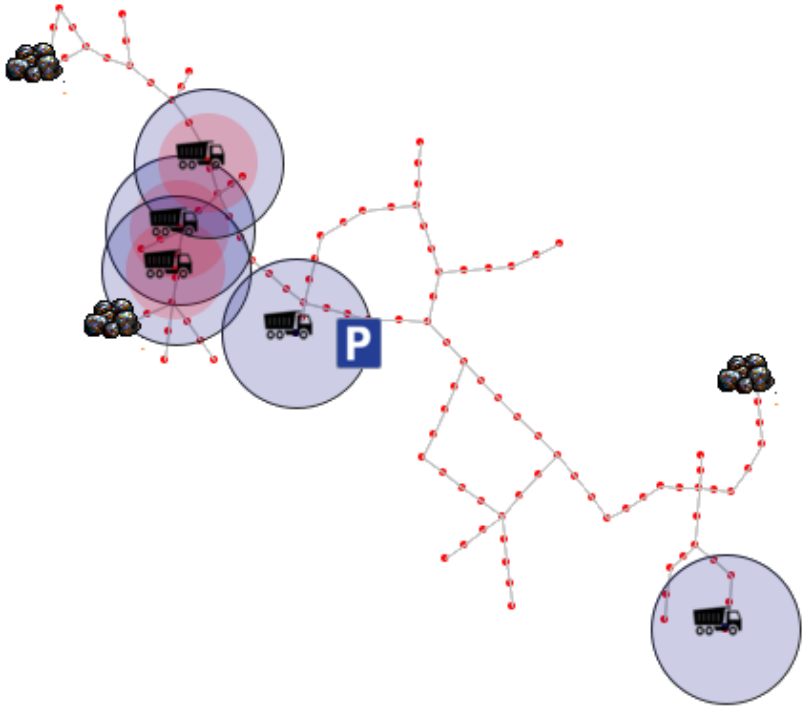
CHALMERS
UNIVERSITY OF TECHNOLOGY



Self-organizing multi-agent systems for shared space operations

Using genetic algorithms and contract net protocols to solve the pickup and delivery problem

Master's thesis in the Master's Programmes:
Complex Adaptive Systems and Computer Science - Algorithms, Languages and Logic

SVANTE KARLSSON
JACOB STEFFENBURG

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018:84

Self-organizing multi-agent systems for shared space operations

Using genetic algorithms and contract net protocols to solve the pickup and delivery problem

Svante Karlsson
Jacob Steffenburg

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Applied Artificial Intelligence Research Group
Chalmers University of Technology
Gothenburg, Sweden 2018

Self-organizing multi-agent systems for shared space operations
Using genetic algorithms and contract net protocols to solve the pickup and delivery problem
SVANTE KARLSSON, JACOB STEFFENBURG

Supervisor: Mauro Bellone, Applied Artificial Intelligence
Supervisor: Åsa Rogenfelt, CPAC Systems AB
Examiner: Peter Forsberg, Applied Artificial Intelligence

Master's Thesis 2018:84
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Applied Artificial Intelligence Research Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A pickup and delivery problem on a graph in which three agents have detected conflicting routes.

Typeset in LaTeX
Gothenburg, Sweden 2018

Self-organizing multi-agent systems for shared space operations
Using genetic algorithms and contract net protocols to solve the pickup and delivery problem
SVANTE KARLSSON, JACOB STEFFENBURG
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Abstract

Automation of on-site scheduling and route planning of units in a mining operation presents interesting challenges. The dynamic properties of real-life operations necessitate AI-inspired decentralized methods, which tend to be more robust under real conditions. Moreover, route planning in a mining environment, consisting of narrow passageways, requires vehicles to communicate and cooperate for safe and efficient transportation.

We model this as a dynamic multiple-agent pickup and delivery problem; a problem in which multiple agents cooperate to complete transportation tasks that are revealed continuously. Taking inspiration from novel solutions, using auction-like bidding systems based on genetically optimized heuristics, we tackle the pickup and delivery problem (PDP) from two different angles. Firstly, we show that existing solutions to the planar variant of the PDP can be improved by giving agents the ability to communicate. Secondly, we present a solution method to the PDP in a shared space environment, applicable for real world scenarios such as a mining operation. Apart from using the aforementioned bidding system to assign tasks to agents, we also implement a method for solving decentralized multiple-agent path finding.

Keywords: artificial intelligence, machine learning, reinforcement learning, multi-agent system, pickup and delivery problem, genetic algorithm, multi-agent path finding.

Acknowledgements

We would like to thank CPAC Systems AB for giving us this opportunity, it has been a great project to work on. We would also like to thank our supervisors, Mauro and Åsa, whose valuable insights have improved the thesis immensely. Finally we want to thank our examiner Peter Forsberg who made this thesis possible.

Svante Karlsson & Jacob Steffenburg
Gothenburg, October 2018

Thesis supervisors: Mauro Bellone, Applied Artificial Intelligence
Åsa Rogenfelt, CPAC Systems AB
Thesis examiner: Peter Forsberg, Applied Artificial Intelligence

Nomenclature

CNET	Contract-Net Protocol
GP	Genetic Programming
MAPF	Multiple-Agent Path Finding
MAS	Multiple-Agent System
PDP	Pickup and Delivery Problem

Contents

List of Figures

List of Tables

1
Introduction

Current trends within autonomous mining utilize Artificial Intelligence (AI) for navigation and low level control of vehicles. In particular, Deep Learning has been found useful for this purpose [1]. The high level control, such as scheduling and route planning of each unit, is however mostly still done manually. One possible way of raising efficiency would be to automate the on-site detailed planning of transportation units.

The objective of these vehicles is to transport ore from deposits to centrally located crushers, which pulverizes the ore into manageable size. This can be modeled as a pickup and delivery problem (PDP), for which the objective is to service requests while minimizing travel time or fuel costs. PDPs have been the focus of a lot of recent research with the rise of taxi cab and peer-to-peer ridesharing services such as Uber, as summarized by Cordeau in [2]. A solution to a PDP can either form a centralized system, featuring a central decision-maker, or a decentralized multi-agent system.

Centralized systems offer a complete overview of each truck's current route. However, there are some issues with a single system having to review the path of each agent, such as loss of communication between agent and system. When circumstances change, which is common in a real world setting, the planning will need to be adjusted accordingly. This necessitates AI-inspired methods, which tend to be more robust under real conditions, as argued by van Lon and Holvoet in [3].

A common solution to the PDP using decentralized multi-agent system is to use a Contract Net protocol (CNET), in which agents bid on each new order according to their current state. Using a CNET protocol, the goal is now to optimize the heuristic used for bidding, representing how well-suited the agent is to service the order. Recent work by van Lon et al. [4] uses genetic programming to perform this optimization, providing a solution for a planar PDP, which shows promising results compared to a centralized solution.

However, going from a planar PDP, where agents do not compete for access to routes, to a shared space graph based PDP introduces a number of problems, chief among them being routing conflicts.

1.1 Problem Statement

The purpose of the project is to examine how genetic programming and reinforcement learning can be used to develop a self organizing multiple agent system (MAS) for the pickup and delivery problem (PDP). The project will build upon existing research and theory by introducing communication between agents to the MAS. Once

this is successful an application will be developed for industrial vehicles in a mining environment.

1.2 Goals

The two main goals of this thesis are:

- To study the effects of introducing communication support to a genetic program in charge of optimizing solutions to the planar PDP.
- To develop a solution method for the shared space variant of the PDP, tackling the issues of schedule optimization and multi-agent path finding.

1.3 Assumptions

For the first part of the thesis, we will limit ourselves to the solution developed by van Lon et al. [4], with the only addition being communication support for bidding heuristics. As for the industrial application, several assumptions will be applied, namely:

- Vehicles are either stationary or travel at a constant speed. Partly since the simulation environment used for the project does not support varying velocities but also because the kinematic aspect is considered to be outside the scope of this thesis.
- Communication devices are 100% reliable, i.e., all sent messages are received as long as vehicles are within communication range.
- Vehicle cargo capacity is limited to one parcel, which is typical for a mining operation. However, there might be fringe cases such as a vehicle depleting an ore deposit only filling its cargo to half capacity.
- Pickup, delivery and depot locations have unlimited capacity for vehicles. This is used to combat situations where a route planner might not be able to find a solution for routing conflicts, causing invalid scenarios which would increase the time needed for evolving bidding heuristics.

1.4 Contribution

In this thesis, we expand on the work of van Lon, Holvoet, and Branke [4, 5] by introducing a way for agents in decentralized planar PDPs to communicate. Communication is enabled by allowing agents to access information about the states and intentions of other agents. This information is utilized when placing bids on parcels, with results suggesting an overall cost reduction of solutions, leading to cheaper and more efficient schedules.

Furthermore, we remodel the problem for graphs with the added constraint that agents are not allowed to collide. By extending the planar PDP solution so that agents are capable of detecting and avoiding collisions, we obtain a general solution method for shared space PDPs.

2
Theory

In this chapter we present the theory behind the main problems addressed in this thesis. We define the pickup and delivery problem in Section 2.1. Section 2.2 describes the contract net protocol, which is used as a building block for solving the pickup and delivery problem. Continuing, the theory behind genetic programming is presented in Section 2.3. Finally, in Section 2.4 we present theory behind multi-agent path planning, which is used for resolving routing conflicts between agents.

2.1 The Pickup and Delivery Problem

The Pickup and Delivery Problem (PDP) is a logistics problem in which one or several agents has to move one or several parcels between different locations. For instance, the agents might represent taxi cabs and the parcels might represent passengers. The problem can either be *static* or *dynamic*; in static problems we are given complete information from the start, whereas in dynamic problems the input data and conditions may change over time. An example of a dynamic pickup and delivery problem is the above mentioned taxi problem, where no fares are known a priori and new fares are introduced as time goes by. Performance can be measured in a number of ways, for example, total waiting time for each pickup, tardiness, total distance travelled, and total pickups completed.

In general, there are two different approaches to choose from when solving the PDP: using state-of-the-art optimization algorithms to solve a static problem every time new input is revealed, or viewing the system as a decentralized multi-agent system. In [3], van Lon and Holvoet argue that the second method provides certain advantages, such as:

- Contrary to centralized systems, decentralized ones scale well with the problem size.
- Decentralized systems are flexible and adaptable to changes and disruptions to the environment.

2.1.1 Problem definition

An instance of a dynamic PDP is defined by the tuple

$$instance := \langle \mathcal{T}, \mathcal{O}, \mathcal{A}, \mathcal{M} \rangle,$$

where $\mathcal{T}$ is the duration of the problem instance, $\mathcal{O}$ is a set of orders, $\mathcal{A}$ is a set of agents and $\mathcal{M}$ is a map in which the agents reside. The interval $[0, \mathcal{T})$ is the time

period in which agents may operate and process orders. Orders $o_i \in \mathcal{O}$ represent parcels to be picked up and delivered and are defined by:

$$\begin{aligned} a_i &\text{ - the announce time,} \\ p_i = [p_i^L, p_i^R) &\text{ - the time window for pickup,} \\ d_i = [d_i^L, d_i^R) &\text{ - the time window for delivery,} \\ ploc_i \in \mathcal{M} &\text{ - the pickup location,} \\ dloc_i \in \mathcal{M} &\text{ - the delivery location.} \end{aligned}$$

The map $\mathcal{M}$ can either be a subset of $\mathbb{R}^n$ or a graph $\mathcal{G}$.

For the general PDP, some more assumptions are made:

- all agents are homogeneous,
- agents travel with constant speed $v \in \{0, V\}$,
- agents have no limits on cargo capacity,
- agents use no fuel and can operate without rest,
- a central depot is used as a starting and finishing point,
- a scenario is completed when every task has been handled and all agents are back at the depot,
- each location in $\mathcal{M}$ can be reached from any other location in $\mathcal{M}$.

Agents are not allowed to pick up or deliver parcels before the corresponding time window is open, i.e., $sp_i \geq p_i^L$ and $sd_i \geq d_i^L$, where sp_i is the time when parcel i is picked up and sd_i is the time when parcel i is delivered. The requirement that the parcel is delivered before the time window closes is modeled as a soft constraint which is put into the objective function:

$$f(\mathcal{T}, \mathcal{O}, \mathcal{A}) = \sum_{j \in \mathcal{A}} vtt_j + \sum_{j \in \mathcal{A}} T(\tau_j, \mathcal{T}) + \sum_{i \in \mathcal{O}} \Big(T(sp_i, p_i^R) + T(sd_i, d_i^R) \Big). \tag{2.1}$$

Here, vtt_j is the total travel of agent j, $T(a, b) := max(0, a - b)$ is the so called *tardiness*, and τ_j is the time when agent j is back at the depot. As can be seen in Equation 2.1, the objective function consists of three different quantities that should be minimized:

1. the total travel time of all agents vtt_j,
2. the total overtime of all agents $T(\tau_j, \mathcal{T})$,
3. the total tardiness of all pickups $T(sp_i, p_i^R)$ and deliveries $T(sd_i, d_i^R)$.

2.1.2 Problem characteristics

A PDP scenario is characterized by three important parameters: **dynamism**, **urgency**, and **scale** [6, 7].

2.1.2.1 Dynamism

The dynamism of a scenario is an indicator for how continuously the scenario will change, where more continuous change means higher dynamism. In this context, a change in the scenario corresponds to a new parcel being announced. Figure 2.1

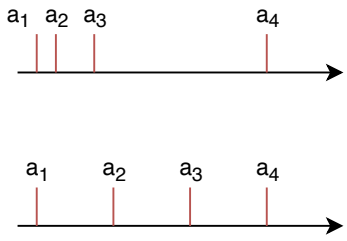


Figure 2.1: Two scenarios with different levels of dynamism. The top-most scenario is less dynamic than the bottom-most one.

shows two example scenarios with different dynamism levels.
A more formal definition of dynamism can be formulated by considering the *inter-arrival times*:

$$\Delta := \{\delta_i = a_{i+1} - a_i | i = 0, 1, ..., |\mathcal{O}| - 2\},$$

which are the times between announcements of two consecutive parcels. The announcement times of parcels in a 100% dynamic scenario would be evenly spaced on on $[0, \mathcal{T})$. This perfect interarrival time is defined as $\theta := \frac{\mathcal{T}}{|\mathcal{O}|}$. For each order in an arbitrary scenario, we calculate the deviation from the perfect interarrival time as

$$\sigma_i := \begin{cases} \theta - \delta_i & \text{if } i = 0 \text{ and } \delta_i < \theta \\ \theta - \delta_i + \dfrac{\theta - \delta_i}{\theta}\sigma_{i-1} & \text{if } i > 0 \text{ and } \delta_i < \theta \\ 0 & \text{else.} \end{cases}$$

Further, the maximum possible deviation to the perfect interarrival time is computed as

$$\bar{\sigma_i} := \theta + \begin{cases} \dfrac{\theta - \delta_i}{\theta}\sigma_{i-1} & \text{if } i > 0 \text{ and } \delta_i < \theta \\ 0 & \text{else.} \end{cases}$$

Finally, the dynamism of a scenario is calculated as:

$$dynamism := 1 - \frac{\sum_{i=0}^{|\Delta|} \sigma_i}{\sum_{i=0}^{|\Delta|} \bar{\sigma_i}}. \tag{2.2}$$

For a more thorough explanation of this formula, see [7].

