact that this algorithm was able to show results comparable to TS, the most powerful heuristic for the VRP, is promising. A potential problem is the previously mentioned difference between the Eurobot task and the VRP. This algorithm is quite fine-tuned to a particular problem type and particular parameters, and shows degradation of the average solution if parameters are changed. Specifically, strong degradation occurs if the population is allowed to have clones, or if the local search is replaced by simple move or swap of nodes.

2.4.7 Solving a dynamic TSP with an evolutionary algorithm

[28] proposes an evolutionary algorithm for solving the *dynamic* TSP (DTSP). This problem, which was first defined in [23], is harder than a regular TSP because the cities involved and the costs of traveling between them vary over time. The DTSP is far less researched than its static counterpart, and [28] claims DTSP research is currently in an initial phase, with many questions waiting to be answered. Like a regular TSP, the DTSP is NP-hard.

The systems

In the DTSP, cities may be added or deleted from the problem over time, and the distances between cities may vary over time. The goal of a DTSP solver will typically be to minimize the total traveling cost at each time step.

[28] handles the problem by using an evolutionary algorithm known to be efficient for solving the *static* TSP, and extending it with three dynamic operators. The EA, known as the Inver-Over EA then drives the solution process forward, and the dynamic operators enable it to respond rapidly to the changing environment.

The three dynamic operators are employed every time the TSP problem changes. The operators are:

Insert: This operator is employed whenever a new city is added to the TSP. It adds the new city, C, to an already generated tour by finding the city closest to C, and placing C directly before or after this city in the tour, whichever yields shortest tour. The operator's complexity is $\mathcal{O}(n)$.

Delete: This operator is called when a city disappears from the TSP definition. It simply removes the disappearing city from the previously generated tour, resulting in its neighboring cities becoming linked together in the new tour. The operator's complexity is $\mathcal{O}(1)$.

Change: The change operator is called whenever the location of a city changes. It consists in first calling delete and then the insert-operator on the city that moved. This is likely to result in a more efficient tour than if the city remained at its previous position in the tour after it

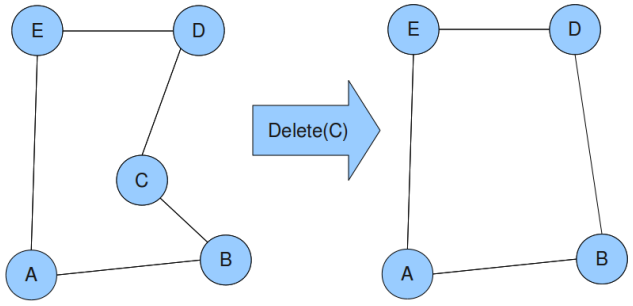


Figure 2.2: The delete operator creating an efficient tour

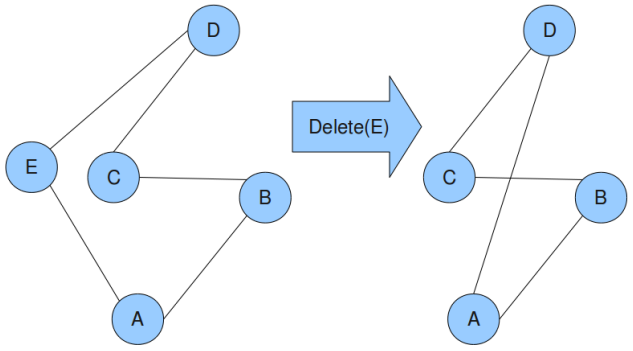


Figure 2.3: The delete operator creating an inefficient tour

was relocated in the problem definition. The time complexity of this operator is $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$.

Even though these operators are likely to produce good tours, they do not guarantee that cities are inserted in the best possible positions. Therefore, the EA is needed for further optimizing the generated tours. Figure 2.2 shows how the delete operator can be efficient in some cases, while Figure 2.3 proves it may produce clearly sub-optimal tours in others.

A similar approach to solving the DTSP was taken in [17]. Here, the authors propose a GA hybridized with the 2-opt or 3-opt local search techniques described in Section 2.3.1. They claim this combination yields an algorithm that both shows rapid response and provides high quality solutions. This approach utilizes more complex heuristics (2-opt and 3-opt have complexities $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively), meaning it will be more time consuming but at the same time have a better chance of adjusting to the changing circumstances without running several generations of the GA.

Usefulness

This method for generating good tours in a changing environment is very promising as a possible solution technique for dynamic in-match planning. The three operators employed in [28] are also very relevant for the Eurobot problem, as both insertion, removal and changing locations of playing elements is likely to occur (insertion occurs if the opponent first picks an element up and later drops it). The fact that the method is based on an EA for solving a *static* TSP problem is also promising, as it indicates that using the same EA for solving the static pre-match planning problem and the dynamic in-match planning problem may prove to be an efficient solution, if the dynamic planning is extended with some good heuristics.

The results on testing the algorithm in [28] on a 100-city dynamic TSP instance are good: Even when all 100 cities change their locations randomly (according to a Gaussian distribution) between every 0.5 and 2 seconds, the average error is only 7%, compared with an extensive 40.000 generation GA used to generate a close to optimal solution, at each timestep. The algorithm was run on a P4 1.4GHz with 256 MB RAM. The good response under large environmental changes makes this a feasible approach for the heavily time-constrained dynamic planning in the Eurobot matches.

Unfortunately, the test results do not present the standard deviation used in the Gaussian distribution governing the random movements of the cities. This makes it hard to say how big the changes to the TSP problem actually were at each time step. In addition, there were no experiments explicitly deleting and inserting cities into the TSP, as the change operator includes both a delete and an insert. For these reasons, the actual performance of this approach on the Eurobot problem is difficult to estimate. Still, the idea of extending an EA with change-handling heuristics seems like an efficient way of dealing with the dynamic planning problem, due to its ability to both quickly respond to changes and later optimize this response incrementally in the EA.

2.4.8 Dynamic vehicle routing

A *dynamic* VRP is a VRP where constraints may change *during* execution of the planned route. Compared to regular VRPs, very little has been published about the dynamic variant. A search in Google scholar returns 12.100 results

for the phrase "vehicle routing problem", but only 616 for "dynamic vehicle routing problem". [23] was one of the first papers to address this problem. It describes 12 points that need to be given particular attention in dynamic VRPs. The most relevant ones for the Eurobot problem are:

The time dimension: Unlike TSPs and many static VRPs, where *distance* is what we wish to minimize, in dynamic problems *time* is the essential factor. This implies that we need to keep track of how the problem and the execution of the plan evolves over time.

Future information may be unknown: In static problems, the entire problem definition is usually given beforehand. However, in a dynamic problem, much may be unknown as execution begins, so some reasoning about unknown future events may be needed.

Near term events are more important: This is an implication of the previous point: As events are increasingly uncertain longer into the future, focus should be on events occurring in the *close* future.

Resequencing may be warranted: Changing conditions may render already planned tours sub-optimal – so some replanning should occur whenever something changes.

Faster computation time: Real-time planning usually has harder time constraints than static planning. This often implies the use of some fast heuristic for modifying plans.

Like the static VRP, the DVRP is NP-hard.

Evolutionary approaches

While far from as commonly researched as the static version, some interesting GA solutions to the dynamic VRP have been reported. A few these solutions are briefly summarized here.

[24] considers the single-vehicle pickup and delivery problem with time windows and capacity constraints. This problem is quite similar to the Eurobot task, the most notable differences being the time windows that restrict *when* each pickup and delivery in the VRP should be made, and the fact that there is no *total* time constraint in the VRP. The problem is solved using a hybrid

genetic algorithm, as the authors expected this to make it possible to generate sub-optimal solutions on demand, and improve these solutions incrementally using the GA whenever more time is available. The algorithm feeds the current state of task execution to a dynamic programming module, which works out solutions to the current problem. The dynamic programming module is time limited, so it will not be able to finish solving the problem; rather unfinished solutions are passed on to the GA to serve as its initial population. Experiments performed show that the average solution cost is lower when hybridizing the GA with the dynamic programming module than when running only the GA.

[12] proposes a GA for the time-dependent VRP (TDVRP). This is a quite complex problem with several vehicles, both delivery and pick-up demands and soft time windows, penalizing vehicles for being too early or too late. In addition, plan adjustments may occur during plan execution due to changing circumstances. A regular GA is employed with crossover and mutation operators specifically tailored to the route representation adopted in the GA. In addition, a special genetic operator called *vehicle merging* is implemented, which tries to merge two vehicle routes into one to reduce costs. Experimental results are good – on mid sized problems (30 demand nodes, 30 timesteps) the algorithm is able to come within 7% of the optimal solution on average, and with substantial time savings compared to the time needed to obtain an exact solution.

[10] considers a DVRP pick-up problem, where some customers are known before planning begins, and some may arrive over time. The DVRP is here viewed as a series of *static* VRPs to be solved at each timestep. In each time step, the algorithm begins with an *event scheduler* that generates a static vehicle routing problem based on the current status of all vehicles and customer demands. Then, for each static VRP, an ordinary genetic algorithm is executed. The algorithm was tested on publicly available VRP benchmark data, with between 50 and 199 customers. The GA was allowed to process each timestep for a maximum of 30 seconds. The results obtained were compared with an *ant colony system* algorithm and a tabu search algorithm, both previously designed for solving the DVRP, and the GA performed favorably: Against the ant colony system the GA was able to generate better average solutions for all 21 benchmark data sets, and against the tabu search algorithm the GA generated better averages in 19 of the 21 data sets.

2.5 Discussion

Reviewing previous work in the domain of optimization provided a lot of different ideas to work on when approaching the Eurobot task. For the static planning implemented as the author's last year specialization project, a genetic algorithm was chosen as the solution method. The GA was chosen, because it is new and interesting solution technique that is beginning to show results comparable with those of the tabu search, one of the most used and powerful heuristics for the type of optimization problems discussed here.

An advantage of using GAs, is that they allow the designer to extract parts of the algorithm (like crossover operators, selection mechanisms etc.) from different previous solutions, combining them in the way that suits the problem best, as long as a proper solution representation and fitness function is chosen.

In reviewing previous evolutionary approaches to both static and dynamic optimization, some common features have become evident:

- Representing tours as simple sequences of integers.
- Using crossover mechanisms known from solving the TSP, like order crossover, partially matched crossover and edge recombination.
- The use of local search methods and heuristics, both for making good initial solutions, as an alternative to random mutation and for providing rapid responses to environmental changes.
- The use of elitism, to avoid losing the best solution.

When extending the task to also include dynamic planning responding to environmental changes, it was necessary to gain an insight into dynamic optimization problems. As a GA was already implemented for solving the static planning problem, special attention was given to GAs as dynamic optimizers. Studying GA solutions to the dynamic vehicle routing problem and the dynamic traveling salesman problem has indicated that regular GAs may be used for solving dynamic optimization problems, and that hybridizing them with good heuristics may give the algorithm the extra responsiveness it needs to deal with a changing environment. Other experiences with evolutionary approaches to dynamic planning indicate that maintaining a diverse solution is important for being able to respond to environmental changes. Therefore,

special care should be taken to avoid convergence towards a single solution in dynamic planning problems.

Chapter 3

Methodology

A genetic algorithm hybridized with a powerful local search technique was implemented as the author's specialization project, and it proved to generate good static solutions to the Eurobot task. It was decided to use this algorithm as the basis for the dynamic problem solver needed for plan adjustments throughout entire Eurobot matches. However, several extensions and adjustments had to be made to move from working on a static problem to solving a task in a rapidly changing environment.

Sections 3.1 and 3.2 describe the previously implemented system, and the rest of this chapter describes how the system was extended and adjusted to cope with the dynamic task.

3.1 The Static GA

The GA implemented as part of the author's specialization project solved the task of finding near optimal plans for any of the 36 table setups, given certain data about the robot, like its capacity for carrying objects, its speed and how fast it can pick the various objects. Figure 3.1 shows the evolutionary loop.

The algorithm has all the typical characteristics of a genetic algorithm: Starting from a randomly initiated population, several individuals compete for the right to become adults and mate, based on their fitness value, to form the next generation of individuals. Mechanisms for both adult selection and

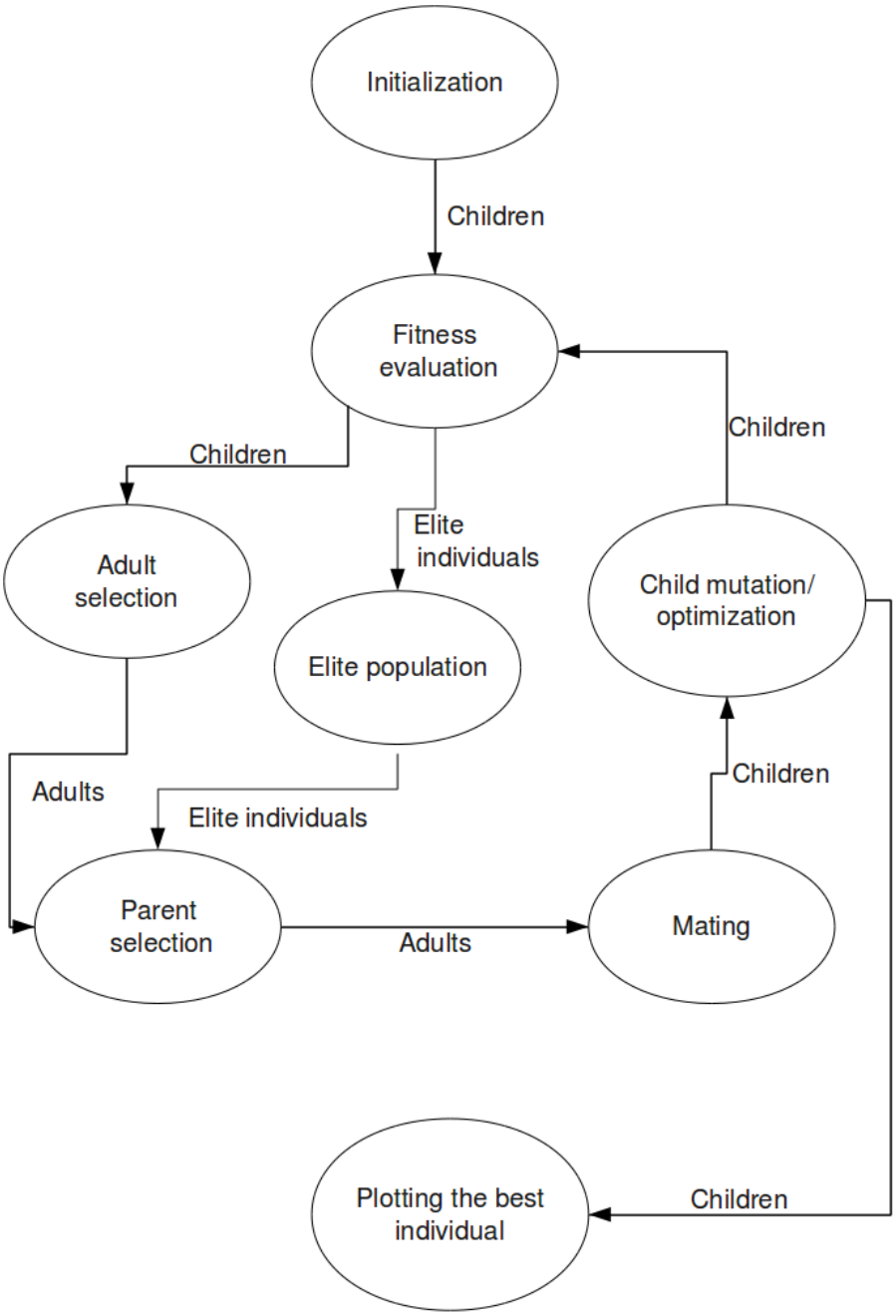


Figure 3.1: The implemented genetic algorithm

mating selection were implemented, meaning there is competition both for growing up and for regenerating, being able to form the next generation.

Mutation as well as a local optimization method was implemented for modifying individual solutions. Finally, elitism was implemented, to avoid losing good solutions.

3.1.1 Representation

Solutions to the GA were represented as permutations of integers, each integer representing a playing object or the goal area. The entire permutation thus becomes the robot's strategy for the match. As not all points will be visited in the 90 seconds of the match, solutions are implicitly cropped during fitness evaluation, by disregarding actions taken after the 90 seconds.

The size of solutions range from 37 integers and up, depending on how many delivery actions the user has allowed for a single match. Typically, three deliveries were allowed, yielding a genotype size of 40, striking a balance between giving the GA a lot of freedom, and limiting its search space.

No distinction was made between genotypes and phenotypes in the algorithm – both were simply the same permutation of integers.

3.1.2 Selecting individuals

To select individuals for the next generation of the GA, a *selection protocol* and a *selection mechanism* is needed.

The *selection protocol* determines *which* children will compete for spots in the next adult population. Three types of selection protocols were implemented, allowing the *selection pressure* of the GA to be adjusted: In *full generational replacement*, there is no selection pressure. All adult spots are simply replaced by the new children. In *generational mixing*, there is more pressure, as both adults and children compete for these spots. Finally, in *over-production*, m children are created to compete for the n adult spots, so the selection pressure can be determined by the user by tuning m and n.

The *selection mechanism* determines *how* the competition among the individuals is performed. Thus, it is also an important factor in adjusting the

selection pressure. Two selection mechanisms were implemented: *Boltzmann selection*, which was used during adult selection, and *tournament selection*, which was used during mating selection.

Boltzmann selection

Boltzmann selection is based on the simulated annealing technique described in Section 2.3.1. This selection mechanism scales the fitness of all individuals according to the factor T, called the *Boltzmann temperature*. By adjusting this factor throughout the evolution, it is possible to move from an initially relatively random exploration process, to a more focused exploration towards the end making the best solutions even better. The scaling equation for fitness values in Boltzmann selection is [8]:

$$ExpVal(i, g) = \frac{e^{f(i)/T}}{\langle e^{f(i)/T} \rangle_g} \tag{3.1}$$

In the above equation, g is the current generation, T is the Boltzmann temperature, $f(i)$ is the fitness of individual i, and the brackets mean the average value of the function during generation g. $ExpVal(i, g)$ is the expected number of times individual i will reproduce in generation g, which of course depends on the individual's fitness value compared to the average fitness value in the current generation. Obviously, scaling the temperature will enable control of this scaling factor, and in turn, scale the selection pressure.

Once fitness values have been scaled, comes the actual selection, and this is done according to *roulette wheel selection*, which simulates assigning all individuals to a roulette wheel, giving them space on the wheel according to their percentage of overall fitness. Then the wheel is "spun", and a random individual wins, and is removed from the wheel and into the next adult population. This gives all individuals a chance of winning proportionate to their scaled fitness value.

Tournament selection

Tournament selection was used for selecting individuals for the mating process. No adults were allowed to mate with themselves, so for each child in the next generation, two different adults needed to be selected.

This selection mechanism was run once for each individual that was to be selected, and all individuals took part in each tournament, meaning that one individual could be picked several times. All tournaments begin with picking k (a user-specified integer) individuals at random from the population. These individuals are this tournament's *contestants*. With a probability of p, the *tournament factor*, a random individual among the contestants is chosen as the winner. With a probability of $(1 - p)$, the contestant with the highest fitness value is the winner.

Thus, by adjusting k and p, the user can tune the selection pressure in this mechanism.

3.1.3 Mating

After selecting adults for mating, comes the actual mating process. Some of the offspring created are the result of crossover, in other words mixing the genetic material from two adults, while others are the result of copying the exact genotype of one parent. The probability of these events is controlled by a factor set by the user, to allow experimentation on how much differentiation is healthy for the solutions. Too little, and premature convergence may be the result. Too much, and there is a risk of never converging at all.

The crossover mechanism is *edge recombination crossover* (ER), a common crossover mechanism for TSP-solving GAs, that was also used in two of the systems reviewed in Section 2.4 for solving the OP and the VRP. The crossover mechanism is described in Section 2.4.1.

