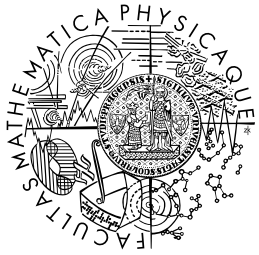

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Lucie Řeháková

Evolutionary techniques utilization in hierarchical task network

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Roman Neruda, CSc.
Study programme:	Computer Science
Study branch:	Theoretical Computer Science

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 27. 7. 2016

Title: Evolutionary techniques utilization in hierarchical task network

Author: Lucie Řeháková

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This master thesis describes the design and the implementation of the algorithm solving the domain-independent partial order simple task network planning problem using the tree-based genetic programming. The work contains comparison of several possible approaches to the problem — it compares different representations, ways of evaluation and approaches to the partial ordering. It defines heuristics to improve the efficiency of the algorithm, including the distance heuristic, the local search and the individual equivalency. The implementation was tested on several experiments to show the abilities, strengths and weaknesses of the algorithm.

Keywords: hierarchical task network, simple task network, planning, evolutionary computation, genetic programming

I would like to thank to everyone who supported me during writting of this master thesis. Above all, I thank Roman Neruda for his incredible patience and for his leathership and guidance.

Contents

1. Introduction

Planning is an inseparable part of rational behavior. It means choosing and organizing actions in order to achieve some goals. But why machines should do it? Some machines need to act autonomously because there is no other option. For example, satellites in space cannot be always guided by humans. Autonomous planning is also useful to help people with their planning problems if they are too big, e.g. it is very difficult to optimize tasks in industrial production. Hierarchical task network planning uses the fact that problems can be often decomposed into smaller ones that are simpler to solve. For instance, to build a house one needs to build a foundation first, then one can build walls, and a roof must be built as the last one. The hierarchical task network uses this decomposition to simplify decision what actions should be next to solve the planning problem.

The evolutionary computation is a robust optimization technique that is inspired by the idea of the natural selection. It starts with a population of partial solutions and it breeds and evolves them until the desired solution, is found. A nice example of application of this algorithm was shown in [2]. Authors of this article used evolutionary computation to find a path for multiple unmanned-aerial-vehicles to the target location, avoiding air defense units, avoiding the ground and avoiding each other. Genetic programming is an evolutionary computation technique that does not need to know the structure or size of the solution in advance. To assure that, the genetic programming has individuals representing programs in a form of trees.

The main goal of this work is to design an algorithm using the genetic programming technique and solving the hierarchical task network planning problem. To be more precise, the goal is to design and implement an algorithm solving the domain-independent partial order simple task network planning problem. The simple task network is a special case of hierarchical task network that does not use all the constraints. Domain-independency means that our algorithm will be working with any planning domain, if it gets its specification. Our algorithm will be using the tree-based genetic programming technique for the following reasons — simple task network has a natural tree-like structure, the size and the final structure of the solution in not clear from the beginning, and the evolutionary computation is a robust technique to produce highly optimized solutions.

Several hierarchical task network solvers exist, let us mention *shop2* [7]. The *shop2* was a winner of one category in the International Planning Competition in 2002. There are also several planners based on evolutionary computation. Among them *sinergy* [6] is based on genetic programming. It has two type of elements

— *terminals* and *functions*. Both of them can be seen as lisp functions, however *functions* have to have at least one parameter, *terminals* are parametless. Then, generated individuals represent computer programs that are built by function composition over the set of *terminals* and *functions*. However, so far we have met only one work including both hierarchical task network and genetic algorithms — by Ruscio, Levine and Kingston [9]. Their individuals are represented as a vector of non-negative integers. If some method should be decomposed during evaluation, this number is the number of decomposition to use. If some variable needs to be chosen, this number is the number of variable instance. Simple encoding seems to be a weak point of this approach — even a very small change at the beginning of the plan can lead to an entire different result, while the same change at the end of the plan not. We are also not fans of the fact, that one number may represent a method decomposition during one evaluation and the number of variable instance during the next one.

Let us briefly describe the structure of this thesis. At first, we will introduce a reader with basics of evolutionary computation (cf. chapter 2). Significant part of this chapter is dedicated to genetic programming, especially to the tree-based genetic programming (see subsection 2.2.1). The next chapter describes all necessary prerequisities to understand the planning problem (cf. chapter 3). A special section is dedicated to the hierarchical task network (and simple task network), since it is the problem we are going to solve (see section 3.2). The following chapter presents the main original contribution of this thesis. It contains the description of our algorithm that solves the domain-independent partial order simple task network planning problem (see chapter 4). It describes the problems we needed to solve, and decisions made. The most important section explains computation of the fitness 4.8, it is the crucial part of our algorithm. Section 4.6 contains also important properties of our algorithm without which it would not work well, namely a local search and running parallel subpopulations. The next chapter contains a simple description of our implementation, and a specification of the problem and the domain (cf. chapter 5). In the following chapter we will show how the implementation of our algorithm works on several experiments (see chapter 6). In the last chapter, we will summarize our work and present some conclusions (see chapter 7).

2. Evolutionary Computation

In this chapter, we will introduce the evolutionary computation [10]. We will define genetic algorithm and its basic parts, then we will look at genetic programming.

2.1 Genetic Algorithm

Genetic algorithm (GA) is a search heuristic based on the idea of natural evolution. It uses techniques such as selection, mutation and recombination to direct the search into regions with better rated solutions within the search space. As such, it is often applied to find useful solutions of optimization problems.

2.1.1 Principle of Genetic Algorithm

GA starts with population of individuals, where each of them has an associated fitness value. The GA iteratively changes the population so that individuals with better fitness value spread in next the generation. During each iteration, best individuals are stochastically chosen, they are recombined and mutated, and in this way the new population is created. Then a combination of old and new populations give us next generation. The whole cycle is shown in figure 2.1.

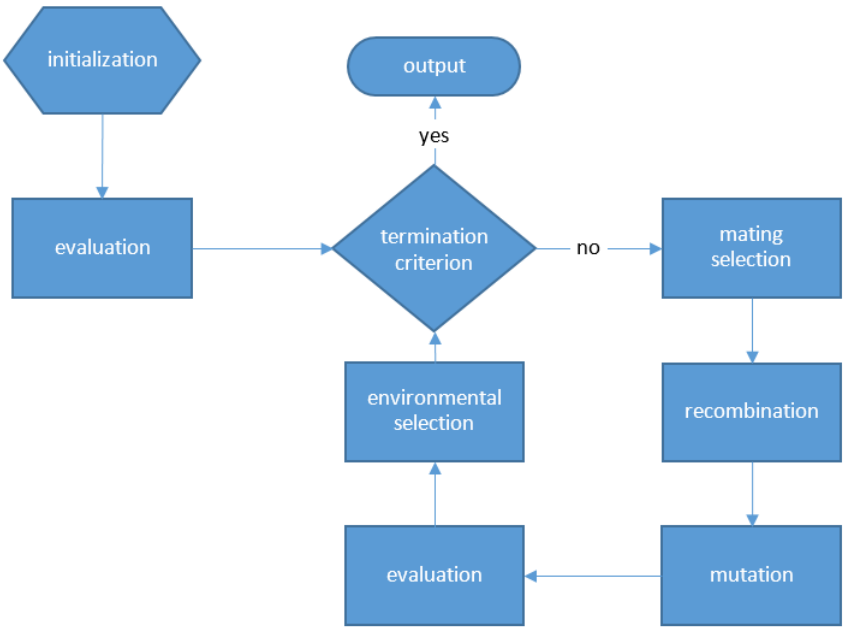

Figure 2.1: Schema of genetic algorithm

Each individual represents a possible solution in the search space. They are

encoded as a vector of a finite length. Such vector is traditionally a binary one. Nevertheless, the best representations usually reflect something about the problem itself. From the point of view of natural selection, these individuals can be likened to chromosomes and values in the vector to genes (see figure 2.2). This designation is also often used in literature [10].

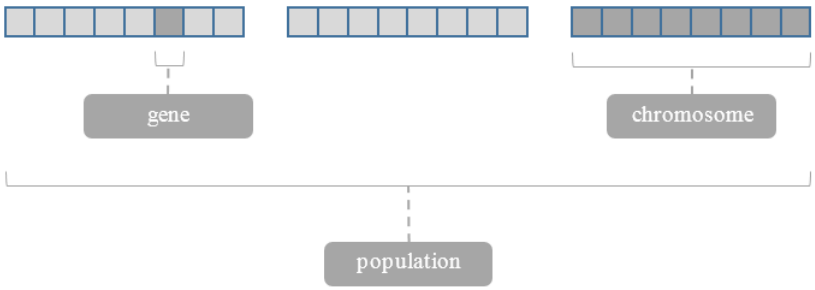

Figure 2.2: Schema of GA population

Fitness function determines how good is the individual (solution). Individuals with better fitness value have higher chance to be selected and reproduced, and thus they transmit their genes into the next generation. Fitness function is always problem dependent, and usually it is the objective function of a given optimization problem.

2.1.2 Basic parts of Genetic Algorithm

Initialization

The first population is often generated randomly, but it can be seeded in areas where the best solutions are assumed, or where previous run of the algorithm found interesting results.

Mating selection

Mating selection choose individuals for *breeding* (recombination and mutation), and their descendants will be present in next population. The key idea of mating selection is giving higher chance to reproduce to better solutions, so their genes will be represented in next generation. The quality of an individual is given by its fitness value. Nevertheless, selection should be stochastic, so even worse individuals have chance to be selected to preserve diversity of population. Otherwise, the algorithm could easily get stuck in a local optimum.

The basic example of mating selection is a *roulette wheel selection*. Imagine a roulette wheel where each individual occupies several pockets on the wheel. The better the individual, the more pockets of the wheel it fills. Then roulette wheel

Algorithm 2.1 Illustration of *roulette wheel selection.*

```
1  Individual rouletteSelection()
2  {
3    int populationFit;
4    for(Individual ind in population)
5      populationFit += ind.fitness;
6
7    double selector = rand.nextDouble();
8    double seeker = 0;
9
10   for(Individual ind in population)
11   {
12     seeker += ind.fitness/populationFit;
13     if(seeker > selector)
14       return ind;
15   {
16 }
```

is spinned and a ball is tossed in. Individual in a pocket, where the ball stops, is selected. Formally, let us define probability of individual i being selected as $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, where f_i is its nonnegative fitness value and N is number of individuals in the population. At first, roulette wheel selector gets random number between 0 and 1, let us call it a *limit*. Then it gradually takes all individuals and it adds their p_i values. Individual, whose p_i value added to the sum causes *limit* overflow, is selected. For better illustration see algorithm 2.1.

Another example is a *tournament selection*. In this case several individuals are chosen randomly, then the best (eventually the worst) of them is selected (cf. 2.2).

Recombination (crossover)

During recombination, genes from two or more parent solutions gained from mating selection are used to create children individuals. A *one-point* crossover is an example of crossover operator, where two parents are split into two parts on the same position in chromosome which is chosen randomly. Then, each child gets one part of mother's and part of father's chromosome (see figure 2.3).

Algorithm 2.2 Illustration of *tournament selection.*

```
1  Individual tournamentSeection(int numOfIndividuals, double
       bestOneProb)
2  {
3    Individual bestInd = chooseRandomIndividual();
4    Individual worstInd = bestInd;
5
6    for(int i = 1; i<numOfIndividuals; ++i)
7    {
8      Individual newInd = chooseRandomIndividual();
9      if(isBetter(newInd, bestInd))
10       bestInd = newInd;
11     if(isWorse(newInd, worstInd))
12       worstInd = newInd;
13   {
14
15   if(rand.nextDouble()<bestOneProb)
16     return bestInd;
17   else
18     return worstInd;
19 }
```

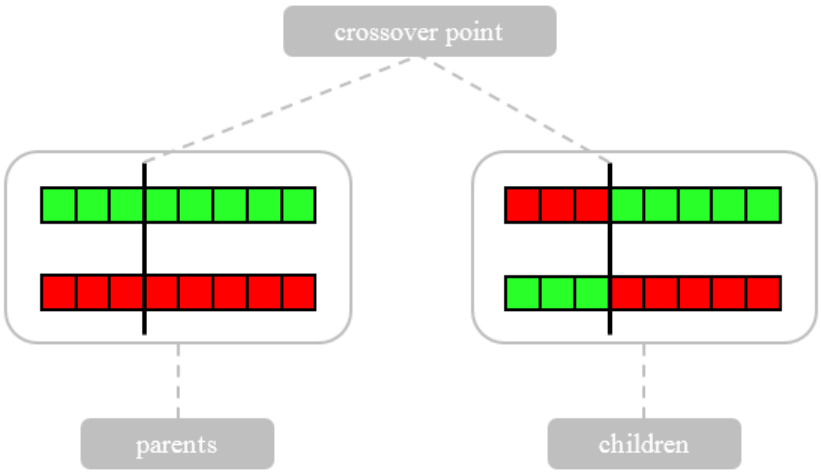

Figure 2.3: Illustration of *one-point* crossover

The importance of recombination was questioned. The success of GA with crossover was compared with success GA with random crossover (one of the parents was entirely random). In some cases GA with random crossover won as it is shown in [4]. This proves that the choice of particular crossover needs a deeper reasoning, whether it makes sense in the given problem and with the used encoding (for more information about encoding see subsection 2.1.4).

Mutation

Mutation maintains diversity of the population. It is also the GA's main power how to escape from local optima. Mutation means small random change in given chromosome, and it is applied on children individuals with some small probability.

In order to speed up the genetic algorithm the so-called informed mutation (crossover) can be used. Informed versions of these operators have some knowledge about the problem being solved. Therefore, the evolution is lead in the right direction. For example, imagine a problem where you have N objects, each with some price, and you want to divide them into M equally precious piles. Chomosome has length of M and the gene i determines on which pile is i-th object. Informed mutation can simply take some object from the most precious pile and put it in the less valuable one. Along with this informed mutation, ordinary mutation should be used, otherwise leaving local optimal could be difficult.

Environmental selection

Environmental selection combines the old and new population to create a new generation. One of the simplest way how to reach this goal it is to forget the old population and accept all individuals from the new one. But it is not the only approach. For example, the next generation may be created from the best individuals of the old population and from the best individuals of the new one.

One way or another, it is practical to add some of the finest individuals from the old population to the next generation together with the new population. This approach is called *elitism*, and it guarantees that fitness function will not decrease across generations.

Termination

GA ends when satisfying solution is found. It is also suitable to terminate algorithm if specified number of generations has been reached, otherwise it could run forever.

2.1.3 Effects of genetic operators

Mating selection itself would fill the population with copies of the best solution from initial population. When recombination is added, the algorithm converges to some local optimum. When some attribute would be missing in initial population, it would not appear during the whole run of the algorithm. For example, if all individuals would have 0 in their first gene, recombination itself cannot ensure another value there. Mutation itself would be the same as a random walk.