2.1.2.2 Urgency

The urgency of an order is defined as the maximum reaction time agents have, i.e., the time from the announcement of the order until the closing of the pick up time window. Two orders with different levels of urgency are illustrated in Figure 2.2. For an entire scenario, the urgency is calculated as the mean of the urgency of all orders.

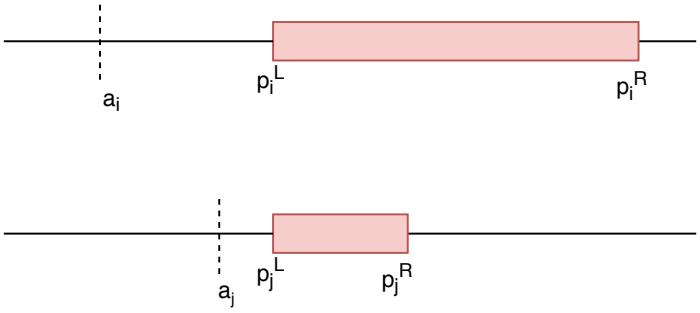


Figure 2.2: Two orders of different urgency. The top-most order is less urgent than the bottom-most one.

2.1.2.3 Scale

An algorithm for solving the PDP is called scalable if it can maintain a fixed cost per parcel when the number of parcels is increased proportionally to the number of agents. If a scenario $\langle \mathcal{T}, \mathcal{O}, \mathcal{A}, \mathcal{M} \rangle$ is scaled by a factor α, a new scenario $\langle \mathcal{T}, \mathcal{O}', \mathcal{A}', \mathcal{M} \rangle$ is obtained, where $|\mathcal{O}'| = \alpha|\mathcal{O}|$ and $|\mathcal{A}'| = \alpha|\mathcal{A}|$. Note that this definition is not very well suited for maps in which collisions have to be avoided.[1]

2.2 Contract-net protocol

To solve task assignment problems in a distributed fashion a contract-net protocol (CNET) can be implemented, as described by Smith [8]. This type of protocol is inspired by how companies use subcontracting to solve a larger problem. In CNET, *contractor* agents compete for tasks in a list set up by a *manager* agent. The bidding process is shown in Figure 2.3. For each task, each contractor calculate a bid using a heuristic that is dependent on the contractor's state. The manager then decides which contractor gets to service the task by assigning it to the contractor with the minimum bid value. This process is shown in Figure 2.3.

Subcontracting systems can be set up either competitively or cooperatively. In a cooperative setting, we can assume that the contractors and the manager all work for the same company and bids are calculated to promote the well-being of this company, instead of bids reflecting personal gain of each contractor.

[1] Increasing the number of agents with a fixed map increases the number of possible collisions, adding yet another dimension of complexity. For collision avoidance in a planar setting, the map could be scaled up as well, but it is not clear how one would scale up a graph in a fair way. This has been left out of the thesis.

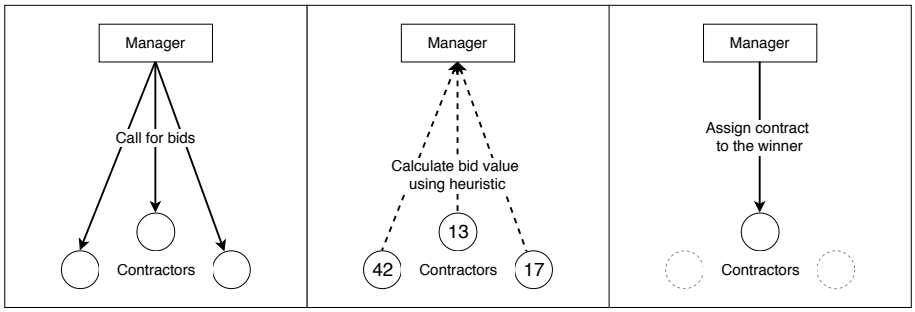


Figure 2.3: The CNET protocol process.

By using a CNET, the problem becomes to optimize the heuristic calculating these bids using the state of the contractor agent. In a dynamic problem where the search space for this function is large and the function needs to be computed several times during a simulation, the use of a genetic algorithm-based hyper-heuristic may be beneficial.

2.3 Genetic programming

Genetic programming is an application of evolutionary algorithms, a set of optimization techniques which, as the name implies, takes inspiration from the biological process of evolution. Evolutionary algorithms, also known as genetic algorithms, maintain a set of *individuals* which model candidate solutions to the optimization problem. These individuals all contain a genome which, in the case of genetic programming, encodes a program or a function, as explained by Koza [9]. The program/function is encoded in some data structure, for instance a tree structure, and consists of different operators and inputs. During optimization, the individuals are combined and mutated to create offspring, as shown in Figures 2.5 and 2.6, thereby creating new candidate solutions to the optimization problem.

Each individual is evaluated, using the objective function, and assigned a fitness value corresponding to its performance. High-performing individuals will more likely be selected as parents to the next generation, whereas individuals with low fitness are unlikely to pass on their genetic material, ensuring convergence.

2.3.1 Encoding

Similar to how organisms encode data in their chromosomes using DNA, the chromosomes in a genetic algorithm encodes a solution to a given optimization problem. Arrays are commonly used for this purpose, which can be used to encode real numbers or weights of a neural network, as shown by Wahde [10]. The solution provided by decoding a chromosome is evaluated, providing a fitness measure for the corresponding individual. Using these fitness values, the individuals may then be recombined using methods such as crossover and mutation, hopefully resulting in better performing individuals and solutions. For encoding a heuristic, a tree data structure can be used. An example of this is shown in Figure 2.4.

Algorithm 1 Genetic programming algorithm

Require: objective function to minimize/maximize
1: *population* ← initialize with random values
2: **repeat**
3: *heuristics* ← decode *population*
4: *fitness* ← evaluate objective function using *heuristics*
5: *generation* ← *generation* + 1
6: *newpopulation* ← perform selection on *population*
7: *newpopulation* ← perform crossover on *newpopulation*
8: *newpopulation* ← perform mutation on *newpopulation*
9: *newpopulation* ← add elite from *population*
10: *population* ← *newpopulation*
11: **until** *generation*=k
12: **return** *heuristics, fitness*

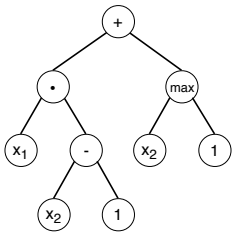


Figure 2.4: Tree based encoding for a genetic program. Decoding the chromosome gives the function $f(x_1, x_2) = x_1 \cdot (x_2 - 1) + \max(x_2, 1)$

2.3.2 Selection

Selection is the process in which individuals are chosen to form the next generation. By favoring individuals with higher fitness, the selection process ensures the evolution of the species. A common selection method is *Tournament Selection*. In regular tournament, selection two individuals are chosen randomly from the population, with or without replacement. The individual with the highest fitness is then selected with probability p_{tour}, where $p_{tour} > \frac{1}{2}$. Tournament selection can be generalized to size n by choosing n individuals, drawing a random number $r \in [0, 1]$ and selecting the most fit individual if $r < p_{tour}$. If not, the process is repeated with the remaining $n - 1$ individuals.

2.3.3 Crossover

After two individuals have been selected, new individuals may be formed using the genetics of the selected individuals. This is commonly done by randomly selecting a crossover point, indicating where to split the individuals chromosomes in two parts. These are then combined so that new individuals are formed. An illustration of this process is shown in Figure 2.5.

Using crossover, the chromosomes of a well-behaving individual may spread

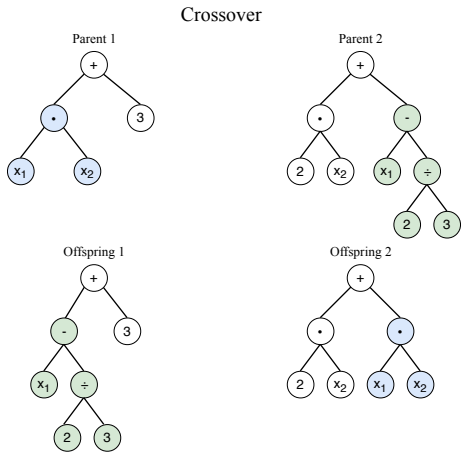


Figure 2.5: Example of how crossover is used for tree-based chromosomes.

rapidly among the population, similar to inbreeding. To counteract this, a crossover probability p_c is introduced. After two individuals have been selected, crossover is performed with probability p_c. If crossover was not performed, the new individuals are simply generated as copies of the previous ones.

2.3.4 Mutation

As with biological mutation, this process aims to make small changes to offsprings' chromosomes enabling new behaviour and providing genetic material for future evolution. Mutations are rarely beneficial for the individual in which they first appear. However, since fit individuals are selected for crossover and unfit individuals are discarded, the mutation process will give rise to long-term evolution of the species.

For a tree-based genetic program, two common types of mutation processes are *point mutation* and *subtree mutation*. In both processes a mutation point is chosen randomly. A point mutation then switches the function or terminal stored at that node with a different one of the same arity. In subtree mutation, the selected node is replaced with an entirely new randomly generated subtree. Both process are illustrated in Figure 2.6.

2.3.5 Elitism

The last addition to the genetic programming algorithm is elitism, where the most fit individual in each generation is directly copied to the next generation. This is done to ensure that the genetic program does not need to rediscover well-behaving partial solutions, resulting in faster convergence. Elitism may be generalized to size n, meaning the top n individuals are copied to the next generation. This may of course also lead to the evolved functions converging to local optima. Therefore, elitism is often chosen to be of size one.

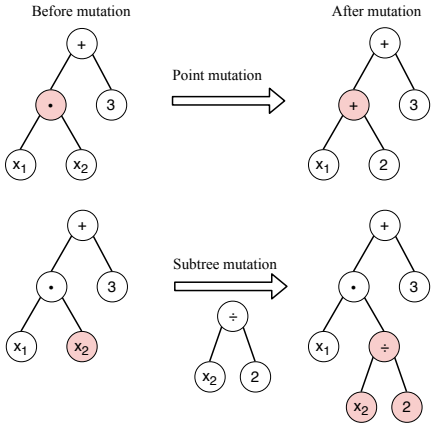


Figure 2.6: Examples of two different mutation schemes for tree-based chromosomes.

2.4 Path finding

A common problem within computer science is the *shortest path problem*. In this problem, we want to find a path between two vertices (s, t) in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that the sum of the weights of the edges in the path is minimized. If the graph is unweighted, we consider the all edges to have unit weight, and therefore search for the path with the least number of vertices.

There are a multitude of algorithms for calculating shortest paths in polynomial time, such as *Dijksta's algorithm*, the *Bellman-Ford algorithm*, the *Floyd-Warshall algorithm* and the *A*-algorithm*, which all can be found in most textbooks on algorithms and data-structures, such as Algorithms by Sedgewick [11]. The A^*-algorithm is a popular choice for pathfinding, and has a runtime complexity of $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}|\log|\mathcal{V}|)$. In short, the algorithm visits vertices one at a time, beginning with the starting vertex, and always chooses to explore the best option available. The next vertex v to explore is determined by a score

$$f(v) = g(v) + h(v) \tag{2.3}$$

where $g(v)$ is the distance from the starting vertex and $h(v)$ is the expected distance to the goal node (via a heuristic). One reason that keeps the runtime complexity down is the fact that the state space is bounded by the number of vertices and edges. As we can see in Figure 2.7, in a given state, we only need to consider the direct neighbors of the current vertex.

2.4.1 Multi-agent path finding

Consider the problem of finding shortest paths for multiple agents in the same graph. If these agents are allowed to share the same node, this problem is trivial: just solve these problems separately using polynomial shortest path algorithms. However, if collisions between agents have to be avoided, things get considerably more complicated.

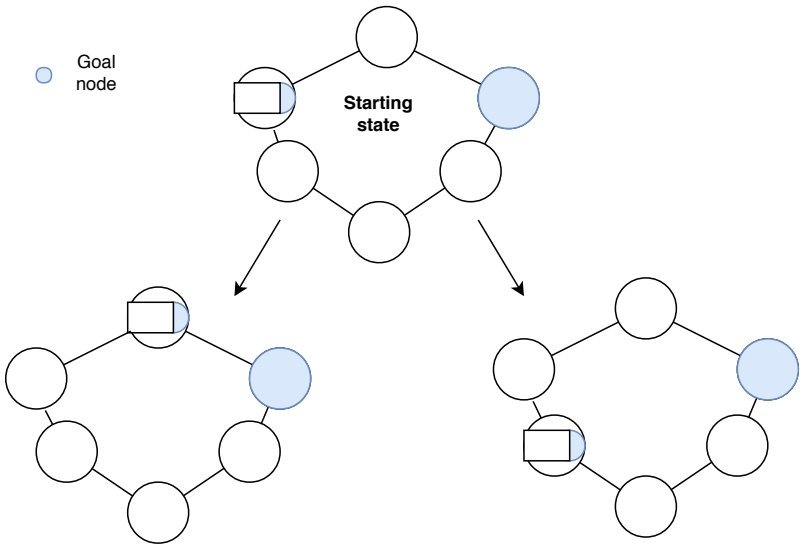


Figure 2.7: Possible next states for one iteration in a single agent pathfinding problem.

By considering all possible moves of all agents, the multi-agent path finding problem can be modelled as a shortest path problem on the *coupled graph* $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$. Given that we want to solve the multi-agent path finding problem with n agents, the vertices of $\mathcal{G}'$ represent state tuples on the form $(v_1, ..., v_n)$, which encode the location of all n agents. As can be seen in Figure 2.8, increasing the number of agents from one to two results in an increase from two to eight neighboring states (as in the single agent case in Figure 2.7).

The number of vertices in $\mathcal{G}'$ is

$$|\mathcal{V}'| = \frac{|\mathcal{V}|!}{(|\mathcal{V}| - |\mathcal{A}|)!}, \tag{2.4}$$

where $\mathcal{V}$ is the vertices in the original decoupled graph $\mathcal{G}$, and $\mathcal{A}$ is the set of agents. This number is derived from the number of possible permutations of $|\mathcal{A}|$ vertices from the set of $\mathcal{V}$.

In order to have a meaningful problem, the number of agents have to be in the range $[1, |\mathcal{V}| - 1]$. If we have $|\mathcal{A}| = 1$, then $|\mathcal{V}'| = |\mathcal{V}|$ and the problem reduces to the single agent path finding problem. On the other hand, if we have $|\mathcal{A}| = |\mathcal{V}| - 1$, then $|\mathcal{V}'| = |\mathcal{V}|!$. Hence, the number of vertices in $\mathcal{G}'$ is super-exponential in $|\mathcal{V}|$ with respect to both $|\mathcal{V}|$ and $|\mathcal{A}|$.