3.1.4 Local optimization

Local optimization took the place of the mutation operator in the GA. Results from the specialization project showed great improvement over random mutations when applying a local search technique. Much of the work reviewed in Section 2.4 has also benefited from hybridizing genetic algorithms with local search techniques when solving complex scheduling problems.

The local search technique that was implemented, is based on the large local search employed by Christian Prins in [22]. It was used as part of a genetic algorithm outperforming the powerful tabu search heuristics on many classical VRP instances. The local search considers all $\mathcal{O}(n^2)$ pairs of integers

Table 3.1: Steps in the large local search

M1	Insert u after v
M2	Insert (u,x) after v
M3	Insert (x,u) after v
M4	Swap u and v
M5	Swap (u,x) and v
M6	Swap (u,x) and (v,y)
M7	If u and v are in the same tour, replace (u,x) and (v,y) by (u,v) and (x,y)
M8	If u and v are *not* in the same tour, replace (u,x) and (v,y) by (u,v) and (x,y)
M9	If u and v are *not* in the same tour, replace (u,x) and (v,y) by (u,y) and (x,v)

in the genotype, and tests nine different rules for each pair, looking for an improvement to the current fitness value. If an improvement is found by one of the rules, the solution is updated, and the next pair of nodes is considered.

As opposed to the VRP that the search technique was developed for, not all steps in the solution genotype of the implemented GA get carried out, as there is a time limit. Therefore, the efficiency of the search was increased, by stopping it from considering pairs of nodes where *both* nodes are outside the region of nodes visited within the 90 seconds of the match. Interchanging these nodes will not change the robot's behavior in the match, and hence, has no effect on the fitness value. On the other hand, considering pairs of nodes where only *one* is outside the 90 seconds of the match, is reasonable, as this will allow for *adding* nodes to (or *removing* nodes from) the robot's strategy for the match.

The nine rules used, shown in Table 3.1 are the same as in [22]. In the rules, u and v are the nodes in the current pair, while x and y are the nodes following u and v, respectively, in the current solution.

Even though these rules appear to be precisely the same rules that Prins used, there is a subtle difference, due to the difference in solution representation. In Prins' solutions, the genotypes did not encode route delimiters. Delimiting routes was done by a separate splitting procedure. In the GA implemented here, however, delimiting routes is done by specific integers in the genotype. This means that the local search has the opportunity to try out different delivery strategies, and not just change the order of visiting the pickup points. Note, however, that rules M7 - M9 were not allowed to fire if *any* of u, v, x or y were the delivery point, as these rules were specifically designed for handling exchanges *within* and *between* tours.

2-opt and 1-interchange are two heuristics commonly employed for path optimization problems, and were used in several of the systems described in Chapter 2. These heuristics are also part of the large local search, and are covered by rules M1, M4 and M7. Briefly explained, 2-opt has the task of *reversing* the order of visiting nodes between u and v, while 1-interchange tries to *exchange* nodes between different tours. Figures 3.2 and 3.3 illustrate how these actions can optimize tours.

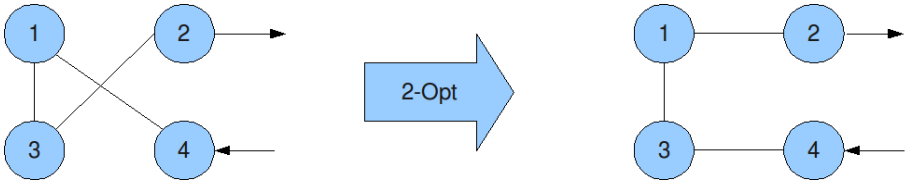


Figure 3.2: 2-opt

3.1.5 Elitism

During the background studies for the specialization project, a common feature of many of the studied scheduling algorithms was the use of *elitism*. Elitism means storing the best solution(s) obtained so far in a GA to avoid losing them in subsequent generations. This is important in complex scheduling tasks, because a small perturbation to a good genotype can destroy the solution. In this task, this can for instance happen if the *delivery* is removed, meaning no points are scored.

In the implemented GA, the number of elites can be specified before the algorithm is initiated. Figure 3.1 illustrates how elites take a "shortcut" past adult selection, going straight to parent selection. The worst elite is replaced as soon as a better solution is found.

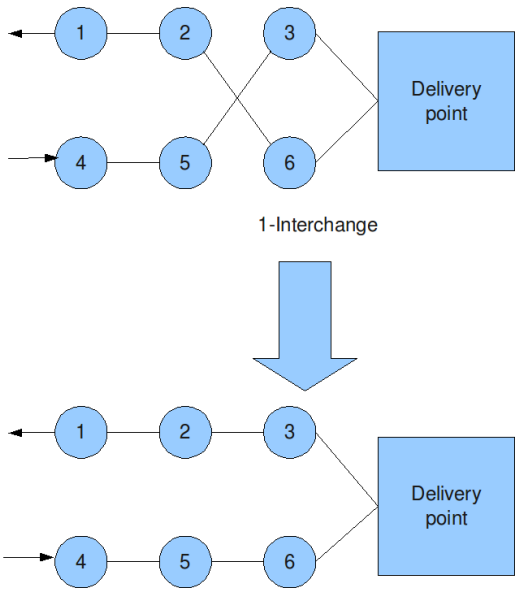


Figure 3.3: 1-interchange

3.2 The Simulator

For evaluating the fitness of any given solution, a simulator was implemented, that simulated an entire 90-second Eurobot match, and assigned a fitness to the solution based on the amount of weight it was able to deliver. Because the robot had not yet been designed at the time of implementing the simulator, it based its calculations on adjustable parameters representing the robot's speed, capacity and other physical attributes. Once the robot is ready, these parameters can be measured, and more accurate simulations can be performed.

3.2.1 Driving

Much of the time during a Eurobot match will be spent driving between objects. Therefore, it is important to have good estimates of the time translations will take. Vector mathematics was used to find the distance D and rotation r of any translation. Together with the number of accelerations, a, these factors gave the time spent on a regular translation from point u to point v:

$$Time(u, v) = \frac{r}{V_r} + (a * P_a) + \frac{D}{V_t} \tag{3.2}$$

Here, V_r is the rotational velocity of the robot, P_a is the acceleration penalty (the approximate time spent on a single acceleration and retardation), and V_t is the translational velocity of the robot. The velocities and acceleration penalty will be based on measuring the properties of the actual robot.

Obstacles

On the Eurobot playing table, there are always 7 black columns inserted in random configurations as obstacles. This has implications for fitness evaluation, because a translation crossing an obstacle is likely to be more time consuming than one that is not. To check if the robot's path intersects an obstacle vector mathematics was once again utilized to find the smallest distance between the current translation and all obstacles on the table. If any

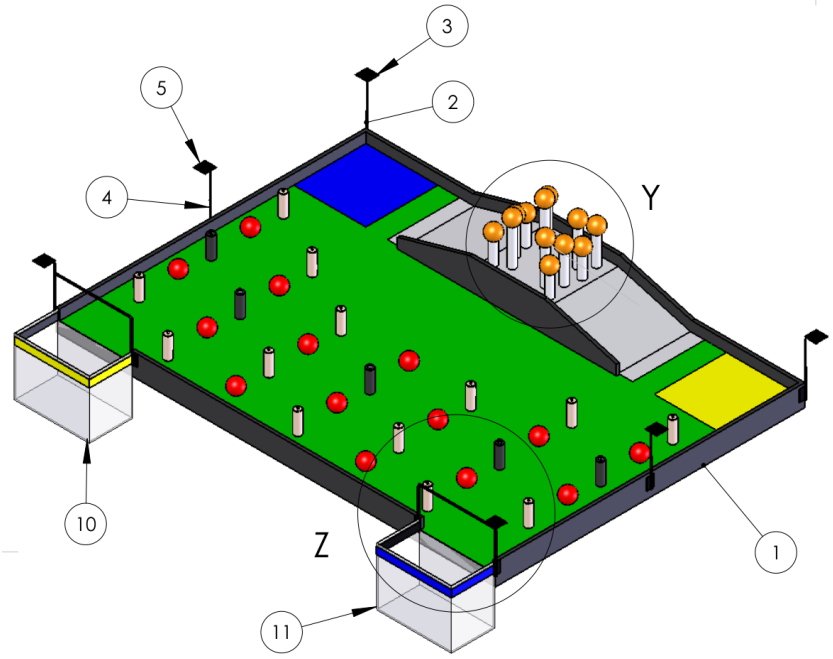


Figure 3.4: View of the game table showing the hill

such distance is smaller than the robot's radius plus the obstacle's radius, an intersection has been found.

Penalizing intersections is simply done by adding a penalty for each obstacle that intersects the current tour to Equation 3.2. What penalty to use will be based on the average time the robot uses to drive around an obstacle.

The hill

At the top of the playing area, is a hill only accessible from the sides (see Figure 3.4). Using Equation 3.2 directly on translations onto or down from the hill, will severely under-estimate the translation time, both because any path to the top of the hill will have to drive via the upper corner of the playing table, and because driving up or down the hill is likely to be slower than driving on the flat table.

To determine if any translation involves the hill, it's coordinates are checked. If the hill is involved, the translation is reformulated as a regular translation to (of from) the bottom of the hill followed by a *hill penalty*. The time spent

on the regular part of the translation is calculated by Equation 3.2, and the hill penalty is added to this time. The hill penalty is the time the robot uses to align itself with the hill, and then drive up or down from it. If either up or down goes faster than the other, the hill penalty will simply represent the *average* of the two. This works, because the robot always has to drive down once it has gone up the hill.

A special case involved translations from one side of the hill to the other, which had to be penalized by adding two hill penalties, as well as the time taken to drive from one side of the table to the other.

3.2.2 Picking up and delivering objects

Calculating the time taken to pick up and deliver objects, is based on parameters defining the time taken to pick up each type of object and for making a delivery of all objects. These parameters can be measured as soon as the picking- and delivery systems of the robot are ready. In addition, *capacities* were defined, that represent the number of each type of object the robot can carry. Once such a capacity is exceeded additional pickups in the plan have no effect until the robot has made a delivery.

3.2.3 Fitness evaluation

The fitness can be represented as the total weight delivered in the 90 seconds of the eurobot match. However, it was found that scaling the fitness helped steer the GA towards better solutions. The formula for calculating the fitness of a solution in the static GA was:

$$fitness(i) = \sum_{k=1}^{n} \left(w_k * (f_{ed})^k \right) + (f_{ef} * t_{rem}) + (f_{co} * objs) + (f_{prox} * (400 - goalDist)) \tag{3.3}$$

The first part of the equation calculates a weighted sum of all n delivered objects. f_{ed} is the *early delivery factor*, set to a number between 0 and 1. This ensures that objects delivered early in the match are weighted more than late deliveries. This is done because unforeseen events may stop the robot from executing the final parts of its plan.

The second part of the fitness calculation is a reward for finishing early. f_{ef} is the *early finish factor*, and t_{rem} is the time remaining until the round is done after the last delivery of the robot. The reason why it makes sense to reward the robot for finishing early, is that giving the GA a preference for such solutions, may allow it to keep promising solutions that in the future will be able to complete *another* round of gathering and delivering objects. It is important, however, not to assign a too high value to f_{ef}, as this will encourage the robot to be "lazy", delivering once early in the round and then never again.

The third part of the equation gives a bonus for picking up objects that the robot does not have time to deliver before the match is over. f_{co} is the *carrying objects factor*, and $objs$ is the number of objects the robot is carrying when the match ends. This behavior is rewarded, because it may steer the GA towards solutions where more pickups are made after the final delivery, and these pickups may prove useful later in the evolutionary process, if the GA is able to find a plan where these objects are actually *delivered*.

The last part of the fitness calculation gives a bonus for finishing close to the delivery area. f_{prox} is the *goal proximity factor*, and $goalDist$ is the robot's distance from the delivery area at the end of the match. This is also used to steer the evolution towards promising solutions. Solutions finishing close to the delivery area are preferable, because they have a good chance of adding another delivery to the robot's plan.

All the different factors can be specified before running the GA, and can therefore be experimented with to see what combination yields the best results. Notice how these factors together give rewards to tours that show promising *trends*, in other words tours that do not necessarily perform very well, but may become very good if some modifications are made. This way of steering the evolution towards promising solutions by modifying fitness values has certain similarities with the *adaptive penalty function* presented in [26], which was discussed in Section 2.4.5.

3.3 Solution Diversity

As discussed in Section 2.2.3, avoiding a too high degree of convergence in the GA is essential in changing environments. Maintaining a diverse population of individuals is therefore very important before and during the dynamic

planning.

3.3.1 Generating diverse elites

When starting the dynamic planner in a Eurobot match, the initial plans will be the same as the best plans obtained during pre-match static runs of the algorithm. These best plans are very likely to be the same as the elites discovered during the evolution, unless, of course, some better solutions are found in the very last generation. Nonetheless, several elites are certain to make it to the pool of initial plans.

For this reason, it would be a good property of the elite population to show a fair level of diversity, so the dynamic planning is not seeded with too similar solutions. Starting the playing round with diverse solutions, can be thought of as having "backup plans", that can quickly take over if problems arise in following the originally best plan.

Section 2.2.3 presented many different ways of ensuring diverse populations, many of which had the problem that they were computationally expensive, and that one somehow has to decide what it means for two solutions to be "different enough". For the Eurobot-task, with its various types of playing elements, this last part becomes particularly challenging, as there are so many options: Diversity measures would range from very *coarse-grained* measures, where the overall strategy is considered (for instance, strategies that first visit the hill to pick oranges, then pick corns and tomatoes on the way to the delivery area can be said to be quite similar) to more fine-grained ones, like counting the number of identical edges in the paths followed.

When deciding what level to compare solutions at, the computational expense of comparisons and the amount of diversity needed to handle the changes that are *likely to occur* were considered. The changes occurring in the match will most likely be the opponent removing, or blocking the path to, many of the elements of the original plan. Therefore, the diversity should be on an *element* level – meaning that the *overall* strategy of all initial plans could be the same. One way to measure diversity on this level, would be to compare the number of identical playing elements or path edges in between plans. A less computationally expensive method, however, is simply comparing the *fitness* of plans. If they do not have the same fitness, they cannot contain the same playing elements in the same order, and this ensures that backup plans exist.

The good thing about this method is that it is computationally inexpensive and straightforward to implement, as no interpretation of genotypes or phenotypes is required. The disadvantage is that the diversity obtained can be very limited. Two paths with different fitness values can actually contain the exact same playing elements, only visited in a slightly different order (due to the fitness scaling parameters discussed in Section 3.2.3). For this task, however, that is not a big problem, as confirmed by some initial experiments described in the next section. Still, a technique similar to the random immigrants approach is used to give a boost in diversity at the beginning of dynamic planning.

3.3.2 Initial results

Early in the development of the dynamic GA, it was considered if a more powerful (and resource demanding) technique for diversifying the population was needed, than simply diversifying based on fitness values. To determine this, some informal experiments were run on the static planner, to get an understanding of just *how* diverse the best results it created were. The only constraint ensuring the diversity of the solutions was the fitness-based constraint discussed above. The results showed a surprisingly high degree of diversity, indicating that a more sophisticated way of avoiding convergence will not be needed.

The experiments on the planner consisted in running it on all 36 game setups and studying how similar the *four best* solutions to each setup looked. Figure 3.5 shows a typical example of how four such solutions may look. These four solutions clearly share the same overall strategy: Two deliveries, and a single hill visit. However, the actual points visited, and the order in which they are visited varies quite a lot, indicating that the plans contain the diversity necessary to deal with changes caused by the actions of the opposing robot.

Explaining the diversity

At first, the great solution diversity obtained only by demanding that no two solutions share the exact same fitness may seem surprising. A further analysis of how the genetic algorithm works, however, gives a certain insight into how this happens. And interestingly, the local optimization, which one would assume contributes to a large degree of *convergence* is also important

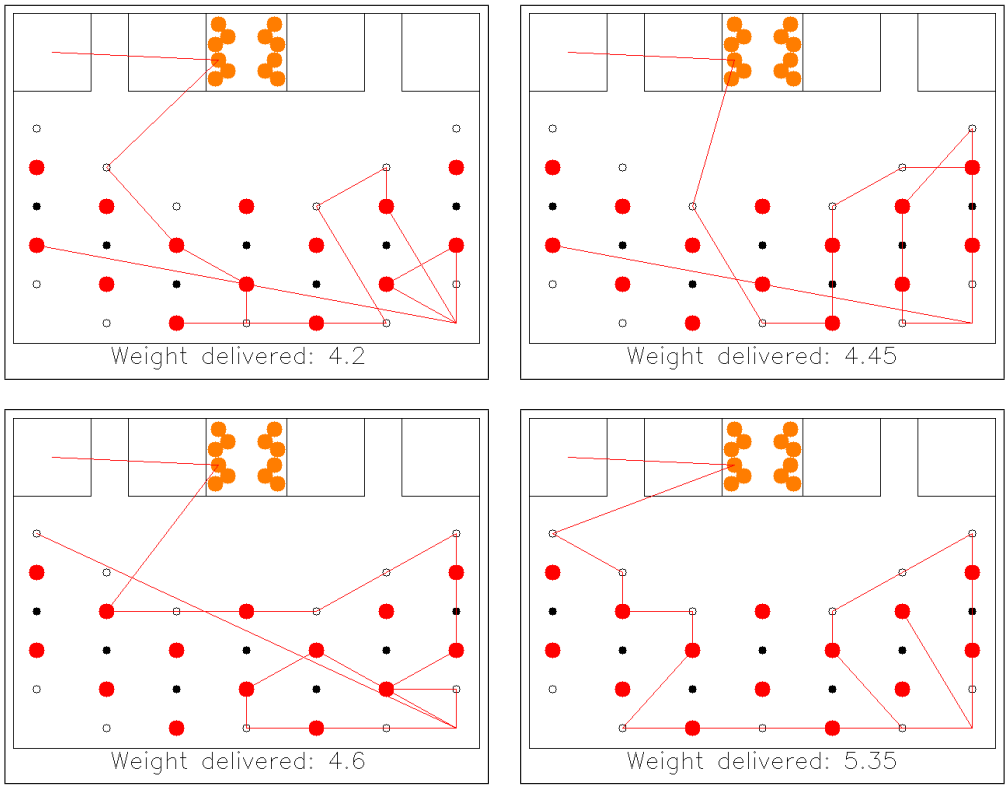


Figure 3.5: Diversity of solutions to the same problem

in diversifying the population.

As discussed above, a problem with the simple fitness-based diversity measure can be that it generates *almost* identical paths, perhaps only ordering the elements visited slightly differently. This will give non-identical fitness values, because of the fitness scaling discussed in Section 3.2.3. This is where the local search turns out to be very useful: If two solutions are almost identical, the local search will *make* them identical (to the local optimum in their neighborhood), meaning that one of them does not get to be part of the population. Notice how this resembles the *crowding* technique discussed in Section 2.2.3, where new solutions that are too similar to old ones, simply replace them in the population.

In addition, the genotypic representation selected helps in generating diversity. Because the last genes of the genotype do not affect the individual's fitness in any way (they are not part of the 90 second match), many different, randomly created, permutations will typically exist for these genes. This enables the edge recombination crossover to keep trying out new edges when generating new children, instead of converging towards a fixed set.

3.3.3 Diversity boost

During the entire dynamic planning period, solution diversity needs to be high. After the static plan optimization, a *certain degree* of convergence will happen, regardless of the care taken to diversify elites. Therefore, not *all* the best individuals obtained from static planning are used as seeds for the dynamic planner, but rather, the n best are selected, and the rest of the population for the dynamic planner are randomly initialized individuals.

This technique resembles the *random immigrants* approach to GAs, where random individuals are inserted in each generation to avoid convergence. Due to the short duration of dynamic planning, adding random individuals each generation will not be necessary, as the diversity created in the match initialization is likely to last for some generations, and because the method for creating diverse elites based on fitness values will still be active during the match.