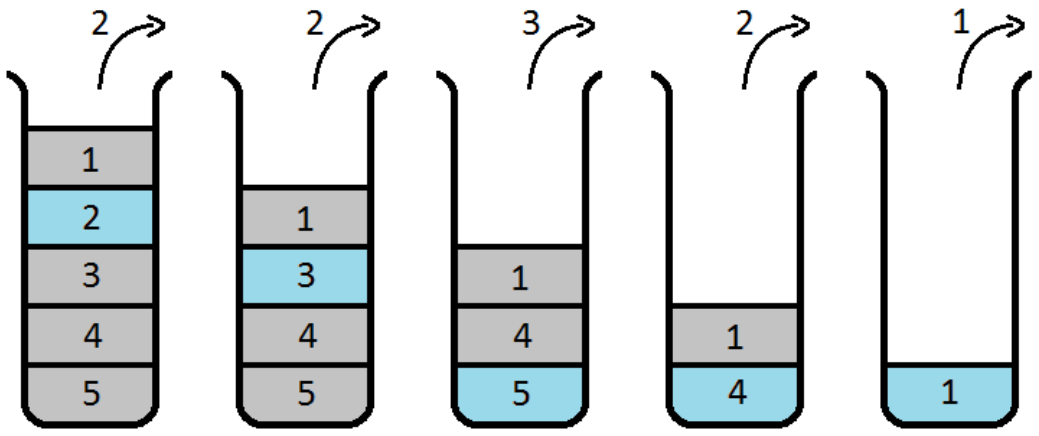

Figure 2.4: Explanation of the buffer encoding for individual $(2, 2, 3, 2, 1)$ representing a path $2 \to 3 \to 5 \to 4 \to 12$.

2.1.4 Representation

Translation from individual to solution could be very simple, but sometimes it needs nontrivial decoding. For example the *travelling salesman problem* could be encoded as a list of cities on the path, i. e. path going from city i to city j is represented as succeeding values i and j in the chromosome. Thus the individual $(2, 3, 5, 4, 1)$ represents a path $2 \to 3 \to 5 \to 4 \to 12$. But the same path could be encoded also into individual $(2, 2, 3, 2, 1)$ using the buffer encoding — cities are arranged in a buffer and they are deleted from buffer when used. Each gene in an individual suggests which city from the buffer is supposed to be the next one on the path. For better illustration see figure 2.4.

Proper encoding is very important, and it influences effectivity of the whole algorithm. In the previous, example the first encoding cannot be used with a basic crossover operators. Offsprings would have to be corrected, so they represents correct path. In the second example it would not be a problem. On the other hand (*one-point* crossover is assumed), when second part of chromosome would represent some part of the path in parent individual, it could represent completely different one in its offspring. In the second encoding also a small mutation can change the path entirely. Therefore we do not consider the second encoding as a proper one.

2.2 Genetic Programming

Authors of [8, chapter 1] define the genetic programming as the following:

> Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without requiring the

user to know or specify the form or structure of the solution in advance.

Basically, GP usually evolves computer programs, as will be shown in next subsections. But it can also used for completely another purposes, e.g. for developing an antenna shape.

2.2.1 Tree-Based Genetic Programming

Tree-based GP is a common encoding of programs, where they are expressed in the form of syntax trees. Leaves of the tree consist of constants and variables, so called *terminals*, whereas inner nodes are called functions. For example, imagine that you want to evolve an approximation of some very difficult mathematical function of two variables X and Y only with basic mathematical operators: +, -, *, /. Therefore we want to evolve the formula of our function. In this case, the allowed mathematical operators are also the allowed functions of GP. Terminals consist of variables X and Y and numeric constants. Then, the figure 2.5 shows an example of an individual in GP.

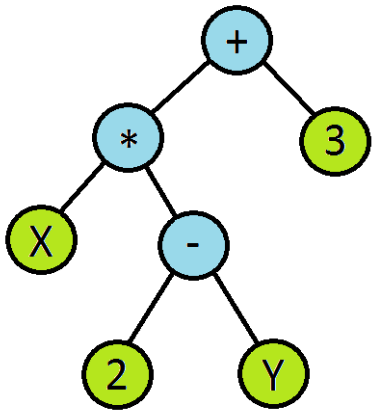

Figure 2.5: Syntax tree of expression $X*(2-Y)+3$. Terminals are colored green, functions blue.

The basic algorithm of GP is similar to GA's. Therefore, we will discuss only those parts where we can see some difference from the GA.

Representation

Individuals are encoded in a form of tree structures. Leaves can be viewed as operands and inner nodes as operators.

Initialization

The first population is chosen randomly or some (or all) of its individuals can be seeded. Nevertheless, unlike with the representations we mentioned in section 2.1, it is not so clear, how a random tree-based individual could look like. We will look on two simplest methods described in the [8, section 2.2]. Both methods need a depth limit of the generated tree to be set in advance, where depth of the tree is number of nodes on the longest path from the root to the leaf.

The first method is called *full*, because it generates full trees of given depth. Basically, it chooses a random function as a node if depth of the node is less then the limit, otherwise a random *terminal*. Notice that this method does not necessary produce trees with the same number of nodes. That depends on arity of chosen functions.

The second method is called *grow*. It chooses random function or random terminal, if depth of the node is less then the limit, otherwise just a random *terminal*. This method allows wider variety of trees to be created than the full method. It is practical to combine these two methods. Example of such a combination is the commonly used *ramped half-and-half* method which says that half of the population should be created by the *full* method and half by the *grow* method. Also a range of depth limits should be used.

Recombination (crossover)

A common method of recombination in the tree-based GP selects one crossover point (node) in each parent tree. Children looks the same as parents but with exchanged subtrees defined by crossover points (see figure 2.6 for better illustration). Results of parents' and children's programs could be very different.

Mutation

GP has a great variety of mutations that can be used. We show some of them.

One of the basic approaches is changing one node with respect to its arity.

Another commonly used form of mutation chooses randomly a node in a given tree. Than it exchanges the whole subtree defined by the chosen node with a random subtree (see figure 2.7). Restrictions on size of random tree can be required, so the size of an individual is similar before and after mutation. This mutation is basically a recombination with a random tree.

Mutation of constants is very important. Otherwise it is not so easy for GP to change $X*(2-Y)+3$ from our example to the correct answer $X*(2-Y)+5$.

Last but not least, we mention mutation that randomly selects node in a tree and replaces it with a random terminal. This mutation is a strong weapon against

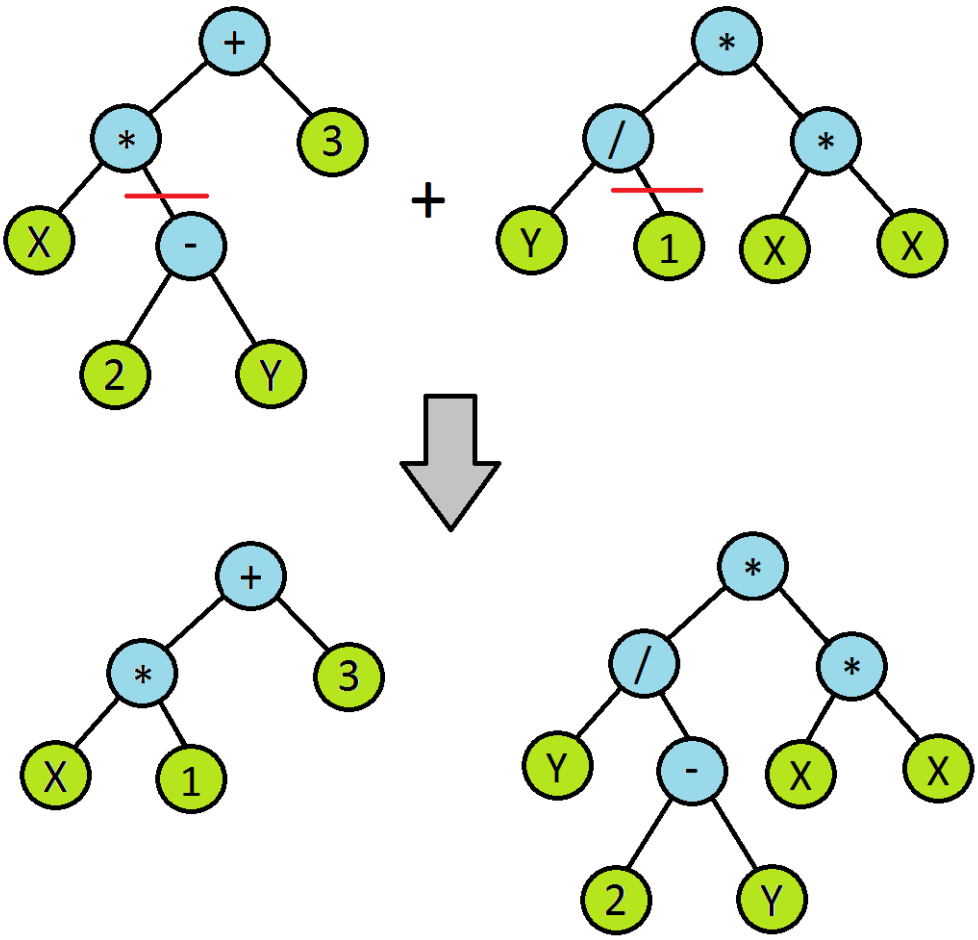

Figure 2.6: Illustration of crossover between two tree-structured individuals.

the bloat issue (see Subsection 2.2.3).

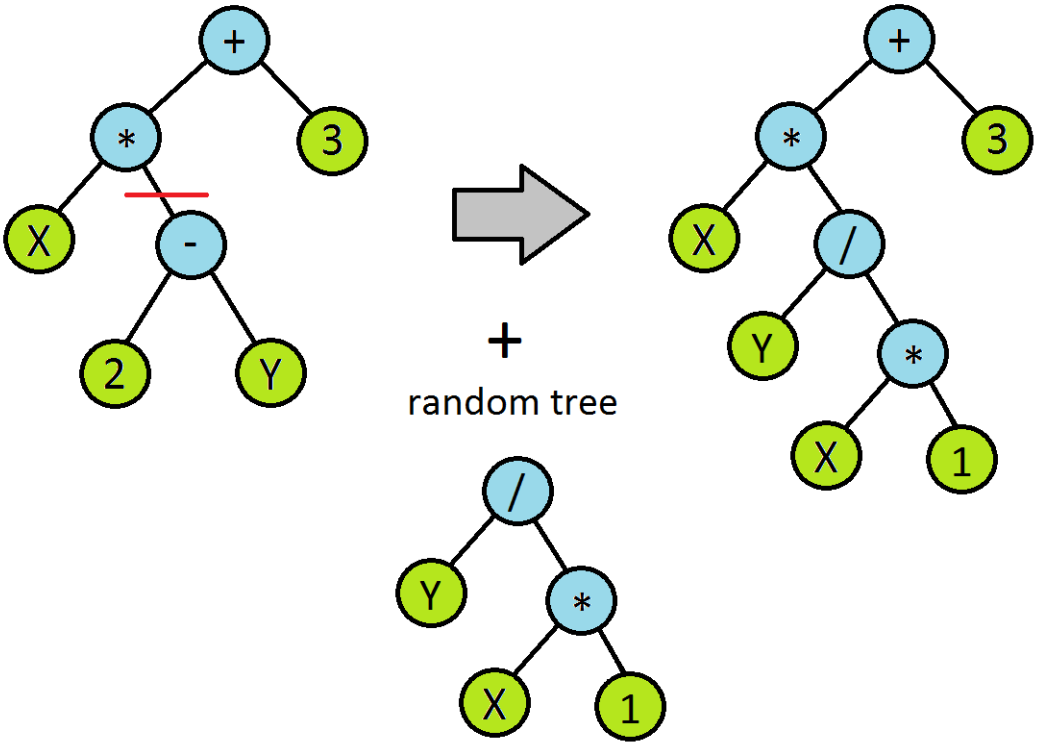

Figure 2.7: Example of tree-based mutation

2.2.2 Other representations

Although tree-based GP is commonly used, there are many other variatns of representations.

Linear Genetic Programming

Source codes of computer programs in many languages (C, Java, ...) are represented linearly (one instruction after another) and with the exception of jumps, program is linearly executed as well. Also with tree-based GP some interpreter has to be used because computers do not usually run tree-shaped programs. That are main reasons why a linear representation might be used instead of tree-based.

The idea of linear GP is that GP evolves executable code. Individual is a list of instructions, that are executed in a given order (with the exception of jumps). Instructions can be executable in CPU of computer or by some higher level virtual machine.

Crossover and mutation can be done straightforwardly — switch parts of the list between parents, change selected instruction, ... But they often need some correction to get a proper code. For example see figure 2.8.

Usually, linear GP is not used on high-level languages source code, but on assembler and similar languages source code instead.

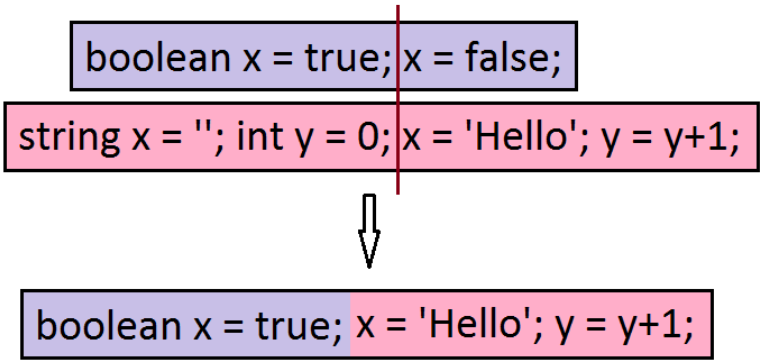

Figure 2.8: Illustration of crossover problem in linear genetic programming. Two runnable parents recombined with *one-point* crossover create non-runnable child. In the descendant there is a string 'Hello' inserted into the boolean variable x and the variable y is not initialized at all. Therefore the gained code is not executable even if the parents' codes are all right.

Graph-Based Genetic Programming

Motivation behind graph-based genetic programming is the fact, that trees are special type of graphs. Therefore graph-based GP is natural extension of the tree-based GP. Example of a simple graph-based individual is shown in figure 2.9 (b).

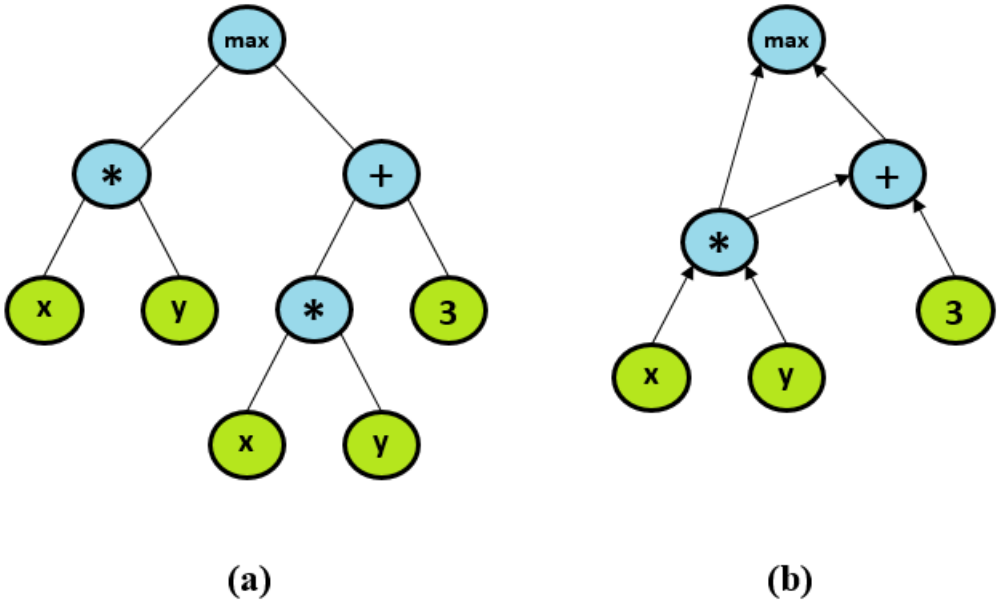

Figure 2.9: Example of individual in graph-based genetic programming (b) and its equivalent in tree-based GP (a)

Crossover and mutation operators are often complicated thus they must guarantee syntactic correctness. Therefore graph-based genetic programming is used rarely.