The number of edges in $\mathcal{G}'$ will potentially be even larger, with an upper bound at

$$|\mathcal{E}'| \leq \frac{|\mathcal{V}|!}{(|\mathcal{V}| - |\mathcal{A}|)!} * |E_{max}|^{|\mathcal{A}|}, \tag{2.5}$$

where E_{max} is the maximum number of neighbors any single vertex has in $\mathcal{G}$. While it is possible to run an ordinary shortest path algorithm on $\mathcal{G}'$, we observe by plugging in Equations (2.4) and (2.5) in the runtime complexity of A^* that this is computationally intractable.

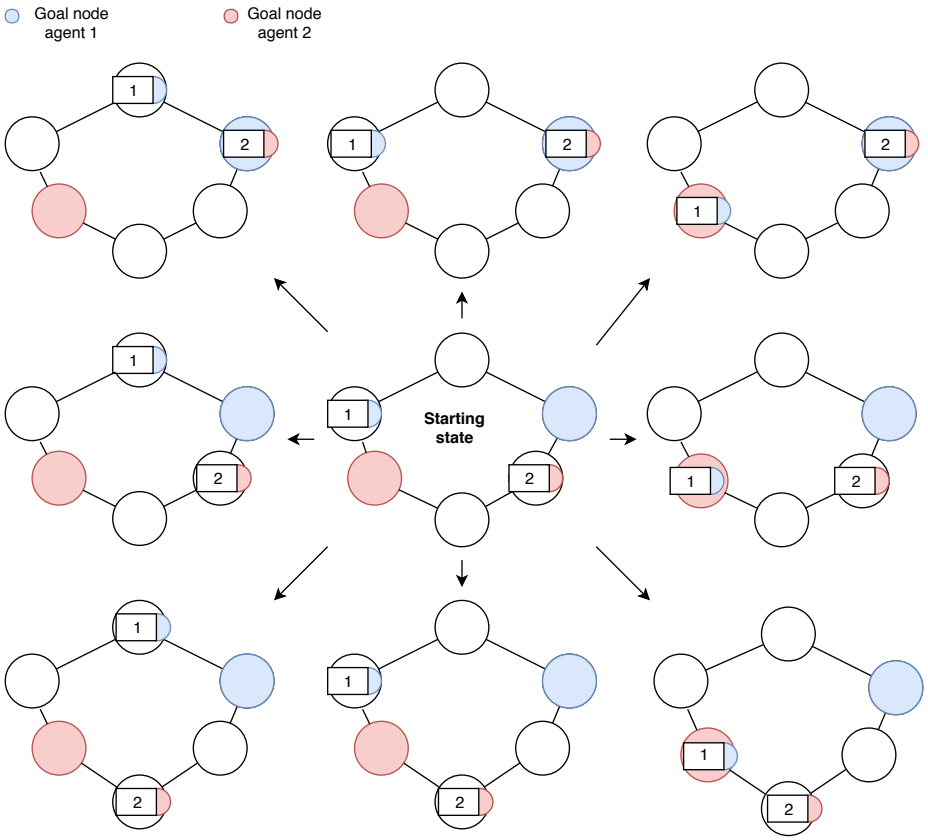


Figure 2.8: Possible next states for one iteration in a multi agent pathfinding problem.

2.4.1.1 M^*

An algorithm specifically tailored for the multi-agent path finding problem is M^*, developed by Wagner in [12]. This algorithm finds an optimal solution to the MAPF problem by letting agents optimistically follow their individually optimal paths until collisions are registered. As collisions are detected, a more thorough search is initiated for the involved agents.

Similarly to A^*, a priority queue *open* of states/vertices is used to store the states not yet visited. The priority queue orders the states in the same way as in A^*, i.e., using the score function f shown in Equation (2.3).

Each state (vertex in the coupled graph $\mathcal{G}'$) keeps track of the following variables:

- **collision set** - set of agents which will eventually collide in some successor state,
- **backpropagation set** - set of states that have considered the current state as a successor,
- **cost** - the cost of the current best path to the state, initialized to ∞,
- **back pointer** - a pointer to the previous state in the best path.

When a collision is registered in a state v, the collision information is propagated backwards to all states in the backpropagation set of v via a backpropagation function. The backpropagation function is then recursively called on all states in the backpropagation set, eventually propagating the information all the way to the starting state. When adding neighboring states to the priority queue, agents that are not in the collision set are forced to follow their individually optimal path, while agents in the collision set try every possible movement.

Because of this, M^* will in the worst case scenario have a superexponential runtime- and space complexity. However, in the average case, M^* should perform much better than A^* as significantly fewer states will be considered. Studies show [12] that for a 32×32 grid based map with 10 agents, median solution times were lower than 0.1 seconds for M^*, while A^* was incapable of solving a single problem instance within 10 minutes. Pseudo code for this algorithm is presented in Algorithm 2.

Algorithm 2 M^*

Require: Graph $\mathcal{G}$
Require: Starting state v_s
Require: Target state v_t
1: $v_s.cost \leftarrow 0$
2: *open* $\leftarrow$ Priority queue comparing state costs
3: *open*.enqueue(v_s)
4: **while** *open* **not** empty **do**
5: $v_{cur} \leftarrow$ *open.poll()*
6: **if** $v_{cur} == v_t$ **then**
7: **return** backtrack(v_{cur})
8: **end if**
9: **for** *neighbor* in limitedNeighbors(v_{cur}) **do**
10: add v_{cur} to *neighbor*.backSet
11: add collisions to *neighbor*.collisionSet
12: backpropagate(v_{cur}, *neighbor*.collisionSet, *open*)
13: **if** no collision **and** v_{cur}.cost + f($e_{cur,neighbor}$) $<$ *neighbor*.cost **then**
14: $neighbor.cost \leftarrow v_{cur}$.cost + f($e_{cur,neighbor}$)
15: $neighbor.backPointer \leftarrow v_{cur}$
16: *open*.enqueue(*neighbor*)
17: **end if**
18: **end for**
19: **end while**
20: **return** No path found

3
Methods

In section 3.1, we describe the main strategy for solving a planar PDP. Furthermore, we introduce a way for the agents to communicate with each other, extending the solution provided by van Lon, Branke and Holvoet in [4].

In section 3.2, we reformulate the problem for an industrial application. The setting of the problem is changed from a plane to an undirected graph, in which collisions between agents have to be avoided at all costs. The reformulation motivates changes in the route planning, parcel assignment, and data-set generation, which will be described in detail.

3.1 Solving a planar PDP

The theory items can be combined to create an auction system, where each agent takes the roll of a subcontractor bidding for requests as they are announced. Bids are calculated using a heuristic evolved by a genetic programming algorithm. This heuristic combines information regarding the state of the vehicle, parcel, and other agents using basic arithmetic and ternary operators.

As requests are auctioned and bids are won the respective vehicles update their list of assigned tasks, as well as optimize their current route so that the current tasks are handled efficiently. Fixing the route planner used by agents, the goal is to optimize the heuristic calculating bids.

This section will show the specifics of the genetic program, how routes are planned, and additional functionality added to the CNET protocol in the form of agents being able to reauction requests. This will also provide the basis of the multiple agent system used for the industrial application described later on in this chapter.

3.1.1 RinSim and ECJ

RinSim is the simulation environment used for simulating the multiple-agent systems described in this thesis. It is an open source, discrete time model simulator, built in Java by van Lon and Holvoet [13]. It has been used for similar projects, most notably *Optimizing agents with genetic programming* [4]. This project implements genetic programming evolution and CNET protocol bidding, and provides a baseline for our thesis.

The genetic programming evolution has been carried out through the use of *ECJ*, a Java based research system for evolutionary computation developed by

Function	Inputs x_i	Description
+,-,×,÷	2	Basic arithmetic functions
if4	4	If statement using 4 inputs which calculates $x_1 < x_2 \ ? \ x_3 : x_4$
pow	2	Calculates the exponent $x_1^{x_2}$
neg	1	Negates the input, $-x_1$
min, max	2	Calculates the minimum/maximum of inputs

Table 3.1: Possible internal nodes of a heuristic function.

Luke et al. [14]. ECJ allows for highly customizable genetic programming with a plethora of different parameters to adjust. In this thesis, we have chosen a tournament selection of size seven and limited the maximum depth of the heuristic trees to 17.

3.1.2 Parcel assignment

Every time a new parcel is announced, an auction is initiated. Each agent places a bid on the parcel using a heuristic that calculates how well suited the agent is for completing the order. When all agents have placed bids on the parcel, the agent that placed the lowest bid gets exclusive responsibility for completing the order. Because of this, it is important that the heuristic produces low values for orders that suits the agent, and high values for orders that would be impractical for the agent.

We employ the same approach as van Lon, Branke and Holvoet [4, 15], in which the heuristic is optimized through genetic programming. As described in section 2.3, the heuristic is represented as a tree. A leaf node either contains a constant value or a value related to the state of the agent. An internal node takes the values of its children as its input. To get the heuristic value for a given parcel, the root function of the tree is evaluated. Evaluation is carried out in a way similar to a post order traversal: evaluate all children before computing the function value at any node.

The functions that are available to the optimization algorithm are displayed in table 3.1, and the set of possible leaf nodes used in [4] are shown in table 3.2.

3.1.3 Introducing communication to multiple-agent systems

Agents could benefit from having information about the intentions of other agents. For instance, if some agents are heading to locations near a newly announced parcel, it could be a good idea to let one of these agents win the auction. Similarly, an agent that knows that all other agents are far away from a newly announced parcel could be the best candidate for servicing this parcel. However, in previous work [3, 4] the state variables used in the optimization were limited to values that are either local to agents, such as *insertion cost*, or local to parcels, such as *pickup urgency*.

In order for the agents to learn how to utilize the states of other agents, we introduce four leaf nodes to the genetic programming algorithm. The reason for not adding more new node types is to not increase the complexity of the problem too much; the search space of the genetic algorithm grows exponentially with respect

Function	Description
Insertion cost	Additional length of route
Insertion travel time	Additional travel time
Insertion tardiness	Expected increase in tardiness, i.e. delayed pickups and deliveries
Insertion over time	Expected increase in overtime, i.e. how long until the vehicle drives back to the depot
Insertion flexibility	Summation of the differences between earliest and last possible arrival times for all pickups and deliveries in the route.
Ado, Mido, Mado	Average, minimum and maximum distance between locations in the current route and the parcel considered for bidding
Pickup urgency	Time until end of pickup time window
Delivery urgency	Time until end of delivery time window
Time left	Time left until end of scenario
Slack	Expected idle time until end of day
Route length	Number of orders in current route
Constants 0,1,2,10	

Table 3.2: The possible leaf node of a bidding heuristic used in [4].

to the number of possible nodes [16]. The new leaf nodes, which are adaptions of those presented by Vonolfen et al. [17], are presented in table 3.3, accompanied by Figure 3.1 which illustrates how the node *common target* works.

Function	Description
Common target	The number of agents that are currently moving towards a destination that is at most r length units away.
Aad, Miad, Maad	Average, minimum and maximum distance from the parcel to all other agents.

Table 3.3: Leaf nodes that uses the states of other agents.

3.1.4 Reauctioning parcels

In a dynamic problem where requests are invisible up until their announce time a_i, decisions made previous to the announcement of an order may turn out to be suboptimal as new information is revealed. Since the nature of the problem is dynamic the solution also needs to be dynamic. This can be achieved by allowing agents to reauction parcels if a better schedule exists.

Similarly to the procedure in [4], we define two situations in which an agent considers reauctioning a parcel:

- the agent has not won an auction in the last five minutes,

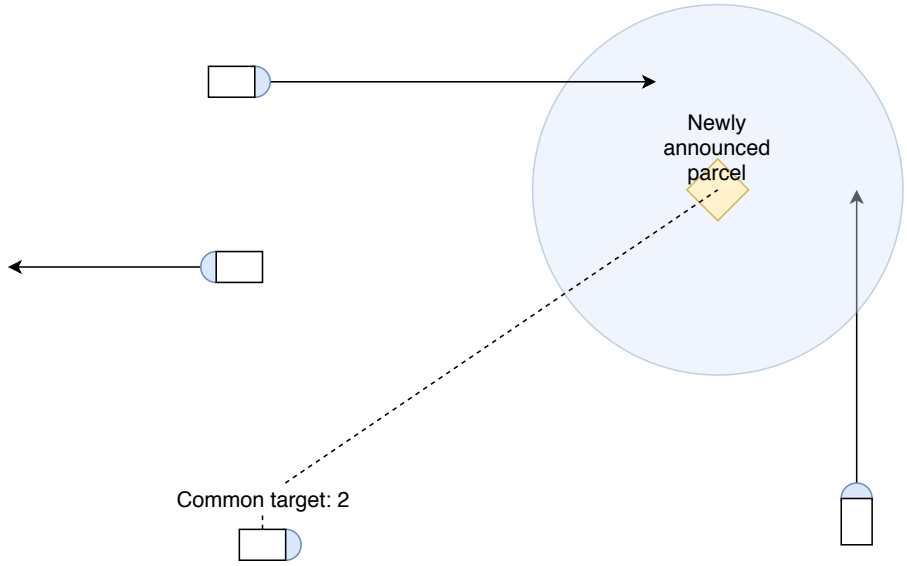


Figure 3.1: Visualization of the *common target* node used by bidding heuristics. The node calculates the number of agents that are heading towards points that are close to the newly announced parcel. For the bottom most agent considering the parcel in the top right, this value is evaluated to 2, as there are two other agents that are moving towards destinations that are close to the parcel. The maximum distance that is considered close is a hyper-parameter which is set before training.

- the agent has an updated schedule.

When a reauction is initiated, the agent has to decide on what parcel to put up for auctioning. The bidding heuristic is evaluated for each parcel and the parcel that is the least valuable to the agent at the given time is selected for reauction.

3.1.5 Route planning

The next problem to address is route planning: given a set of parcels, in what order should the parcels be processed and how should the agents move to accomplish this? Since agents are residing in the plane, movement is not an issue as all agents can move in a straight line to any position. To determine the order in which parcels are processed the *Cheapest Insertion* algorithm is utilized, where each time a parcel is assigned to an agent, it is inserted in the schedule such that the expected cost of the new schedule is as small as possible.

3.1.6 Training data

In order to evolve and optimize the bidding heuristic, several simulations have to be evaluated. This requires a large amount of training data in the form of *scenarios*. A scenario $S(d, u, s)$ is characterized by a (dynamism, urgency, scale)-tuple and contains vehicle information and parcel information. Vehicles are defined by their starting time and position, whereas parcels are defined by their pickup- and delivery locations together with corresponding time windows.