3.4 Heuristic plan modification

It takes the GA some time to adjust its plan. How much time depends on how far into the match we are (the shorter the time horizon for planning, the shorter the time to finish a generation of the GA), but a generation takes from about half a second to about fifteen seconds to finish. Therefore, the planning system needs the ability to make good and fast adjustments to the current plan *without* completing a generation of the GA. Because at any time the changes made to the table are likely to be quite small, a *local search* around the current solution may be sufficient in the short term whenever a playing object is added, moved or removed from the table. A solution close to the *global* optimum will later be generated by the GA when it has been given sufficient time to deliberate.

3.4.1 The implemented heuristic

A simple heuristic, proposed by [28] for solving the dynamic TSP, was chosen for implementation. Alternative heuristics were the 2-opt and 3-opt local search techniques, which were used for solving the dynamic TSP in [17], or even performing the entire local search presented in Section 3.1.4 whenever changes are made. The choice fell on the simpler heuristic because it has a short execution time, which is important because it serves as the robot's *immediate* response to changes. Its runtime is constant for objects removed from the table, and $\mathcal{O}(n)$ for inserted and moved objects (where n is the number of nodes in the robot's plan). 2-opt and the large local search, on the other hand have a complexity of $\mathcal{O}(n^2)$, while 3-opt is of complexity $\mathcal{O}(n^3)$. Another reason for not using these more time consuming searches, is that the large local search (and, hence, 2-opt) is already *part of* the GA, so the result of this search will be compared with that of the quick heuristic as soon as a GA generation is finished.

The implemented heuristic was presented in Section 2.4.7, but as it was intended for a TSP, some changes had to be made to adjust it to the problem faced in the Eurobot competition. The heuristic has three parts corresponding to three typical events resulting from interference by the opposing robot. The three parts are described below.

Insert: This happens whenever the opposing robot has removed an object from the table, which it then re-inserts into the match, for instance by

dropping it, or when we *think* the opponent has removed an object (due to our limited field of vision), but it turns out to still be there. Unlike the TSP, the Eurobot problem has no demand of visiting all nodes, so it must be determined *if* the object should actually be picked, and if so, *when* to pick it. This is done by finding the object in the robot's plan closest to the inserted object, and then inserting the new object before or after this closest point in the plan, whichever yields the best fitness. Thereafter, the object is inserted *outside* the objects reached in the 90 seconds, to see if *not* picking it yields a higher fitness than actually picking it (for instance because picking it makes the robot miss its final delivery). The three cases considered are illustrated in Figure 3.6.

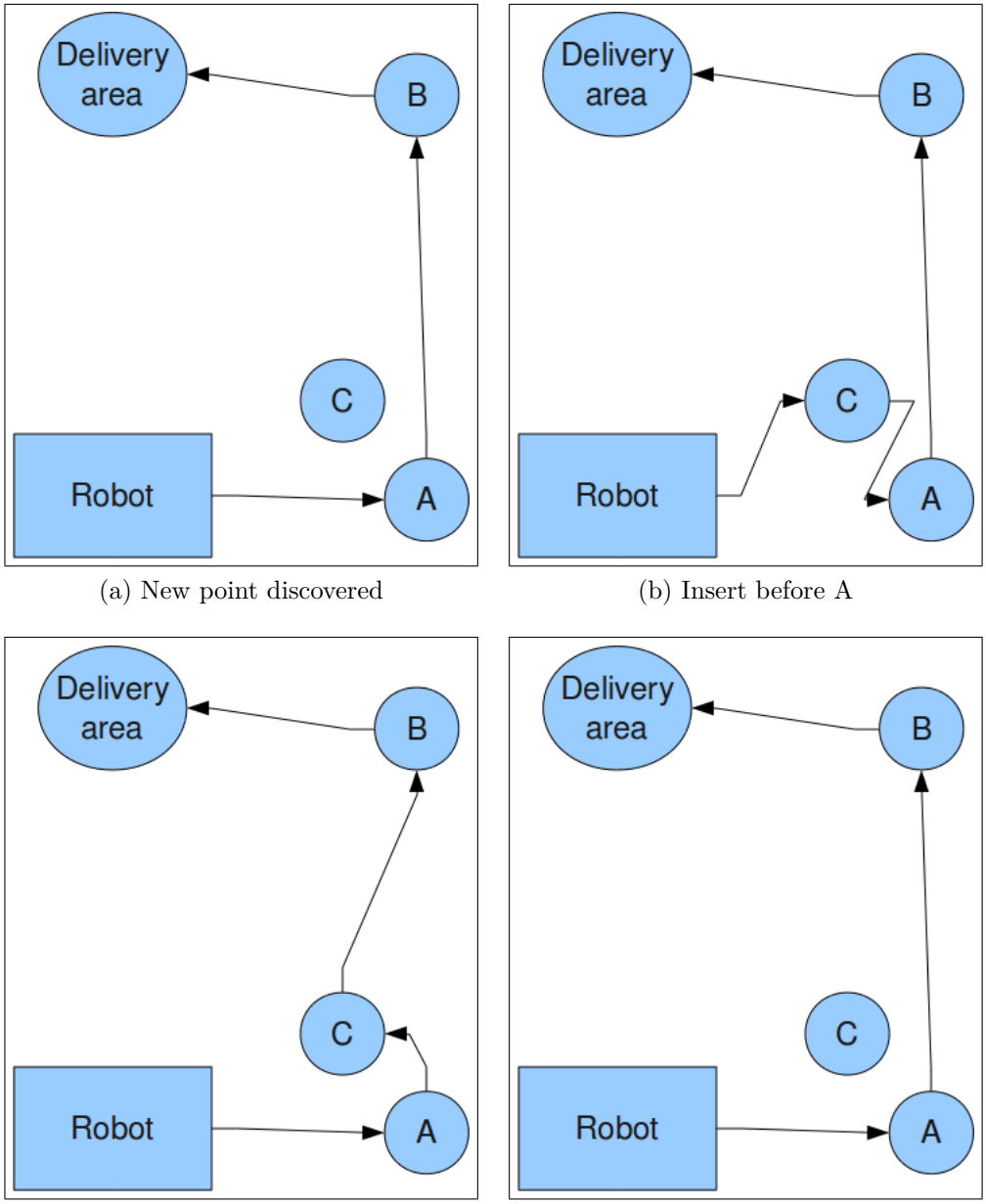
Delete: The delete action is performed when we think an object has been removed from the playing area. This action simply removes the object from the current plan, and connects its neighboring objects. If the object is not part of the plan, it is removed from the objects *outside* the plan, so it is not considered for insertion into the plan any more. The two cases are illustrated in Figure 3.7

Move: This happens when an object is moved from one spot on the playing area to another, for instance because the opponent (or our own robot) pushes it. When this happens, the delete heuristic is first called, and then the insert heuristic is called. This means that the only difference between removing and inserting an object, and simply moving it, is the time elapsed between the delete and the insert. This makes sense, as it will not always be known if the opponent picks up an object, or simply pushes it out of the way, because the robot's cameras may be pointed in the opposite direction.

As demonstrated in Section 2.4.7, this heuristic is likely to generate good tours, but can also sometimes generate sub-optimal tours. This problem is alleviated by letting the GA replace the heuristic solution as soon as it generates one that is better in the current environmental setting.

3.5 Changing the Phenotype

When moving from static to dynamic planning, an issue with the phenotypic representation of solutions became apparent. Earlier, the phenotype and



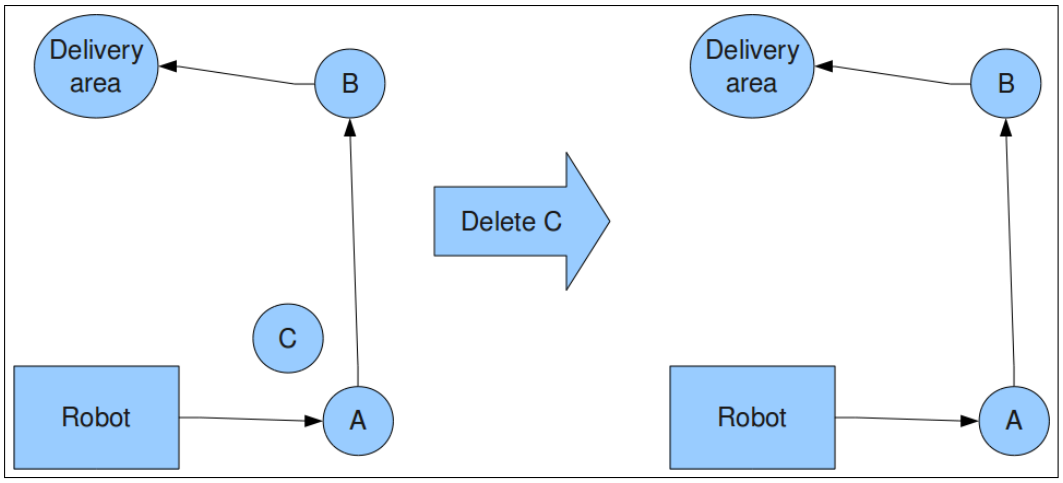
(a) New point discovered

(b) Insert before A

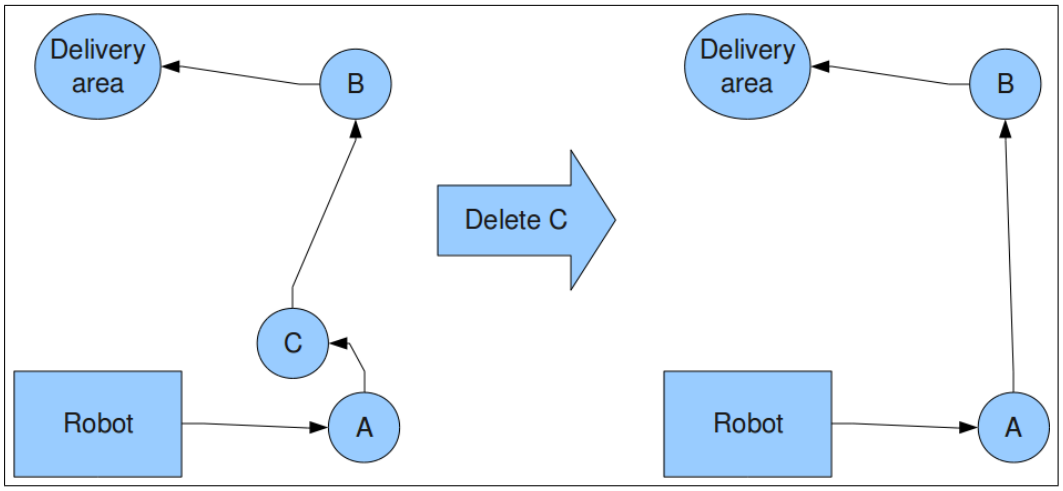
(c) Insert after A

(d) Insert outside of plan

Figure 3.6: Inserting point C in existing plan

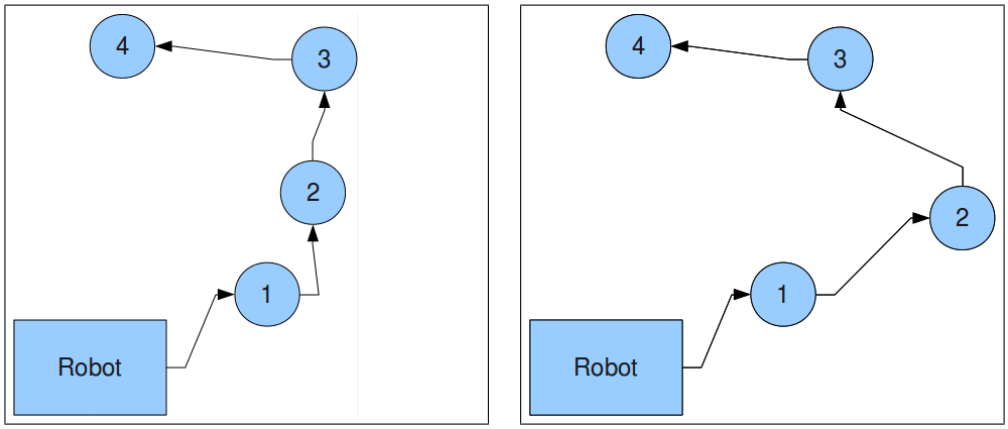


(a) Deleting a point outside of the plan



(b) Deleting a point in the plan

Figure 3.7: Deleting point C from existing plan



(a) Phenotype before change to point 2 (b) Phenotype after change to point 2

Figure 3.8: Phenotypes for two *identical* genotypes

the genotype were exactly the same: A permutation of integers representing the order of visiting all points on the playing table. This is fine for static planning, as the integers always map to the same point on the table. In the dynamic planning scenario, however, playing objects may be moved around on the table, meaning that one integer in the genotype can correspond to different points on the table, as the match progresses.

To handle this changing mapping between genotypic indexes and table points, it was decided to change the phenotype into a permutation of *point objects*, and store the current position of each point object in a separate class. This had the effect of simplifying the match simulation, as it could work directly on *points*, with coordinates and associated playing objects, at the cost of making the genotype - phenotype mapping a bit more complex.

To see why this mapping is useful for dealing with changes to the playing table, and especially *small* changes, like the ones likely to occur when a robot "bumps into" an object, consider Figure 3.8. Figure 3.8a shows the initial situation: The genotype of this plan, is [1, 2, 3, 4], and the phenotype is the visiting order of the points shown in the figure. In Figure 3.8b, a small change has been made to the table setup: Point 2 has moved to the right. Now, when developing the *same* genotype as before, the phenotype becomes the order of points shown in the figure, because the genotype - phenotype mapping takes this new table setup into consideration. This way, small changes are dealt with without even changing the genotype. When bigger changes occur, they are handled either by the fast-response heuristic, or by the GA itself, both making appropriate changes to the genotype.

3.5.1 What about the genotype?

When considering this new phenotype representation, a new question arises: What should happen when points are removed from or added to the table setup? Two options are available: Handling these changes in the genotype, or handling them in the genotype - phenotype mapping. In the first case, the genotype would contain the same number of points as the phenotype, and integers would be added to or removed from the genotype as points are added to or removed from the table. In the second case, the genotype size always remains the same, but the phenotype size changes as points are removed or added: When, for instance, a point is removed from the table, this is simply not added to the phenotype when its corresponding integer is met in the genotype.

The choice fell on the first option, changing the genotype, for the sake of performance. The number of elements in the genotype greatly affects the runtime of the GA, due to the large local search around the current genotype, and as several points are likely to disappear from the table during the match, being able to remove these from the genotype makes the GA a lot faster towards the end of a match.

3.6 The Opposing Robot

Another new factor that needs to be dealt with when working on dynamic planning, is that of an opponent that moves around and makes changes to the playing table. The actions of the opponent has four main implications that will affect the strategy choices of our robot.

1. Objects can disappear from the playing table.
2. Objects can re-appear on the playing table.
3. Objects can move from one spot to another.
4. The opponent can block our path.

The first three groups of table changes are handled by the new genotype - phenotype mapping, the fast-response heuristic and the actual GA, by giving

fitness values based on simulations updated with the *current* table setup. The last type of event is very important because colliding with the opponent gives penalty point subtraction in the Eurobot competition, and may even lead to disqualification. Therefore, enemy avoidance works on two levels in the planning system: Firstly, plans operating in a safe distance from the enemy are preferred over those that do not, and secondly, if the two robots should get *too close* to each other, anti-collision and avoidance is activated, overriding the plan made by the GA.

3.6.1 Long-term avoidance

Long-term avoidance is concerned with generating plans that are not likely to operate in the vicinity of the opponent. As the GA already does the main part of planning, it was decided to also leave this task to the GA. To make the GA able to make plans where enemy avoidance is considered, the game simulator had to be extended with information on the enemy's position, and how the enemy's presence is likely to affect the fitness of a solution. This section describes how this was implemented.

Avoidance while driving

When driving between two points on the playing table, the *expected time* taken to drive between these points is greatly affected by the enemy's position. In the worst case, if the enemy is blocking the path between the two points, our robot may have to stop, and perhaps turn around, to avoid a collision. How time consuming such an action will be, is of course dependent on the physical properties of our robot and on how the opponent reacts to our presence. The physical properties of our own robot will be known before the competition, but the actions of the opponent will never be known, so it will not be possible to calculate the time spent on an enemy encounter with a 100% accuracy. Miscalculation of this will lead to a shorter or longer time remaining than estimated, but the dynamic planning system will hopefully be able to quickly generate close to optimal plans with the new time constraints.

It was decided to estimate the time spent on driving between two points as a function of the shortest distance from the enemy and the number of seconds between the last enemy observation and the beginning of the translation.

The function for calculating the driving time begins with the original driving time as calculated by Equation 3.2, and adds a penalty for being close to the enemy in the near future:

$$Penalty = (DrivingDistLimit - MinEnemyDist) * f_{edp} - f_{epd} * (t_{trans} - t_{obs}) \tag{3.4}$$

The penalty can be positive or negative, but if it turns out to be negative, it is discarded; the fastest traversal of a distance occurs when *not considering* the enemy. The $DrivingDistLimit$ is an adjustable parameter that specifies how close the enemy can be without changing the expected driving time at all. $MinEnemyDist$ is the actual closest distance to the enemy's last observed position during the translation considered. If this distance is bigger than the $DrivingDistLimit$, the first part of the equation will be negative, and as the second part of the equation is always negative, no penalty is incurred. f_{edp} is the *enemy driving penalty factor* – in other words a factor scaling the penalty given for driving close to the enemy. In summary, this first part of the equation gives a penalty for a translation inversely proportionate to the smallest distance to the enemy's last observed position during the translation, as long as this distance is lower than a certain threshold – if not, no penalty is given. Figure 3.9 shows the variables $DrivingDistLimit$ and $MinEnemyDist$ for a translation from A to B close to the enemy.

The second half of the equation *decreases* the penalty of an enemy encounter based on *how far* into the future the translation is estimated to happen. t_{trans} is the starting time of the considered translation, t_{obs} is the time of the last enemy observation (this is mostly within a second of the current match time), and f_{epd}, the enemy penalty decrease factor, is a factor scaling the penalty reduction. The logic behind decreasing the penalty of future enemy encounters, is simply that the enemy is likely to move around, so encounters far into the future should not be penalized a lot. A separate parameter, $enemyLookAheadSeconds$, sets an upper bound for how far into the future translations should be penalized – so when working with translations very far into the future, the enemy is not considered at all.

Avoidance while picking

Since picking objects close to the enemy entails a high degree of uncertainty and danger, an additional penalty is added for making such pickups. This

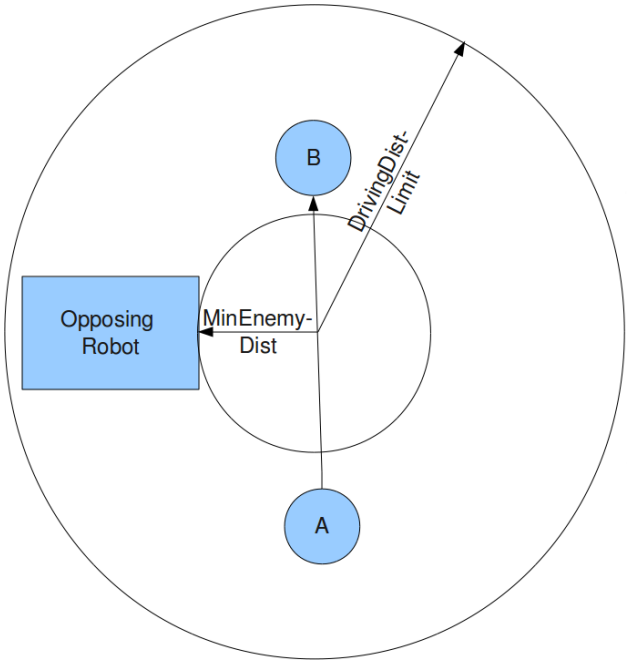


Figure 3.9: Variables involved in penalty calculation when driving close to the enemy

uncertainty has several reasons: The enemy may pick the object before us, he may block access to the object, or he may push it away from its current location. All these factors add to the time estimated to pick the object up, and they cannot be considered as part of the driving penalty, as that penalty can be thought of as the penalty for choosing an *alternate route* to the object, and is not really concerned with what happens once the object has been reached.