2.2.3 Bloat

The term bloat describes a situation where the average size of individuals in population grows quickly in time. Usually there is no significant improvement of fitness function.

It is obvious that size of infividuals must rise at the beginning of, if the algorithm started with small individuals. Nevertheless, it is unwanted to maintain too big individuals with a lot of code that does not influence fitness (e.g. $X - 3 + 3$). Moreover bigger individual means more space taken and larger cost of evaluation. That is why anti-bloat operators and measures were invented. Now we will briefly describe some common methods that hold back the bloat. As you will see, anti-bloat methods can be integrated basically within any part of the algorithm.

Size and depth limits

In this approach, size and depth of the tree is restricted. It can be simply done by checking whether new offspring fulfills size or depth limitations. If not, new

mating could be done, or fitness of such an offspring could be set so small, that it is destroyed during next generation steps.

Anti-bloat selection

Basic idea of anti-bloat selection is taking size of individual into account during the selection, e.g. it can be done by setting fitness of random *bigger than average* individual to 0.

Anti-bloat genetic operators

In this case, operators that do not increase size of an individual or even shrink it are added (e.g. the last mutation mentioned in subsection 2.2.1).

Checking and correction

If no other option is on the table, selected supervisor may check individuals and delete the code that does not influence fitness function manually.

2.2.4 Automatically Defined Functions

Modularity and usage of procedures are natural for human programmers. Automatically Defined Functions (ADF) is an approach that allows GP to do the same thing — reuse some part of code. Thus, ADF is a program that computes some part of the main program. ADF can have different functions and terminals than the main program, and ADF itself is its new function (or a terminal if the arity is 0). ADF can be specific for every individual or it can be common to the whole population. ADF can show modularity or symmetries of the problem, and so accelerate the computation.

3. The Planning Problem

In this chapter, we will show the basics of the planning problem. We will introduce the *conceptual model*, and then we will take a brief look at some algorithms solving this problem. At the end of this chapter we will show a simple comparison of mentioned algorithms.

3.1 Planning Overview

Planning has three inputs: an initial state of the world, a goal, and a list of possible actions. Then, the planning problem consists of finding such a sequence of actions that, when gradually applied to the initial state of the world, at the end satisfies the goal. This sequence of actions is called a *plan*.

The goal can be described as one particular state, but it can be much more complex, e.g, a set of constraints that need to be accomplished within the state, or even several visits of some state during the whole plan can be required.

3.1.1 Planning vs. Scheduling

Planning is often associated with scheduling, but these two terms have slightly different meaning. Planning chooses a set of actions to be used regardless of the time or resources needed for their completion. Scheduling has a set of actions given. It decides when and how those actions should be performed, with respect to time and resource constraints.

3.1.2 Domain-Independent Planning

There are two basic approaches to solve the planning problem: domain-specific and domain-independent. Domain-specific planners use information about the world to enhance effectivity of the planner. This approach is often very effective, but it has disadvantages. E.g., it is more costly to create new specific-domain planner for each planning problem than just adapt it to some existing general tool. Also, it is not a very good tool for autonomous intelligent machines, since they would be limited with their planning skills. Domain-independent planners cannot use any domain-specific information. To solve the planning problem they need to get as input the problem specification and the description of the domain. Domain-independent planners are not opposites of domain-specific planners. As general planning tools they need to be able to respond at any domain, but to achieve that they can integrate some domain-specific planners.

3.1.3 Conceptual Model

The real world is very complex. Therefore some abstraction should be used. Within this abstraction, only things that affect planning itself should be represented. For example if you are planning a car trip, you are interested whether your car is in the garage or in another city. But when your car is in front your house, you do not need to take into account whether it is on the right or left side of the road.

State-transition system (STS) is such an abstraction of the world described in [3, section 1.4]. STS has abstract possible states of the world, abstract actions, abstract events and state transition function. These *states* of the world represent all possibilities how the world can look like. By *actions* we mean all options of the planner how to change the world. *Events* also change the world, but the planner cannot influence them. Then, the *state transition function* gets actual state, chosen action and current event and returns the new state of the world.

Conceptual model interconnects STS with the planner. It can be describe through interaction between the following three components (see figure 3.1):

1. Planner — creates the plan for the controller. It gets description of the STS, the initial state, and the goal.

2. Controller — provides actions to the system according to some plan.

3. State-transition system — represents the world, changing as described above.

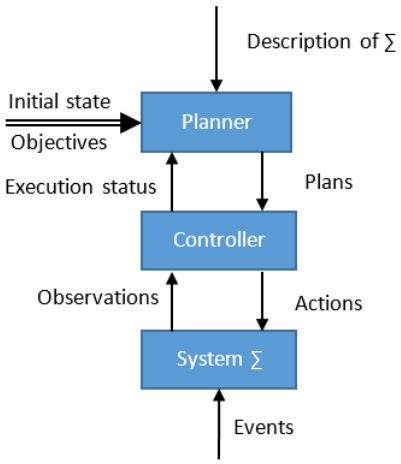

Figure 3.1: Schema of a conceptual model.

The planner creates a plan, the controller chooses the action from the plan and sends it to the system which evolves. This simple approach is called offline planning — a plan is created once, and it is unchangeable. That does not always

lead to the success, because the planner has no information about events that are going to come. For example the planner decides to use underground, but the electricity blacks out. The scheme on figure 3.1 shows online planning. The system sends some info about the new state of the world to the controller, then the controller may force the planner to replan. STS does not necessarily gives the new state of the world to the controller. For example, imagine that you are driving on the highway and you see a long queue of cars standing and blocking the way. You know that something happened so cars cannot drive through, but you do not have to know what exactly caused the traffic jam.

Conceptual model is a great abstraction, but it is still unnecessarily complex for some applications. Let us consider the following restrictions on the model:

1. *Finite system* — system description consists of finite number of states, actions and events.
2. *Fully observable system* — STS informs the controller about the whole new state of the world.
3. *Deterministic system* — each triple state-action-event leads to only one state of the world.
4. *Static system* — no events can occur in the system.
5. *Restricted goals* — goal is represented only by a set of goal states.
6. *Sequential Plans* — a plan is linearly ordered sequence of actions.
7. *Implicit Time* — actions (and events) takes instant unit of time.
8. *Offline Planning* — dynamic of the world is ignored.

These restriction can make solving the planning problem much easier, and the restricted model is still sufficient in many cases. For example, static and deterministic system can be solved by offline planning with guarantee of successful execution, if plan is found (see figure 3.2 left). On the other hand even non restricted model does not have to suffice. E.g., when plans are not sequential and time is not implicit, some scheduler has to be added to the model between the planner and the controller (see figure 3.2 right).

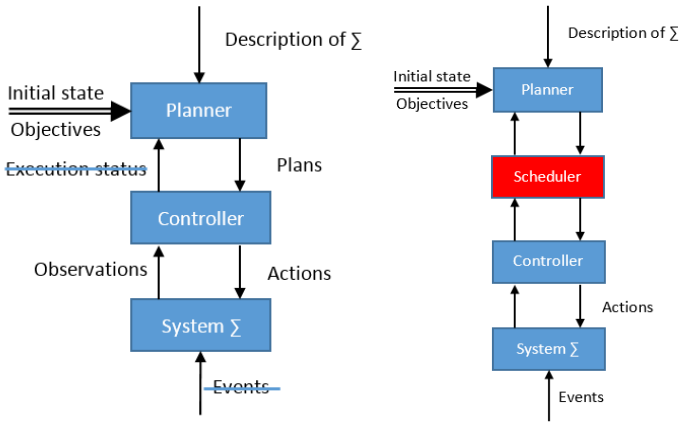

Figure 3.2: Scheme of restricted (left) and extended (right) conceptual model.

3.1.4 Classical Planning

Classical planning solves planning problem with all eight restriction mentioned above. Two basic algorithms of classical planning are *state-space* planning and *plan-space* planning. They are often used with some heuristics to reduce the time consumption.

State-Space Planning

Imagine a graph where nodes are states of the world, and two nodes are connected if one state can be obtained from the second one using some action. Then, planning means finding a path in the graph from the initial state to one of the goal states or vice versa.

Plan-Space Planning

A *plan-space* planning starts with a partial plan and expands it until the whole plan is created. During the plan expansion, a new action can be added or two actions can be ordered. Usually some variables are used, and in that case variables can be instantiated during expansion. As an example of a partial plan expansion see figure 3.3.

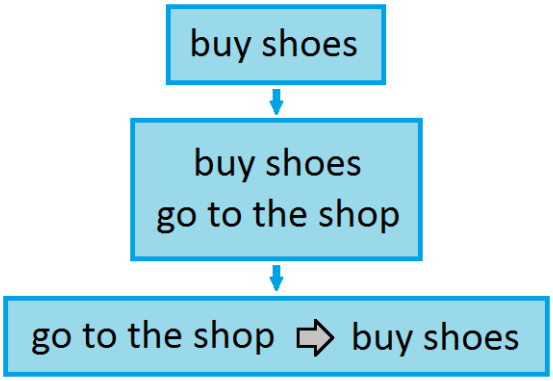

Figure 3.3: Example of plan-space planning. The goal is to have some shoes. Let us say that action *buy shoes* is already in the partial plan. In order to buy shoes we need to move to the shop. Therefore the action *go to the shop* is added to the partial plan. Going to the shop must happen before buying shoes, and so these two actions are ordered.

3.1.5 Representation

There are many representations used for describing the planning problem, and many of them uses some kind of logic. For example, in *the classical representation*, the states are encoded as a set of atoms (no variables) which determine what is true in the current state of the world. Everything not included in a state is considered as false. To understand the encoding of actions, an operator must be defined. Operator is some abstraction of action. It is defined by its name. Each operator has some preconditions (what has to be true or false in the world in order to be applicable) and some effects (how the world will change after using the operator). Operator has some variable parameters, and actions are fully instantiated operators.

Not all of the representations use logic. In *the state-variable representation* states are represented as a n-tuple of variables and actions are partial functions from one tuple to another one. If a state is seen as a set of attributes with finite domains, this approach is very useful.

Regardless of their difference, both mentioned representations are equal in expressive power.

3.2 Hierarchical Task Network Planning

A lot of tasks in the real world has a natural hierarchical structure. For example (see figure 3.4), building a house consists of building foundations, walls, roof, interior etc. During planning, there are many ways how to build a wall, but

they are all wrong if no foundation is built ahead. The *hierarchical task network* (HTN) planning is an approach which uses a natural hierarchy of tasks.

In HTN planning, two types of tasks can be solved: primitive and nonprimitive. Primitive tasks are similar to those in the classical planning, they can be executed directly. In contrast, nonprimitive tasks represent description, how to decompose some nontrivial task into a set of subtasks. The HTN planner starts with some nonprimitive tasks (e.g. build the house). It recursively decomposes its nonpromitive tasks until only primitive tasks remain in the plan.

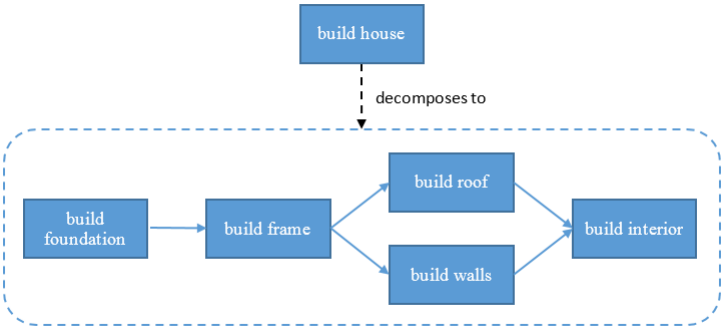

Figure 3.4: Example of hierarchical structure of task.

3.2.1 Simple Task Network

The *simple task network* (STN) planning is simplified version ofthe HTN. To understand the STN planning we need to define the following notions.

- **Operator** — Operator is an abstraction of an action, the same as in the classical planning. It is apliced to solve primitive tasks.
- **Method** — Method is a generalized operator for nonprimitive tasks. It is not an abstraction of one action, but of several ones. More precisely, a method has a list of task (subtasks) that replaces original nonprimitive task, when the method is applied. As well as the operator, in order for method to be applicable, some preconditions can be defined.
  Methods can be totally or partially ordered. While totally ordered methods define sequence of subtasks, partially ordered methods define a partially ordered set of tasks, or in other words a *task network* (see figure 3.5).

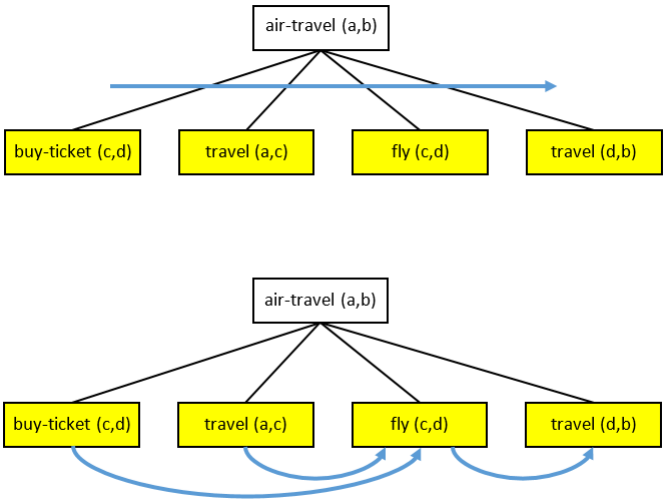

Figure 3.5: Illustration of a totally (top) and a partially (bottom) ordered method.

- **Task network** — Essentially, task network is a partially ordered set of tasks. It is represented by an acyclic directed graph. Nodes (tasks) are connected by an edge if one task must precede the second one.

A STN planner starts with an initial state of the world, an initial task network, and a list of operators and methods. *Initial task network* defines what needs to be done. Therefore, the goal of the STN planner is to decompose the initial task network with available methods into primitive tasks, and then to apply operators on them. In this manner, a list of actions that solves STN planning problem, is obtained.

3.2.2 Hierarchical Task Network

STN planning has two kinds of constraints on methods — preconditions and ordering of subtask. HTN extends a set of allowed constraints. For example *before-constraint*, generalization of preconditions, defines what has to be true before some task (or even set of task) is processed. Another example is *between-constraint* that describes what must be true right after one set of tasks is completely processed until the execution of another set of tasks begins. For more information see [3, section 11.5].

3.2.3 Comparison with classical planning

STN and HTN is more expressive than the classical planning. Authors of [3, section 11.6] claims that STN, unlike the classical planning, can express undecidable problems. Even a totally ordered STN has a higher expressive power than the

classical planning. To even the expressive power, a STN must be restricted as follows: initial task network and all methods' subtasks can contain at most one nonprimitive task and it has to be the very last task of the list.

4. Algorithm proposal

In this chapter, we will explain our evolutional algorithm created with respect to the planning problem. At first, we will explain how is the planning problem represented (cf. section 4.1). Then, we will show our solution of the representation of the individual (see section 4.2), and our choice of environment and mating selections (4.3). Next part of this chapter describes initialization of individuals (see section 4.4) and our implementation of all genetic operators (cf. section 4.5). Besides that we will also take a look at some techniques used to improve the efficiency of the algorithm (cf. section 4.6), namely a local search and running parallel subpopulations. Then, we will introduce our solution of partial ordering of methods (cf. figure 3.5) in section 4.7. After that, our design of the fitness function will be explained in section 4.8. We will define the distance heuristic (see subsection 4.8.2) and introduce the concept of the individual equivalence (cf. subsection 4.8.3) in this chapter. In the end, we will show some optional parts of the algorithm and we will discuss their effects.