Name	Dynamism	Urgency [min]	Scale	Individuals	Generations	Evaluations
DCGP-20-35-1	20%	35	1	500	100	50
DCGP-50-20-1	50%	20	1	500	100	50
DCGP-80-5-1	80%	5	1	500	100	50

Table 3.4: The data sets used to evolve bidding heuristics. *Evaluations* correspond to the number of evaluation for a single individual in a generation. DCGP is an abbreviation for *decentralized communication genetic program.*

The data set that has been used for the optimization of the planar PDP system is the same as the one that van Lon, Holvoet, and Branke used in [4–6]. This data set consists of a training set and a validation set.

The training set $\mathbf{S}_{train}$ contains scenarios with three different combinations of dynamism, urgency and scale, and was used to train three different systems. These systems with their corresponding parameters are shown in table 3.4

The validation data set $\mathbf{S}_{val}$ contains 270 scenarios; 10 scenarios from 27 different parameter configurations and is on the form

$$\mathbf{S}_{val} = \{S(d,u,s)_i | (d,u,s) \in D \times U \times S, i \in [0,10]\},$$

where

$$\begin{aligned} D &= \{20,50,80\} - \text{set of dynamism levels,} \\ U &= \{5,20,35\} - \text{set of urgency levels,} \\ S &= \{1,5,10\} - \text{set of scale levels.} \end{aligned}$$

The best performing individual in the last generation of each evolution was chosen as the representative heuristics. The validation data set was then used to evaluate representative heuristics for different parameter settings.

3.2 Industrial application

This section will describe the industrial application part of the thesis, which has been the primary focus of the work. The system layout is shown in Figure 3.2 and the individual components will be discussed in the following sections.

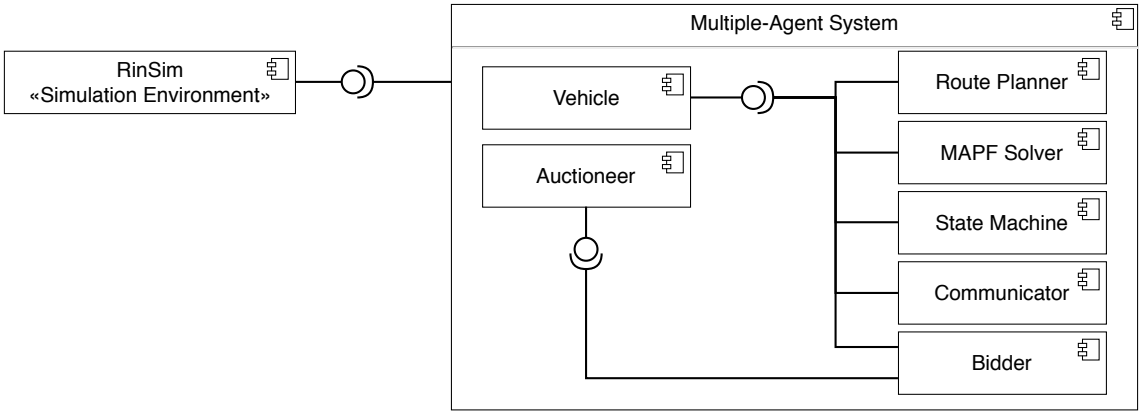


Figure 3.2: System layout for the industrial application.

3.2.1 Problem redefinition

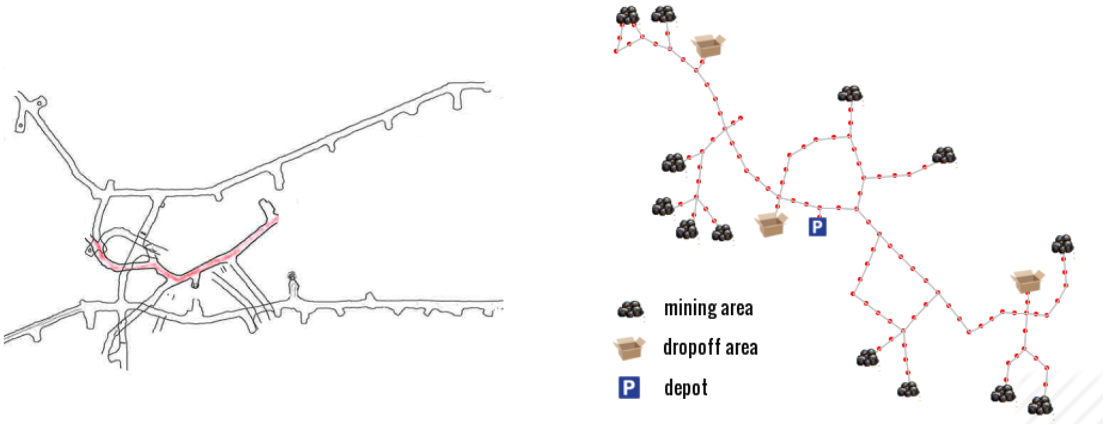
Going from a planar map with no physical limitations to a real world mining operation drastically changes the problem and introduces new constraints and assumptions.

3.2.1.1 Topology

The most significant change from the planar setting is the introduction of a mining environment. Mines typically span large underground spaces using a few long corridors which branch off into mining areas and delivery points, as can be seen in Figure 3.3a. The corridors are often narrow and do not support vehicles passing each other. This means that trucks might not be able to follow their individually optimal routes as this may set them on a collision course with other vehicles.

To account for the aforementioned changes, the map is modelled as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. The set of vertices is itself split into $\mathcal{V} = (\mathcal{V}_C, \mathcal{V}_U)$, where $\mathcal{V}_C$ is a set containing *constrained vertices* and $\mathcal{V}_U$ is a non-empty set containing *unconstrained vertices*. Constrained vertices are defined as vertices that can only be occupied by one vehicle at a time, whereas any number of vehicles are allowed to be in the same unconstrained vertex simultaneously. Unconstrained vertices represent in our case locations such as mining areas, delivery points and central hubs, while constrained vertices represent locations in narrow passageways.

A graph based map was created to model a typical mine, which is displayed in Figure 3.3b. The graph contains one central depot, 13 pickup locations and three dropoff locations, which are the vertices in the unconstrained set. All other vertices (the red dots in Figure 3.3b) are constrained vertices.



(a) Layout of a typical mine.

(b) The graph used for the simulations.

Figure 3.3: Layout of a real world mine and the model used for the thesis.

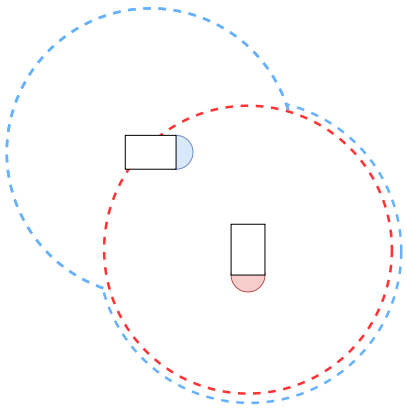


Figure 3.4: By relaying messages from the blue vehicle its communication range is extended to include that of the red vehicle.

3.2.1.2 Limited communication range

Trucks were equipped with communication devices to able to communicate their positions and intentions with each other. Since a mining environment features a lot of obstacles for messages to travel through, the devices were modelled to have a limited range of a few hundred meters. To improve intention spreading, communication devices were also set to relay known information to other vehicles. This process is depicted in Figure 3.4.

3.2.1.3 Agent capacity

In a real world setting, it is not realistic that agents are able to carry an infinite amount of parcels. Picking up a parcel is in our case analog to loading a truck with ore, which motivates restricting the cargo capacity of agents. We have set the cargo capacity to one, meaning agents have to complete an entire order before being able to process another. Note that agents are still able modify their schedule with respect to parcels that are not yet picked up.

3.2.1.4 Service queuing

In a mining operation, trucks typically travel from a central hub to an ore pickup destination where a wheel loader is waiting, ready to fill the truck to capacity with ore. The truck then travels back to the central hub, depositing the cargo in a crusher, which reduces the ore into smaller chunks. Thus, both the pickup and delivery locations may serve a limited amount of trucks at a time. This behaviour can be modelled by placing first in first out queues at each service location, where only vehicles in the front of the queue are serviced.

3.2.2 Shared space route planning

By introducing a shared space environment, routing conflicts become an issue. In a centralized system, an optimized route could be calculated each time the problem changes since we would then have access to complete information. However, with a decentralized system, as well as limited visibility of other agents, route optimization and conflict avoidance need to be handled separately as it is impossible to know at which points during a route conflicts may emerge.

Thus, route planning is divided into two parts:

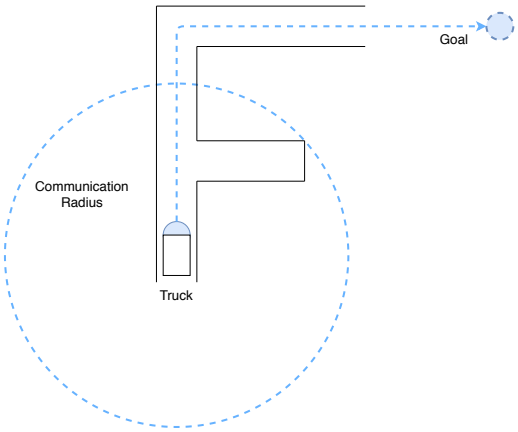
- Global route planning: Given a schedule of pickups and deliveries, find a route that minimizes the objective function 2.1 assuming that shared space is not an issue.
- Local route planning: Given a routing conflict, find routes for all involved vehicles such that they are able to reach their respective destinations without colliding.

Global route planning was solved using cheapest insertion cost, which inserts the new request such that it minimizes the objective function in polynomial time. Local route planning requires more thought.

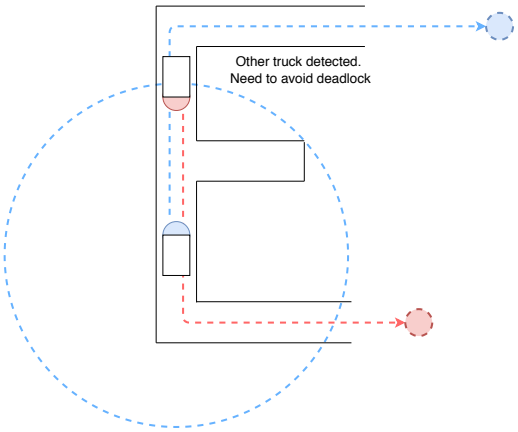
3.2.2.1 Local route planning

When an upcoming collision is detected, as exemplified in Figure 3.5, a MAPF-algorithm is executed with the current assignments of the agents in the vicinity of the collision area as input. If an idle agent is involved in a collision, its goal will be to remain at its current vertex. After a solution has been found, each involved agent will receive an updated route to follow. The algorithm used in this project was chosen to be M^*, which is described in Section 2.4.1.1.

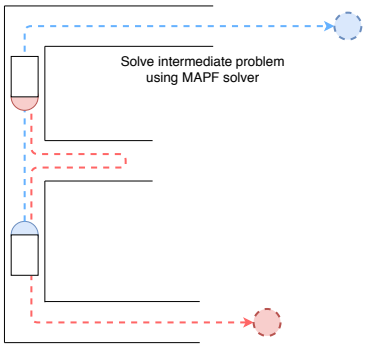
Agents in conflict will either be moving towards an unconstrained vertex or be idle at a constrained vertex. The only time an agent will have a constrained vertex as its target is when the agent blocks the way for another agent heading towards an unconstrained vertex. Therefore, there will never be any MAPF-problem instances where two agents want to move to the same constrained vertex. As the set of unconstrained vertices $\mathcal{V}_U$ is guaranteed to be non-empty, any given problem instance will be solvable.



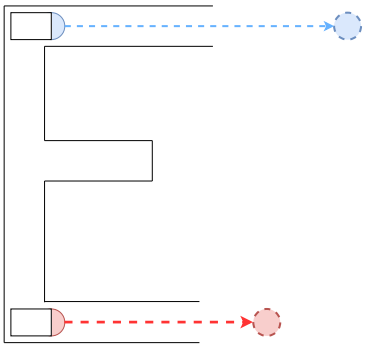
(a) Truck travelling towards a pickup or delivery destination.



(b) Communication picks up another truck travelling in the opposite direction.



(c) To avoid deadlock, a partial solution is found.



(d) The routing conflict has been resolved.

Figure 3.5: Routing conflict with two vehicles.

3.2.2.2 Unweighted edges and map resolution

Examining the mining map 3.3a, we find that if the graph only includes vertices at junctions and dead ends the length of edges will have widely different proportions. Such edges could range from pockets, barely able to fit a vehicle, to tunnels hundreds of meters long. This would incur three major problems:

1. Applying a MAPF solver on such an instance would give the algorithm little room for maneuverability. Where in reality an edge between two vertices might fit a large number of vehicles there is instead only room for one vehicle.
2. Solutions provided by M^* show a set of state transitions, where all agents move from one vertex to the next in unison. This means agents have to move synchronously from vertex to vertex in order for no collisions to occur. Thus vehicle travel times are limited by the longest edge travelled by a vehicle in each iteration.
3. Without extensive modification to the simulator, vehicles in RinSim have to strictly follow an edge once they have started traversing it. Since M^* needs unique starting positions for each vehicle, a collision has to be detected before involved vehicles are travelling toward the same vertex, as this is the vertex given to M^*.

As a solution, the graph was extended with local vertices, as shown in Figure 3.6. These local vertices were added so that no edge is longer than half of the vehicles communication range, meaning there will always be two vertices between vehicles when they register a routing conflict. This also applies a limit to the longest traversable edge. The additional vertices will not add much in the way of maneuverability however, as pocket edges already fulfill the criteria of being at most half the range of the vehicles' communication range. The reason for not increasing the graph resolution further was due to the heavy penalty it would incur to the worst case complexity of M^*, which is discussed in Section 2.4.1.

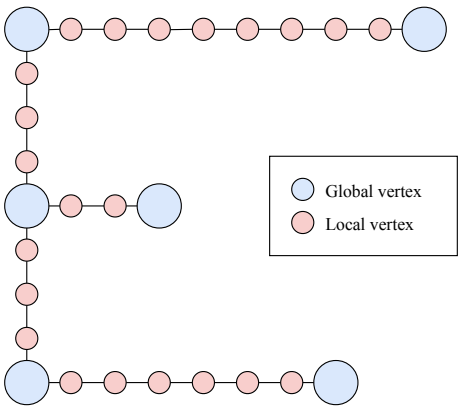


Figure 3.6: Graph expanded with local vertices, which is useful for multi-agent path finding solution quality.

3.2.3 Vehicle behaviour

As with any decentralized MAS, the performance of the system relies heavily on the perception and decision making of its agents. Vehicles have access to their own

Variable	Description
Assignment Version	Integer that is incremented each time the vehicles changes state, i.e. whenever its assignment is updated.
Current Assignment	Current position and destination. This, along with nearby vehicles assignments gets passed to the MAPF solver whenever a conflict emerges.
Time Map	Mapping between vertices in the graph and time windows for which the vehicle expects to occupy the vertex.
Collision Map	Stored points of collision between current vehicle and nearby vehicles. Look-ups use the involved vehicles current assignment versions.

Table 3.5: Long term memory variables.