The equation for this penalty is

$$Penalty = (PickingDistLimit - PickingDist) * f_{epp} - f_{epd} * (t_{pickup} - t_{obs}) \tag{3.5}$$

This equation is the same as Equation 3.4, but with different factors for scaling the penalty, so the pickup penalty can be scaled independently of the driving penalty. $PickingDistLimit$ is equivalent to $DrivingDistLimit$, but can be set to a different value, if for instance a larger safety distance is required for picking than for driving. $PickingDist$ is the enemy's distance from the pickup position. f_{epp} is the *enemy pickup penalty factor*, which scales the penalty - distance relationship for picking.

The second part of the equation is the same as above, except the time of translation has been exchanged with the time of the pickup. This part, concerning time, uses *the same* time limit ($enemyLookAheadSeconds$) and scaling factor (f_{epd}) as the previous equation, because the expected change in the enemy's position over time will be the same in both cases.

For a single translation followed by a pickup, both penalties are calculated and added to the expected elapsed time. This means that translations in the close future that pass by an enemy that is also blocking the access to the playing element, will be penalized the hardest. This makes sense, as plans including such translations are very risky.

3.7 Stability

A good planning algorithm needs to find a balance between being able to adapt to changes and being stable – in other words, not changing its mind too often. Too rapid redecisions may occur in this case, because the plans made are so highly optimized, that a small discrepancy between the estimated time to perform a task and the actual time taken during the match may cause the final delivery to happen after the 90 second limit. In this case, no points are given for the last picked elements, and the robot has to make a new plan including fewer elements, discarding the old one. Making new plans is a good thing if circumstances demand changes, but if it happens too often, the resulting plans may be poor, because the GA is given *too little time* to optimize the active plan.

3.7.1 Penalizing late deliveries

This technique aims to make the difference in fitness value smaller for plans *making* the last delivery and plans *not making* it. Figure 3.10 shows the idea: The relationship between phenotypes and fitness values has a sudden steep decrease for phenotypes not being able to make their final delivery within the 90 seconds. This is clearly realistic, as plans not delivering within 90 seconds will not score points for the objects remaining within the robot at the end of the match. However, there are a couple of reasons why a less realistic but smoother phenotype-fitness mapping, like the one shown in Figure 3.10b may be preferable.

(a) Fitness values without penalties (b) Fitness values with penalties

Figure 3.10: Smoothing the fitness landscape

Firstly, GAs in general perform better on smooth fitness landscapes, as they are easier to search through. Sudden peaks will make it hard to search, because similar solutions can have very different fitness values, making attempts to reach optima in the landscape through local improvements difficult. Secondly, smoothing the fitness landscape in this problem has the effect of decreasing the utility of making over-optimized tours, delivering at the very last moment. An experiment in the author's final year project, showed that most solutions generated by the static planner were of this kind, making final deliveries between the 89^{th} and 90^{th} second. As the match simulator is not 100 % accurate, and as unforeseen events may occur during the match, these types of over-optimizations are likely to lead to tours that will be discarded during the match. While it is good to discard infeasible tours, over-optimization will lead to tours being discarded way too often, meaning more resources will be spent on making new tours, than on further optimizing the current solution.

Alternatively, when giving the GA a reason to finish its deliveries earlier, as in Figure 3.10b, some degree of inaccuracies in the simulator, and some unforeseen events may be dealt with *without* discarding the current plan, thereby giving the GA more time to further optimize this plan.

The way this penalty was implemented, was by subtracting a given value from the fitness of solutions making their last delivery after 85 seconds. The value subtracted is inversely proportionate to the remaining time before the

full 90 seconds are exceeded. The penalty is calculated as:

$$Penalty = (t_{delivery} - 85.0) * f_{ldp} \tag{3.6}$$

$t_{delivery}$ is the time of the final delivery, so the first part of the equation will range from 0 to 5 (deliveries made *after* the 90 seconds are given no points at all). f_{ldp} is the *late delivery penalty factor*, which was adjusted to give penalties for over-optimizing tours, but still letting late deliveries happen, if the benefits of an additional pickup outweigh the risk of making a late delivery.

Notice how this somewhat resembles the *adaptive penalty function* introduced in Section 2.4.5, which also resulted in less abrupt changes in the fitness landscape for solutions around the feasibility limit. The difference is that the adaptive penalty function reduced the fitness of solutions that were actually *infeasible*, but still looked promising, to guide the evolutionary process. The approach taken here, instead reduces the fitness of solutions that are feasible but unsafe, and gives *no fitness* to infeasible deliveries. The reason for doing this, is that giving positive scores for solutions delivering after the 90 seconds may make solutions risky, leading to a higher frequency of plans not making their final delivery.

3.8 Contingency Planning

Because all playing elements in this year's Eurobot competition are common to both competing robots, there is a high probability that the opposing robot may pick up or block access to elements that are in our plan. When this happens, the heuristic discussed in Section 3.4 will make a quick modification to the plan, discarding the missing object. However, the new plan is not likely to be optimal. Therefore, being able to quickly generate a new, close to optimal solution through the GA in such cases will be beneficial. Preferably, the solution would be ready even *before* the opponent picks or blocks the object, and enabling this to happen, is the goal of *contingency planning*.

Planning for contingencies has been extensively researched in classical AI systems. [7] argues that for robots controlling dynamic and unpredictable processes, *reactivity* is an important ability. Reactivity in this context means "the ability to act appropriately in a broad range of situations without deliberation" [7]. The author goes on to argue that such reactivity can be achieved

by planning for contingencies, but that limiting the range of expected contingencies is also important, to reduce computational complexity. Therefore, it is important to efficiently divide the processing capabilities of the robot between reacting to events and planning for contingencies. A heuristic suggested in the book, is focusing on events and actions in the *near* future in the contingency planning.

Combinations of genetic algorithms and contingency planning, however, have not been researched a lot, because GAs already generate a lot of solutions, meaning contingency plans may already have been created. Additionally, GAs focus on *reacting* to environmental changes as they occur, instead of planning ahead for them. Even so, it was chosen to investigate a potential combination of GAs and traditional contingency planning, for two main reasons. Firstly, because the reactivity in the GA may not always be sufficient in a Eurobot match, and secondly because some events to plan ahead for were readily available, as there is a quite limited degree of environmental change in this task.

The idea of performing contingency planning was born from the robot's increased processing capabilities this year. A new computer with a quad-core processor was installed in this year's Eurobot robot, meaning that while one CPU core runs the regular GA, based on the playing table's current condition, another can perform contingency planning, making new plans based on *possible future events*. If the future events that are planned for are carefully selected, "unpleasant surprises" may be handled very efficiently. The choice of future events was based on their probability of occurring, and on how severe their consequences would be for our strategy. Two kinds of future events were considered, and they are discussed in the following sections. They will both be tested, to see which type of contingency plan gives the highest average solution fitness when competing against another robot.

Contingency planning was implemented by extending the GA to multi threaded use, and starting two separate evolutionary threads, each working on slightly different problems – one on the original problem, and the other on the same problem, but missing *one* playing element. If the thread working on the contingency plan in any generation provides a solution of a higher fitness than the regular thread, their roles are switched: The contingency planner becomes the main strategy generator and vice versa. However, the status of an evolutionary thread is never changed *within* a single generation. This also applies to the event that is planned for: It is only updated at the start of a new generation, to make the entire generation work on the same problem.

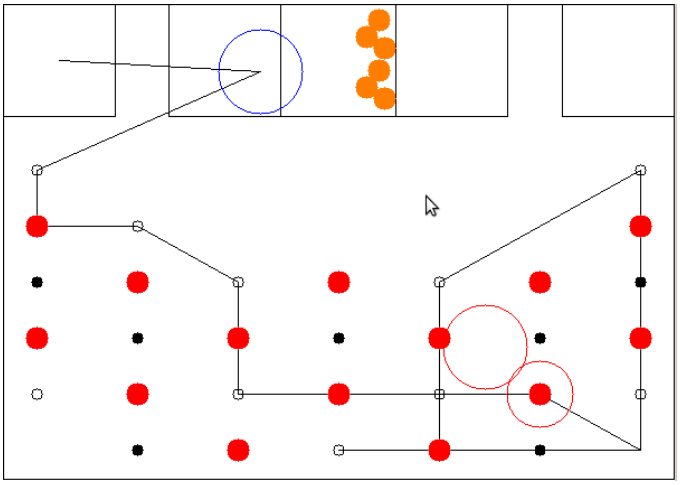


Figure 3.11: Contingency planning based on enemy position

3.8.1 Contingency based on the opposing robot

The first kind of contingency plan, is based on identifying what object on the table is *most likely* to disappear or be blocked, and make a plan without this object. So, if this object can no longer be part of the robot's plan, an alternative plan has already been made. The object considered to be most likely to disappear is simply the object closest to the opposing robot. However, only objects planned to visit in the relatively close future are considered, as the enemy is likely to move around on the table. A simple limit of visiting the hill or making a delivery was set as the horizon of contingency planning, as these processes are thought to be quite time consuming, and by the time such a process is done, the enemy is likely to have moved. If no object which has a planned visit in the close future is close enough to the enemy, both evolutionary threads are used for regular planning until such an object exists.

Figure 3.11 shows a plot of the playing table demonstrating how this works. The plot shows the playing table from above, with our robot as a big blue circle, and the enemy as a big red circle. The solid line shows our plan for the entire match, picking objects and making two deliveries. The smaller red circle indicates the object that the contingency plan would be based on removing if a new evolutionary generation were to begin. Obviously, the enemy is *closer* to another point in the plan, but as picking this object will happen so far into the future, a contingency plan is rather made for the object below the enemy to the right, which is also part of our plan, at an

earlier time.

3.8.2 Contingency based on planning horizon

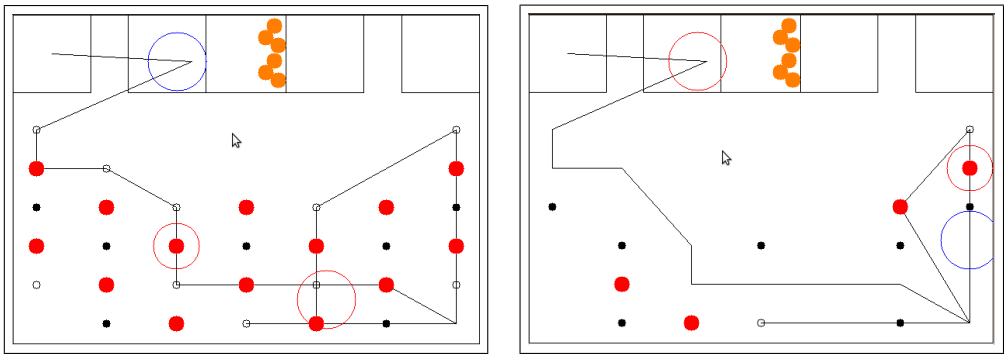
The object that would cause the most disruptive change to our plan if removed by the opponent, is the *next* object in our plan. If this object is removed, having prepared a plan for such an occasion will be very valuable, as very short time is available for replanning our next move. However, making a contingency plan for the next move will not always be reasonable, due to the computing time of the GA. In particular, at the beginning of a match, the contingency planner will often not be able to finish a generation of planning before reaching the next object. Therefore, a contingency planner deciding which object to plan for the removal of based on the time used on a GA generation was implemented.

This way of planning always measures the time the evolutionary threads used on finishing the previous generation, and uses this time together with the match simulator as an indication of which object will be reached when the contingency planner has finished its *next* generation. For instance, if the last evolutionary thread took ten seconds to complete a generation, the first object reached ten seconds from now will be the basis of the contingency plan. This will be a *pessimistic* estimate, as the GA runs faster and faster as the match progresses, due to playing elements being removed, and shorter simulation time remaining. Towards the end of the match, the contingency planner will mainly work on the next object in the robot's plan, as GA generations at this time take less than a second to complete.

Figure 3.12 shows this in action: Early in the match, the planning horizon is long, and an action quite far into the future is selected for contingency planning. Towards the end of the match, however, the object considered is the next object to be reached.

3.9 Changes to the Simulator

Quite a few changes had to be made to the previously implemented simulator, both to enable it to run in a dynamic environment (in other words, being able to start *in the middle* of a match), and to make sure it accurately reflects the constraints that the actual robot performs under. As the main focus of



(a) Early in the match (b) Late in the match

Figure 3.12: Contingency planning based on planning horizon

this report is on the genetic algorithm itself, going into the details of all these changes and adjustments is out of the scope of this chapter, but the most interesting ones will be briefly discussed here.

3.9.1 Avoiding the narrow spot

As the size of the robot was being determined, the team discovered that half of the playing table setups would have a "narrow spot", which the robot would be too wide to pass through. Figure 3.13 shows one such table setup. The narrow spot is in the middle of the table, at the bottom. The distance between the two lowest black corns and the wall is too small for the robot to pass by. Firstly, this means that if the robot makes a plan where it is supposed to follow the lower part of the table, it has to make a detour, going via the upper part. Secondly, it means that the two tomatoes under these black corns will not be pickable in their initial position.

When dealing with this problem, it was first assumed to be something to address in the simulator. However, thanks to the new phenotype-representation, discussed in Section 3.5, it was easier to address the issue in the genotype-phenotype mapping. To indicate that the two lowest tomatoes were blocked in certain table setups, the coordinates of all points in the genotype were simply checked for these setups. If the point was right below one of the two narrow spots, it was not inserted into the phenotype.

To penalize driving *through* the narrow spots, the genotype was checked for paths crossing one or both spots, by examining positions of consecutive

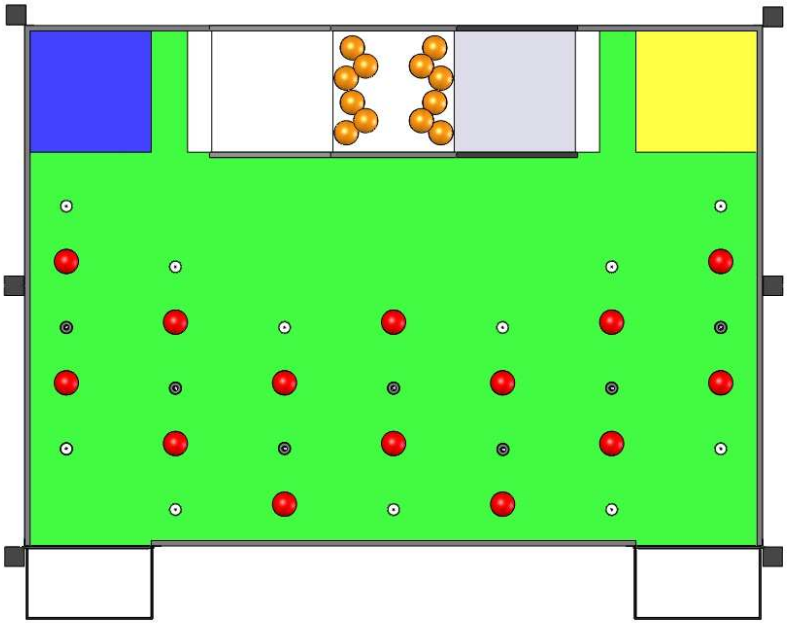


Figure 3.13: Game setup with a narrow spot

genotype-items. If such a path is found, an additional *viapoint* is inserted into the phenotype. So, when simulating a plan, the penalty for driving via this point is taken into consideration. The viapoint is inserted in the row above the two blocking corns, it's horizontal position determined by which tight spot is being passed: If only the left spot is passed, the viapoint is placed right above the left narrow spot, if both are passed, the point is placed horizontally in the middle of the table, and if only the right spot is passed, it is placed above the right narrow spot. Figure 3.14 shows how this works. The dotted arrows show the robot's initial plan, driving through the narrow spot, while the solid arrow shows how the developed phenotype will look.

When the GA knows of the big penalty for driving along the bottom in half the table setups, it will respond by avoiding this area altogether, if there are enough points to be gathered in other places. Figure 3.15 shows two examples of this behavior.

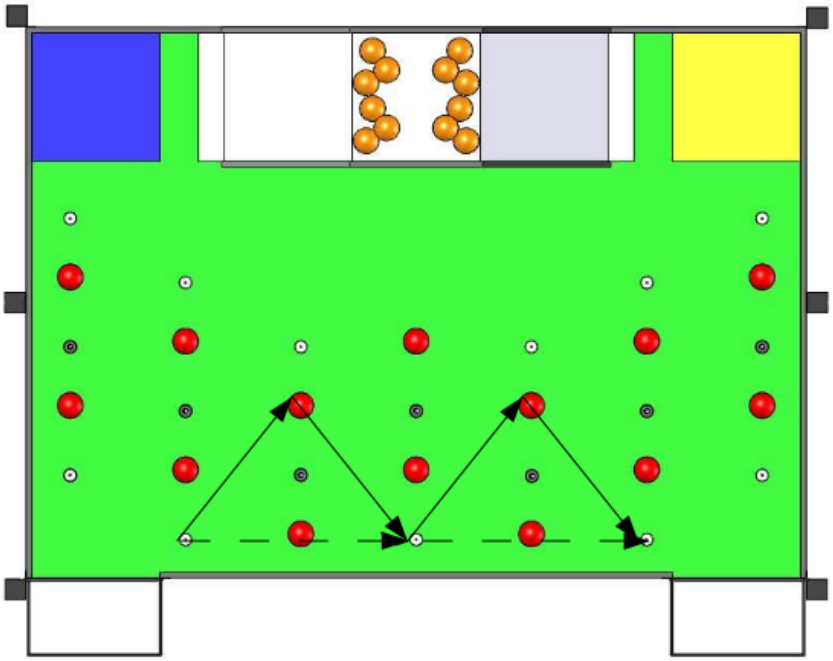


Figure 3.14: Navigating around the narrow spot

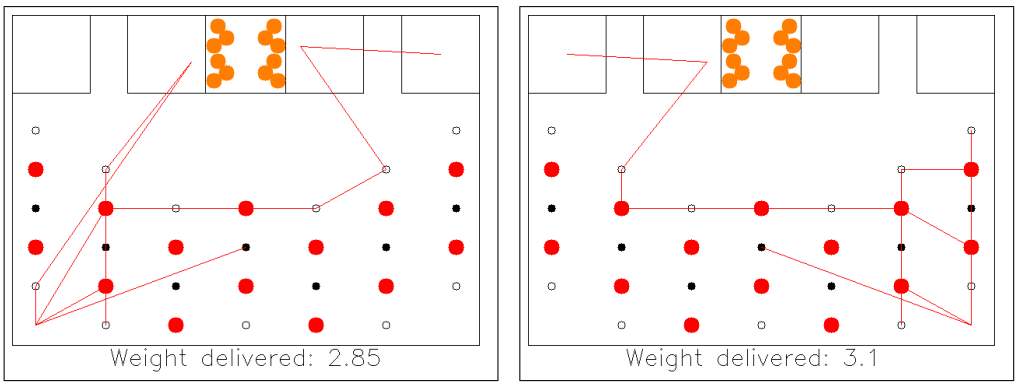


Figure 3.15: Avoiding the narrow spots

3.9.2 Decreasing the genotype size

As the runtime of the local search has a complexity of $\mathcal{O}(n^2)$, where n is the genotype size, limiting the size of the genotype will be very important to enable the robot to efficiently replan in a changing environment. Therefore, all actions that need to be performed in a specific order should be eliminated from the genotype.

As the robot design was completed, it was decided to have an orange capacity of three, and to pick all these three oranges simultaneously with a "fork" designed specifically for this purpose. Due to the positions of the oranges, this design also meant that only three specific oranges could be reached on each side of the hill, so once these three are picked, that side can be considered as "empty".