4.1 Planning problem representation

In order to explain our algorithm we need to resolve how the planning problem is represented in a first place. We are using classical representation described in section 4.5 and extended to support HTN. To be more precise we define domain that consists of:

1. list of objects and properties located in the world, and their quantity;
2. list of predicates that can be true in the world, and their parameters;
3. list of operators, their preconditions and effects;
4. list of methods, their decomposition, time constrains (order of decomposed parts), and another information that will be further discussed in this chapter — distance heuristic (see subsection 4.8.2 for more information).

To remind, operators are actions that can be done in the world, methods cannot be directly used, but they explain how to decompose some task into set of subtasks. Now, let us define an example word to easier describe our algorithm (see figure 4.1, figure 4.2, figure 4.3, and figure 4.4). We will describe this world without distance heuristic first and we will extend our example when needed. We will also describe just the domain, but not a particular world (e.g. how many

locations are in our world and how they are connected?). The world used will differ across examples, but we will always describe it when necessary.

box — container that can be transported
vehicle — means of transport
location — space to transport vehicles
crane — tool used to load box on vehicles

Figure 4.1: Objects and properties of the example domain.

Let us briefly explain objects and properties of our example, that can be seen on figure 4.1. There are four kinds of objects in our example domain — box, vehicle, location and crane. However that does not mean that there are 4 objects in this world. Each type can be presented there in different copies, e.g. if there are 50 *locations*, we will be referring to then as *location 1*, *location* 2, ..., *location 50*.

isAdjacent (loc #1, loc #2) — location #1 and location #2 are connected
isGround (loc #1) — location #1 is a ground
isWater (loc #1) — location #1 is a water
isCar (veh #1) — vehicle #1 is a ground type of transportation
isBoat (veh #1) — vehicle #1 is a water type of transportation
occupied (loc #1) — some car is occupying location #1 (no other car is allowed)
boxOnVehicle (box #1, veh #1) — box #1 is on vehicle #1
boxOnLocation (box #1, loc #1) — box #1 lays in a pile of boxes on location #1
boxOnCrane (box #1, crane #1) — box #1 is on crane #1
boxOnBox (box #1, box #2) — box #1 is on box #2
boxOnTop (box #1) — box #1 is on top of a pile
vehOnLoc (veh #1, loc #1) — vehicle #1 is on location #1
craneOnLoc (crane #1, loc #1) — crane #1 is on location #1

Figure 4.2: Predicates of the example domain. Please take into account that location #1 is not the same as *location 1*. *Location 1* represents real location in the world, that got label 1. Location #1 represent the first location parameter of some predicate.

Now we will focus on meaning of chosen predicates, present on figure 4.2. There are two types of *locations* in our domain — *ground* and *water*. Therefore, we also have two types of transportation — *cars* and *boats* — they can visit respective *locations*. Some *location* can be of both types, e.g. ports or bridges and so it can be visited by both types of *vehicles*. Only one *car* can be on a *location* at a time, but any number of *boats* can be presented there. A *car* and *boats* can also share *location*.

Boxes can be placed on some vehicle, on a crane or they can lay directly on some location (eventually they can be stacked up on itself on that location).

moveGround (vehicle V, location startL, location targetL)
description: moves car from startL location to targetL location if it is possible
preconditions: isCar(V), isGround(targetL), vehOnLoc (V, startL), -occupied(targetL), isAdjacent (startL, targetL)
effects: -occupied(startL), occupied(targetL), -vehOnLoc (V, startL), vehOnLoc (V, targetL)

---

moveWater (vehicle V, location startL, location targetL)
description: moves boat from startL location to targetL location if it is possible
preconditions: isBoat(V), isWater(targetL), vehOnLoc (V, startL), isAdjacent (startL, targetL)
effects: -vehOnLoc (V, startL), vehOnLoc (V, targetL)

---

take (crane C, box B, location L)
description: lift box from the ground to the crane on a given location
preconditions: boxOnLocation(B, L), boxOnBox(B, nextB), boxOnTop(B), craneOnLoc(C, L)
effects: -boxOnLocation(B, L), -boxOnBox(B, nextB), -boxOnTop(B), boxOnCrane(B, C), boxOnTop(nextB)

---

load (crane C, box B, vehicle V, location L)
description: puts box from crane to vehicle on given location
preconditions: vehOnLoc (V, L), craneOnLoc(C, L), boxOnCrane(B, C)
effects: -boxOnCrane(B, C), boxOnVehicle(B, V)

---

reload (vehicle V, vehicle W, location L)
description: reloads a box from one vehicle to another
preconditions: vehOnLoc (V, L), vehOnLoc (W, L), craneOnLoc(C, L), boxOnVehicle(B, V)
effects: boxOnVehicle(B, V), boxOnVehicle(B, W)

Figure 4.3: Operators of the example domain.

Let us take a look at actions that can be used in our example domain. They are all described at figure 4.3. *Boats* can move to adjacent *water location*, *cars* can travel to adjacent *ground location*. *Crane* can lift a *box* from the ground and put it on a *vehicle*. The last operator allows reloading a *box* from one *vehicle* to another, if they are on the same *location* and a *crane* is present.

l*oadRobot* (crane C, box B, vehicle V, location L)
description: loads vehicle with a box
child 1: take C, B, L
child 2: load C, B, V, L
time constraints: child 1 < child 2

---

summonVehicle (vehicle V, location targetL)
description: moves vehicle to the target location
child 1: movingAction V, startL, targetL

---

transhipmentMove (vehicle V, location startL, location targetL)
description: allows change of a vehicle carring a box
child 1: summonVehicle W, startL
child 2: reloadBox V, W, startL
child 3: movingAction W, startL, targetL
time constraints: child 1 < child 2< child 3

---

recursiveMove (vehicle V, location startL, location targetL)
description: allows moving vehicles through bigger distance
child 1: movingAction V, startL, middleL
child 2: movingAction V, middleL, targetL
time constraints: child 1 < child 2

---

transportBox (box B, location targetL)
description: transports a box to target location
child 1: loadRobot C, B, V, startL
child 2: movingAction V, startL, targetL
time constraints: child 1 < child 2

---

transport2Boxes (box B1, location L1, box B2, location L2)
description: transports two boxes to target locations
child 1: transportBox B1, L1
child 2: transportBox B2, L2

---

moveTwoVehicles (vehicle V1, location L1, vehicle V2, location L2)
description: moves two vehicles to target locations in a given order
child 1: summonVehicle V1, L1
child 2: summonVehicle V2, L2
time constraints: child 1 < child 2

Figure 4.4: Methods of the example domain. MovingAction reffers to any method or operator connected with moving: *moveGround*, *moveWater*, *recursiveMove* or *transhipmentMove*.

Now it is time to introduce HTN methods shown at figure 4.4. At first let us say what *movingAction* is. It is any method or operator that involves moving, thus it is *moveGround*, *moveWater*, *recursiveMove* and *transhipmentMove* from our example. If some method uses it, it says that any of those can be used.

RecursiveMove method splits path that needs to be overcome into two smaller parts that is easier to overcome. The *transhipmentMove* method makes the necessary arrangements to transship *box* from one *vehicle* to another. Thanks to these two methods, any distance — even with different types of locations — can be overcome. Then l*oadRobot* method ensures that a *vehicle* is loaded with a *box* laying on a ground. Now we have prepared everything to a transport chosen *box* to the target *location* via *transportBox* method or even two *boxes* via *transport2Boxes* method. Except transporting *boxes*, we can also transport *vehicles* itself. The *moveTwoVehicles* method ensures that two vehicles get on target *location* in a given order.

SummonVehicle serves as auxiliary method to bring a *vehicle* to the target *location*. Calling this method can be simply replaced by calling *movingAction*. Later in this chapter we will show how this auxiliary method is useful.

4.2 Representation of the individual

Since we are utilizing the tree-based GA, the individual is represented as a tree. Hierarchical structure of the planning problem can be encoded in a tree naturally in the following straightforward way: nonterminals are methods of the problem, and terminals are operators. For example, if we want to load up some *vehicle*, we will use the l*oadRobot* method and the *take* and the *load* operators. Then, the tree will look as in figure 4.5 up.

Now let us take a look at slighltly different problem — loading *vehicle 1* and loading *vehicle 2* are different actions. Therefore we need to take parameters of each method and each operator into account. We can assume that each parameter of each method will be also parameter of at least one operator. Otherwise that parameter would not matter, as it would not influence the world, since only operators can change it. Accordingly, we can extend our representation. This time both methods and operators are nonterminals, and parameters are terminals (see figure 4.5 down). However, it is true, that this tree without parameters is equivalent to the first approach. Hence, we will call nodes representing operators *pseudo-terminals*. We decided for the second mentioned representation over the first one with some inner parameters representation for the following reasons: it is natural and easy to work with. For example, it is obvious what crossover does with parameters. It is also possible to correct almost good tree with only bad parameter by a simple mutation. Hence, no specialized operators are needed.

Our trees are typed. That means that each node has some type, and it also requires each of its children to be some specific type. Thanks to that it is possible to ensure that a *box* is always a *box* during initialization of a tree, breeding of a

tree and so on. It is also an easy way how to deal with *movingAction* introduced in figure 4.4. Simply all method and operators representing movement will have the same type.

To sum it up, we are using typed trees, and we have tree types of nodes — nonterminals, pseudo-terminals and terminals. Pseudo-terminals are nonterminals whose all children are terminals. Nonterminals represent methods, pseudo-terminals represent operators and terminals represent parameters of the planning problem.

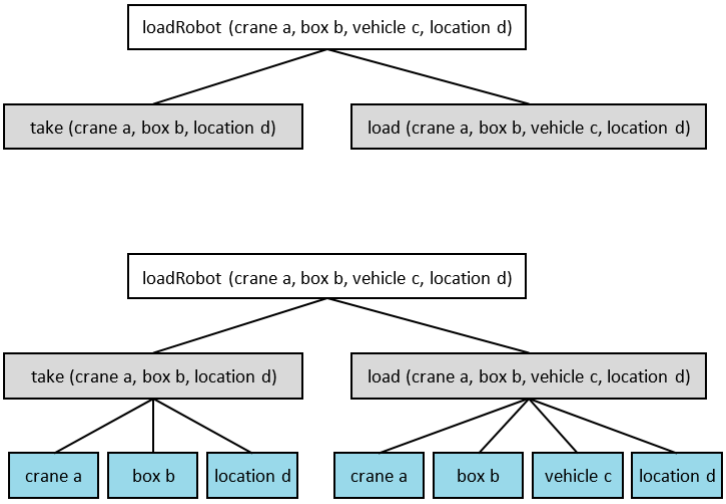

Figure 4.5: Comparison of a tree with operators as terminals (up) and a tree with parameters as terminals (down). White color represents methods, gray represents operators and blue represents parameters.

4.3 Selection

As we mentioned in chapter 2 we have two basic selections in evolutional algorithms. The environmental selection is in our case very simple. We choose one — the best — individual from the old population, and then we fill up the new population with new individuals. The mating selection is a little more complex. We are using a modification of the *tournament selection*. This selection chooses only two individuals to compete each other. The first one is chosen randomly from the entire population. The second one is also chosen randomly from the whole population with the 50% chance. Nevertheless, the second individual can chosen randomly from the set of equivalent individuals with the probability of 0.5 (for more information about equivalency between individuals see Subsection 4.8). For a better illustration see algorithm 4.1.

Algorithm 4.1 Environmental selection used in our algorithm.

```
1  Individual tournamentSelection(boolean firstChoice)
2  {
3    if (firstChoice)
4      return randomIndividual;
5    else
6      if (rand.nextDouble()<0.5)
7        return randomIndividual;
8      else
9        return randomEquivalentIndividual;
10 }
```

4.4 Initialization

To initialize our trees, we are using the *ramped half-and-half* method described in chapter 2 in section 2.2.1. However, since we have three types of nodes, both *full* and *grow* methods had to be modified. Otherwise they both would behave as *grow* method without any boundaries. For exaple imagine that you want a full tree with a depth of five, so at a depth of one *full* method chooses some nonterminal. Unfortunatelly it chooses a pseudo-terminal, and therefore at a depth of two it has no choice than to choose a terminal. Vice versa, if that method chooses a real terminal at a depth of four, no terminal at a depth of five is available. Therefore, both of our methods are modified to look one step ahead and choose pseudo-terminal if in the next step a terminal is required, and a real nonterminal if nonterminal is desired. Those modifications are shown in algorithm 4.2 and in algorithm 4.3. However, no modification does ensure full trees. For instance, if the goal of the planning is to transport a *box*, the left part of the tree at figure 4.6 will always have a depth of four.

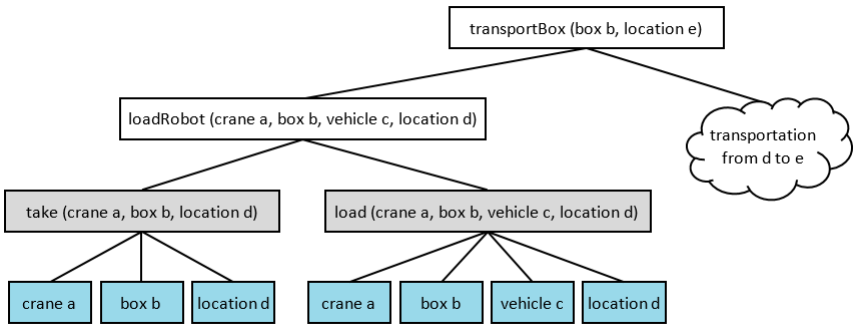

Figure 4.6: Example of a tree whose left part will have a depth of 4, regardless the chosen initialization method.

Algorithm 4.2 Full method modification.

```
1 Node full(int depth, int limit)
2 {
3   if (depth == limit)
4     return randomTerminal(type);
5   else if(depth + 1 == limit)
6     return randomPseudo-Terminal(type);
7   else return randomNonterminal(type);
8 }
```

Algorithm 4.3 Grow method modification.

```
1 Node grow(int depth, int limit)
2 {
3   if (depth == limit)
4     return randomTerminal(type);
5   else if(depth + 1 == limit)
6     return randomPseudo-Terminal(type);
7   else return randomNode(type);
8 }
```

4.5 Genetic operators

In order to evolve our trees we chose four genetic operators — *basic crossover*, *basic mutation*, *one-node mutation* and *all-nodes mutation*. When an individual is selected to breed, one of those operators is chosen randomly, but not uniformly, and applied on a given individual. The probability of individual operators was set experimentally. Let us describe the chosen operators in detail:

1. **basic crossower** — this operator is the only one that needs two individuals and also produces two new individuals. It chooses one node in each individual and simply swaps two subtrees rooted in selected nodes. Its probability is set to 0.1.

2. **basic mutation** — this mutation chooses one node and by using the *grow* method it creates new subtree. It is the most often used operator with the probability of 0.6.

3. **one-node mutation** — this operator chooses one node and replace it with another node with the same arity and type constraints. It occurs in 10% of cases.

4. **all-nodes mutation** — last operator on the list is very similar to the *one-node mutation*. It also chooses one node and replace it with another node with the same arity and type constraints, but it also does it to all nodes in a subtree rooted in a selected node. It has a probability of 0.2.

So far, we were talking about operators choosing nodes to work with. Now it is time to explain how they do that. Operators choose the node randomly with the uniform distribution, but not from the all nodes of a tree. Let us define the problematic node as a first node in which some problem occurs during evaluation, e.g. if some operator should be applied and his preconditions are not fulfilled. Then, operators can choose from the problematic node, its ancestors and any of its offspring.

4.6 Efficiency improvements

During the development of the algorithm, it became clear that in our case the exploration is more crucial than exploitation. Since the search space of our algorithm is highly constraint, the algorithm tends to stuck in a local minima. That is why most of the decisions was made to support exploration.