Variable	Description
Vehicle positions	Mapping between nearby vehicles' identification number and their respective positions
Vehicle destinations	Mapping between nearby vehicles' identification number and their respective destinations

Table 3.6: Short term memory variables.

position as well as the position and intent of other nearby vehicles using vehicle-to-vehicle communication. To modularize the behaviour of vehicles they were modeled as state machines. To make informed decisions, based on the vehicle's current assignment as well as the intent of other vehicles, each vehicle calculates the time windows for which they expect to occupy each vertex in their path.

3.2.3.1 Communication

Vehicles share information by broadcasting and receiving asynchronous messages within their communication range. These messages contain the vehicles' current position, assignment, time map, and state. Whenever a vehicle receives a message it also broadcasts it to other nearby vehicles, allowing information sharing as described in Section 3.2.1.2. To prevent an overflow of messages being passed around, only the most recently created message from each vehicle is relayed during an update tick.

3.2.3.2 Long term and short term memory

To be able to detect and solve routing conflicts, vehicles need to store information regarding their current state as well information pertaining to other nearby vehicles. Some of this information needs to be updated continuously, such as vehicle positions, while other information remains constant until a destination is reached or the vehicles route is disrupted. Thus stored information is divided into long and short term memory. The stored variables are shown in Tables 3.5 and Table 3.6.

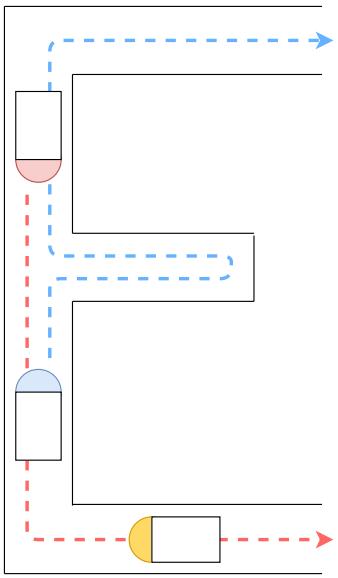


Figure 3.7: Route conflict with more than two vehicles. The blue and red vehicle will register that they are on a collision course and enter the collision avoidance state. Should the MAPF solver only consider the red and blue vehicle and decide that the blue vehicle should enter the pocket which only fits one vehicle, the subsequent collision situation between the red and yellow vehicle will either require an unnecessarily convoluted solution or be unsolvable.

3.2.3.3 Collision detection and avoidance

When another vehicle is encountered a comparison of time maps is made. If the trajectories of both vehicles overlap during some time interval we expect an imminent collision, unless vehicles reroute, and the collision point is stored along with the vehicle's identification number in a collision map. If a vehicle is within collision range, a check is made to see if its identification number is contained within the collision map. If so, both vehicles will stop moving and enter a collision avoidance state. Once a MAPF solution has been obtained the vehicles start moving again and the situation is resolved.

When vehicles enter the collision state they also signal other nearby vehicles to enter this state, ensuring that they will be part of the MAPF solution. This is done to make sure the routes received from the MAPF solver does not interfere with routes of nearby vehicles. A situation where this is necessary is presented in Figure 3.7. Including as many vehicles as possible in a conflict resolution minimizes the travel time in the system and the amount of unsolvable conflict scenarios. However, as the worst case time complexity of M^* is superexponentially increasing with the number of agents involved, as shown in Equation (2.4), the execution time may be heavily increased.

3.2.3.4 State machine

Vehicles were modelled as finite state machines with states and transition events as described in Table 3.7. The different states contain the following logic:
Wait: Stand still until a parcel is assigned to the vehicle or a conflict emerges nearby.
Goto: Move towards the destination provided by the global route planner. Should a new parcel be assigned to the vehicle this may create a more beneficial route, or assigned parcels might be reauctioned, leaving the vehicle with no assigned parcels. If a conflict emerges nearby, then an intermediate path planning instance needs to be solved.

State	Event	Next State
Wait	GOTO	Goto
	CONFLICT	CollisionAvoidance
Goto	NOGO	Wait
	ARRIVED	WaitAtService
	REROUTE	Goto
	CONFLICT	CollisionAvoidance
WaitAtService	REROUTE	Goto
	NOGO	Wait
	READY_TO_SERVICE	Service
Service	DONE	Wait
CollisionAvoidance	RESOLVED_WAIT	Wait
	RESOLVED_GOTO	Goto
	CONFLICT	CollisionAvoidance

Table 3.7: Transition table for the finite state machine in charge of vehicle logic.

WaitAtService: Wait at a service location until the pickup or delivery is available, as specified by p_i^L and d_i^L, and the vehicle is first in queue. Should the assigned parcel be reauctioned then wait for a new assignment. If the route planner finds a more beneficial route then move towards the provided parcel.
Service: Stand still while cargo is picked up or delivered.
CollisionAvoidance: Solve a MAPF instance as described in Section 3.2.2.1. If the vehicle reaches its destination then return to the previous state. If a new conflict emerges nearby then solve the new MAPF instance.

Note that neither of the two states **WaitAtService** and **Service** contain a transition to collision avoidance. This is because a vehicle may only be in one of these states if it is at a service location. Since service locations always belong to the unconstrained set $\mathcal{V}_U$, no collision will occur on these vertices.

3.2.4 Optimization and system evaluation

The system described above, using collision detection and the M^* algorithm for solving MAPF instances, was evolved by having a genetic program optimize the bidding heuristic. To generate training and validation data sets a scenario generator was developed, based on the generator described in [6], with additional support for graph maps. To create the graph used for training and evaluation a script was written in Matlab with capabilities to draw vertices and edges, along with placement of depot, pickup and delivery locations. The resulting graph used for both training and validation is shown in Figure 3.3b.

For heuristic evolution to be successful large amounts of computational power was needed. To this end, a computer cluster built on Intel 2650v3 CPUs consisting of 5 vertices with 20 cores each was used for training.

Name	Dynamism	Urgency [min]	Scale	Individuals	Generations	Evaluations
DGGP-20-35-1	20%	35	1	400	50	50
DGGP-50-20-1	50%	20	1	400	50	50
DGGP-80-10-1	80%	10	1	400	50	50
DGGP-mixed	20/50/80	35/20/10	1	400	50	54

Table 3.8: Scenario and genetic programming parameters used to evolve solutions for shared space PDPs. DGGP is an abbreviation for *decentralized graph genetic program*.

3.2.4.1 Limiting evolution runtime

Since the running time of M^* is superexponential in the number of agents, evaluations requiring an extraordinary long time due to state space explosion is to be expected. As genetic algorithm evolution requires large amounts of evaluations, minute long runtimes are simply not feasible.

Therefore we had to find a way to limit the running time for each scenario evaluation. Tuning test showed that 99% of scenarios using five vehicles were evaluated within two seconds. The remaining scenarios could potentially require up to several minutes, and so evaluation runtime was limited by setting a cap on the number of states explored by M^* to 70 000. Any scenario evaluation exceeding this limit was stopped prematurely and assigned minimum fitness.

3.2.4.2 Training data

Similar to the process described in Section 3.1.6, three different training data sets $\mathbf{S}_{train}$ were generated for heuristic evolution. These data sets consist of scenarios $S(\mathbf{d,u,s})$, defined by parameters **d**ynamism, **u**rgency and **s**cale. Three heuristic, referred to as **DGGP**, was optimized for each of these data sets. A forth heuristic, named **DGGP-mixed**, was evolved using a mixture of the three training data sets.

Scenario parameters along with genetic programming hyperparameters for each training set are shown in Table 3.8. Scenario length was set to four hours, with five vehicles and 40 orders per scale. Pickup and delivery locations for orders were uniformly randomly distributed among the mining and dropoff areas shown in Figure 3.3b.

Training data sets contain 2500 scenarios each, enabling the genetic program to evaluate its population on 50 new scenarios each generation for 50 generations. Using new scenarios in each generation ensures the evolved heuristics are not overfitted for a small data set.

3.2.4.3 Validation data

Since increasing the scale parameter slowed down scenario evaluation to unreasonable runtimes, multiple validation sets were created. The first validation set was used to measure the evolved heuristic **performance** for varying levels of dynamism and urgency. The other validation data sets were used to measure the systems **failure rate** for increasing levels of scale.

In a similar vein as the validation set used for the planar PDP heuristics, the first validation data set $\mathbf{S}_{val,1}$ was created containing 300 scenarios; 20 scenarios from 15 different parameter configurations of dynamism and urgency and is expressed by

$$\mathbf{S}_{val,1} = \{S(d,u,s)_i | (d,u,s) \in D \times U \times S, i \in [0,20]\}, \tag{3.1}$$

where

$$D = \{20,50,80\} - \text{set of dynamism levels},$$
$$U = \{10,20,35\} - \text{set of urgency levels},$$
$$S = \{1\} - \text{one scale level}.$$

The second validation set $\mathbf{S}_{val,2}$ consists of 10 000 scenarios; 1000 scenarios generated from 10 different parameter configurations of varying scale, on the form

$$\mathbf{S}_{val,2} = \{S(d,u,s)_i | (d,u,s) \in D \times U \times S, i \in [0,1000]\}, \tag{3.2}$$

where

$$D = \{50\} - \text{one dynamism level},$$
$$U = \{20\} - \text{one urgency level},$$
$$S = \{0.2,0.4,0.6,0.8,1.0,1.2,1.4,1.6,1.8,2,0\} - \text{set of scale levels}.$$

where the values $s \in S$ translates to $5 \cdot s$ vehicles and $40 \cdot s$ orders.

3.2.4.4 M* runtime analysis

To measure the runtime of M^*, nine data sets of routing conflict instances were generated. Each data set contains 1000 instances, with the number of agents ranging from two to ten. Each instance was generated by first inserting an agent on a randomly selected vertex. To get the desired number of agents, new agents were inserted on randomly selected vertices within the visibility range of previously placed agents. Agent destinations were then chosen among the vertices in the unconstrained set $\mathcal{V}_U$. Problem instances were then solved by M^* and runtimes up to one minute were recorded.

4
Results

In this chapter we present the main results of the thesis. Section 4.1 covers the results from adding communication nodes to the genetic programming algorithm in the planar PDP. Section 4.2 presents training- and evaluation results from the shared space PDP, as well as the results from the scalability test and M^* runtime measurements.

4.1 Introducing communication to the planar PDP

After implementing communication terminals to the planar PDP solution, bidding heuristics were optimized using genetic programming, one for each of the three training sets containing scenarios of varying dynamism and urgency, shown in Table 3.4. The training took approximately 88 hours for one parameter setting (500 individuals, 100 generations and 50 evaluations per individual in each generation) and was carried out on a computer with an Intel Core i7-6950X hyper-threaded CPU with 10 cores at 3.00 GHz.

To get the desired heuristics for validation, the best performing individual in the last generation of each training was chosen. The chosen heuristics were then compared to those not using communication terminals. The validation data set and genetic algorithm parameters were the same as those used by van Lon et al., [4].

4.1.1 Training performance

Figure 4.1 shows the objective value cost per parcel for the best individual in each generation for all three heuristic evolutions. For each training set, the best performing individual of the last generation where chosen as the representative heuristic. We can see a significant decrease in the cost function during the earlier generations, after which the overall trend of cost function seems to start plateauing. Note that the trend is still slowly decreasing, which indicates that the training may not have fully converged.

4.1.2 Comparison with previous results

A comparison with the heuristics evolved in *Optimizing agents with genetic programming* [4] are shown in Table 4.1. The results produced by van Lon et al., named **DGP**, are average results for ten evolution runs of each configuration, while the results presented for DCGP are only for one evolutionary run.

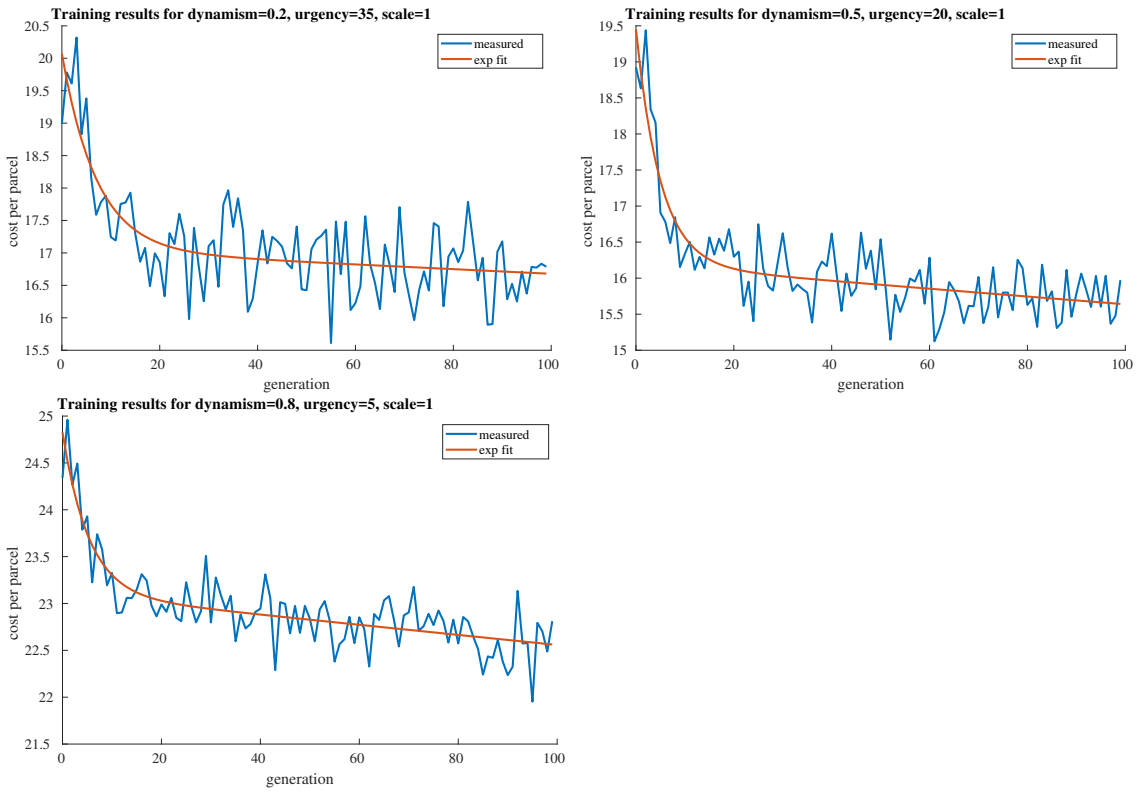


Figure 4.1: System performance during training of planar PDP solution. Note that the heuristic evolution uses new scenarios for each generation, meaning that a well performing individual is not guaranteed to have the same success in the next generation, resulting in the jaggedness of the graphs.