What this meant for the GA was that all oranges but one on each side of the hill could be removed from the genotype, as the *order* of picking them is insignificant. This allowed the genotype to decrease from 40 elements to 30, yielding significantly faster local searches.

3.9.3 Dynamic simulations

When running the GA in a Eurobot match, it is clearly beneficial to only simulate the *rest* of the match, instead of simulating from the beginning every time. Simulating from the beginning will both yield less accurate estimates of the current state than those received from the robot's sensors, and it will be more time consuming. Therefore, the simulator was extended to being able to start from any given point in the match, and simulating the rest. Due to the fact that more than one evolutionary thread will be using the simulator, a separate class, *CurrentState*, was created for keeping the *states* of the match and the robot, and data from this class are loaded into the simulator every time it is run. The CurrentState class will be getting its data from the various sensors of the robot during a Eurobot match. For simulated runs on the computer, its data are generated from the simulated results of executing the robot's plan. This state is also used for dynamic plotting of the robot's position and the playing table's configuration throughout the match when simulating matches, as discussed in the next section.

3.10 Real-Time Match Plotting

One of the requirements stated in the project goal was being able to show a real-time match simulation, complete with playing objects, robots and the current plan. This was important, to enable testing of the planning system without the actual robot.

As a simulator had already been implemented as part of the fitness calculation, it was decided to use data from this simulator when plotting the state of a match. This way, what is plotted is a match going *exactly* as the fitness evaluator expects it to, which is of course not very realistic. For this reason, two ways of making the match less predictable were implemented: A simulated enemy, and random errors to time estimates.

The simulated enemy is an enemy robot with the same specifications (speed, capacity and so on) as our own, which runs a plan produced by the static GA. This simulated enemy will interfere with our plan, as it picks up objects and gets in our way. Whenever it interferes with our plan, the state of the game will be different than what the fitness evaluator earlier anticipated, so changes may have to be made to the active plan. Screen shots from a complete match against a simulated enemy are presented in Appendix A, and discussed in Section 4.3.1.

Random errors to time estimates were simulated by adding a uniformly distributed error within user-defined bounds (for instance, between -3 and +3 seconds) to *every* translation of a match. The error is only added when calculating the state of the robot during the match, and not during the actual fitness evaluation. This way, the match reaches a state that the fitness evaluator didn't expect, forcing it to consider altering its plan. Experiments on adding random errors to time estimates are presented in Section 4.3.3.

3.11 The Other Robot Modules

This section briefly describes the other modules of the robot, and how the strategy module interacts with them. This intends to let the reader get a better idea of where this work fits in to the 2010 Eurobot project at NTNU. Figure 3.16 shows all the other robot modules that the strategy system needs to interact with. These interactions are briefly explained below.

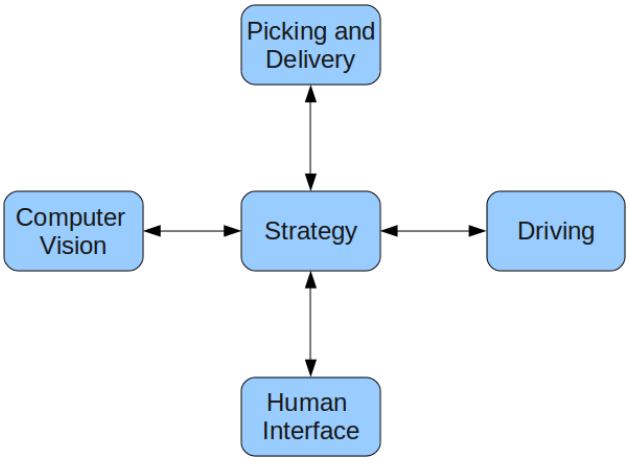


Figure 3.16: The robot modules

3.11.1 The driving system

The driving system regulates the speed of the robot's engines to ensure it gets to all the points it needs to visit. It is also responsible for aggregating positioning estimates from both our own and the opponent's robot. These estimates are sent to the strategy system, as they are important for determining an optimal strategy for the rest of the match.

Communication with the driving system happens by sending waypoints that the robot needs to visit, and stop commands if the robot needs to stop immediately at its current position. As the strategy system only determines *what* points to visit, and not *how* to get there, an additional waypoint generator was implemented, that translates the robot's strategy into a series of waypoints that allows the robot to safely navigate around the table without colliding with walls or obstacles. As the main focus of this master's thesis is the strategy system, the waypoint system was kept quite simple, using geometrical equations for calculating safe ways to get around fake corns, pick up objects and so on.

Figure 3.17 shows an example of how this works. Following a straight line takes the robot dangerously close to fake corns, but the waypoints (the numbered circles) ensure that the robot avoids collisions. When the driving system gets close enough to a waypoint, it sends a "waypoint reached" message back to the strategy system.

In addition to sending waypoints to the driving system, sometimes the strat-

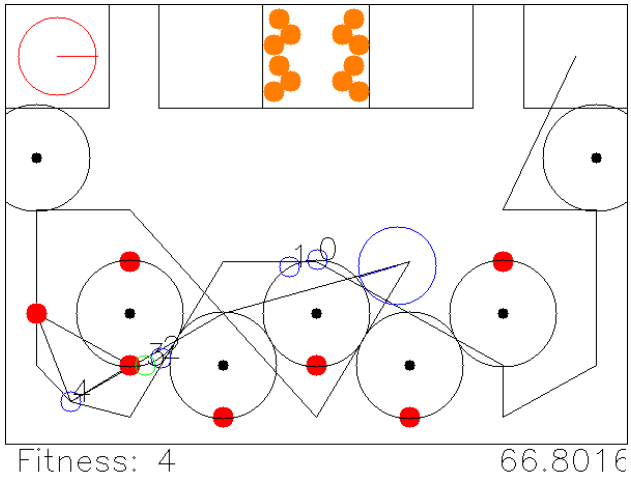


Figure 3.17: Translating the strategy into waypoints

egy system needs to send a stop command. This may happen for two reasons: Firstly, after 90 seconds, the robot needs to stop driving, or it will receive a penalty in the Eurobot competition. Secondly, if the opponent is a short distance ahead of the robot, it needs to stop to prevent a collision, or it may face penalty points or even disqualification from the competition. Collision detection is done by checking if the opponent's current position is close to the path we are planning to take in the immediate future. If it is, a stop message is sent to the driving system.

3.11.2 The human interface

The human interface of the robot is a box placed on top of it with a few buttons and a starting cord. The buttons are used for telling the robot what side of the table it is starting at. This information is received by the strategy system before the match begins, and is of course important for determining where the robot is starting and, hence, how it will navigate around on the table.

A starting cord is a requirement in the Eurobot competition. When this cord is pulled, a message is sent to the strategy system, which causes it to start its main loop, and also to send initialization messages to all other systems.

3.11.3 Computer vision

The computer vision is responsible for updating the robot's view of the playing elements at all times, and for finding the initial configuration of the black corns. Both pieces of information are essential for determining a good strategy for the rest of the match.

As soon as the start cord of the robot is pulled, a request is sent to the computer vision system to receive the initial corn configuration. When the answer is received, the stored plan for this configuration is loaded, and waypoints for the first few steps are generated and sent to the driving system.

During the entire match, updated information about what playing elements are present on the table, and where they are is sent from the computer vision system to the strategy system. Whenever a significant change is made, the internal game state in the strategy system is updated, and the heuristics discussed in Section 3.4 are applied to the new state. Also, the always-running GA uses this new state for all fitness calculations.

3.11.4 The picking- and delivery system

Whenever an object is to be picked, or whenever a delivery should be made, the picking- and delivery system needs to be informed. The way this happens, is that when the next waypoint is either a pickup- or delivery point, and a waypoint reached message is received from the driving system, the strategy system immediately sends the appropriate pickup or delivery message to the picking- and delivery system. For pickups, this message depends on the object present at the pickup location, whereas for deliveries the message will depend on what elements the robot is currently *carrying*, to avoid wasting time on performing unnecessary delivery actions.

In addition, initialization- and stop-messages are sent to this system, the latter important because robots with actuators moving after 90 seconds cannot be qualified for the Eurobot-competition.

Acknowledgements of finished pickups or deliveries are sent back from this system, to allow the strategy system to continue performing its plan.

Chapter 4

Results and Discussion

4.1 Results From the Static GA

The static GA that the work in this report is based on was thoroughly tested as part of the author's specialization project, to see if it could solve the task faced in the Eurobot competition to a satisfying degree. The results from these tests are summarized here.

The first experiment aimed to find out whether the GA was able to generate good plans for all the 36 possible table setups in the competition. The algorithm was therefore run with the parameters in Tables 4.1 and 4.2 on all 36 setups, generating plots of its plans. The results were good – all the plans looked reasonable, with few crossing lines, and focus on picking objects giving many points and taking little time to pick. For a few of the setups, however, manual inspection showed that at least slightly more optimal plans existed. Furthermore, it was found that plans found for one table setup, could also be very efficient on other setups, as some of them are quite similar. These observations led to the second experiment: GA runs based on a *seeded* population.

Whereas all plans generated in the first experiment had started from randomly generated plans, the second experiment aimed to see if even better plans could be generated when initiating all searches with some random individuals, and some seeded individuals, representing the best solutions generated in the first experiment. The results were good: The GA was able to find

Table 4.1: The GA parameters

Generations	25
Population size	16
Selection protocol	Generational mixing
Initial Boltzmann temperature	10°
Boltzmann-decrease per generation	0.36°
Number of elites	3
Tournament contestants	4
Tournament factor	0.1
Crossover probability	0.5
Mutation probability	0
Large local search probability	0.4
Extra local searches	5
Max number of deliveries	3

Table 4.2: The simulator parameters

Early delivery factor (f_{ed})	0.99
Early finish factor (f_{ef})	0.01
Carrying objects factor (f_{co})	0.1
Goal proximity factor (f_{gp})	$\frac{1}{4000}$
Obstacle radius (robot radius + corn radius)	15 cm
Translation speed (V_t)	15 cm/sec
Rotation speed (V_r)	120°/sec
Acceleration and breaking penalty (P_a)	1.5 sec
Obstacle penalty	1 sec
Hill penalty	5 sec
Ball (oranges + tomatoes) capacity	4
Corn capacity	4
Can pick oranges	True
Can pick tomatoes	True
Can pick corns	True
Orange weight	0.3
Tomato weight	0.15
Corn weight	0.25
Orange penalty	1 sec
Tomato penalty	0.5 sec
Corn penalty	0.5 sec
Delivery time	2 sec

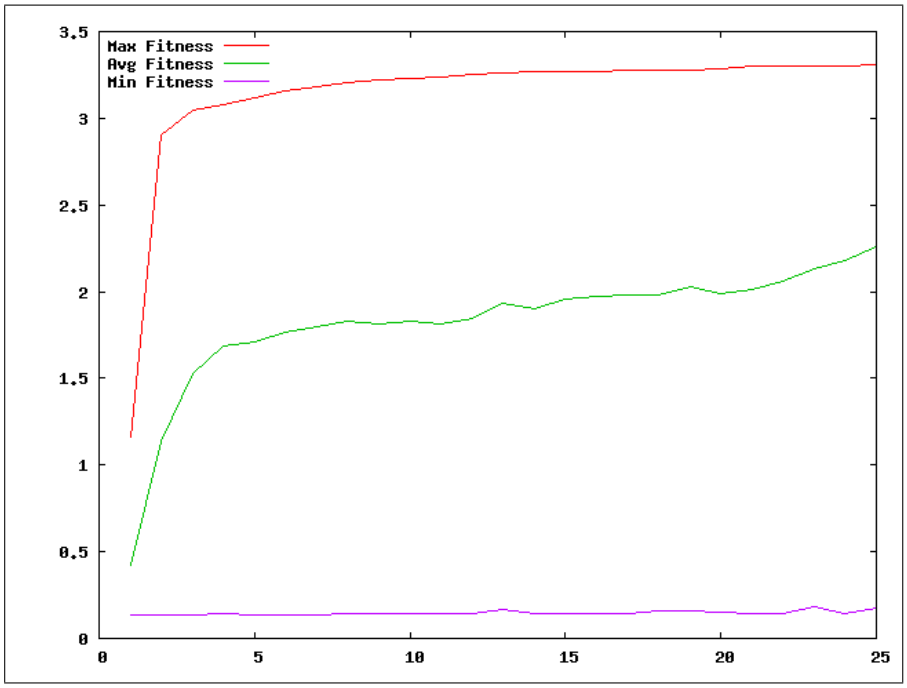
better solutions than in the first experiment in *eight* of the 36 configurations, and none of the configurations resulted in worse plans than before. Further studies on the cases where the GA was *not able* to improve previous solutions proved that this was often due to solutions being so highly optimized that no improvement was possible. Many solutions made their final delivery between the 89^{th} and the 90^{th} second of the match. It was suggested that such plans may be unsafe, and that work on dynamic replanning should attempt to alleviate this problem. Work on this issue was discussed in Section 3.7.

The final experiment performed on the static GA sought to find out how important the local optimization was to the success of the GA. To test this, the hybridized GA was tested against a GA using regular mutation, that evaluated the *same amount* of individuals per generation as the local search did. A comparison of the two gave an indication as to how *intelligently* the local optimizer searches for solutions. It was found that on average, the hybridized GA tested 3850 individuals per generation. Therefore, the non-hybridized GA was run with a population of 3850. This made it perform very slowly, due to the task of performing selection and mating on this huge population. The average results from running both algorithms 50 times are given in Figure 4.1. They show that the hybridized GA is much better both at *finding* and *improving* solutions to the Eurobot task than the non-hybridized version, indicating that the task's complexity is simply too large to leave it up to a standard GA.

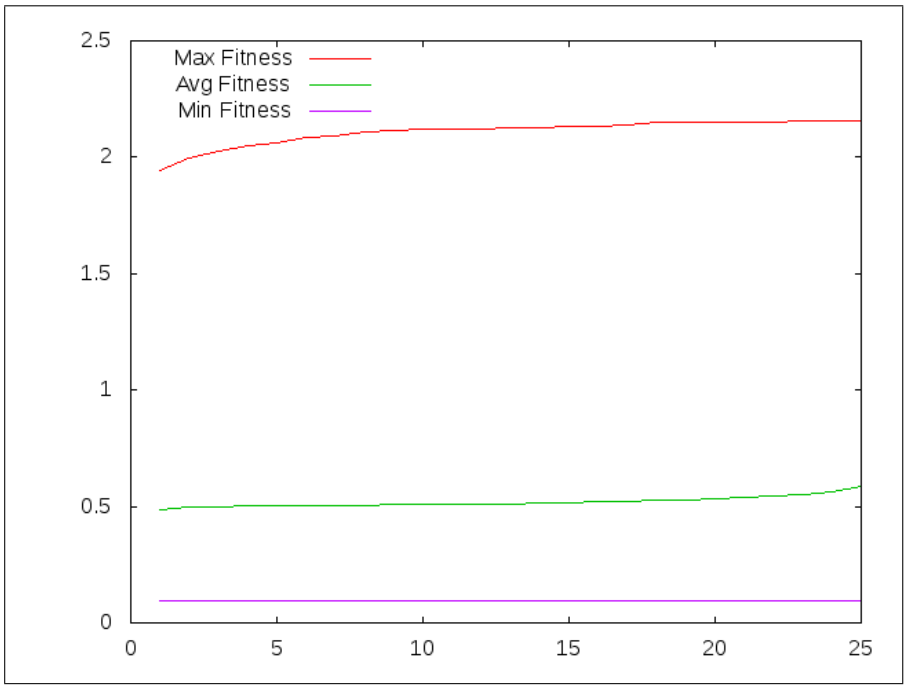
4.2 Experiments on the Dynamic GA

The experiments on the dynamic GA focused on investigating how well it would do in an actual Eurobot-match. This included both testing its responsiveness and its behavior in changing and unpredictable environments. The parameters used in these experiments are listed in tables 4.3 and 4.4. How these parameters are used in the GA and the match simulator is explained throughout Chapter 3.

All experiments were run on the author's private laptop, except for a timing experiment that was also run on the robot computer intended for use in the Eurobot competition. Hardware specifications for the computers are given below.



(a) With local optimization



(b) Without local optimization

Figure 4.1: Fitness plots for the GA run with and without local optimization

The computer on the robot has the following specifications:

- Intel Mini-ITX Socket LGA775 board (Intel DG45FC)
- Intel Core^{TM}2 Quad Q9550s (4 cores @ 2.83 GHz, 12 MB L2 Cache)
- 4 GB RAM DDR2 (2 x 2 GB)

The author's laptop (an Acer Aspire 6935G) has the following specifications:

- Intel Core^{TM}2 Duo P7450 (2 cores @ 2.13 GHz, 3 MB L2 Cache)
- 4 GB RAM DDR3

Table 4.3: The dynamic GA parameters

Population size	16
Selection protocol	Generational mixing
Initial Boltzmann temperature	10°
Boltzmann-decrease per generation	0.4°
Number of elites	4
Tournament contestants	4
Tournament factor	0.1
Crossover probability	0.5
Mutation probability	0
Large local search probability	0.4
Max number of deliveries	3
Number of evolutionary threads	2

4.2.1 Plotting a full simulated match

This experiment consists in running a single simulated match against an opponent that is executing a plan generated by the static GA, and plotting the game state with five second time intervals. As this is a single match, and a GA is a stochastic process, this experiment is not meant for drawing any general conclusions about the performance of the dynamic algorithm, but rather on highlighting some of the issues discussed in Chapter 3, by showing how they come into play in a Eurobot match. The rest of the experiments will focus on more comprehensive tests generating statistically valid results.

Table 4.4: The dynamic simulator parameters

Early delivery factor (f_{ed})	0.999
Early finish factor (f_{ef})	0.01
Carrying objects factor (f_{co})	0
Goal proximity factor (f_{gp})	0
Late delivery factor (f_{ldp})	0.05
Obstacle radius (robot radius + corn radius)	22.5 cm
Translation speed (V_t)	25 cm/sec
Rotation speed (V_r)	180°/sec
Acceleration and breaking penalty (P_a)	1 sec
Obstacle penalty	2 sec
Hill penalty	6 sec
Orange capacity	3
Tomato capacity	5
Corn capacity	3
Orange weight	0.3
Tomato weight	0.15
Corn weight	0.25
Orange picking time	3 sec
Tomato picking time	1 sec
Corn picking time	0 sec
Delivery time	3 sec
Safe enemy driving distance	50 cm
Safe enemy picking distance	50 cm
Enemy picking penalty factor (f_{epp})	0.02
Enemy driving penalty factor (f_{edp})	0.01
Enemy lookahead seconds	10.0
Enemy penalty decrease (per second) (f_{epd})	0.5

4.2.2 Contingency planning

This experiment aims to compare the two kinds of contingency planning described in Section 3.8 against each other and against no contingency planning at all. To do this, 36 matches are simulated (one for each table setup) against an enemy executing the best plan found by the static GA for that setup. Our own robot adjusts its plan dynamically throughout the match, while the enemy simply keeps executing its pre-defined plan. The reason for not having the enemy respond dynamically to environmental changes, is partly due to the computational complexity of running two dynamic GAs on one machine, and partly because to test contingency planning, it is not important that the enemy performs optimally, but rather that it in various ways interferes with our plan.

4.2.3 Simulator inaccuracies

When simulating Eurobot-matches, many simplifications are made. For instance, acceleration is modeled simply as a constant penalty, and the robot's speed is regarded as constant. These simplifications are intentional – the goal of the dynamic planner is to be flexible enough to handle discrepancies between simulated matches and the reality. To test whether the dynamic planner is flexible enough, erroneous time estimates in the fitness evaluation were simulated by adding random errors to translations in the robot's plan during a simulated match against an opponent. However, during fitness evaluation, these errors were not inserted, so the GA needed to continuously adjust its plan when these errors occurred.