4.6.1 Exploitation techniques

Most of the choices in our algorithm prefer exploration at the expense of exploitation due to mentioned reasons. But it does not mean that no decisions strengthening exploitation were made. Most of them, however, only prevents exploring obviously wrong parts of the planning space.

At first, let us look one more time at figure 4.5. As can be seen both *take* and *load* operators have a parameter *crane a*, which is obviously the same (and their parent has also the same parameter). But they are two different nodes in a tree, which can lead to a situation, that they represent different *cranes*. Then, the tree is obviously wrong. Fortunately we can simply avoid such a situation. There are two solutions — the parameters representing the same object in the world would be one node, or we can utilize parameter passing. The first option would change the structure of the individual into a general graph. Therefore, we would loose the advantage of the tree structure. We would have to create our own special operators and initialization instead of using existing and optimized ones. The second approach — the parameter passing — is more simple, yet with the same result. The pseudocode of this approach is shown in algorithm 4.4. The first time we visit terminal during evaluation we randomly choose what object it is representing, and remember it for future runs of the algorithm (line 10) as

its *id*. The terminal sends this information to its ancestors (line 12) and they send it to all their sons requiring that information (see figure 4.5). Then, when we visit the terminal with additional information about the representing object, we simply set it (its *id*) as required (line 8). This approach can be imagined as function parameters passing — nodes are functions calling other functions (their children) using the same parameters. But what happened when some of the terminals would be changed by mutation or crossover? If it is the first one visited, all terminals representing the same object will be changed. If it is another one, it will be repaired back. Both choices cost us constant amount of time, since changes are processed when visiting the nodes during evaluation and we need to visit every node anyway.

As a part of the world initialization, compulsory passed parameters can be set. This assures that the final tree will have some parameters set as required. This also helps preventing exploring the wrong parts of the planning space, but it depends on domain initialization, and so it is not guaranteed in general.

Algorithm 4.4 Illustration of a terminal evaluation function.

```
1  class Terminal : Node
2  {
3    Integer id = null;
4
5    void eval(ref Integer parameter)
6    {
7      if(parameter != null)
8        id = parameter;
9      else if(id == null)
10       id = rand.nextInt(numOfThisObjects);
11
12     parameter = id;
13   }
14 }
```

Now let us focus on a different problem that is not a problem for humans usually, but it is rather a big one for algorithm described so far. Imagine world described at figure 4.7. Obviously, *crane 1* needs to be used to pick up the *box*, and it will load it on *car 1*. But when parameters are chosen, this information is not taken into account. Therefore, nothing stops the algorithm to choose *crane 15* to lift the *box*. Problem will be found when the *take* pseudo-terminal will check its preconditions, such as that a *crane* and a box has to be at the same *location*. We have 12000 possibilities how to choose *location*, *crane, vehicle* and *box* $(1 * 20 * 20 * 30)$ in this world, and only 1 is the correct choice. This problem leads us to define the local search on pseudo-terminals (see algorithm 4.6). The

local search is triggered only if some parameter is not defined (see algorithm 4.5), and it guarantees that only a valid set of terminals (with respect to preconditions of the pseudo-terminal and passed parameters) would be taken into account in most cases. However, an invalid set of operators can still appear. For example, when no valid one is possible, but also after the crossover or after change of passed parameters.

Last but not least, the exploitation support was already mentioned in section 4.3. This is the only exploitation technique that is created at the expense of exploration. We are talking about our modification of tournament selection. The ordinary tournament selection is the best for exploration. The higher the probability of finding equivalent individual, the higher is exploitation.

We have a big world with 50 *locations*, 20 *cranes*, 20 *vehicles and 30* boxes. *Crane 1* and *car 1* is on *ground location 1* and they are the only *crane* and the only *vehicle* standing on that *location*.
Target: load some *box* that lays on the *location 1* on some *vehicle*.

- Number of correct combination of parameters of *take* operator: 1
  - only one *box* is on top of stack, location is given and only one *crane* is on that location
- Number of all possible combination of parameters of *take* operator while passing parameters: $600 = 20 * 30 * 1$
  - location is given, the rest is chosen randomly
- Number of all possible combination of parameters of *take* operator: $12000 = 20 * 30 * 20$
  - even location must be chosen randomly
- Number of correct combination of parameters of *load* operator: 1
- Number of all possible combination of parameters of *load* operator while passing parameters: $20 = 1 * 1 * 20 * 1$
  - *vehicle* must be chosen randomly, other parameters was already chosen by *take* operator
- Number of all possible combination of parameters of *load* operator: $600000 = 20 * 30 * 20 * 50$

Figure 4.7: Motivation example for local search.

Algorithm 4.5 Pseudo-terminal evaluation function.

```
1  void eval(ref Integer[] parameters)
2  {
3    foreach(node child in children)
4      if(child.id == null)
5        doLocalSearch(parameters);
6
7    for(int index = 0; index<children.length; ++index)
8        children[index].eval(parameters[index]);
9
10   bool ok = checkConstraints();
11
12   if(ok)
13     changeTheWorld();
14 }
```

Algorithm 4.6 Illustration of local search function in *take* pseudo-teminal. It is assumed that *location* parameter is given.

```
1  void doLocalSearch(ref Integer[] parameters)
2  {
3    int loc = parameters[2];
4
5    for(int cr = 0; cr<craneNum; ++cr)
6      for(int box = 0; box<boxNum; ++box)
7        for(int box2 = 0; box2<boxNum; ++box2)
8          if(world.boxOnLocation(box, loc)&&
9              world.boxOnBox(box, box2)&&
10             world.boxOnTop(box)&&
11             world.craneOnLoc(cr, loc)
12           saveCombination;
13
14   parameters = randomConbination;
15 }
```

4.6.2 Exploration techniques

The majority of the algorithm was selected to improve exploration. Namely, we are using tournament selection, and it only takes two individuals to compete. Further, despite the fact we are using elitism, we choose just one — the best — individual to join the new population. Even the probabilities for the operators were chosen to support exploration. Even the fitness was bent to support unique individuals (as you can read in section 4.8). Local search in pseudo-terminals is triggered on all parameters of the pseudo-terminal even if just a single parameter

is unknown as it is shown in the algorithm 4.6. Of course, this does not involve passed parameters that are superior to local search.

Moreover, our algorithm consist of four small separated subpopulations rather than one big population. This subpopulations never communicate to each other and the algorithm is considered successful if any of those subpopulations find the ideal individual, and therefore, the plan. Number of subopulations was chosen experimentally.

4.7 Partial order

In totally ordered trees, order of the nodes is given — from up to down, from left to right. But with partial order, the order of nodes has to vary. However, the plan is actually specified just by the order of actions. Therefore, the order of pseudo-terminals in a tree is the only crucial information to determine the plan, even in partial ordered trees. Then we could look at this problem from the other side. Order of nodes in a tree can remain unchanged, all we need is an order of pseudo-terminals. The comparison of these two approaches is shown at figure 4.8. These approaches are equivalent. The way the tree is traversed needs to be changed in both cases and it is clear how to do it with the second representation of a tree. Traversal starts in the root, and goes down to the first pseudo-terminal. Then, it starts in the root again, and continues into second pseudo-terminal etc. That's the reason why we chose the second approach.

At the beginning the order of pseudo-terminals is random, fulfilling time constraints. At the beginning of evaluation order of pseudo-terminals is updated. Nodes that was in the tree during last run of the algorithm stays on their positions. New (created by a mutation or received through a crossover) pseudo-terminals' order is chosen randomly with the respect to the time constraints as shown in algorithm 4.7. Then, with a small probability, one of the pseudo-terminals is selected and its order is changed (randomly with the respect to the time constraints).

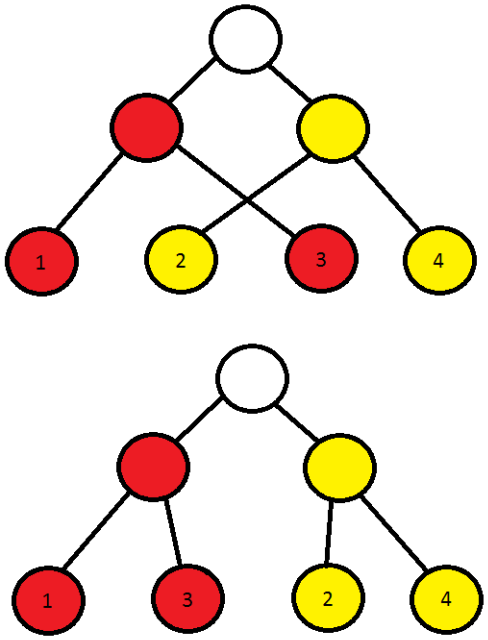

Figure 4.8: Illustration of two versions of a partially ordered tree.

Algorithm 4.7 Update of partial order of a tree.

```
1  void updateOrder()
2  {
3    foreach(newNode in newPseudo-Terminals)
4    {
5      min = 0;
6      max = orderedPseudo-Terminals.length;
7      foreach(oldNode in orderedPseudo-Terminals)
8      {
9        if(oldNode.isBefore(newNode))
10         min = Math.min(min, oldNode.order);
11       if(oldNode.isAfter(newNode))
12         max = Math.max(max, oldNode.order);
13     }
14
15     int index = min + rand.(max-min+1);
16     orderedPseudo-Terminals.add(newNode, index);
17   }
18 }
```

4.8 Fitness function

Creating fitness function was the most crucial part of our algorithm, since it is not clear at all how to evaluate a partial plan. When an individual is evaluated,

it starts with the initial state of the world, and ends with some other state of the world. If the result is the goal state of the world, it is also the required plan. If two individuals end in the same state of world, they will have similar fitness value. So we can also look at a comparison of two individuals as a comparison of two states of the world. Of course, we must take into account, that the individual ending in some state of the world doing nothing should be better than the individual ending in the same state of the world doing many things.

4.8.1 Tree Evaluation

Our individuals are always fully initialized (all operators and methods are chosen, and all parameters are set if corresponding pseudo-terminal was evaluated). But it does not assure that all action acquired from the individual are permissible. E.g., a plan can contain action to *move* a *car* from some *location* even if no car is on that *location* present. Of course, this action is wrong and does not change the world. However, there are two possible ways what to do with such an action. We can ignore it and continue evaluating the individual, or we can stop the evaluation.

The benefit of ignoring such an action is that the whole tree can be evaluated every time. Moreover, some parts of the plan can be evolving even if there are some problems at the beginning of the plan. For example, we can find the path to the target *location* before we figure out how to *load* up the *car*. Unfortunately, this is also the major disadvantage of this approach. Once some subtree repsesenting a part of the plan is relatively coherent, it is very hard to change it. But part of the plan that makes perfect sense under certain conditions can be completely wrong under different ones. For example, look at figure 4.9. Unfortunately, when this tree evaluation is selected, similar situations to the example are common and often occurs for different reasons. Therefore, it is not possible to detect them before the prior part of the plan is actually done.

Because of problems connected to the first mentioned approach, we decided for the second one. The procedure stops tree evaluation when any problem appears. Let us define that node is OK, if it was visited during evaluation, and no problem occurred in it or any of its children. This approach is indicated in algorithm 4.8. The disadvantage of this evaluation is that application of some genetic operators does not influence the evaluation at all if it is done in the wrong part of the tree. To avoid that we are using a very specific node selector for genetic operators as mentioned in section 4.5.

The world displayed here consists of 8 *locations* connected to a circle, and two *cars*. The goal is to get the blue *car* to the blue target *location* and the red *car* to the red target *location* in this order. To do so, we can use the *moveTwoVehicles* method. In this example, the whole individual is evaluated, bad actions are skipped.
Unfortunately, during evaluation some problems occurred in the blue *car* movement, however the red *car* finished its path by going clockwise (there were the blue *car* blocking the way on the left). This plan finished a lot from its objectives, and so the individual gets a high rating. Then, the blue *car* movement is being fixed. It moves first, so it cannot go right, and so it moves counterclockwise instead. Once it moves on the *location 5*, the red *car's* plan cannot be used anymore, and the individual gets much lower rating, since no goal is achieved even if it is the correct solution.

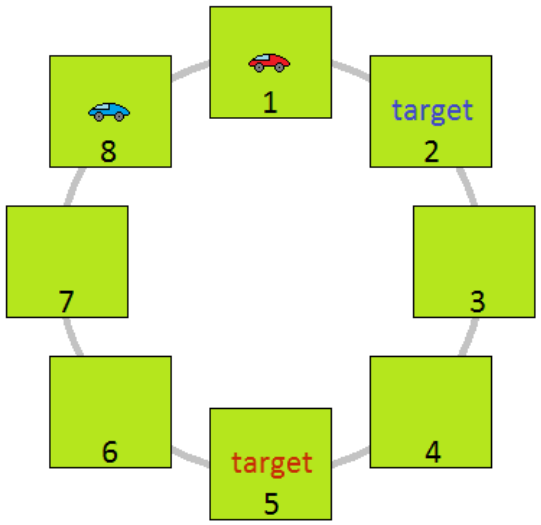

Figure 4.9: Illustration how dynamism of the world can change conditions and destroys plans when done ahead.

4.8.2 Distance Definition

Let us consider how to compare the states of the world. What state of the world is better than the other? The one that fulfills more goals is obviously better. But what if all of them have the same amount of fulfilled goals? Then it should be the one that is closer to fulfill thr next goal. But how far is the state of the world from fulfilling some goal? The answer to this question is not so easy, and it is depends highly on the description of the domain and the description of the problem. Now, we will explain one of the most important heuristics of our algorithm that can dramatically improve its efficiency.

As you can remember from section 4.1, our domain representation consists of list of predicates. This predicates basically say what is true and what is false in the world. They are defined by their name and parameters. Let us take a closer

look at one predicate from our earlier example: *isAdjacent* (with two *locations* as parameters). This predicate determines which two *locations* are connected to each other. You may notice that this predicate is actually defining a distance between *locations*. If for two locations the predicate *isAdjacent* is true, then their distance is one. If two *locations areAdjacent* to the same *location*, but not to each other, their distance is two etc. Actually this observation applies for all similar predicates — predicates with two parameters with the same type. Now we can define a distance heuristic: if a part of the objective is to overcome some distance, we can compare two states of the world by comparing their remaining distances to the goal. The distance we have defined estimates the value of the objective function. For example, if two *locations* have distance of five, it means that *at least* five steps need to be done to get from one *location* to another, because some obstacles can be on the path. In order to use the distance, it must be defined on methods. Then, when the node representing given method is in the tree, the distance can be counted. To be more precise, the distance is defined all the time. Each method has to define between what it is counted, what distances it is using (more than one can be defined by domain representation), and eventually, how to combine them. Let us extend our example 4.4 with distance definition as shown in example 4.10.

We are actually using two types of distance heuristics in our algorithm — *compulsory* and *optional*. They are both counted equally, the difference is when they are counted. The *compulsory* distance heuristic is computed for each node representing a given method in a tree. It makes no difference whether the node was visited during evaluation. Therefore, it is good to use it on methods with a predetermined number of appearances in a tree. On the other hand, the *optional* distance heuristic is counted only on nodes that were visited during the evaluation. To be more precise, it is counted on nodes, that are considered an *actual problem* (cf. 4.8.3 for more information about actual problem). For example, if you want to transport two *boxes*, you can count distance of the *location* of the *boxes* and their target *locations* as shown in the *transportBox* method (4.10). Since there are always two *boxes*, the *compulsory* distance heuristic can be used. Then, this heuristics will be representing the minimal distance needed to be overcome in order to fulfill both goals. Another example is the *summonVehicle* method. It counts the distance between a wanted *vehicle* and a target *location*. There is no guarantee how many summoning actions will be needed to fulfill the goal, and so the *optional* distance heuristic should be used. We are saying that distance heuristics is *fulfilled*, when it is zero.