Class	DGP-20-35-1	DCGP-20-35-1	DGP-50-20-1	DCGP-50-20-1	DGP-80-5-1	DCGP-80-5-1	Best
20-5-1	$^{6}31.03 \pm 13.87$	$^{2}25.48$	$^{5}26.48 \pm 1.17$	$^{1}\mathbf{25.42}$	$^{3}25.75 \pm 0.58$	$^{4}25.98$	DCGP-50-20-1
50-5-1	$^{6}25.84 \pm 8.49$	$^{1}\mathbf{21.31}$	$^{5}22.90 \pm 1.46$	$^{4}21.68$	$^{3}21.67 \pm 0.37$	$^{2}21.59$	DCGP-20-35-1
80-5-1	$^{6}25.97 \pm 10.07$	$^{1}\mathbf{21.27}$	$^{5}22.66 \pm 1.26$	$^{4}22.02$	$^{2}21.28 \pm 0.38$	$^{3}21.50$	DCGP-20-35-1
20-20-1	$^{5}18.99 \pm 0.36$	$^{2}18.70$	$^{3}18.71 \pm 0.30$	$^{1}\mathbf{18.12}$	$^{6}19.15 \pm 0.33$	$^{4}18.88$	DCGP-50-20-1
50-20-1	$^{4}15.27 \pm 0.38$	$^{2}15.17$	$^{3}15.22 \pm 0.24$	$^{1}\mathbf{14.68}$	$^{5}15.51 \pm 0.68$	$^{6}15.77$	DCGP-50-20-1
80-20-1	$^{5}15.02 \pm 0.34$	$^{2}14.73$	$^{3}14.80 \pm 0.16$	$^{1}\mathbf{14.34}$	$^{6}15.34 \pm 0.49$	$^{4}14.97$	DCGP-50-20-1
20-35-1	$^{5}16.48 \pm 0.34$	$^{1}\mathbf{15.68}$	$^{3}16.37 \pm 0.28$	$^{2}15.76$	$^{6}17.02 \pm 0.82$	$^{4}16.45$	DCGP-20-35-1
50-35-1	$^{3}14.44 \pm 0.45$	$^{2}14.41$	$^{4}14.54 \pm 0.32$	$^{1}\mathbf{13.73}$	$^{5}14.96 \pm 0.46$	$^{6}15.07$	DCGP-50-20-1
80-35-1	$^{4}13.74 \pm 0.24$	$^{2}13.27$	$^{3}13.61 \pm 0.18$	$^{1}\mathbf{13.01}$	$^{5}14.18 \pm 0.55$	$^{6}14.49$	DCGP-50-20-1
20-5-5	$^{6}22.22 \pm 9.25$	$^{1}\mathbf{17.14}$	$^{5}19.10 \pm 2.52$	$^{2}17.66$	$^{3}17.78 \pm 0.23$	$^{4}17.80$	DCGP-20-35-1
50-5-5	$^{6}18.62 \pm 6.24$	$^{1}\mathbf{14.35}$	$^{5}16.01 \pm 1.83$	$^{4}14.87$	$^{3}14.76 \pm 0.18$	$^{2}14.74$	DCGP-20-35-1
80-5-5	$^{6}18.49 \pm 6.16$	$^{1}\mathbf{14.39}$	$^{5}15.91 \pm 1.72$	$^{4}14.87$	$^{2}14.66 \pm 0.16$	$^{3}14.68$	DCGP-20-35-1
20-20-5	$^{4}13.94 \pm 0.30$	$^{3}13.84$	$^{2}13.83 \pm 0.14$	$^{1}\mathbf{13.38}$	$^{5}14.69 \pm 0.73$	$^{6}14.71$	DCGP-50-20-1
50-20-5	$^{4}9.76 \pm 0.19$	$^{2}9.34$	$^{3}9.50 \pm 0.12$	$^{1}\mathbf{9.09}$	$^{6}10.30 \pm 0.73$	$^{5}10.07$	DCGP-50-20-1
80-20-5	$^{4}10.08 \pm 0.24$	$^{2}9.52$	$^{3}9.82 \pm 0.16$	$^{1}\mathbf{9.50}$	$^{6}10.61 \pm 0.73$	$^{5}10.47$	DCGP-50-20-1
20-35-5	$^{3}11.08 \pm 0.18$	$^{2}10.96$	$^{4}11.29 \pm 0.19$	$^{1}\mathbf{10.64}$	$^{5}11.94 \pm 0.66$	$^{6}12.54$	DCGP-50-20-1
50-35-5	$^{3}8.88 \pm 0.20$	$^{2}8.71$	$^{4}8.97 \pm 0.25$	$^{1}\mathbf{8.46}$	$^{5}9.72 \pm 0.67$	$^{6}10.16$	DCGP-50-20-1
80-35-5	$^{3}9.10 \pm 0.25$	$^{2}8.97$	$^{4}9.25 \pm 0.22$	$^{1}\mathbf{8.61}$	$^{5}9.92 \pm 0.61$	$^{6}10.22$	DCGP-50-20-1
20-5-10	$^{6}20.17 \pm 7.78$	$^{1}\mathbf{15.49}$	$^{5}17.16 \pm 2.67$	$^{2}15.80$	$^{3}15.86 \pm 0.14$	$^{4}15.99$	DCGP-20-35-1
50-5-10	$^{6}16.83 \pm 6.56$	$^{1}\mathbf{12.44}$	$^{5}13.92 \pm 1.83$	$^{4}13.15$	$^{3}12.84 \pm 0.18$	$^{2}12.75$	DCGP-20-35-1
80-5-10	$^{6}16.52 \pm 5.57$	$^{1}\mathbf{12.74}$	$^{5}14.10 \pm 1.83$	$^{4}13.25$	$^{2}12.90 \pm 0.16$	$^{3}13.00$	DCGP-20-35-1
20-20-10	$^{3}10.76 \pm 0.22$	$^{2}10.66$	$^{4}10.78 \pm 0.12$	$^{1}\mathbf{10.56}$	$^{5}11.55 \pm 0.72$	$^{6}11.77$	DCGP-50-20-1
50-20-10	$^{4}8.80 \pm 0.25$	$^{2}8.51$	$^{3}8.59 \pm 0.07$	$^{1}\mathbf{8.26}$	$^{6}9.48 \pm 0.81$	$^{5}9.38$	DCGP-50-20-1
80-20-10	$^{4}8.71 \pm 0.25$	$^{2}8.42$	$^{3}8.49 \pm 0.11$	$^{1}\mathbf{8.23}$	$^{6}9.35 \pm 0.76$	$^{5}9.21$	DCGP-50-20-1
20-35-10	$^{3}9.16 \pm 0.19$	$^{2}9.13$	$^{4}9.44 \pm 0.31$	$^{1}\mathbf{8.93}$	$^{5}10.07 \pm 0.64$	$^{6}10.62$	DCGP-50-20-1
50-35-10	$^{3}7.84 \pm 0.20$	$^{2}7.77$	$^{4}8.06 \pm 0.41$	$^{1}\mathbf{7.63}$	$^{5}8.74 \pm 0.70$	$^{6}9.53$	DCGP-50-20-1
80-35-10	$^{3}7.77 \pm 0.19$	$^{2}7.61$	$^{4}7.98 \pm 0.39$	$^{1}\mathbf{7.50}$	$^{5}8.58 \pm 0.68$	$^{6}9.38$	DCGP-50-20-1

Table 4.1: Comparison between evolved heuristics with and without communication terminals. The listed values are mean cost per parcel. Values for the heuristics named **DGP** are averages of 10 evolutionary runs, while values for those named **DCGP**, which use communication terminals, are only for one evolutionary run. Since DGP values are taken from multiple runs deviations are also reported for these results. Superscript values are the ranking for each different validation setting.

4.2 Evolving solutions for graph based PDP

The results for the solution to the shared space PDP will primarily focus on how well the evolved heuristics perform versus a basic heuristic. For this purpose *Insertion cost* and *Route size* will be used as baseline heuristics to be compared with heuristics evolved using different sets of training data.

Since the problem is considered safety critical, the success rate of simulations will also be an important metric. The most important factor influencing the success rate is the complexity of problems given to M^*. Since the algorithm has an superexponential time bound, it may very well encounter problems which are not solvable within a reasonable time. To this end, both success rate of simulations as well as runtime evaluations of M^* for different numbers of agents will be quantified.

4.2.1 Training results

To enable heuristic evolution three training data sets were generated. Each data set contains scenarios of a specific configuration of dynamism and urgency as specified in Table 3.8. A fourth heuristic was also evolved using a mix of the scenarios in the training data sets.

The training was conducted on a computer cluster consisting of five nodes, each equipped with 20 CPUs at 2.30 GHz, and took approximately 27 hours to complete for each parameter setting (with 400 individuals, 50 generations and 50 evaluations for each individual per generation).

For a fair comparison, evolutionary runs were repeated five times for each of the four configurations. Uniqueness of evolutionary runs are guaranteed by initializing the populations of the genetic algorithm using different seeds. Examples of how the individuals perform on the training and validation data sets are shown in Figure 4.2.

To evaluate the heuristics created by the genetic program a validation data set was created with 20 scenarios each of 15 different settings of dynamism and urgency. These settings are shown by Equation 3.1. In each evolutionary run, the best performing individual with respect to the validation data set was used as the representative heuristic. Representative heuristic performance on the validation set was then averaged across all five rollouts. The results of this process, along with the performance of the baseline heuristics, are shown in Table 4.2

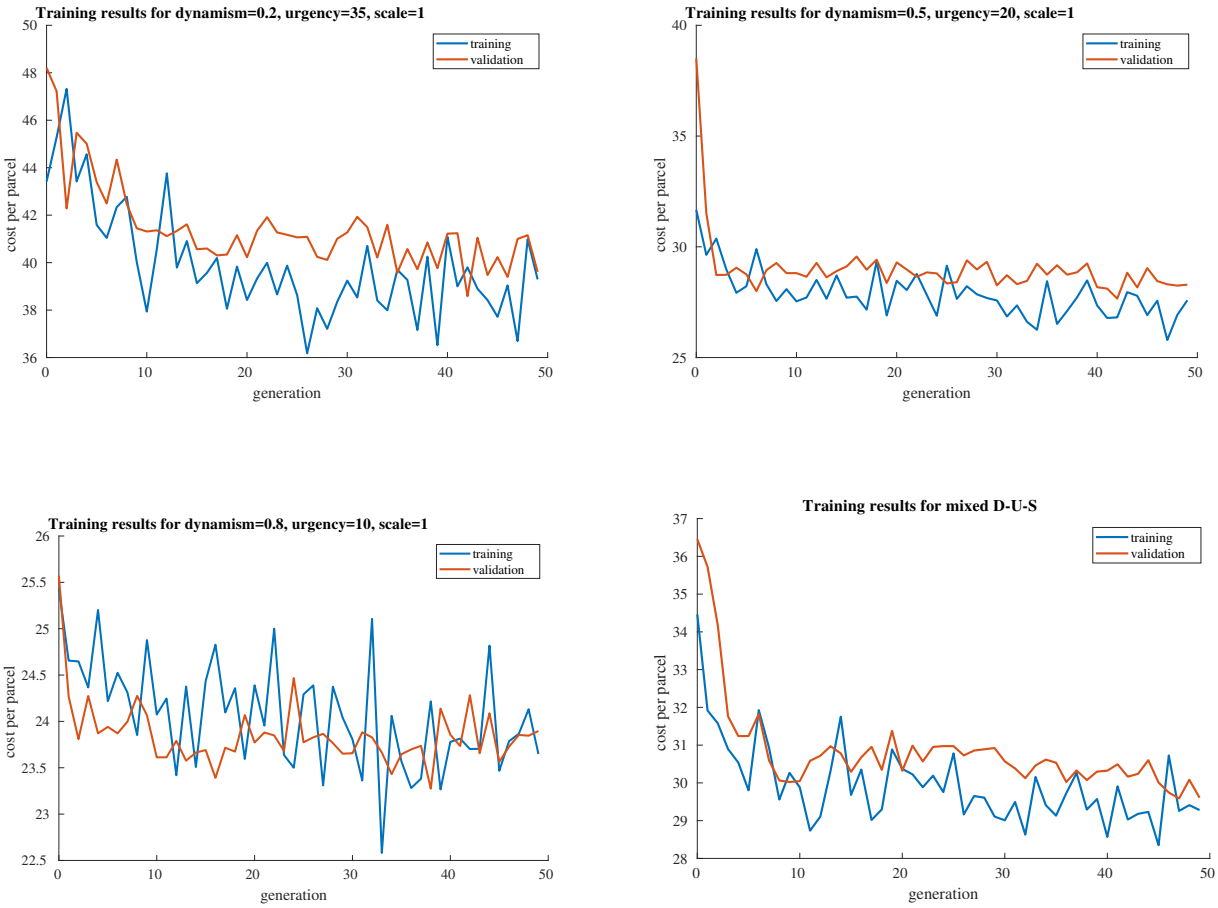


Figure 4.2: System performance on training and validation data sets during training. For each generation, the best performing individual is displayed along with its performance on the validation set. Validation performance is only measured against the configuration of dynamism, urgency, and scale for which they were trained. Note that the heuristic evolution uses new scenarios for each generation, meaning an elite individual is not guaranteed to be the elite of the next generation, causing the jaggedness of the graphs.