Only erroneous *translation* estimates were simulated, because the time taken to pick up and deliver objects is likely to vary very little. The estimate errors were simulated by adding a random error in the interval from -3 to 3 seconds to *every* translation. The dynamic algorithm's capability of dealing with these errors was compared with the performance of the *static* GA when exposed to erroneous estimates, yielding the experimental setup shown in Table 4.5

Table 4.5: The combinations tested

Plan type	**Random error**
Static plan	0 sec
Static plan	-3 to 3 sec
Dynamic plan	0 sec
Dynamic plan	-3 to 3 sec

4.2.4 Efficiency of the GA

The project goal (see Section 1.2.2) stated that "The algorithm should be fast enough to be able to efficiently handle rapid changes to the robot's playing environment". Part of achieving this was implementing the heuristic for rapidly responding to changes. But it is also desirable that the GA itself performs very efficiently, so heuristically modified solutions are quickly optimized. Therefore, experiments were made to test the runtime of a single GA generation throughout a simulated match. This was done both on the PC used for the main part of development and testing of the algorithm, and on the robot's PC which would actually be used in the competition. On each PC, 36 simulated matches were performed, and average runtimes stored.

The reason for focusing on the time taken to complete a *generation* is simply that it is a straightforward unit of measurement for a GA, and that due to the local search embedded in the GA, each generation will typically bring a new and quite well adapted individual. Of course, later generations will optimize this individual further, so completing *several* GA generations between each large environmental change will be preferable.

4.3 Results

4.3.1 Plotting a full simulated match

The result of taking screen shots of the game simulator every five seconds in a match against a simulated enemy is given in Appendix A. The blue circle shows our robot, continuously updating its plan, while the big red circle shows the opposing robot, which runs a plan generated by the static GA that is never updated. The little red circle shows what object the contingency planner considers to be most likely to disappear next, based on the enemy's

position, as discussed in Section 3.8.1. The solid black line shows our robot's plan for the future, as well as the part of the plan that has been completed. The plan visits playing objects and makes *two* deliveries in the bottom right corner. An additional line from the delivery area towards a far-away object is plotted after the second delivery. This is the translation that the robot initiates as the match is almost done, so this last action has no impact on the fitness of a solution, and will most of the time be selected at random by the GA. The bottom left of the plots shows how much weight the current plan would deliver, if it was executed exactly as planned.

The first plot shows both robots starting in their separate corners and shows our initial plan of delivering 3.75 kg. Plots up until Figure A.1i show the opponent picking up several objects from our plan, decreasing the plan's fitness, resulting in small changes to the plan, by use of the "delete"-heuristic presented in Section 3.4.1. The plots also show the little red circle representing the focus of our contingency plan moving around as the opponent moves.

After about 45 seconds, a major plan modification happens, as the GA finds a better solution than the one generated by the heuristic. This increases the expected weight delivered after carrying out the entire plan from 2.8 to 3.05 kg. Later in the match, in Figure A.1l, the enemy is once again about to interfere with our plan. This late in the match, the runtime of the GA is only a few seconds, so the plan is quickly adapted as the enemy approaches. This enables the robot to pick another plan, delivering the same weight, as seen in Figure A.1m.

Simplifications

The match plots show some obvious simplifications in the match simulator. These simplifications and their significance are discussed below.

Firstly, Figure A.1h shows the two robots *driving through* each other. Secondly, Figure A.1l shows our robot driving over a white corn without picking it up. And, finally, Figure A.1e shows our robot driving through a black corn. The reason why collisions were not handled more realistically in the simulator, was simply that what is interesting to study, is how the *robot's plan* evolves as changes are made to the playing area, and not how the *playing area* changes as the robot interacts with it. In other words, it is interesting to see whether the robot makes plans avoiding the enemy and black corns, not

what happens if it actually collides with them. Anti-collision and obstacle-avoidance are implemented in the waypoint generator discussed in Section 3.11.1, which is a process *external* to the GA, that is responsible for making the GA-generated plan into commands sent to the robot's driving module.

Another simplification in this simulation is that the robot has a complete view of the playing table and all the playing objects. In a real Eurobot match, however, the robot will only get updated information on the playing objects in view of the robot's cameras. What this implies will be that the robot has shorter time to deliberate after observing the changed table state. This problem, however, was greatly alleviated by the fact that the algorithm runs much faster on the computer used during the competition, than the one the system was made and tested on. Section 4.3.4 discusses the runtime of the algorithm on the two computers further.

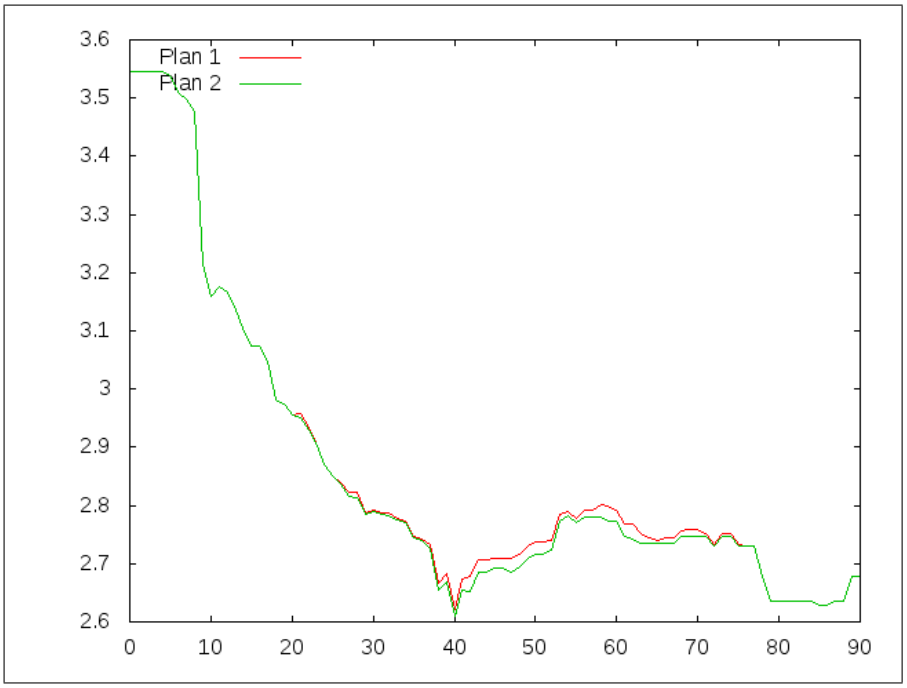
The plotting of oranges is also simplified. Even though the robot can only pick three oranges, all six disappear once it has made its pickup. This was done because there are only three *specific* oranges on each side that the robot can pick, due to the way it was designed. Therefore, the GA will not be able to gather any more points once it has visited one side of the hill, and the rest of the oranges there can be discarded.

As mentioned, this section has mainly tried to give an insight into the typical behavior of the GA and the game simulator during a match, and not provide any statistical data analyzing the performance of the algorithm. Gathering statistical data through running simulated matches with the GA on all 36 possible match setups will be the focus of the remainder of this chapter.

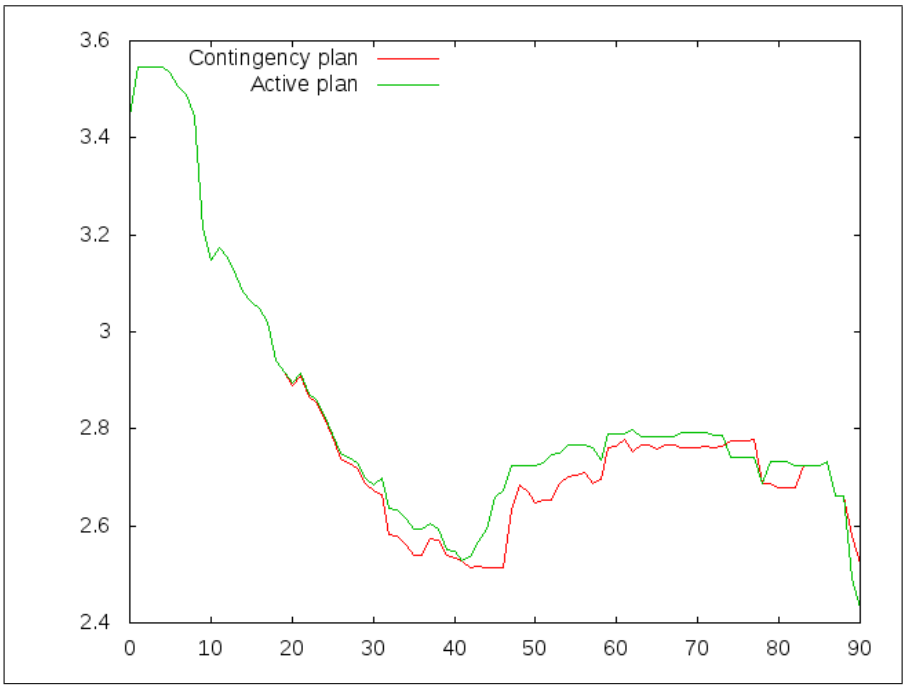
4.3.2 Contingency planning

This experiment tested the performance of the two types of contingency planning presented in Section 3.8 against each other and against a GA running two evolutionary threads, but no contingency planning. Matches against a simulated enemy, like the one presented in the previous section, were run for all 36 possible match setups, and fitness values were gathered throughout the match for each type of planning.

Fitness plots showing the average fitness values of the best solutions found through 36 90-second matches for each type of planning are shown in Figure 4.2.

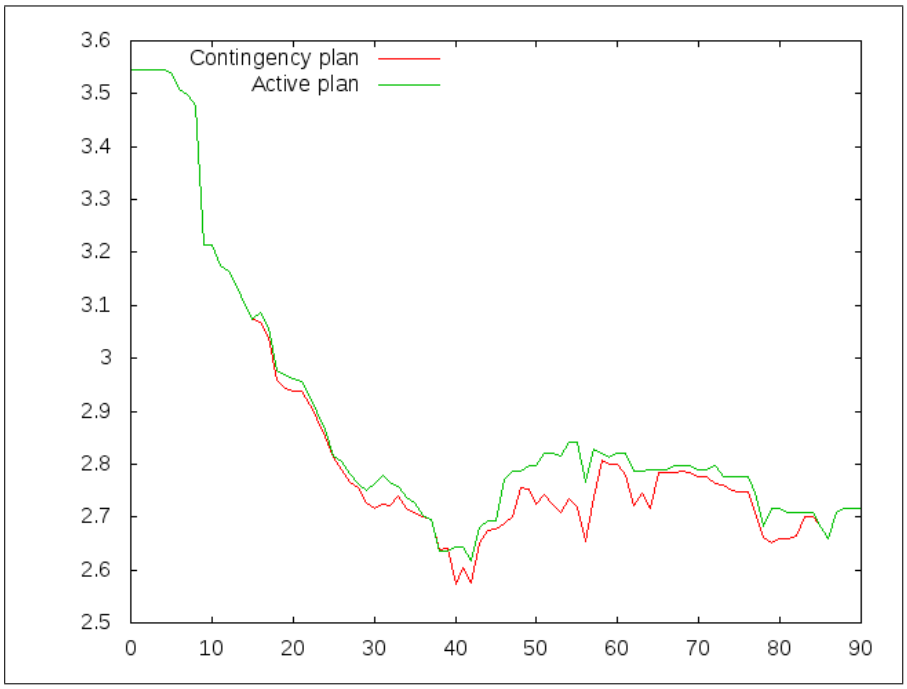


(a) No contingency planning



(b) Enemy-based contingency

Figure 4.2: Fitness plots for 90-second matches with different types of contingency planning



(c) Horizon-based contingency

Figure 4.2: Fitness plots for 90-second matches with different types of contingency planning (cont.)

Common features

The fitness plots clearly have more commonalities than differences. They all begin with the same high fitness value, which then decreases steadily towards a level of about 2.6 after 40-45 seconds. In this period, the opponent removes more and more of the element our robot has planned to pick, and thus, the fitness value decreases. The increase beginning halfway into the match comes as a result of the planning module finally being able to find a new and better plan to replace the old one. This trend was also confirmed by the match plot in Appendix A.

The increase in fitness coming this late in the match may seem surprising – after all, the GA uses about 15 seconds on its first generation, and then shorter and shorter. Studying the match plot in Appendix A may help explain why. Notice that while the fitness increase does not happen until about 45 seconds into the match, the fitness of the old solution is not below the fitness of the *new* solution until about 35 seconds. The fact that the enemy most often visits his hill first, means that his interference with our solution often comes quite a few seconds into the match, and may therefore explain why the major plan adjustment comes as late as halfway through the match.

From 45 seconds to about 60 seconds comes a period of improving the plan, to handle the environmental changes. After this, the plan's fitness stays stable until about 75-80 seconds where a decrease is seen. This decrease seems to be the effect of unforeseen events in the very last moments of the match, making the robot miss its final delivery in a few of the matches, leading to a large decrease in fitness in these matches, and a slight decrease in fitness in the average case.

The fitness of the contingency plans and the regular plans are, not surprisingly, highly correlated. The contingency plan typically has a little bit lower fitness than the regular plan, as it is normally working on a table with one less point-giving element.

Differences

When it comes to the *differences* between the plots, it is hard to find any significant results. The plots all have the same general form, and the small variations that do exist may simply stem from the stochastic nature of GAs. However, the horizon-based contingency planning *does* seem to perform slightly

better than the two other planning methods, yielding both a higher minimum value and a higher maximum value in the second half of the match. While it is hard to say whether or not this is an indication that horizon-based contingency planning is *better* than enemy-based contingency planning and two threads running regular evolutions, this does seem to imply that contingency planning GAs are worth looking into for other dynamic planning systems.

One obvious advantage horizon-based contingency planning has over the enemy-based one, is that it is independent of the type of enemy we are competing against. It bases its choice of contingency-action only on the runtime of the GA and the parameters simulating our own robot's movements, while the enemy-based version plans for the removal of the object closest to the enemy. The latter will clearly work better against some enemies than others: Against enemies staying a long time close to the same object while picking it, this will work a lot better than against enemies driving and picking very fast, leaving the contingency planner very little time to take their anticipated actions into consideration. The parameters governing the movements and picking actions of the simulated enemy are the ones given in Table 4.4. These describe a relatively fast robot, using little time to pick up objects (no time for corn, one second for tomatoes). It is expected that enemy-based contingency planning will perform better against slower enemies.

Evaluation

The fitness plots do not show any significant difference between GAs with and without contingency planning, but the results from contingency planning based on the robot's planning horizon stand out as the best ones. Therefore, a combination between GAs and contingency planning seems to be an idea that should be explored further in dynamic planning systems.

4.3.3 Simulator inaccuracies

Throughout Chapter 3, simplifications made in the simulator have been highlighted. The reason why a lot of simplifications have been made is firstly, because of the huge complexity of making an accurate Eurobot simulator, and secondly, because the GA is intended to cope with inaccuracies in the simulator dynamically. In other words, if something takes a bit longer or shorter than expected, the plan should be adjusted to fit the new and updated time

frame. This way, the GA not only adapts its plans due to environmental changes, but also due to inaccurate time estimates.

As pickups and deliveries are likely to have a quite stable and predictable duration, the focus of these experiments have been on investigating how inaccuracies in *translation time* affect the genetic algorithm. To do this, random errors have been added to *each* translation in a simulated match, as summarized in Table 4.5. The simulations used in fitness evaluation for the GA were, of course, unaware of this error, and used the standard timing estimates from Table 4.4. For each experimental setup, all 36 table setups were tested, and the average results stored. Plots of these results are shown in Figure 4.3. The planner ran two regular GA threads during these experiments.

Dynamic planning

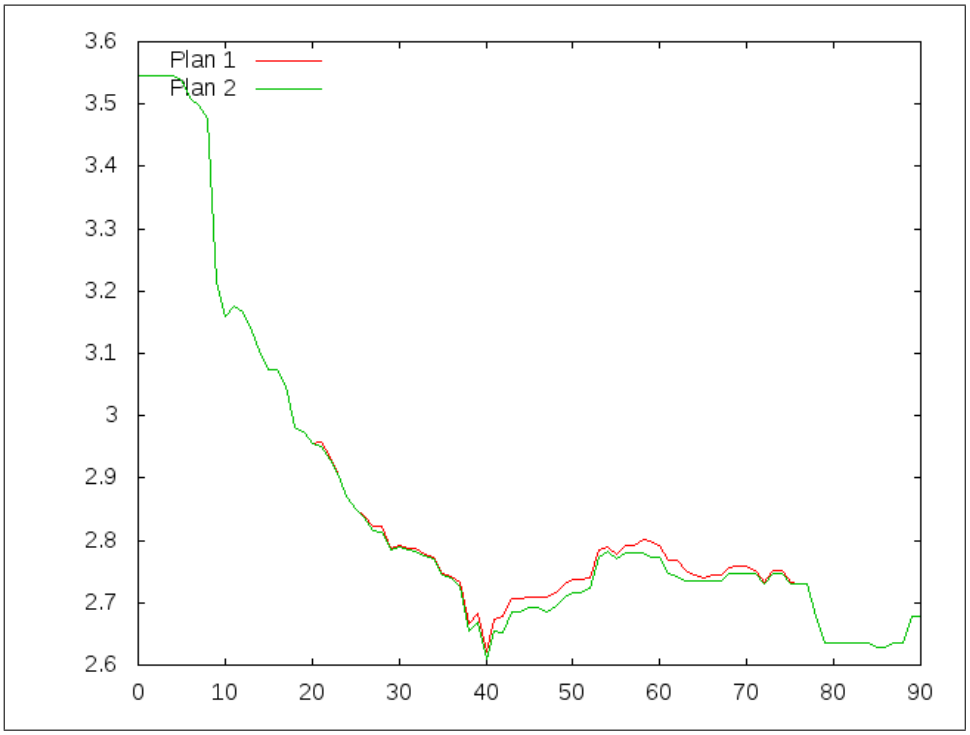
The first two plots in Figure 4.3 show how the dynamic planner reacts to wrong timing estimates. The first plot shows the case where timing estimates are completely precise, meaning that the decrease in fitness experienced here is completely due to the opponent's influence on the plan.

The next plot shows the case where a random, uniformly distributed error between -3 and 3 seconds has been added to each translation in the match. Results here are surprisingly good: The fitness values are actually *better* when a random error is inserted. This indicates that the GA is able to use additional time to its advantage, and that it can efficiently replan when something takes longer than expected.

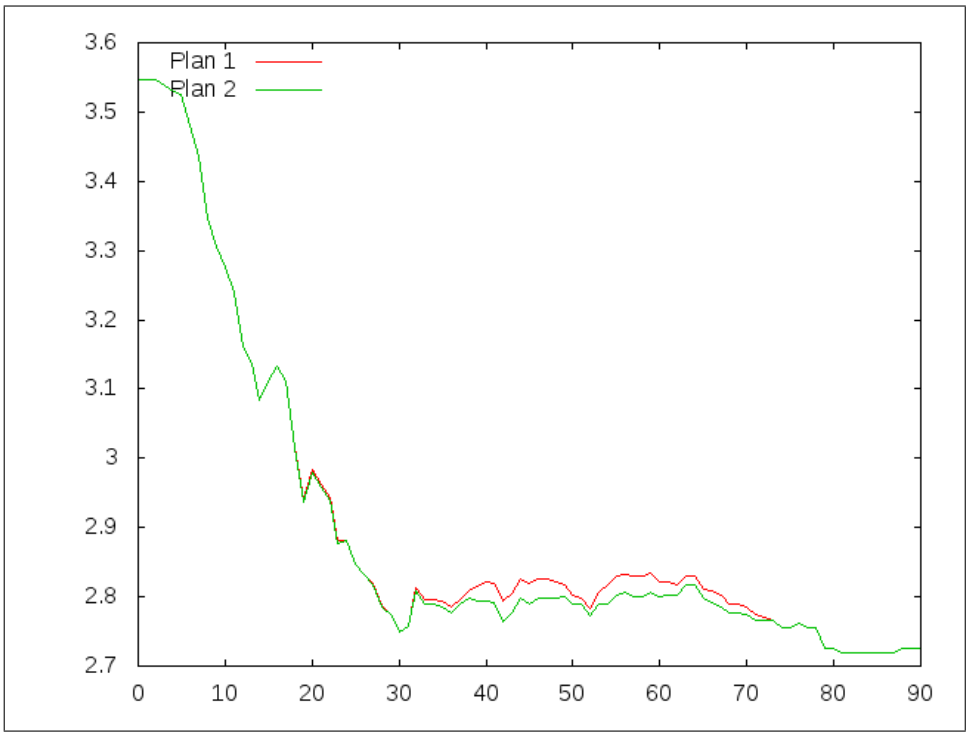
Static planning

The static plan that was used on each game setup was generated by the static GA, and was the same as the best plan used for initializing the dynamic planners. The plan was never modified during the match, except that elements removed from the table were also removed from the plan, saving the robot the time taken to visit these objects.