4.8.3 Individual Equivalence

In this subsection we will describe which individuals are equivalent. Simply stated, two individuals are equivalent if they fulfill the same goals and they are solving the same problems. The first part of this statement should be clear. But what problems are the individuals actually solving? Is it simply the rest of the goals? The example at figure 4.11 shows the problem of our current approach — that some individuals should not be compared together straightforwardly. This leads us to define the *actual problem* with assistance of the distance. The actual problem of an individual is a type of each node that is not done, defines the distance, and none of its offsprings define an actual problem (see fiure 4.8). Then, individuals are equivalent if they fulfill the same goals and they have the same set of actuals problems. To be more precise, the actual problem can be defined also by some parameters. For instance, all the individuals who are requesting a *vehicle* to *location 5* are equivalent. These parameters must be defined in the domain representation, like in example 4.10.

Since distance heuristic defines the actual problem, it should not be defined on recursive nodes (where type of some child is the same as the type of the node). To prevent problems from figure 4.12 from happening, some other method should take care of counting distance. This new method can only call the problematic recursive method, and it can be called only if a new distance is needed. Actually, example of such a method is the *summonVehicle* mentioned above.

summonVehicle (vehicle V, location targetL)
description: moves vehicle to the target location
child 1: movingAction V, startL, targetL
equivalency: summonVehicle to targetL
optional distance:
isAdjacentDistance (L, targetL) :- *vehOnLoc* (V, L)

- distance between target location and actual location of a vehicle is counted.

---

transportBox (box B, location targetL)
description: transports a box to target location
child 1: loadRobot C, B, V, startL
child 2: movingAction V, startL, targetL
time constraints: child 1 < child 2
equivalency: transportBox B to targetL
compulsory distance:
min
isAdjacentDistance (L, targetL) :- *vehOnLoc* (V, L) & *boxOnVehicle* (B, V)
isAdjacentDistance (L, targetL) :- *craneOnLoc* (C, L) & *boxOnCrane* (C, V)
isAdjacentDistance (L, targetL) + *boxOnBoxDistance* (topB, B) :- *boxOnLocation* (B, L) & *boxOnLocation* (topB, L) & *boxOnTop* (topB)

- if the box is on a vehicle, the distance between actual location of that vehicle and target location is counted
- if the box is on a crane, the distance between actual location of that crane and target location is counted
- if the box lays in the pile on some location, the distance between the location and the target location is counted. However, also the distance between selected box and box on the top of stack is counted, since it will be necessary to dig it up first
- then, minimum of those distances is chosen as a real distance of this method

Figure 4.10: Example 4.4 extension with distance definition.

Imagine the following situation. Your task is to transport the *box* over a *water*. You can use a bridge, or you can transship the *box* onto the *ship*. One individual is waiting for the *ship*, another one took a bridge. The distance of the *box* and the target *location* is the same for both individuals. Therefore, they would have the same fitness. However, the first one needs to wait untip a *ship* arrives and that can be far away from the transshipment *location*. Therefore, its real distance to the target place is actually increased by the distance of the *ship*. But if that would be done, no transshipment would ever occurred — its fitness would be much worse than a fitness of an individual moving across the bridge. Not even if it is the only right path since some obstacle is blocking the bridge. Now the individual evolved and moved *ship* one step towards the transshipment *location*. Its fitness should be better than before evolution, but is should still be the same as the second individual fitness, since their distances from the target did not change.

Figure 4.11: Motivation example leading to defining individual equivalence.

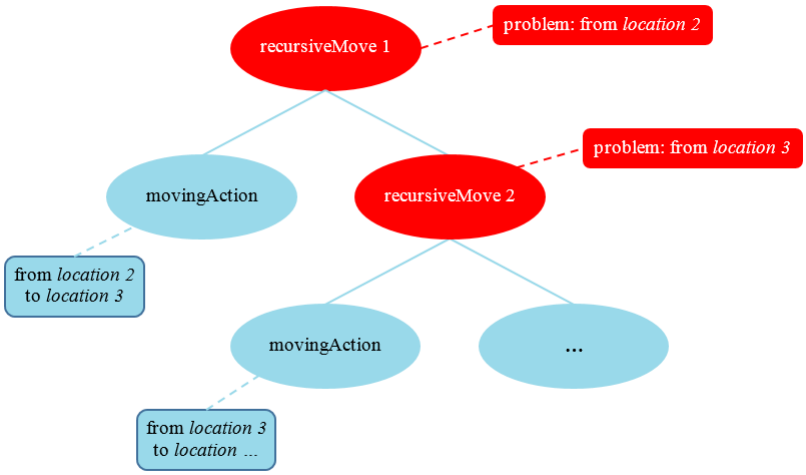

Figure 4.12: Illustration of problems connected with defining the actual problem on recursive node. Equivalent are all individuals moving from *location 2*, therefore even this individual. The recursiveMove 1 node wants to move form *location 2* as required and the movement to *location 3* succeeds. Now, the recursiveMove 2 wants to move from *location 3* and fails. Since the last mentioned node must be defining the actual problem, this individual would not be equivalent to others in spite of the fact that it is actually moving from *location 2*.

Algorithm 4.8 Terminal evaluation function.

```
1  void eval(ref Integer[] parameters, int[] path, int ok)
2  {
3    int nextChild = path[depth];
4
5    if(ok)
6    {
7      Integer[] childParams = prepareParameters(nextChild);
8      children[nextChild].eval(childParams, path, ok);
9      parameters = fixParameters(childParams);
10   }
11
12   if(!ok)
13   {
14     if(problematicNode == null)
15       problematicNode = this;
16
17     boolean canBeProblem = true;
18     foreach(Node child : children)
19       if(child.hasSomeProblems())
20         canBeProblem = false;
21
22     if(canBeProblem && iHaveDistance())
23       setActualProblem(this.type, distanceParameters);
24   }
25 }
```

4.8.4 Antibloat Treatment

As we mentions in subsection 2.2.3, bloat is unwanted, since it makes evaluation more costly without any fitness improvement. Now, we will explain how we fight against it.

In our algorithm we are using two bloat penalizations. The firts of them penalizes the fitness of the individual by value $\frac{1}{10}$ for each recursive node in a tree. This penalization causes that no too big or even infinite trees are developed. The second one penalizes fitness by value between 0 and 1. It is defined recursively as shown in algorithm 4.9. The second bloat penalize forces trees to finish nodes as high as possible. It will also influence recursive pair of nodes (A calls B, B calls A) or even more hidden recursion.

Algorithm 4.9 Illustration of second antibloat penalization. This method is called with root node and maxPenalization = 1.

```
1  double penalization2(node actNode, double maxPenalization)
2  {
3    if(node.ok) return 0;
4
5    double nextPenal = maxPenalization/(node.children.length
         + 1);
6    double correctPenal = nextPenal;
7
8    foreach(child in node.children)
9      correctPenal += penalization2(child, nextPenal);
10
11   return correctPenal;
12 }
```

4.8.5 Final Result

We have described the most complex parts of the fitness function so far. Now it is time to put it together with the remaining parts of our fitness function. The ideal fitness function is 0, and its computation is summarized in algorithm 4.10. One can notice, that point 5 and 6 ensure that our algorithm operates as a simple coevolution, since fitness of an individual depends on the fitness of the rest fo the population.

1. As a part of the world initialization interesting states of the world (including goal states) can be defined. If the individual ends int such a state, its fitness value is improved. The fitness function then starts at the worst possible option according to the defined interesting states and it is improved if necessary.

2. $fitness = fitness - \frac{\#good}{\#all}$, where fitness is acatual fitness value, #good represents number of nodes visited and left without a problem and #all means number of all visited nodes. This part of fitness function assures that every single node in final individual will be OK.
   Now if the fitness is equal to 0, the ideal individual is found. Otherwise, the penalizations has to be added.

3. The distance heuristics are counted and added to the fitness.

4. The first antibloat penalization of $\frac{1}{10}$ for each recursive node is added.

5. The actual fitness is divided by the average fitness of equivalent individuals. This assures that inequivalent individuals can live together, regardless to

their distance heuristics.

6. The ctual fitness is divided by 2 if less than one hundred equivalent individuals live in the population. This supports exploration.

7. The second antibload penalization is used. Using this after averaging ensures that individuals with more fulfilled goals will be better.

Algorithm 4.10 Fitness evaluation summary.

```
1  void evaluate(Individual individual)
2  {
3    individual.world = World.init();
4
5    individual.fixPartialOrder();
6    individual.eval();
7
8    World endWorld = individual.world;
9    fitness = 1 + numberOfGoals;
10
11   for(int i = 0; i<numberOfGoals; ++i)
12     if(endWorld.isTrue(goal[i]))
13       fitness -= 1;
14       else if(endWorld.isTrue(sub-goal[i]))
15       fitness -= 0.5;
16
17   fitness -= individual.good/individual.all;
18
19   if(fitness != 0)
20   {
21     fitness += distanceHeuristic(individual.root);
22     fitness += penalization1(individual.root);
23     fitness = fitness/averageFit(individual.type);
24
25     if(numOfIndividual(individual.type<100))
26       fitness = fitness/2;
27
28     fitness += penalization2(individual.root, 1);
29   }
30 }
```

4.9 Optional parts of the algorithm and their effects

As mentioned above in this chapter, some parts of the algorithm depend on definition in the domain description and in other external sources. Let us describe what parts we have in mind, what is their purpose and what happened if they are not defined.

- *Compulsory passed parameters* — Their purpose is to force the individual to set correctly parameters that are known from the beginning. They are usually connected to the goal. When not defined, this parameters are set randomly, and any error is not identified until the tree is finished.

- *Interesting states of world for fitness* — Interesting state means state/states of the world whose achievement should be awarded. E.g., if we want to transport a box, loading a box on a car is always a good move and should be awarded. Goal states of the world are also interesting states of the world. If non-goal interesting states of the world are not defined, run of the algorithm can be longer. If goal interesting states of the world are not defined, individual is perfect if all nodes are OK after evaluation. Especially, when no compulsory passed parameters are defined, any tree with all OK nodes is considered as ideal.

- *Distance heuristics* — This helps algorithm to overcome bigger distances by considering the distance of objects in the fitness. Without this, it is impossible or very demanding to overcome a bigger distances. However it, can be replaced by a set of interesting states of the world, but it is necessary to create them with respect to the current worlds and the current problem.

5. Software Implementation

We will briefly describe the software tool created to demonstrate our algorithm in this chapter. At first we will show the structure of the planner, then we will explain how domain independence is assured. Then, we will look ar the planning domain specification and the planning problem specification in detail. In the end, we will describe how to launch our planner, what arguments are available and where to find settings and results.

5.1 Structure of the Solver

Our planner is written in java, and it uses the ECJ library [11] for evolutionary computation. For ease of explanation, let us assume that our solver is domain-specific and it is written for one specific domain and problem. We will deal with domain independence in the following section.

Except for general parts of the algorithm that were already created in the library as initialization, genetic operators etc., it was necessary to create several classes to describe the given domain and the problem. The first of them is the *HTNProblem* class that is basically the heart of the program. In this class, the world is initialized, individuals are evaluated and fitness function is computed. Another important class is the *ControlledData*, that provides opportunity to pass data througt the individual from one node to another. It is used for passing parameters, but also for maintaining the current state of the world and some other needed information. The *World* class describes the actual state of the world. It holds three functions. The first is that it can tell what is true or not in the world, and change it. The second one provides information about number of objects of specified type in the world. The third one computes the distance between two objects in the world, if the distance heuristic exists. To be more specific, the distance heuristic is computed in individual nodes. *World* class only provides a breadth-first search algorithm for them. This is actually the only class that is not required by ECJ library, but it is useful for planning problem solver. The last part needed to create is a set of classes representing nodes for the tree of the individual.

An individual in ECJ library consists of several classes. The most important ones are *GPIndividual* and *GPNode*. *GPIndividual* class represents actual individual and provides all information about it, e.g. its fitness, its trees (it is possible to have more than one tree for the individual, but we are not using it) or whether it was already evaluated. The abstract class *GPNode* represents a node in a tree

of an individual. Its inherited classes represent specific parameters, operators and methods of the planning problem.

Now let us take a look how specific *GPNodes* works. The prameter node is very simple. It is created once for each object in the world and it remembers what number of that object it is as its ID. If passed parameter is present, it changes its ID as required, if ID is not set, it is chosen randomly. The operator node provides local search if any of its children is not defined (it has an empty ID and no passed parameter is given). It also ensures that all its parameter are evaluated, and therefore their ID is set. Then, it checks the world if it can be applied, and if so it also changes the world. The method node can compute distance heuristic, even choosing between several heuristics, or combine more of them if required. For example, we want to know how far is a box from the target location. A box can be in the pile or on the car. Then, the required distance is number of boxes above our chosen one, and number of location from pile location to the target one, or the distance is number of location from car location to the target one, depending where our box is actually placed. The method node also remembers time constrains for its children. Further more, it also provides evaluation of its children and ensures right passing parameters throught them. Unlike parameter node, operator node and method node can be set as problematic. For each operator/method, one operator/method node class had to be created. In the end, two configuration files are needed. *HTNProblem.params* file obtains definition of nodes' type and type of their children. *HTNSettings.params* file consists of information about chosen genetic operators, node selector and mating selection.

Moreover, several modifications had to be done to the ECJ library. Briefly, the most important changes were: new class securing individual equivalence had to be created and added to the *GPIndividual*, new interface for pseudo-terminals was created, initialization methods were modified to work with pseudo-terminals, new node selector was invented, and *GPIndividual* was changed to work with partially ordered trees.

5.2 Solver Generator

To provide domain-independence, the program has to be able to adapt individuals, fitness and all other parts required for the correct function of the algorithm to assigned domain. This adaptation is ensured by processing domain and problem definition and generating classes described in previous section 5.1, as well as the *HTNProblem.params* file. The second configuration file is domain-independent so there is no need to generate it. To generate classes we are using the CodeModel library [1]. Domain and problem definition will be described in the next section.

5.3 Planning problem specification

Description of the planning domain and problem specification can be found in the *PartialOrderRules.txt* file. This file starts with the name of file with evolution settings. Then, it is divided into seven parts that gradually describe everything necessary. For the record we should mention that lines starting with *"#"* symbol are comments and have no meaning, empty lines have no meaning and separator is enter or space depending on the situation. The parser is not too robust, so the exact format needs to be followed. Please bear in mind that this file does not contain *"..."*, it is used in examples to denote omitted parts.

1. The first part describes objects and properties in the world. It starts with the number of objects followed by their names and count as shown at algorithm 5.1.

Algorithm 5.1 Illustration of object and properties description.

```
# define parameters
6
Type
3
Vehicle
6
...
```

2. Then, the description of the predicates follows. It again starts with the count of predicates in the world, followed by their list. Each predicate starts with its name, followed by the number of its parameters and the list of its parameter types. For a better illustration see algorithm 5.2.

Algorithm 5.2 Illustration of predicates description.

```
# define predicates
18
IsWater
1
Location

AdjacentLoc
2
Location
Location
...
```

3. The next part describes initialization of the world. It is the only part that does not start with number of records, instead it ends with "− − −". This solution was chosen because the world description can be long and it needs to be changed often. Each row contains name of the predicate with its parameter values as you can see in algorithm 5.3.