	DCI	DRS	DGGP-20-35-1	DGGP-50-20-1	DGGP-80-10-1	DGGP-mixed	Best
20-10-1	$^{6}46.01 \pm 23.35$	$^{5}45.74 \pm 10.51$	$^{2}37.51 \pm 6.46$	$^{3}37.82 \pm 7.20$	$^{4}39.90 \pm 11.89$	$^{1}\mathbf{36.99} \pm 7.13$	DGGP-mixed
20-20-1	$^{6}52.50 \pm 28.58$	$^{5}49.02 \pm 10.78$	$^{3}33.76 \pm 6.49$	$^{2}33.10 \pm 5.87$	$^{4}37.96 \pm 16.59$	$^{1}\mathbf{32.99} \pm 6.11$	DGGP-mixed
20-35-1	$^{6}70.81 \pm 29.64$	$^{5}57.91 \pm 7.30$	$^{2}37.69 \pm 6.88$	$^{3}38.56 \pm 8.62$	$^{4}57.82 \pm 26.04$	$^{1}\mathbf{37.38} \pm 7.44$	DGGP-mixed
50-10-1	$^{5}30.50 \pm 7.64$	$^{6}39.82 \pm 7.18$	$^{2}29.12 \pm 4.26$	$^{4}29.51 \pm 4.54$	$^{3}29.28 \pm 5.01$	$^{1}\mathbf{28.67} \pm 4.28$	DGGP-mixed
50-20-1	$^{6}46.01 \pm 34.17$	$^{5}42.26 \pm 6.91$	$^{3}29.61 \pm 5.18$	$^{1}\mathbf{27.86} \pm 4.02$	$^{4}31.37 \pm 12.15$	$^{2}28.70 \pm 4.96$	DGGP-50-20-1
50-35-1	$^{6}62.92 \pm 31.86$	$^{5}53.02 \pm 11.03$	$^{2}31.97 \pm 6.63$	$^{3}32.27 \pm 7.25$	$^{4}46.76 \pm 25.53$	$^{1}\mathbf{31.35} \pm 6.89$	DGGP-mixed
70-10-1	$^{6}36.21 \pm 25.57$	$^{5}34.88 \pm 7.63$	$^{4}26.56 \pm 3.71$	$^{2}25.31 \pm 3.11$	$^{3}25.45 \pm 7.18$	$^{1}\mathbf{25.11} \pm 3.13$	DGGP-mixed
70-20-1	$^{5}32.43 \pm 23.59$	$^{6}38.75 \pm 10.69$	$^{3}25.65 \pm 3.84$	$^{2}24.78 \pm 3.46$	$^{4}26.40 \pm 9.32$	$^{1}\mathbf{24.48} \pm 3.58$	DGGP-mixed
70-35-1	$^{6}59.59 \pm 33.22$	$^{5}52.31 \pm 11.90$	$^{3}29.30 \pm 4.86$	$^{2}28.64 \pm 5.49$	$^{4}35.69 \pm 17.53$	$^{1}\mathbf{28.26} \pm 5.10$	DGGP-mixed
80-10-1	$^{5}27.96 \pm 18.19$	$^{6}31.92 \pm 4.74$	$^{4}24.62 \pm 3.13$	$^{3}24.06 \pm 3.34$	$^{1}\mathbf{23.35} \pm 2.95$	$^{2}23.85 \pm 3.12$	DGGP-80-10-1
80-20-1	$^{6}40.80 \pm 26.22$	$^{5}35.11 \pm 11.44$	$^{3}24.24 \pm 4.26$	$^{2}23.98 \pm 4.43$	$^{4}26.25 \pm 14.31$	$^{1}\mathbf{23.58} \pm 4.54$	DGGP-mixed
80-35-1	$^{6}72.25 \pm 35.53$	$^{5}52.36 \pm 10.81$	$^{3}28.20 \pm 4.82$	$^{2}27.69 \pm 4.72$	$^{4}41.41 \pm 26.31$	$^{1}\mathbf{26.87} \pm 4.48$	DGGP-mixed
90-10-1	$^{5}26.02 \pm 13.16$	$^{6}30.09 \pm 6.23$	$^{4}23.83 \pm 4.31$	$^{2}22.91 \pm 3.99$	$^{3}22.91 \pm 4.41$	$^{1}\mathbf{22.77} \pm 4.22$	DGGP-mixed
90-20-1	$^{4}22.97 \pm 4.56$	$^{6}31.08 \pm 5.66$	$^{5}23.05 \pm 3.40$	$^{2}22.11 \pm 3.09$	$^{3}22.28 \pm 3.17$	$^{1}\mathbf{21.39} \pm 2.82$	DGGP-mixed
90-35-1	$^{6}67.61 \pm 40.82$	$^{5}53.14 \pm 9.20$	$^{3}28.30 \pm 5.87$	$^{2}26.91 \pm 5.25$	$^{4}41.10 \pm 24.85$	$^{1}\mathbf{26.44} \pm 5.09$	DGGP-mixed

Table 4.2: Baseline heuristics and evolved heuristics performance for varying levels of dynamism and urgency. The listed values are mean and standard deviation of cost per parcel. Superscript values are the ranking for each different validation setting.

4.2.2 Scalability

By introducing more vehicles into the system, while keeping the size of the map constant, the frequency and scale of possible collisions will increase. To evaluate how the increasing runtime of M^* effects the solution, two different tests were conducted, using data sets with varying numbers of agents. These tests were run on a computer with an Intel i7-7700HQ CPU with eight cores at 2.80 GHz.

In the first test, we measured the running times of M^* on a data set containing randomly generated problem instances, as described in Section 3.2.4.4. This data set consists of nine groups with differing number of agents (from two to ten), each with 1000 problem instances. The results are presented in Figure 4.3. M^*-evaluations that took longer than 60 seconds were terminated prematurely and put in the right-most bin of the histogram. We can see that the runtime of the algorithm increases dramatically when the number of agents increase.

The second test was conducted with the goal of measuring the scalability of the system. Ten different datasets with scale levels varying from 0.2 to 2.0 were used, where the number of agents and parcels increase proportionally with the scale level, ranging from $(1, 8)$ to $(10, 80)$. Each of the 1000 scenarios in each dataset were evaluated using the DGGP-50-20-1 heuristic and the failure rate was logged.

In this test, we used the same failure condition for the simulations as we did when training the system, i.e., a scenario fails if the search space of M^* grows to include more than 70 000 states. These results are presented in Figure 4.4. We can see that a higher number of agents greatly increases the risk of encountering problems that are too complicated for M^* to solve, causing the simulations to fail.

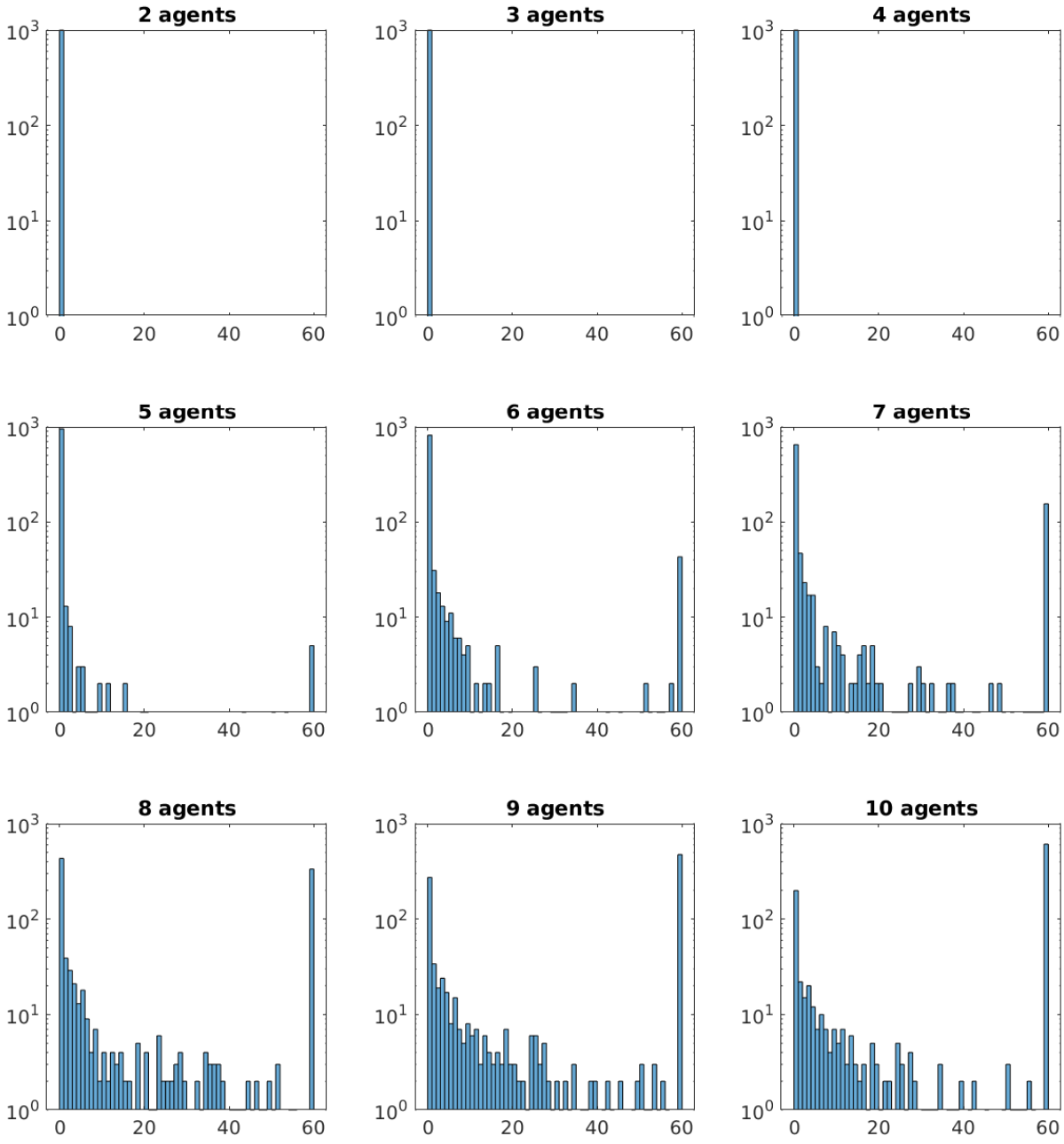


Figure 4.3: Distribution of computation times in seconds for 1000 instances of M^* with differing numbers of agents.

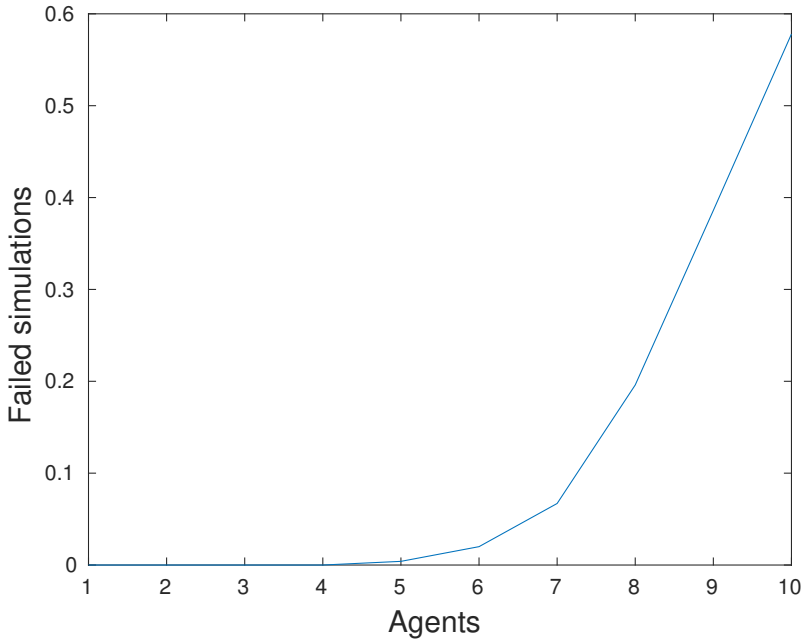


Figure 4.4: The failure rate of the simulations with respect to number of agents.

5
Discussion

In this chapter, we discuss and interpret the results presented in Chapter 4. We also give our thoughts on how the solutions could be improved as well as areas we consider interesting for further exploration.

5.1 Extending the solution of the planar PDP

Does adding communication terminals to the solution of the planar PDP provide a qualitative improvement? As stated in Section 4.1.2, data for **DGP** heuristics was generated based on 10 evolutionary runs, while values of the **DCGP** heuristics are based on the results of one evolution run for each GP setting. This certainly complicates comparison of the two systems.

5.1.1 System evaluation

Comparing each communication system with its counterpart from [4] in Table 4.1, we note that the cost per parcel is lower for DCGP heuristics in all categories compared to the mean values of DGP. This result may seem unsurprising, as providing the optimization algorithm with more information should not worsen the result. However, as the space of possible solutions grows exponentially with the number of possible nodes, adding too many nodes can hinder convergence. Nevertheless, our results indicate that valuable information indeed can be extracted from the added communication nodes, resulting in better overall performance when solving the planar PDP in a decentralized fashion.

5.1.2 Future work

The reason for the discrepancy in the amount of evolutions performed boils down to time constraints and availability of computational power. Ideally, the same amount of rollouts should be performed for each GP configuration, making a more fair assessment possible.

5.2 Solving a shared space PDP

The results show both good news and bad news for the graph based PDP solution. In this section, we will offer insight regarding both the results as well as the multitude of extensions that did not make it into the thesis due to time constraints and scope.

5.2.1 Objective value evaluation

As there are no benchmarking datasets available for the graph based dynamic PDP with collision avoidance, we decided to compare the performance of the evolved heuristics with the performance of two simple heuristics: *Insertion cost* and *Route size*. We note that the simple heuristics are outperformed by the evolved heuristics in almost every single problem class. This indicates that the GP is able to find some valuable underlying structure regarding how to assign parcels. However, due to the sheer size of the evolved heuristics, it is very difficult to analyze what they actually do. Therefore, one can view them as black boxes that assigns parcels to agents in an intelligent way.

From Table 4.2, we can see that the heuristic **DGGP-mixed** is the best heuristic in 13 out of 15 problem classes, suggesting that training on multiple different problem classes helps developing a more general purpose heuristic. This result is quite similar to what van Lon, Branke and Holvoet present in [4], where the specialized systems outperform the mixed system on the problem classes that the specialized systems have been trained on, but the mixed system performs better overall. Yet, as the graph based setting is considerably more complicated than the planar setting due to the added problem of collision avoidance, it is not obvious that the results would be similar.

In real problems, it might be difficult to know which problem class to use. For instance, scenarios could contain multiple successive parts with different characteristics. This further motivates the use of an all-purpose system that has been trained on a mixed set of scenario classes.

5.2.2 Cost correlation to urgency

The urgency of a scenario tells us how much time in average agents have to pick up announced parcels before the pickup time window closes. Urgency is measured in time, and a higher urgency value means, somewhat counterintuitively, that agents have more time to react to new parcels. Thus, one could expect that scenarios with high urgency values should be easier than scenarios with low urgency values. This is certainly the case in the planar PDP, as shown in Table 4.1: the cost per parcel is consistently lower for scenarios with higher urgency, but with the same dynamism and scale. However, inspecting the results in Table 4.2, we can clearly see that this is not the case for the graph based PDP.

This could be explained in part by measuring the maximum allowed duration of orders, i.e. $d_i^R - p_i^L$. Using the scenario generator, described in Section 3.2.4.2, we find this quantity to increase with the urgency level. An example of this, comparing maximum allowed order durations in scenarios generated using 10 and 35 minute urgencies, is shown in Figure 5.1. Since vehicles have a binary capacity, a longer order duration means the vehicle will be occupied by a single parcel for a longer duration. Thus, order processing throughput decreases and the average cost per parcel increases.

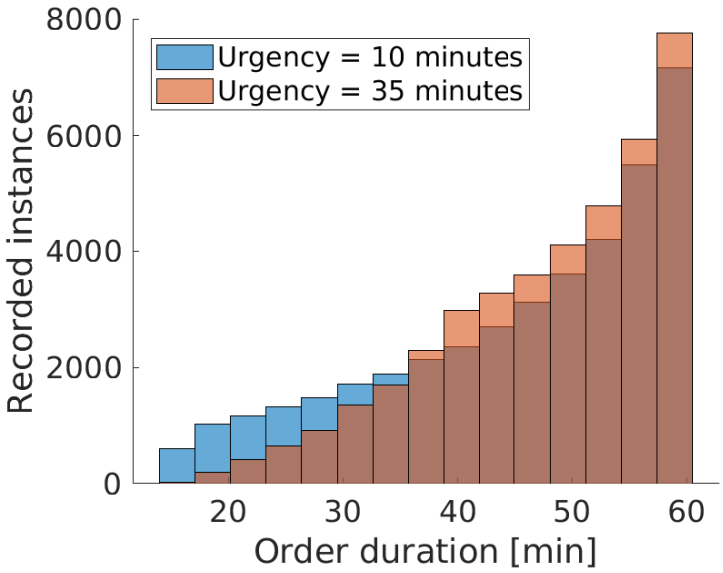


Figure 5.1: Histogram of maximum order durations for orders from different urgency classes.