Figure 4.3c shows how the static plan performs in a match against a simulated opponent, but with completely precise timing estimates except for the opponent's influence. Its fitness values are lower than those for the corresponding dynamic planner throughout the match, and the plan ends up

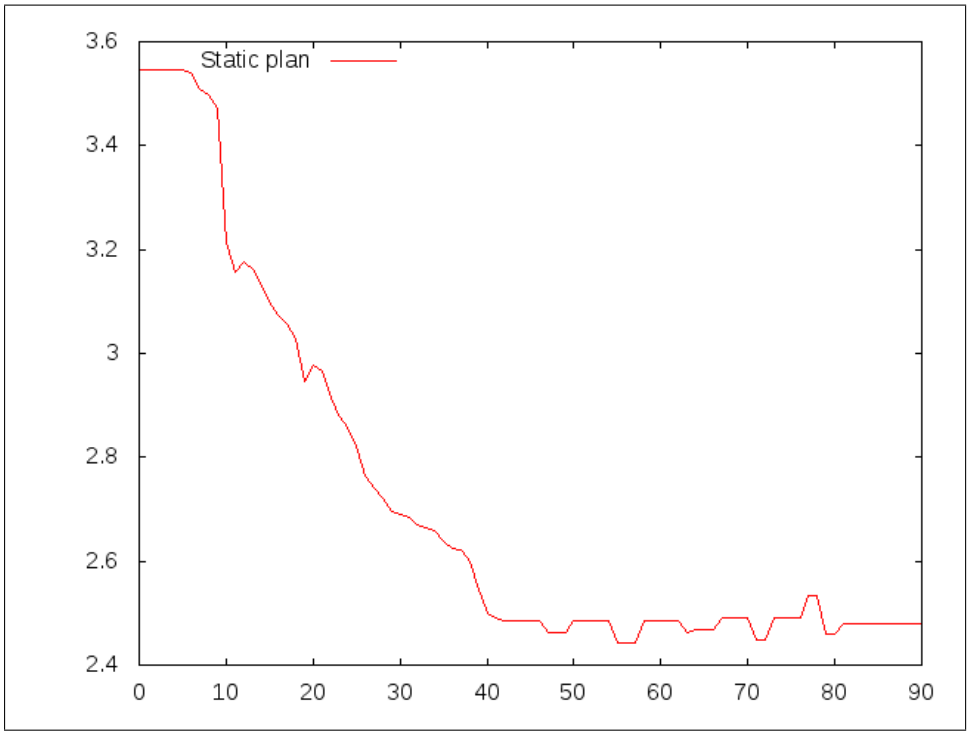


(a) Dynamic planning, no inaccuracies

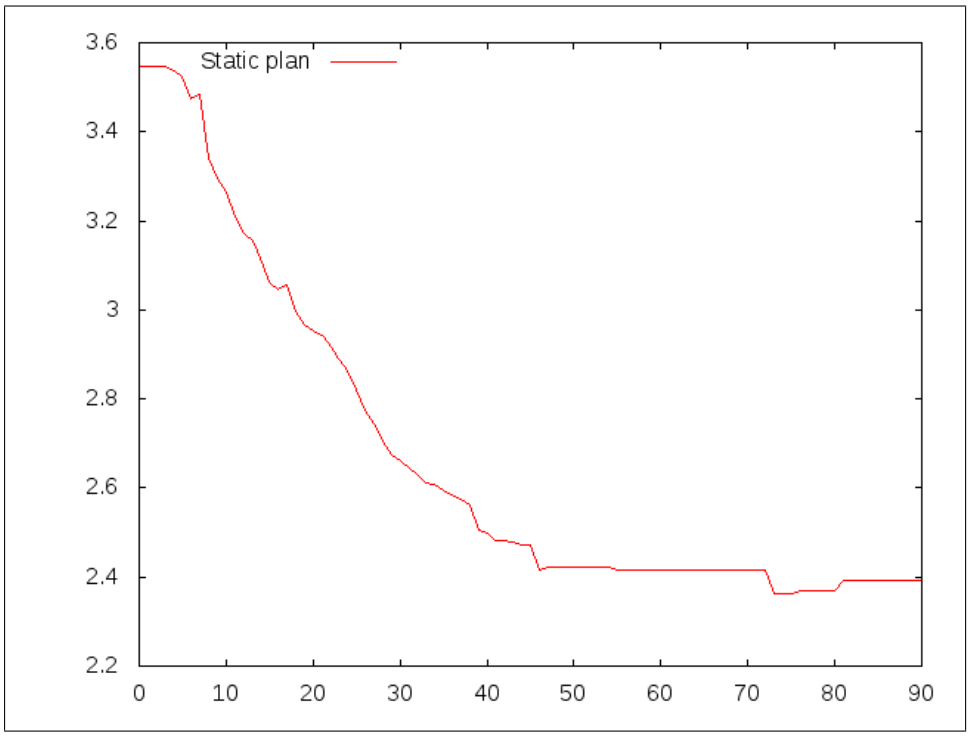


(b) Dynamic planning, 6 sec random error

Figure 4.3: Fitness plots for matches with an inaccurate simulator



(c) Static planning, no inaccuracies



(d) Static planning, 6 sec random error

Figure 4.3: Fitness plots for matches with an inaccurate simulator (cont.)

delivering about 0.2 kg less than the dynamically adjusted one.

Finally, figure 4.3d shows how the static plan does when translation times are subject to a random error ranging from -3 to +3 seconds. The results are worse in this case, and the final fitness value after 90 seconds shows that the random error makes the plan deliver more than 0.3 kg less than the dynamically adjusted one in the average case.

Evaluation

The fact that the static plan performs significantly worse than the dynamically adjusted one both when timing estimates are precise and faulty, proves that the hybridized GA approach combined with good plan-modifying heuristics works for the changing environment found in the Eurobot competition. That the dynamic planner is also able to use erroneous time estimates to its *advantage* is also a very useful result, as this means that a completely accurate match simulator is not needed, and the complexity of fitness evaluation can be kept low.

It may seem surprising that the static plan performs as well as it does in a dynamic and changing environment. After all, its final fitness isn't that much lower than the fitness of the dynamic plan. The reason why a lot of points are still gathered when running a static plan, is that the robot is still able to pick up all the objects it *reaches before* the opposing robot, and it is most of the time also able to complete all its planned deliveries. The robot is able to complete its deliveries despite delays incurred by enemy encounters and wrong time estimates, because it *saves* a lot of time by not having to visit the objects that the opponent has already picked. In other words, a good robot will perform well with a good static plan, but can add extra weight to its deliveries by dynamically adjusting its plan.

4.3.4 Efficiency of the GA

This experiment aimed to investigate whether the implemented GA is responsive and efficient enough to handle the dynamic environment of a Eurobot competition. To do so, simulated matches were run on both the computer used for development and the main part of testing of the GA, and on the computer used in the actual competition. The average times spent on each

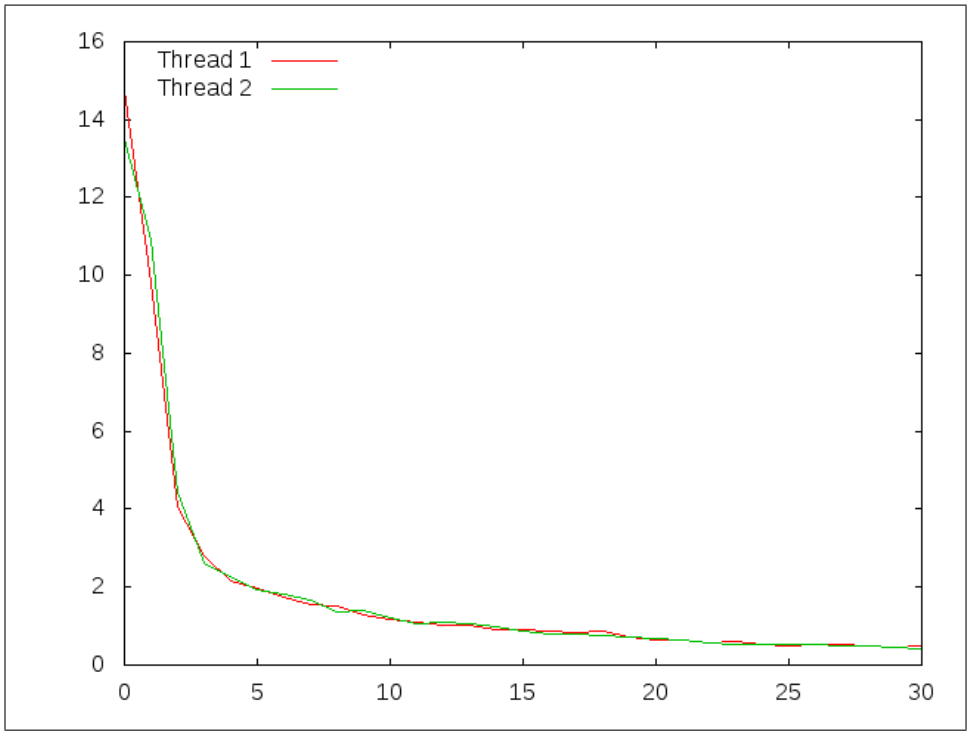
generation in these 90-second matches are plotted in Figure 4.4. The x-axis shows the generation number, and the y-axis the average time spent on that generation.

There are two main reasons why the main part of testing and development had to happen on a computer separate from the one used in the competition: Firstly, the competition computer was bought a couple of months into the project period, so development had to begin on a different computer, and secondly, many of the other team members also needed to use this computer for testing of their subsystems, so it was beneficial to be able to use a separate computer as much as possible.

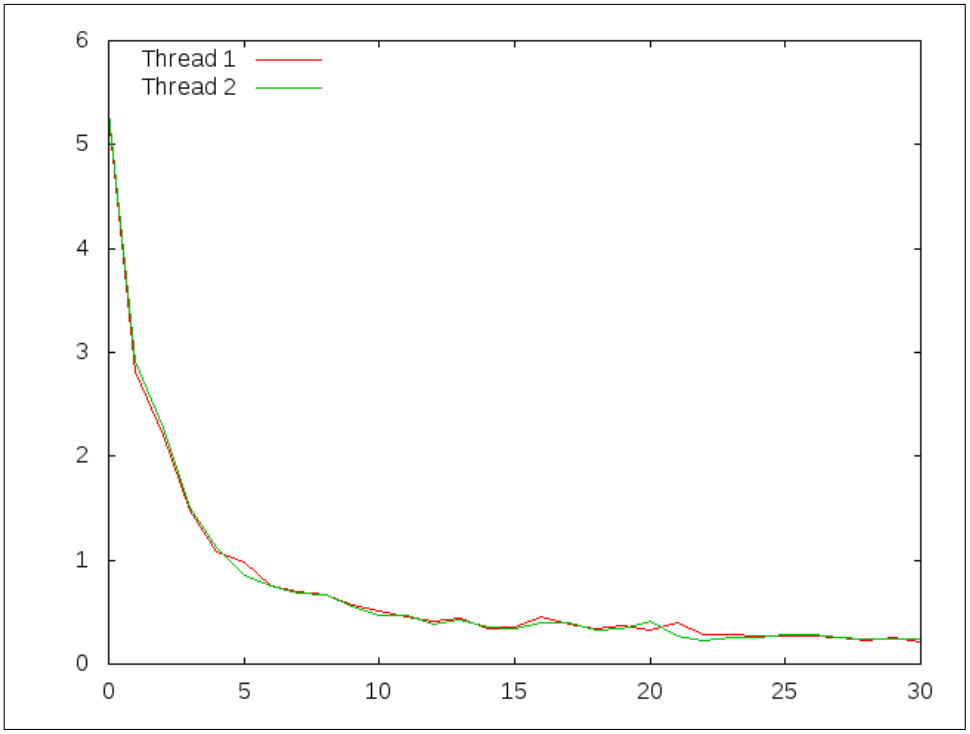
Results and evaluation

Figure 4.4 shows a quite big difference in runtime on the two computers that the GA was tested on. This is partly due to hardware differences between the two, and partly because the algorithm was compiled in a more optimized manner on the competition computer. Both computers show a rapid decrease in the time it takes to complete a generation as the match progresses. In fact, only thirteen seconds into the match, the robot-computer takes less than *a second* to complete a generation. As the robots start on opposite sides of the playing table, and as they will often be interested in visiting the hill first, this means that the responsiveness of the GA will be very good during the part of the match where it is most needed.

The reason why the runtime of GA generations decreases so rapidly throughout the match, is twofold: Firstly, objects are all the time removed from the table, effectively decreasing the genotype size, and thus, decreasing the time spent on the large local search (which has runtime $\mathcal{O}(n^2)$, where n is the genotype size). Secondly, the remaining time to plan for decreases as the match progresses, as fitness evaluation is always based on a match running from the *current state* of the game and up until the 90 seconds are done.



(a) The testing- and development computer



(b) The competition computer

Figure 4.4: Time to complete a GA generation throughout a match

Chapter 5

Conclusion

5.1 Goal Achievement

Looking back at the problem definition in Section 1.2.2, the main task was implementing a system capable of adapting previously generated plans for the Eurobot-competition throughout the match based on the opposing robot's influences. For this purpose, the genetic algorithm implemented for static planning as the author's specialization-project was adapted and extended to be able to handle a dynamic environment.

Chapter 4 presented several experiments whose aim was to determine if this goal had been reached. The first experiment, showing a full plot of a Eurobot match indicated that the dynamic GA was successful in adapting an already good plan to changing circumstances, and further experiments proved that statistically, the robot performed better when adjusting its plan throughout the match than when following a static plan. This improved performance was partly caused by the robot being able to deal with environmental changes, and partly by its ability to adjust to its own faulty time estimates in the initial planning phase.

The problem-definition also had three specific sub-goals that further specified the requirements for the envisioned system. The first sub-goal was the requirement that the algorithm should be efficient enough to handle the rapid changes of a Eurobot competition. To enable *immediate* responses to changing circumstances, the GA was extended with the ability to plan ahead for

contingencies, and with a fast heuristic making simple changes to plans on demand.

These extensions ensured changes were handled rapidly, but it was also desirable that the main evolutionary loop be as efficient as possible, so plans are rapidly optimized. Making the evolutionary loop as efficient as possible was done by boosting solution diversity, thereby giving the GA a higher degree of responsiveness, and by decreasing the genotype size, allowing the local search embedded in the GA to perform faster. Section 4.3.4 discusses the result of running performance tests on the GA, finding that the GA's performance is very good, using less than a second to complete a generation of evolution on the computer used in Eurobot matches, except for in the first 10-15 seconds of a match, where it may use up to six seconds. Due to the local search embedded in the GA, each generation typically brings a fairly well adapted individual, meaning that one second per generation should be rapid enough to respond in a good and efficient way to environmental changes.

The second sub-goal was having enough flexibility in the algorithm's fitness evaluation to allow it to easily be adjusted to the type of robot being built. This was important because design and construction of the robot was performed in the same time period as the planning system was being implemented. For this purpose, the simulator used for fitness evaluation was made highly parameterized, allowing important robot parameters to be specified as soon as they were known. These parameters include the robot's speed, capacity for carrying the various types of objects and many more. The parameters that were used for experiments in this report are listed in Table 4.4.

Despite the parameterized design of the simulator, some changes had to be made as soon as the robot was ready. For instance, it was found that picking too many corns could block access to the tomato-picker. These types of interactions between the various capacities of the robot had not been considered when designing the simulator, so a few changes needed to be made. However, all changes to the simulator caused by the robot's designed were found to be fairly straightforward to implement by making simple extensions to the methods controlling simulated pickups, deliveries and so on.

The third subgoal stated the need for a real-time simulator and plotter for testing the algorithm without having to run the robot. This was also important because the robot was not expected to be ready until quite shortly before the competition. Such a simulator and plotter was implemented, and screen shots from a simulated match taken with 5 second intervals are presented in

Appendix A. Although the physics of this simulator are quite simplified (see Section 4.2.3), it has proved absolutely essential in testing and understanding the GA's responses to environmental changes.

5.2 Research Value

As discussed in the above section, the project had a practical value in making a good planning algorithm. But what has been the *theoretical* value of the project? And where does this work fit in among the previous work done in this field?

First of all, the study of previous work presented in Chapter 2 has provided a good insight into common ways of solving the type of path optimization problems faced here, both with genetic algorithms and through other approaches. Solutions for both static and dynamic planning problems were studied, and it was found that the same type of solution technique may be employed in both cases, as a dynamic problem can be viewed as a *series* of static problems with time constraints.

5.2.1 Static scheduling problems

The study of static scheduling problems performed as part of the author's specialization project, resulted in the discovery that the Eurobot task was best viewed as a *combination* of two well known scheduling problems: The orienteering problem (OP) and the vehicle routing problem (VRP). They both encompass different parts of the Eurobot problem: The OP has the flexibility of not having to visit all nodes, while the VRP shares the issue of a capacity constraint with the Eurobot task.

Studying previous GA solutions to OPs and VRPs thus gave a valuable insight into how the Eurobot task could be represented. The choice fell on letting the GA handle both capacity constraints and the selection of a subset of nodes to visit, by adding a time limit and a capacity limit to the fitness evaluation, and by adding delivery nodes to the genotype that the GA could distribute freely.

The background study also led to the realization that the GA would need to be hybridized. All the previous systems that were studied were using a local

search technique to guide the GA in a complex fitness landscape. The type of local search selected for this task, was a technique developed by Christian Prins [22], a resource demanding yet very powerful technique. Experiments on the GA proved that it had no chance of finding good solutions without hybridization.

5.2.2 Dynamic scheduling problems

Extending the system to also be able to handle a *dynamically* changing environment was a challenging task. Optimization techniques for dynamic problems have been far less researched than static ones, and research into dynamic solutions for the TSP and the VRP can be said to still be in an initial phase.

Studying the task of dynamic planning, it was soon apparent that classical, mathematical planning methods would not be good solution techniques, as they always need to recalculate solutions from scratch as the environment changes. Search techniques like GAs, however, have the ability to continuously adapt their solutions to a changing environment. Also, as a GA had already been implemented for the task of static planning, adapting and extending this system seemed like a good choice for the dynamic planner.

A study of previous work using GAs in dynamic environments revealed some issues that were given special attention when implementing the dynamic planner. Firstly, the importance of *solution diversity* was argued to be more important in dynamic problems, because a diverse population will give a more responsive system. Different ways of boosting and maintaining diversity were reviewed, but a study of solutions generated after static pre-match planning indicated that they may already be diverse enough, with a simple fitness-based diversity measure. It was discussed whether it may be the local optimization that was causing this, by making similar solutions the same, thus giving them the same fitness. Even though a certain degree of diversity is generated by the static planner, an additional boost is given before dynamic planning is initiated by adding random individuals to the initial population, a technique somewhat resemblant of the *random immigrants* technique.