Algorithm 5.3 Illustration of world initialization.

```
# initWorld
AdjacentLoc 0 1
AdjacentLoc 0 4

...

Boat 4
OccupiedWa 16
EmptyVehicle 4
IsType 4 2
---
```

4. The next section allows user to set starting passed parameters, these are the parameters that are known from the beginning. It starts with the number of definitions. Each definition starts with number of parameters followed by type of parameters and their value. See algorithm 5.4 for more details. Please bear in mind that starting parameters will be applied if and only if root of the tree expects here the specified starting parameters. Also, if more than one applicable set will be written, first one will be always applied.

Algorithm 5.4 Illustration of starting passed parameters.

```
# define starting parameters
1
4
Type 0
Location 30
Type 1
Location 30
```

5. Now it is time to specify interesting states of the world, e.g. the goal states. At first, the number of inequivalent goal states is written. Than the number of interesting states belonging to one goal followed by their description is

written down (see algorithm 5.5). If there are defined N interesting states belonging to the one goal, award of the fitness is following: If the first state is achieved, the award = 1. If the first one is not achieved, but the second one is, the award = $1 - \frac{1}{N}$. If just the last one is achieved, the award = $\frac{1}{N}$. Interesting state can be one state of the world, but it could also be a set of them. They are connected by *&*, if they should be true at the same time, or —, if just one of them is required to earn the award. The *&* symbol has a higher priority. Therefore if A & B — C is written, it means (A & B) — C. No brackets are allowed in the description. States of the world can be fully defined by numbers, but they can also contain variable, if that particular value does not matter. These variables are shared throught *&*, but not throught —. Thus *IsType I 0 & OnPl I 3* means: box *I* is type *0* and it also lays on place *3*. While *LayPlAtLoc A 30 — BuildCrAtLoc A 30* means: some place lays on location 30 or some crane lays on location 30. To be complete, the following expression is also enabled: *IsType I -0*. It means box *I* is not type *0*. Now you should be able to fully understand the algorithm 5.5.

Algorithm 5.5 Illustration of the fitness specification.

```
# define fitness
2
2
IsType I 0 & OnPl I A & LayPlAtLoc A 30 | IsType I 0 & OnCr I A & BuildCrAtLoc A 30
IsType I 0 & OnVehicle I A
2
IsType I 1 & OnPl I A & LayPlAtLoc A 30 | IsType I 1 & OnCr I A & BuildCrAtLoc A 30
IsType I 1 & OnVehicle I A
```

6. The next part contains description of operators. It starts with number of operators as usual, followed by the list of individual operators. Each operator starts with its unique name, then its type follows, succeeded by number of parameters and the list of them (with their names). Names of parameters can be any sequence of symbols, that can represent a variable in java. Notice that this is actually simultaneously the list of operator's children and their parameters. Then number of preconditions and list of them follows. If a new parameter appears here, it is taken as existential quantifier — there exists such parameter, that ... Then the number of effects and their list is written. The last part of the operator is the number of parameter changes and their list. For better illustration see algorithm 5.6.

   Preconditions and effects are always predicates fully defined with parameters, with one exception — inequality. Preconditions may look like this: *!= B C*. However since this operation is not a predicate, it has special limita-

tions. It always has two parameters, those parameters have to be defined earlier (no existencional quantifier, not even later definition suffice). It is always checked, but since it is very complicated to iterate over it, it is not always considered during local search (see subsection 4.6.1). To be more precise it is considered during local search if both of its parameters are not null when the inequality is checked. Thus, it is always good to write it as the last precondition. Notice that no equality symbol is necessary, since parameters are shared through the whole operator. Therefore, if two children gets the same parameter as their argument, it ensures equality. Both preconditions and effect may start with the *"-"* symbol. This simply means that this predicate cannot be true in the world before the use of operators or it cannot be true in the world after the use of operator.

Parameter change is a simple tool how to change parameters for the next use. For example, imagine that you have box on *car 0*, then you transship it on *boat 3*. Since moving methods and operators remember *vehicle*, they would want to continue with a *car*. This is the place where operator can announce this change to its parent.

Not all operators changes the world. First, there are operators who do nothing as part of the job, e.g. staying as movement. But there are also finding operators that uses local search to ensure that all parameters are set before some important part, e.g. the *FindPlace* operator.

Finally, an important note must be done. All methods/operators with the same type should have the same parameters, and operators should make sure that their parameters are defined correctly. It may sound naturally, but imagine the following example. For a correct move you need to know the *vehicle*, the start *location*, and the end *location*. But if you wan to stay on same *location*, only two parameters are necessary — *vehicle* and start/end *location*. How do you ensure that start and end *location* will be the same? Variable defined by parameters has to have different name. Our solution is written in algorithm 5.6. We simply set two preconditions, the first one ensures that the *vehicle* stays on the start *location* and the second one do the same for the end *location*.

7. Definitions of methods are contained in the last part of the file. It starts with the number of methods, followed by their list. Each method starts with its unique name, then its type is written down. After that, the number of parameters and their list is written. Each parameter has defined its name here. The next is the number of children, the list of its children

and their parameters. After that, the number and the list of conditions follows. Then, the number of time constraints and their list follows. Each time constraint is written as a pair A B, it means, that child A (and all its ancestors) has to be done before child B (and any of its ancestors). Time constraints are transitive and cannot contain a circle. The next part of description contains the number of parameter changes and their list. In the end, the distance is defined. See algorithm 5.7 for better illustration.

There are two types of conditions — inequality and postconditions. Postconditions are specified as predicates fully defined with parameters. They are the same as preconditions in operators. However, since it is not clear how will be parameters instantiated, it is impossible to check them until all children are finished. That is why they are check after that. Inequality is not checked, it is required from node's descendants. It does not mean, that these two parameters are not the same, it means, that if one of this parameters is already set, the second one is going to be set differently. If both of them are set in the same operator, this inequality is not applied — inequality of operators should be set to prevent equality.

The distance definition starts with the number of distances that will be counted. Then it is decided whether minimum or maximum of all distances is wanted. It is followed by individual distances — whether minimum or maximum is wanted if more values are possible and their formula. Then, the number of parameters that are taken into account when equivalency is checked is written, followed by their names. On the last row is true/false whether this distance is compulsory/optional.

The distance formula has two parts — definition and condition. They are separated by the *":"* symbol. The definition can contain a name of the distance, parameters, integer value, +, -, *, /, min, max in postfix notation. The name of the distance is the same as predicate from which it is derived, just with *"Dist"* string at the end, e.g. *AdjacentLocDist*. The condition consists of states of the world and their parameters separated by the *","* symbol. All words/symbols have to be separated by one space.

Finally, a few comments. Parameters are shared throught the whole method. Method that should be the root has to have *nill* type.

5.4 Folder Structure

Let us explain a folder structure that is used by our program. Our planner can be launched by the *HTN.jar* file that is placed in *HTNPlanner* folder. There are

two more directories located in this folder — *GenericHTN* and *result*.

- GenericHTN directory contains a *partialOrderRules.txt* file, a *HTNSettings.params* file and a *HTN* folder. The *HTN* folder contains generated files described in section 5.2. *HTNSettings.params* file describes all not generated evolution settings. The *partialOrderRules.txt* file describes planning domain and problem specification. This file was be described in detail in the previous section.

- *Result* directory contains files describing last run of the planner, or last runs if planner is set to complete more of them (default is 10 runs on a launch). There are two files for each run — *job.No.out.stat* and *job.No.out2.stat*, where No stands for number of the run. Names of these two files are simplified to *out.stat* and *out2.stat* respectively if only one run is launched.

  - *Out.stat* file contains description of the best individual of each subpopulation of each generation. In addition to fitness, the tree is described and ordered list of pseudo-terminals determining partial order is written here. See figure 5.1 for illustration.
  - *Out2.stat* is filled with statistical information about the run. It contains five columns — number of generation, mean fitness of generation, best fitness in generation, best fitness so far in the run and variance.

Each time the planner is launched, it performs specified number of runs. Not to override earned results, *HTN.jar* can be launched with one argument — the number of the first run. To launch the program, you must follow these steps:

- Set java environment variables to the location of installed JDK.

  - Set JAVA_HOME variable and add it to the beginning of the PATH environment variable:
    - %JAVA_HOME%/bin on Windows OS
    - $JAVA_HOME/bin on UNIX OS
  - On Mac OS use following command: export JAVA_HOME=$(/usr/libexec/java_hom

- Start cmd/terminal and change directory to the folder with HTN.jar file.

- Launch the program by *java -jar HTN.jar* command.

(FinishBox (StartAtGroundDeliver (FindOnPlace Box0 Type1 Location3) (FindCar Vehicle2 Location5) (GroundArrivingRating (RGroundMove (RGroundMove (BaseGroundMove3 Vehicle2 Location5 Location1) (BaseGroundMove Vehicle2 Location1 Location2)) (BaseGroundMove2 Vehicle2 Location2 Location3))) (IsOnTop Box0 Location3) (LoadUpGoodCr (Load Cranenull Boxnull Vehiclenull Locationnull)) (GroundStay Vehiclenull Locationnull Locationnull) (UnloadFullEmptyCr (Put Cranenull Boxnull Placenull Locationnull) (Unload Cranenull Boxnull Vehiclenull Locationnull) (Put Cranenull Boxnull Placenull Locationnull))))

Partial order:
FindOnPlace(Box0, Type1, Location3) FindCar(Vehicle2, Location5) BaseGroundMove3(Vehicle2, Location5, Location1) BaseGroundMove(Vehicle2, Location1, Location2) BaseGroundMove2(Vehicle2, Location2, Location3) IsOnTop(Box0, Location3) Load(Cranenull, Boxnull, Vehiclenull, Locationnull) GroundStay(Vehiclenull, Locationnull, Locationnull) Put(Cranenull, Boxnull, Placenull, Locationnull) Unload(Cranenull, Boxnull, Vehiclenull, Locationnull) Put(Cranenull, Boxnull, Placenull, Locationnull)

Figure 5.1: Illustration of an individual record. The upper part shows the tree, the bottom part determines the plan. Methods and operators with null were not visited during evaluation.

Algorithm 5.6 Illustration of operators' description,

```
# define operators
22
GroundStay
GroundMove
3
Vehicle A
Location B
Location C
3
StayVehicleAtLoc A B
StayVehicleAtLoc A C
Car A
0
0

BaseWaterMove
WaterMove
3
Vehicle A
Location B
Location C
5
Boat A
IsWater C
AdjacentLoc B C
-OccupiedWa C
StayVehicleAtLoc A B
4
-StayVehicleAtLoc A B
StayVehicleAtLoc A C
-OccupiedWa B
OccupiedWa C
0

FindPlace
ChooseUnloadingPlace
3
Crane A
Location B
Location C
3
BuildCrAtLoc A B
LayPlAtLoc P B
!= B C
0
0
...
```

Algorithm 5.7 Illustration of methods description.

```
# define methods
26
TranshipmentGroundWaterMove
GroundMove
3
Vehicle A
Location B
Location C
5
UnloadRobot CR K A B
ChooseBoat Q Y
ArriveOnWater Q Y B K
LoadUpRobot CR K Q B
WaterMove Q B C
0
4
0 3
1 2
2 3
3 4
1
A Q
0

GroundArrivingRating
ArriveOnGround
4
Car A
Location B
Location C
Box D
1
GroundMove A B C
0
0
0
1
min
min
AdjacentLocDist C Loc : StayVehicleAtLoc A Loc
2
D
C
false

...
```

6. Experiments

6.1 Datasets

We will explain the domain description and the specification of the problem used in our experiments. We have defined two versions of the world under one domain to illustrate different problems that can occur during evaluation. We will briefly introduce these worlds in subsection 6.1.2 and subsection 6.1.3. But first, we will take a look on the domain description (see subsection 6.1.1).

We will make just a brief introduction to the domain description and the specification of the problem. For the full description see the *partialOrderRules_finalBig.txt* file and the *partialOrderRules_finalWater.txt* file on the attached CD.

Sometinmes, we are using a domain and a problem derived from those described below in our experiments. The changes will be explained in each experiment. These changes are small and were created to show some problem or to allow to achieve different goals.

6.1.1 Domain Description

We are using our extension of *the dock-worker robots* (DWR) [3, section 1.7] domain in our experiments. Our domain consists of six objects/properties: *type*, *box*, *vehicle*, *crane*, *location* and *place*. We have two kinds of *vehicles* — a *boat* and a *car*. *Boats* can move on water *locations* and *cars* on ground ones. Only one *car* and one *boat* can be present on one *location* in a time. *Vehicles* can only moves through connected *locations*. They can be loaded with maximum of one *box* and ferry them to different *locations*. *Cranes* are attached to the *locations* and provide loading and unloading of *vehicles*. *Places* represents areas where *boxes* can be stored, to be more precise, *boxes* can be stack up in one pile here. Each *box* has a specific *type*. The goal is usually to transport one or more *boxes* of specified *types* to the target *location*s. It is allowed to successfully finish transportation when a chosen *box* ends up on a *crane* in a target *location*.

Among the basic actions is a *movement* of a car, a *movement* of a boat, *taking* a box from the place, *loading* it on a vehicle, *unloading* a box from a *vehicle* and *putting* it on a place. Other operators presented in the domain description just exist to provide local search. Example of such an action is *findCar* that simply chooses a car, or *findAnotherCar* that chooses a car different from the given one. The *FindCar* operator can be used in three ways — to get a car presented on the given location, to get a location of the given car, and to get some car on any location.

You might notice that *movement* of a car and *movement* of a boat were introduced separately instead of simple introduction of *movement* of a vehicle. This introduction was chosen intentionally. There are two ways how to describe movement. The first approach considers a movement of any vehicle, but then each move operator must be be either a ground type, and contain a car and a ground location, or it must be a water type, and contain a boat and a water location. This representation proved to be non functional. That is why we are using another representation. When movement is done, it must be decided beforehand whether it is a ground movement or a water movement. Then, all operators are either ground, containing a car and a ground location, or they are water, containing a boat and a water location depending on the type of movement.

It is assumed that some box is on the top of pile on each place every time. This is achieved by adding one imaginary box to the world. It is on the top of each place that is empty. This box in set as absent in each place. This setting ensures that this box is never loaded on any car.

It is also assumed that all cars starts empty (no box loaded) and our domain ensures that all cars ends empty.

The whole description of our domain, whose target it to transfer two boxes of different types, can be seen in the *partialOrderRules_finalBig.txt* file and the *partialOrderRules_finalWater.txt* file on the attached CD. Both appendices have the same domain, they differ by the definition of the world. For comparison, a domain, whose target it to transfer two boxes of the same type can be seen in *partialOrderRules - 2 same.txt* file on the attached CD.

6.1.2 The Big World

The first world we have created is called the *big world*, since it is designed to show the functionality of our algorithm if the world is rather crowded. There are 50 locations, 21 places, 21 cranes, 17 vehicles and 20 (21, respectively) boxes of 4 types. The illustration of this world is shown on figure 6.1. The whole initialization of the world can bee seen in appendix 6.4.5.

Type of individual boxes is not shown on figure 6.1, since their type is often changed to easily force algorithm to transport different boxes. Similar result can be achieved by changing the starting passed parameters and definition of fitness function. The main reason why we are changing type of boxes is that it allows to obtain greater variability of problems for algorithm to solve.