5.2.3 System scalability

Evaluations for different levels of scale are notably missing from Table 4.2. When increasing the scale parameter, both the number of vehicles and orders increase for the scenario. As shown by Figure 4.4, the failure rate of scenarios rises quickly as the number of agents increases. Thus, showing evaluations for increasing levels of scale without taking into account the number of failed scenarios would be misleading. Figures 4.3 and 4.4 also quantify the issue of solely relying on an optimal MAPF solver. For an operation using more 5 agents in a similar graph structure, an alternative to M^*, such as a suboptimal solver, is essential.

Another issue is that the definition of scale as described in Section 2.1.2.3 does not map well to a shared space graph scenario. Increasing the number of vehicles on a map of constant size will increase the rate of vehicle conflicts, which in turn increases the cost per parcel. Having the size of the map increasing with scale could be a solution, however the amount of intersections, pickup locations, and delivery locations would have to increase as well for the scenario to correctly model reality. Another solution could be to implement additional maps of varying size. These maps could then also be used to test how the system adapts to different environments, i.e., if the evolved heuristics can be used for general maps or if they are overfitted to the map they have been trained on.

5.2.4 Future work

A number of ideas and implementations for the industrial application have been left out of the thesis, largely due to time constraints and the limitations set for the project. In this section we will discuss the necessity for these extensions and how they could be implemented.

5.2.4.1 Communication nodes in shared space MAS

As discussed in Section 4.1, adding communication nodes for the genetic program to use may improve the overall performance for solutions to the planar PDP. Thus it would be interesting to see how this could effect the performance of the decentralized shared space PDP solver. Additionally, introducing more ways of communicating may be specially important in a shared space environment, as this may improve agents ability to avoid unnecessary routing conflicts.

The difficulty lies in implementing these nodes without while keeping within the limitations of agents' communication ranges. One naive solution is to let agents store the last known route of other agents. Using this information along with the agent velocities and the current time, agent positions could be extrapolated. With a rough estimation of other agents position as well as their last known routes, all communication nodes described in Table 3.3 can be evaluated.

5.2.4.2 Modelling capacity

As mentioned in Section 3.2.1.3, the capacity of the agents has been limited to a single parcel at a time. This constraint means that vehicles always have to fill their cargo entirely when picking up ore. However, in reality there may be situations when a vehicle should/can only fill parts of its cargo at a single location. This could be added to the model by associating a weight to each parcel and a capacity limit to each vehicle. New nodes containing capacity information should then also be supplied to the genetic programming algorithm.

5.2.4.3 Supplement M* with suboptimal solver

As discussed in Section 5.2.3, relying solely on an optimal, superexponential, MAPF solver will eventually lead to state space explosions, even for a small number of agents. Therefore M^* is simply not sufficient for a real world implementation and needs to be supplemented by a suboptimal solver. These solvers each have with their own area of usage and limitatons, such as Bibox [18] which outperforms many state of the art suboptimal solvers, but is applicable only on biconnected graphs. Another issue with suboptimal solvers is the nature of the solutions provided, often having agents move sequentially instead of synchronously. A number of algorithms have been developed for more simultaneous movement, such as Parallel Push and Swap [19]. Post processing algorithms to improve suboptimal solutions have also been put forward, for example the **condense** function described by de Wilde et al. [20].

5.2.4.4 Increase parallelization for conflict resolution

Whenever a route conflict emerges, all vehicles involved decelerate to a stand still and one vehicle gets the responsibility of providing a MAPF solution. Thus, all but one of the vehicles' on-board computers are unutilized. One way to increase parallelization would then be to let all passive vehicles instead use the suboptimal solver. Thus, if the M^* solver would time out, either by a limit on the number of

states expored or a limit on computation time, the best suboptimal solution could be used as a backup.

Another way to increase parallelization, as proposed by Cohen et al. [21], is to use *Rapid Randomized Restarts*. They propose that MAPF solvers commonly exhibit heavy-tailed distributions of runtime, reaffirmed by our results in Figure 4.3. By introducing randomness to the solver, such as changing the order in how limited neighbours are added in M^*, and restarting the solver whenever a time limit is reached, the runtime may be shortened by avoiding the solution paths that cause the heavy-tailed distribution. Provided that a suboptimal MAPF solver has also been implemented, conflicts could be solved by having all but one vehicles run Rapid Randomized Restarts on an optimal solver, such as M^*, and the last vehicle using a suboptimal solver as a fail safe.

5.2.4.5 Post processing of emergency paths

As discussed in Section 3.2.2.2, MAPF solutions provided by M^* require agents to move synchronously from state to state, which introduces waiting times since agents have to wait for the agent traveling the longest edge in each iteration. Increasing the map resolution, thereby shortening the length of long edges, would then reduce waiting times. However, this will not circumvent the problem of agents having to stop and start since edges will still have different lengths, and in a real world environment decelerating and accelerating a vehicle carrying several tons of cargo is a gas-guzzling operation.

To handle this issue, MAPF solutions can be passed through a post-processing step, where vehicle speeds are adjusted in order for all vehicles to arrive at the next solution state in unison. This process is described in detail by Hönig et al. [22], where a polynomial time post processing algorithm is described able to take kinematic constraints into consideration.

5.2.4.6 Introduce additional classes of vehicles

A real world mining operation uses a number of different types of vehicles, each with different assignments. Personnel needs to be moved around for extracting new ore via blasting and wheel loaders need to be accessible at pickup destinations in order to load vehicles with ore. Therefore, the system needs additional classes of vehicles. The system could be optimized using the same heuristic in each class of vehicle, with the difference being that these classes bid on different sets of assignments:

- The personnel transport problem is identical to that of a pickup and delivery problem and assignments do not need to be modified except for only being available to staff transportation vehicles.
- Wheel loaders on the other hand do not have a pickup or delivery destination, instead they simply need to be available at pickup locations during certain times. This behaviour could be modeled by letting wheel loader assignments have the same locations for pickup and delivery, one for each vehicle assignment.

Another optimization method that could be implemented in a system with several classes of agents is co-evolution, as described by Koza [23]. During co-evolution,

the genetic algorithm will consist of several populations. In each generation, the individuals from one generation is evaluated using the individuals from other populations as their environment. However, if each individual is evaluated against all individuals from the other populations this will dramatically increase the required computation time for optimization.

6
Conclusion

The results presented in this thesis suggest that genetically programmed solutions to the planar pickup and delivery problem can be improved by supplementing agents with abilities in communication. This result should also carry over to the shared space variant of the pickup and delivery problem, where agent interaction is of even greater importance.

As for the shared space PDP, a solution method has been presented which is successful in solving instances with a few number of agents and adaptive for different levels of dynamism and urgency. Using a genetically programmed heuristic improves the solution quality compared to using naive bidding heuristics such as cheapest insertion cost. Our results suggests that these heuristics should be evolved using scenarios of varying difficulty in order to minimize travel time and tardiness for a wide range of problem instances.

Moving forward, there is wide range of extensions that can be implemented to further improve the quality of the shared space PDP solver. The largest bottleneck of the system is the multi-agent path finder M^*, due to its superexponential worst case time complexity. We propose the greatest benefit could be gained by supplementing vehicle logic with a suboptimal multi-agent path finder, as this would increase the scalability of the system.

Bibliography

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[2] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of operations research*, 153(1):29–46, 2007.

[3] Rinde RS van Lon and Tom Holvoet. Evolved multi-agent systems and thorough evaluation are necessary for scalable logistics (position paper). In *Computational Intelligence In Production And Logistics Systems (CIPLS), 2013 IEEE Workshop on*, pages 48–53. IEEE, 2013.

[4] Rinde RS van Lon, Juergen Branke, and Tom Holvoet. Optimizing agents with genetic programming: an evaluation of hyper-heuristics in dynamic real-time logistics. *Genetic programming and evolvable machines*, 19(1-2):93–120, 2018.

[5] Rinde RS van Lon and Tom Holvoet. When do agents outperform centralized algorithms? *Autonomous Agents and Multi-Agent Systems*, 31(6):1578–1609, 2017.

[6] Rinde RS van Lon and Tom Holvoet. Towards systematic evaluation of multi-agent systems in large scale and dynamic logistics. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 248–264. Springer, 2015.

[7] Rinde RS van Lon, Eliseo Ferrante, Ali E Turgut, Tom Wenseleers, Greet Vanden Berghe, and Tom Holvoet. Measures of dynamism and urgency in logistics. *European Journal of Operational Research*, 253(3):614–624, 2016.

[8] Reid G Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, (12): 1104–1113, 1980.

[9] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[10] Mattias Wahde. *Biologically inspired optimization methods: an introduction*. WIT press, 2008.

[11] Robert Sedgewick. *Algorithms*. Pearson Education India, 1988.

[12] Glenn Wagner. Subdimensional expansion: A framework for computationally tractable multirobot path planning. 2015.

[13] Rinde RS van Lon and Tom Holvoet. Rinsim: A simulator for collective adaptive systems in transportation and logistics. In *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*, pages 231–232. IEEE, 2012.

[14] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and A Chircop. Ecj: A java-based evolutionary computation research system. *Downloadable versions and documentation can be found at the following url: https://cs.gmu.edu/~eclab/projects/ecj/*, 2006.

[15] Rinde RS Van Lon, Tom Holvoet, Greet Vanden Berghe, Tom Wenseleers, and Juergen Branke. Evolutionary synthesis of multi-agent systems for dynamic dial-a-ride problems. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 331–336. ACM, 2012.

[16] William B Langdon and Riccardo Poli. The genetic programming search space. In *Foundations of Genetic Programming*, pages 113–132. Springer, 2002.

[17] Stefan Vonolfen, Andreas Beham, Michael Kommenda, and Michael Affenzeller. Structural synthesis of dispatching rules for dynamic dial-a-ride problems. In *International Conference on Computer Aided Systems Theory*, pages 276–283. Springer, 2013.

[18] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3613–3619. IEEE, 2009.

[19] Qandeel Sajid, Ryan Luna, and Kostas E Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *SoCS*, 2012.

[20] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research*, 51:443–492, 2014.

[21] Liron Cohen, Glenn Wagner, TK Kumar, Howie Choset, and Sven Koenig. Rapid randomized restarts for multi-agent path finding solvers. *arXiv preprint arXiv:1706.02794*, 2017.

[22] Wolfgang Hönig, TK Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *ICAPS*, pages 477–485, 2016.

[23] John R Koza. Evolution and co-evolution of computer programs to control independently-acting agents. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats. MIT Press, Cambridge, MA*, pages 366–375, 1991.

A

Appendix 1

The tree structure of one of the highest performing heuristics from DGGP-mixed is presented in Figure A.1. This tree encodes a complicated function used for calculating bid values for auctions. Note that this example is one of the smaller trees produced by the genetic algorithm.

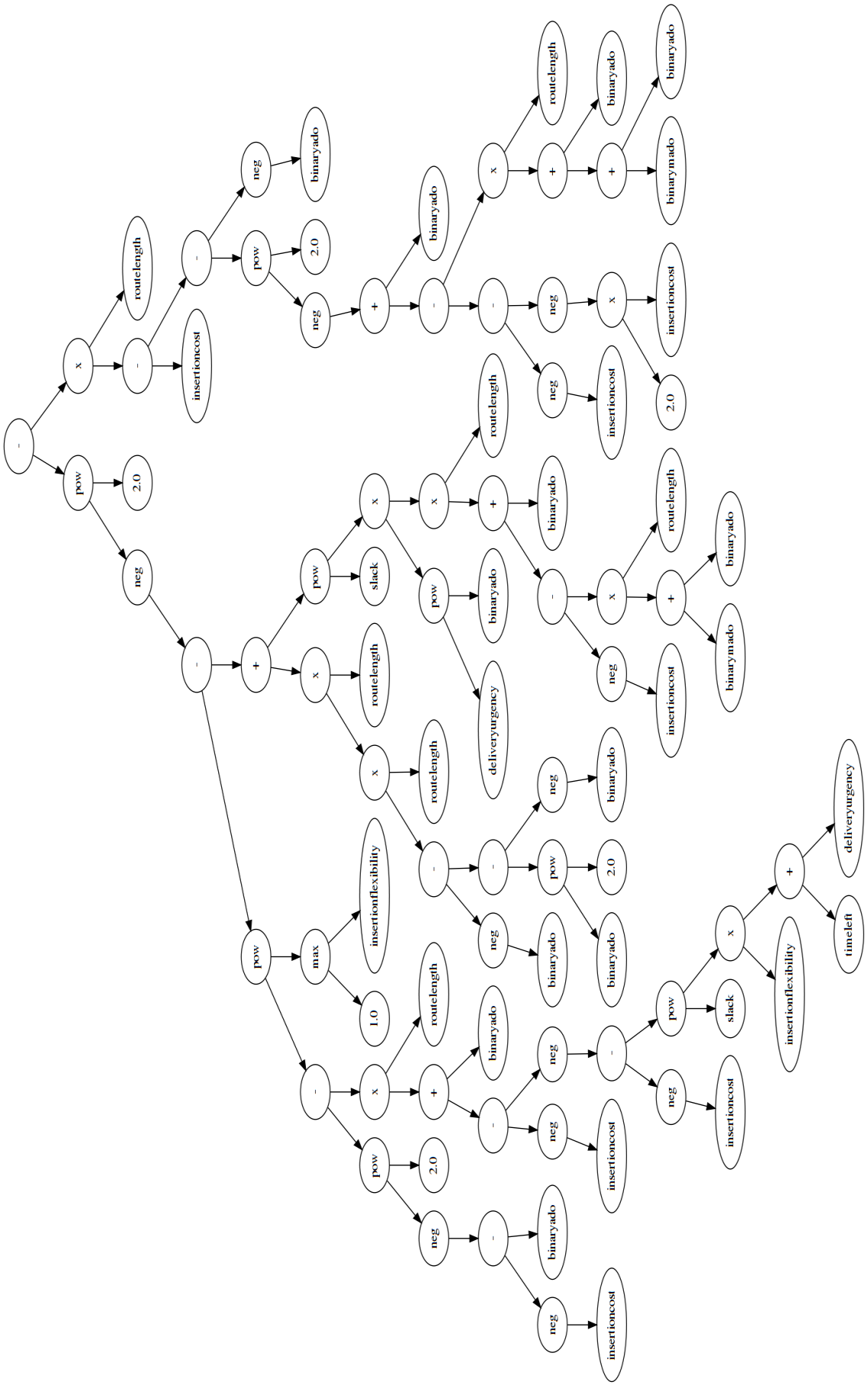


Figure A.1: One of the highest performing heuristics from DGGP-mixed.