Studies of systems used for solving dynamic TSPs and VRPs with GAs showed that they often employed a separate heuristic for generating rapid responses to environmental changes. A combination of a GA and a good heuristic can give the planner a combination of rapid response to a changing environment, and the ability to generate better and better plans the longer it

gets to deliberate. Different heuristics were considered for this task, and the choice fell on a modified version of a simple and very efficient one that had been used as part of a GA solving dynamic TSPs [28]. More complex and time demanding heuristics were considered, but the choice fell on the simple one, as a complex heuristic was already embedded in the GA, meaning that solutions from this heuristic will be ready as soon as a GA generation finishes.

A final technique that was used to make the GA more responsive, was the use of a separate contingency-planning GA, running alongside the regular GA on a problem where one object has been removed. This type of combination of GAs and contingency planning has to the author's knowledge never been researched before, probably because GAs are implicitly contingency planners by maintaining a large population. The reason for making the contingency planning explicit, was to enable a guaranteed, immediate response to certain critical events, such as removal of the next object in the robot's plan. Experiments on the performance of the contingency planning GA show no conclusive results, but indicate that this could be worth looking into for other GAs doing dynamic planning.

All the approaches used to make the algorithm more responsive and faster may seem ambiguous. For instance, both the contingency planner and the heuristic aim to enable the system to give immediate responses to changes. The idea is, however, that the approaches can complement each other by working on different levels in resource usage and quality of the solution they generate. In addition, the system can easily be run *without* contingency planning, enabling the planning algorithm to be adjusted to the computational resources available.

5.3 Further Work

This section discusses what research questions raised in this report remain unanswered. These questions could be good starting points for further research into dynamic planning with GAs.

Firstly, the idea of using a genetic algorithm with a good heuristic for rapid response turned out to be a good fit for a dynamic planning problem. As previously mentioned, research into solving dynamic scheduling problems with GAs is at an initial phase, and in the author's opinion more work on

this field should try to uncover the relationship between properties of the GA and its responsiveness. For instance, how does the choice of diversity ensuring techniques, crossover techniques etc. affect the GAs responsiveness? Also, finding good and fast heuristics that provide the GA with good starting points for further optimization whenever the environment changes would be an interesting topic.

Also, the idea of explicitly running contingency planning parallel to the GA needs more research. The results from experiments in this report are inconclusive, but do indicate that such an idea works well if carefully selecting the type of contingency to plan for. Further experiments in this area should keep exploring whether the computational effort to run contingency planning give a better result than using the same resources on the GA.

References

[1] Enrique Alba and Bernabé Dorronsoro. Solving the vehicle routing problem by using cellular genetic algorithms. In *Evolutionary Computation in Combinatorial Optimization*, pages 11–20. 2004.

[2] Enrique Alba and Bernabé Dorronsoro. Computing nine new best-so-far solutions for capacitated VRP with a cellular genetic algorithm. *Information Processing Letters*, 98(6):225–230, June 2006.

[3] M. Bellmore and G. L. Nemhauser. The traveling salesman problem: A survey. *Operations Research*, 16(3):538–558, June 1968. ArticleType: primary_article / Full publication date: May - Jun., 1968 / Copyright © 1968 INFORMS.

[4] J. Branke, T. Kaussler, l Schmidt, and H. Schmeck. *A multi-population approach to dynamic optimization problems*. 2000.

[5] I-Ming Chao, Bruce L. Golden, and Edward A. Wasil. A fast and effective heuristic for the orienteering problem. *European Journal of Operational Research*, 88(3):475–489, February 1996.

[6] Yu-Wang Chen, Yong-Zai Lu, and Gen-Ke Yang. Hybrid evolutionary algorithm with marriage of genetic algorithm and extremal optimization for production scheduling. *The International Journal of Advanced Manufacturing Technology*, 36(9):959–968, April 2008.

[7] Thomas L. Dean and Michael P. Wellman. *Planning and control*. Morgan Kaufmann Publishers Inc., 1991.

[8] Keith L. Downing. Introduction to evolutionary algorithms. `http://www.idi.ntnu.no/emner/it3708/lectures/evolalgs.pdf`, 2009.

[9] David Floreano and Claudio Mattiussi. *Bio-Inspired Artificial Intelligence*. The MIT Press, 2008.

[10] Franklin Hanshar and Beatrice Ombuki-Berman. Dynamic vehicle routing using genetic algorithms. *Applied Intelligence*, 27(1):89–99, 2007.

[11] Yaochu Jin and J Branke. Evolutionary optimization in uncertain environments-a survey. *Evolutionary Computation, IEEE Transactions on*, 9(3):303–317, 2005.

[12] Soojung Jung and Ali Haghani. Genetic algorithm for the Time-Dependent vehicle routing problem. *Transportation Research Record: Journal of the Transportation Research Board*, 1771:164–171, January 2001.

[13] C. Peter Keller. Algorithms to solve the orienteering problem: A comparison. *European Journal of Operational Research*, 41(2):224–231, July 1989.

[14] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, June 1992.

[15] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345–358, 1992.

[16] P. Larrañaga, C. M. H Kuijpers, and R. H Murga. Tackling the travelling salesman problem with evolutionary algorithms: Representations and operators. 1994.

[17] Zhao Liu and Lishan Kang. A hybrid algorithm of n-OPT and GA to solve dynamic TSP. In *Grid and Cooperative Computing*, pages 1030–1033. 2004.

[18] Zbigniew Michalewicz and David B. Fogel. *How to solve it: modern heuristics*. Springer-Verlag New York, Inc., 2000.

[19] F. Oppacher and M. Wineberg. The shifting balance genetic algorithm: Improving the GA in a dynamic environment. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 504–510, 1999.

[20] Christos H. Papadimitriou. The euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237–244, June 1977.

[21] Planéte-Sciences. Eurobot 2010 - Feed the World. `http://www.eurobot.org/commonfiles/docs/2010/E2010_rules_and_drawing_EN.pdf`, September 2010.

[22] Christian Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31(12):1985–2002, October 2004.

[23] Harilaos N. Psaraftis. Dynamic vehicle routing problems. In *Vehicle Routing: Methods and Studies*, pages 223–248. 1988.

[24] Wan rong Jih and Jane Yung jen Hsu. Dynamic vehicle routing using hybrid genetic algorithms. *Proceedings of the 1999 IEEE International Conference on Robotics & Automation*, pages 453–458, May 1999.

[25] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.

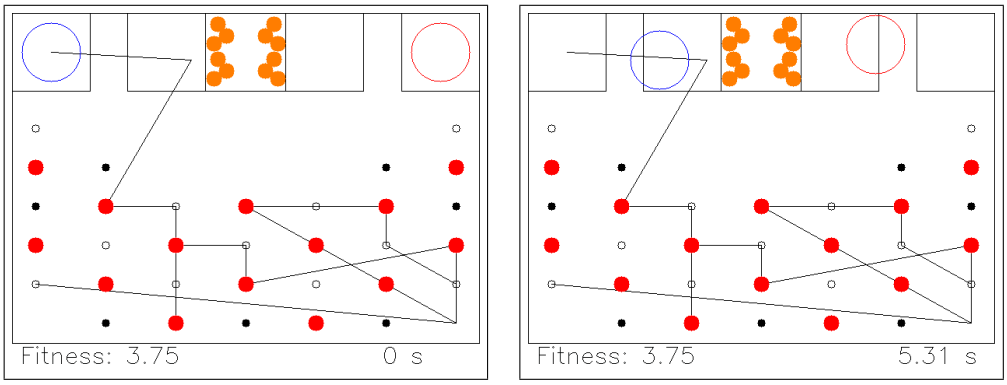
[26] M. F Tasgetiren. A genetic algorithm with an adaptive penalty function for the orienteering problem. *Journal of Economic and Social Research*, 4(2):1–26, 2001.

[27] Xia Wang, Bruce L. Golden, and Edward A. Wasil. Using a genetic algorithm to solve the generalized orienteering problem. In *The Vehicle Routing Problem: Latest Advances and New Challenges*, pages 263–274. 2008.

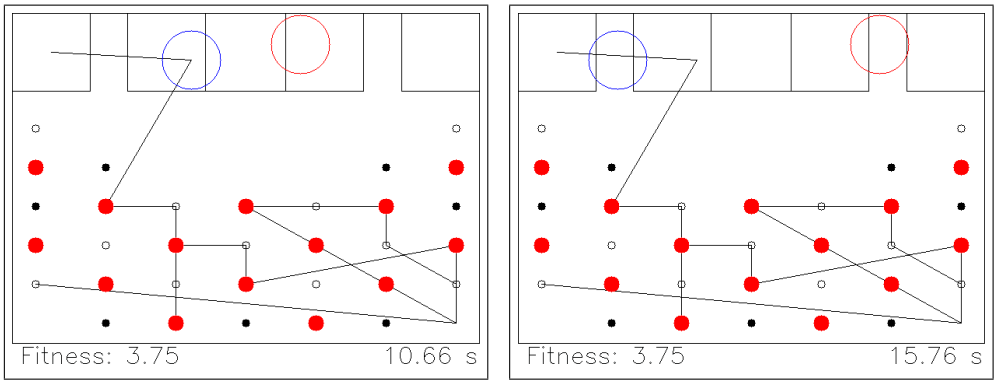
[28] Aimin Zhou, Lishan Kang, and Zhenyu Yan. Solving dynamic TSP with evolutionary approach in real time. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 2, pages 951–957 Vol.2, 2003.

Appendix A

Plots of a Full Simulated Match

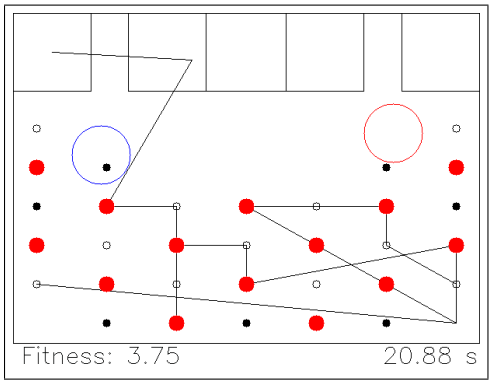


(a) (b)

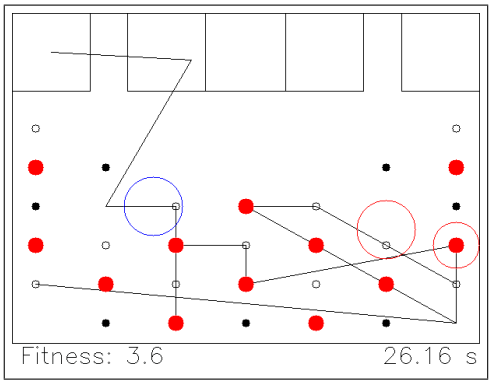


(c) (d)

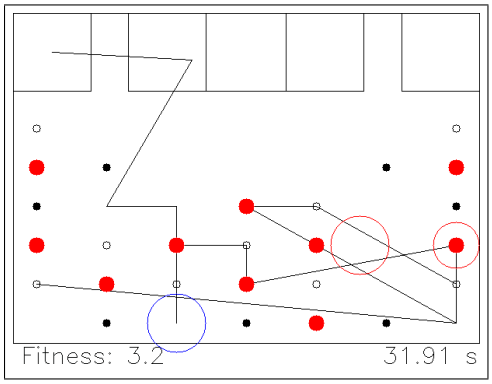
Figure A.1: Screenshots from a full simulated match



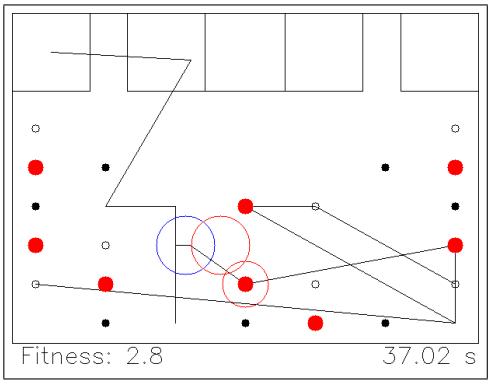
(e)



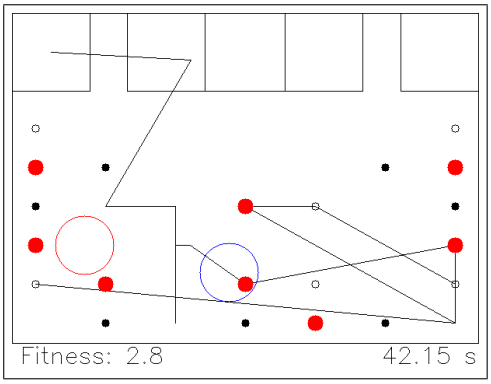
(f)



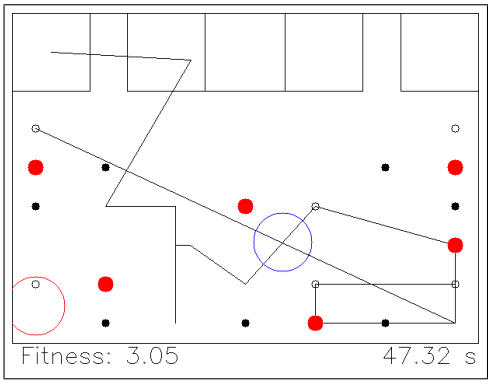
(g)



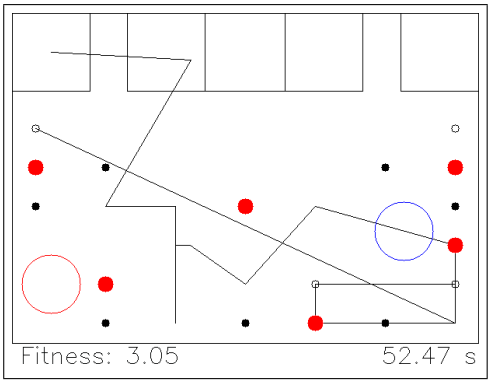
(h)



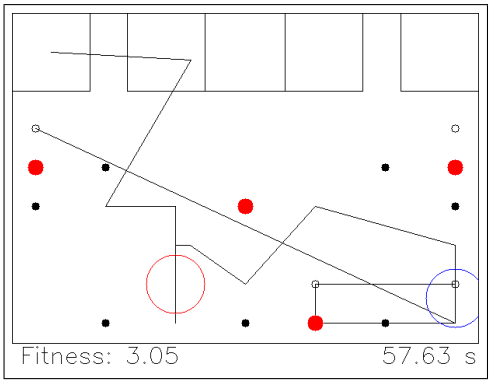
(i)



(j)

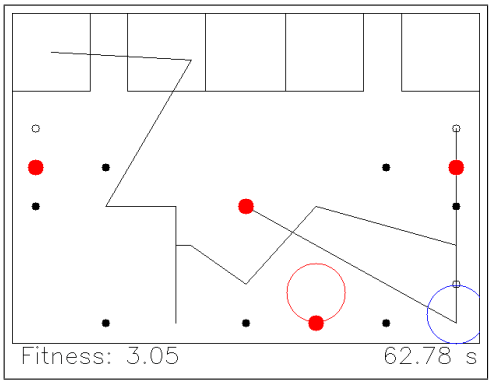


(k)

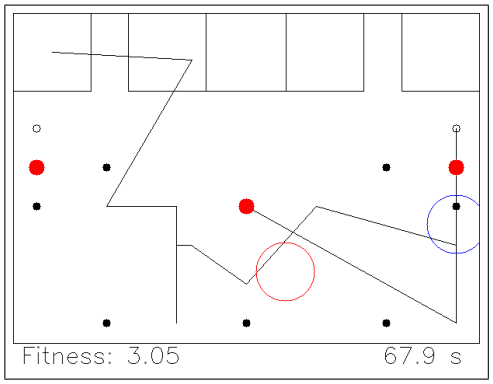


(l)

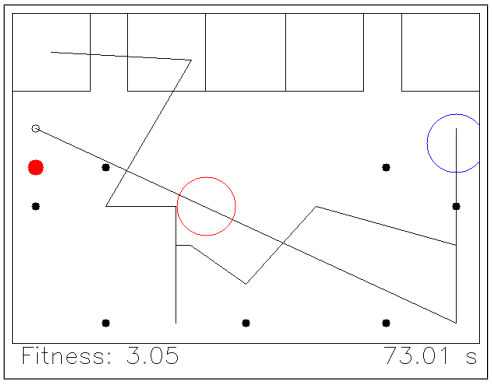
Figure A.1: Screenshots from a full simulated match (cont.)



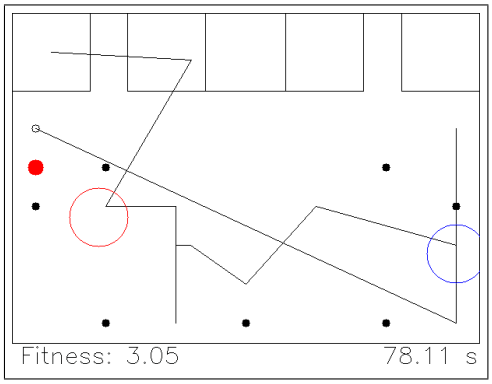
(m)



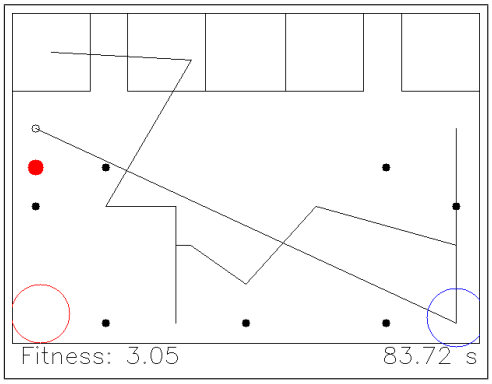
(n)



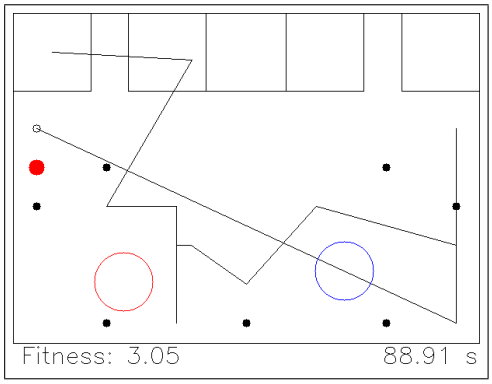
(o)



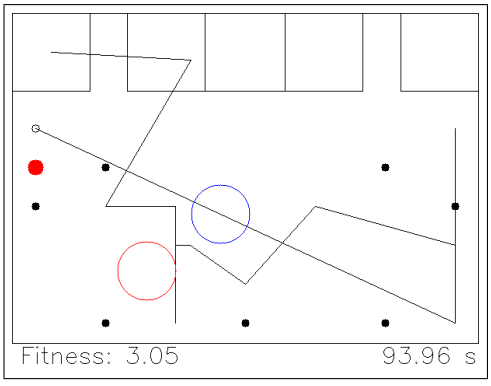
(p)



(q)



(r)



(s)

Figure A.1: Screenshots from a full simulated match (cont.)

Appendix B

Experiences From the Competition

This chapter sums up some of the experiences the Eurobot team made from working as a large interdisciplinary team creating a single product, and from taking part in the Eurobot competition. Hopefully, it may be of use to future Eurobot teams.

B.1 Results From the Competition

The team was unable to qualify the robot for the Eurobot-competition. The reason was that many of the systems (both hardware and software) on the robot were not ready until shortly before, or even after, departure to Switzerland. This meant that a lot of integration and testing had to happen in Switzerland, and there simply wasn't enough time to make the robot work properly.

B.2 What Went Wrong?

The reason why things were not ready in time, was partly due to bad planning and coordination between the team members. Also, a few weeks before the competition, a design flaw in the robot's mechanical system was found, that called for a major redesign of the robot in the very last weeks. This delayed integration and testing of the various subsystems further.

This is the first year where a team of this size has participated in the Eurobot-competition, and in hindsight it is obvious that the management and coordination of such a team requires an experienced team leader with a thorough understanding and overview of everybody's work. The task's large complexity could also be handled better by a more *continuous* integration process, beginning as soon as each member had something ready. This way, basic functionality of the robot can be ensured early in the project period, and more advanced functionality can build upon this.