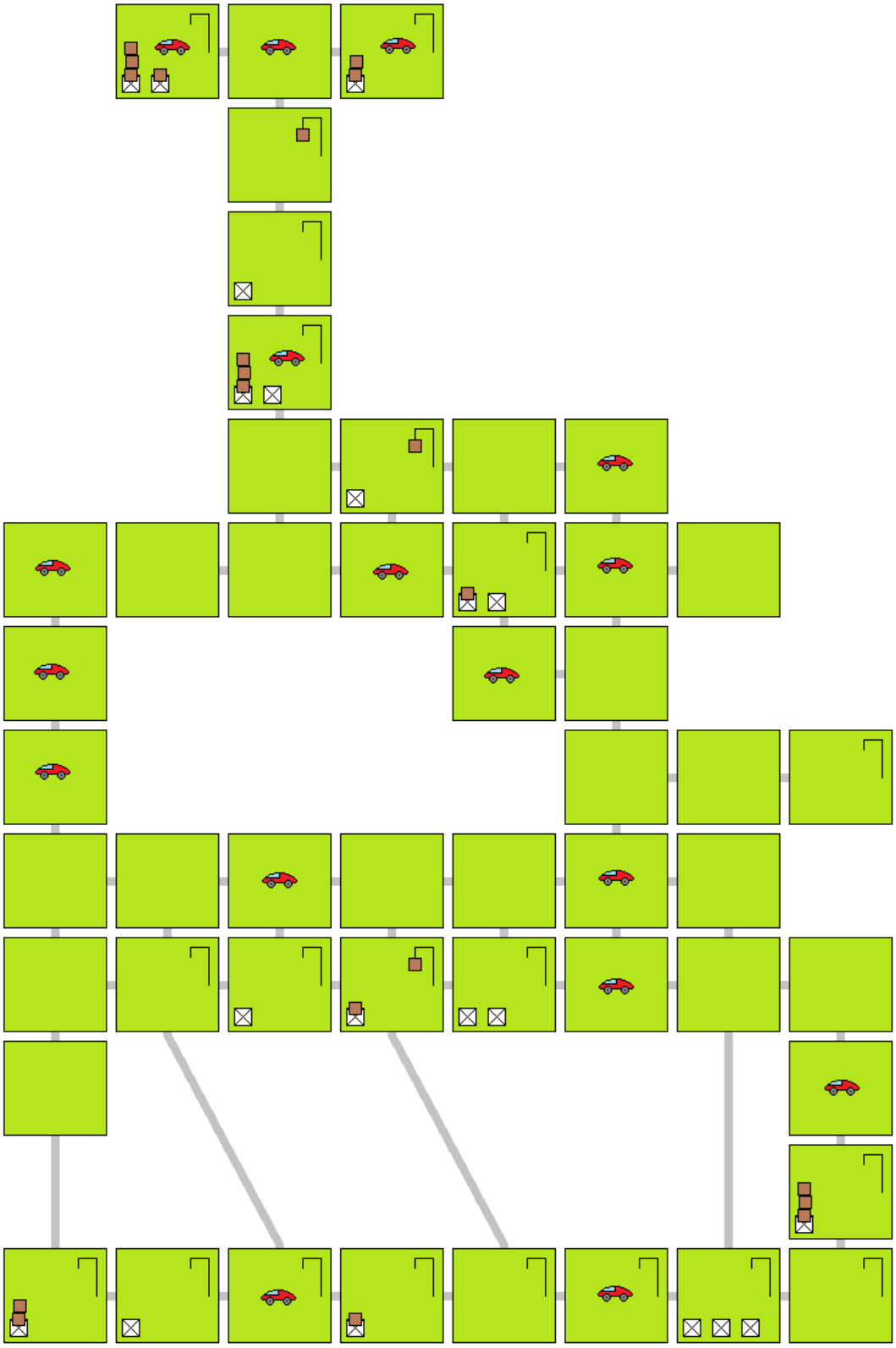

Figure 6.1: Illustration of the big world.

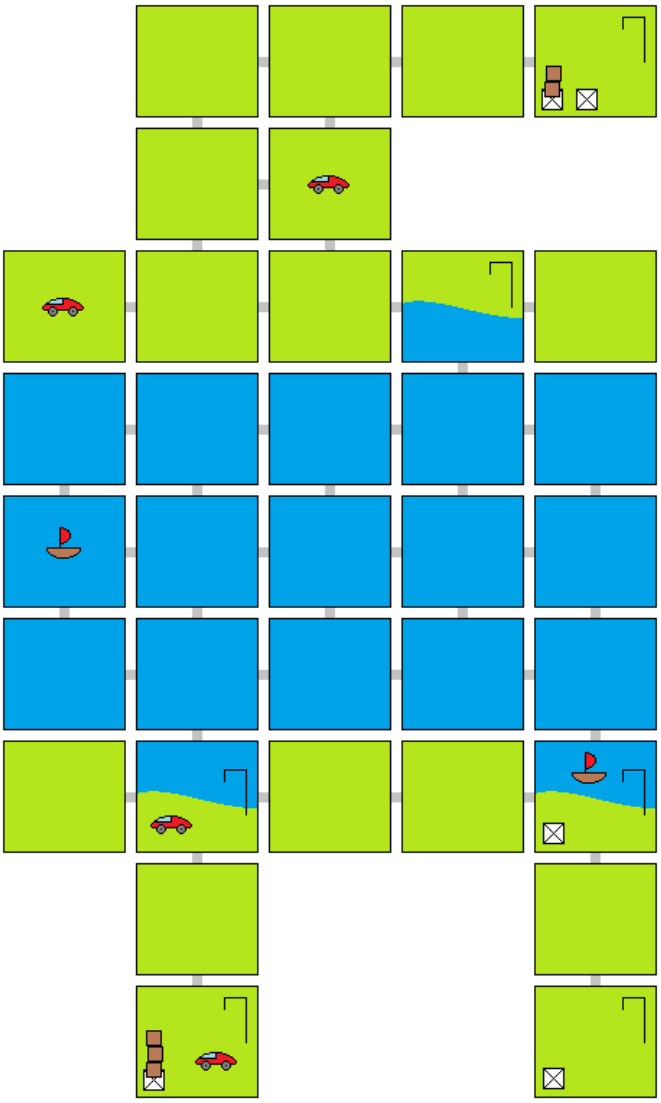

Figure 6.2: Illustration of the water world.

6.1.3 The Water World

The second world we have created is called the *water world*, since it uses water locations. It is designed to show the functionality of our algorithm when the world is sparsely populated and objects are distant from each other. There are 35 locations, 5 places, 6 cranes, 6 vehicles and 5 (6, respectively) boxes of 3 types. Illustration of this world is shown on figure 6.2. The whole initialization of the world can bee seen in appendix 6.4.5.

6.2 Quality Measure

The experiments will demonstrate that our algorithm can solve the given taks, and compare functionality of our algorithm under different conditions. Our al-

gorithm will be running 100 times under each condition with the exception of experiment 2 (see subsection 6.4.2). This experiment will be running 50 times. Then, the mean value of the fitness of the best individual of each run and their 95% *confidence interval* will be computed for each condition. According to authors of [5, section 12.4], if two 95% confidence intervals do not overlap, those conditions are statistically significantly different.

The 95% confidence interval will be counted in the following way. At first, mean fitness $\bar{f}$ will be counted as $\bar{f} = \frac{\sum_{i=1}^{N} f_i}{N}$, where N is the number of individuals ($N = 100$ in our case). Then, the *sample standard deviation* will be computed as $s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (f_i - \bar{f})^2}$. Next, the *standard error of the mean* will be counted as $SE = \frac{s}{\sqrt{N}}$. In the end, we will get 95% confidence interval from formula $Conf = \bar{f} \pm (SE * 1.96)$, cf. [12].

6.3 Evolution settings

We have executed 50–100 independent runs of each test in each experiment. They were running for 5051 or 8051 generations. The algorithm had 4 subpopulations with 1024 individuals each, except for one test in subsection 6.4.3, that had 1 subpopulation with 4096 individuals. The best individual from each subpopulation will be chosen directly to the next generation. All GA settings are summed up in table 6.1 left.

The *ramped half-and-half* method is used for initialization with probability 0.5 for both *grow* method and *full* method. One of the following operators is always chosen as genetic operator — basic crossover has a probability of 0.1, one-node mutation has also a probability of 0.1, all-nodes mutation has a probability of 0.2, and finally, basic mutation has a probability of 0.6. Order of one of pseudo-terminals is changed with a probability of 0.1. Probabilities of all operators are summed up in table 6.1 right.

As we have mentioned, the best individual from each subpopulation will be chosen directly to the next generation. However it does not mean that the best fitness of the following generation could not get worse. An individual fitness depends on fitness of the equivalent individuals, and so it can become smaller through generations.

GA setting			Operators probabilities	
generations	5051/8051		basic crossover	0.1
subpopulations	4/1		one-node mutation	0.1
subpopulation size	1024/4096		all-nodes mutation	0.2
elitism	1		basic mutation	0.6
			order change	0.1

Table 6.1: Basic evolution setting of the algorithm.

6.4 Results

We will illustrate the ability of our algorithm to solve planning problems in this section. We will show that our algorithm can solve planning problem within our domain, and we will focus on the influence of chosen settings.

Our experiments were launched single threaded on computers with unix system, intel core i7 processor, and 16 GB RAM. The time required to finish the evaluation is hard to estimate, since it depends on several factors, e.g. the size of individuals in the population. Simple estimation for the *water world* is up to approximately 1 hour and for the *big world* up to two hours, depending on the number of generations.

One optimal solution from each test was added on the attached CD for an illustration. If no such individual exists, the best individual from the last generation of some run of the test was added instead. Statistic logs from experiments are also added on the attached CD.

6.4.1 Water World environment test

The target of this experiment is to show that our algorithm can transfer two boxes across the *water world* — one of them needs to be transported over water. Moreover, we will check the influence of the partial order of deliveries to the evaluation.

The goal of the tests A and B is to transfer the box 0 (the only box of type 1) and the box 3 (the only box of type 0) to the location 30. For a better idea, box 0 is on the bottom of a pile at the topmost, rightmost location, box 3 lays in the middle of a pile at the lowest, leftmost location and location 30 is the ground/water location with a boat, as shown in figure 6.2.

The second goal of the tests C and D is to transfer the box 0 and the box 1 to the location 30. For a better idea, both boxes lay on the topmost, rightmost location. Location 30 is the ground/water location with a boat, as shown in figure 6.2.

Each test was launched 100 times. The algorithm was running with 1024 individuals in each of 4 independent subpopulations, for maximum of 5051 generations. All tests are using partial ordering, however some tests have predetermined order of deliver methods. We will refer to such tests as *full order* test (tests B and D), we will refer to the rest as *partial order* tests (tests A and C).

Let us evaluate the tests results. It is not surprising that the more ordered tests are significantly better than the less ordered ones (cf. figure 6.4), since the less ordered ones have a bigger planning space and more constraints. While 91% and 95% of runs found the correct plan in tests B and D respectively, just 60% of runs found an ideal individual in test A, and actually no ideal individual was found during test C. Comparing tests A and C shows that it is significantly harder to deal with the partial ordering when actions from one part of the tree can influence actions from the second part of the tree. For all statistical information see table 6.2.

Let us take a look an figure 6.3, it is a correct plan gained during the test A. One may notice that the plan is almost ordered — transportation of the first box is finished before transportation of the second box truly starts. This is true for most of the runs solving the less ordered problem. These results lead us to conviction, that in this particular case, a problem with ordered deliveries is a more natural representation. That is why we will use the ordered deliver methods to transfer two boxes.

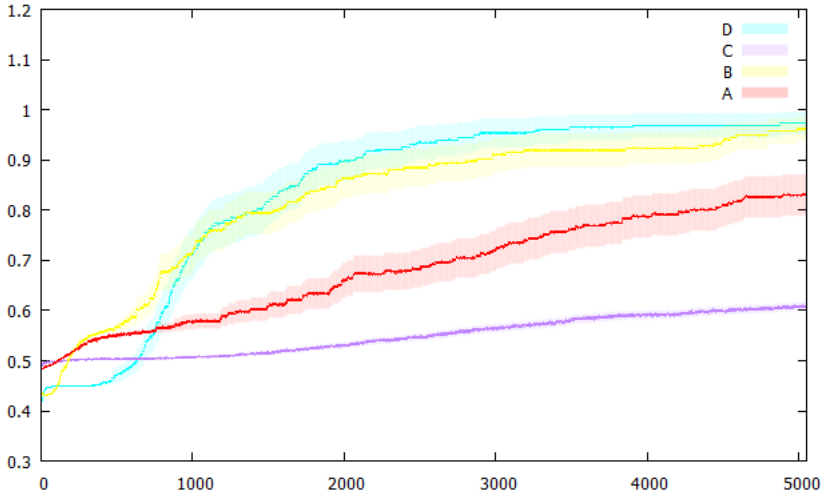

Figure 6.4: Comparison of two set of runs of the algorithm differing by the order of delivery methods.

FindOnPlace(Box3, Type0, Location32) FindCar(Vehicle0, Location32) GroundStay(Vehicle0, Location32, Location32) Take(Crane0, Box4, Place2, Location32) Put(Crane0, Box4, Place2, Location32) Take(Crane0, Box4, Place2, Location32) Put(Crane0, Box4, Place2, Location32) Take(Crane0, Box4, Place2, Location32) Put(Crane0, Box4, Place2, Location32) FindOnPlace(Box0, Type1, Location3) FindEmptyCr(Crane1, Location27, Location32) Take(Crane0, Box4, Place2, Location32) Load(Crane0, Box4, Vehicle0, Location32) BaseGroundMove(Vehicle0, Location32, Location31) BaseGroundMove2(Vehicle1, Location27, Location26) BaseGroundMove3(Vehicle0, Location31, Location27) Unload(Crane1, Box4, Vehicle0, Location27) FindCar(Vehicle2, Location5) BaseGroundMove3(Vehicle0, Location27, Location31) BaseGroundMove2(Vehicle0, Location31, Location32) IsOnTop(Box3, Location32) Take(Crane0, Box3, Place2, Location32) Load(Crane0, Box3, Vehicle0, Location32) BaseGroundMove3(Vehicle0, Location32, Location31) BaseGroundMove(Vehicle0, Location31, Location27) BaseGroundMove3(Vehicle0, Location27, Location28) BaseGroundMove2(Vehicle2, Location5, Location1) BaseGroundMove3(Vehicle0, Location28, Location29) BaseGroundMove3(Vehicle0, Location29, Location30) Unload(Crane3, Box3, Vehicle0, Location30) BaseGroundMove2(Vehicle2, Location1, Location2) BaseGroundMove(Vehicle2, Location2, Location3) Put(Crane3, Box3, Place4, Location30) GroundStay(Vehicle2, Location3, Location3) Take(Crane4, Box1, Place0, Location3) Put(Crane4, Box1, Place1, Location3) IsOnTop(Box0, Location3) Take(Crane4, Box0, Place0, Location3) Load(Crane4, Box0, Vehicle2, Location3) BaseGroundMove3(Vehicle2, Location3, Location2) BaseGroundMove3(Vehicle2, Location2, Location1) BaseGroundMove3(Vehicle2, Location1, Location5) BaseGroundMove(Vehicle2, Location5, Location8) BaseGroundMove3(Vehicle2, Location8, Location9) Unload(Crane2, Box0, Vehicle2, Location9) FindBoat(Vehicle5, Location30) BaseWaterMove(Vehicle5, Location30, Location25) BaseWaterMove(Vehicle5, Location25, Location24) BaseWaterMove2(Vehicle5, Location24, Location19) BaseWaterMove2(Vehicle5, Location19, Location14) BaseWaterMove3(Vehicle5, Location14, Location9) WaterStay(Vehicle5, Location9, Location9) Load(Crane2, Box0, Vehicle5, Location9) BaseWaterMove2(Vehicle5, Location9, Location14) BaseWaterMove3(Vehicle5, Location14, Location19) WaterStay(Vehicle5, Location19, Location19) BaseWaterMove(Vehicle5, Location19, Location24) BaseWaterMove2(Vehicle5, Location24, Location25) BaseWaterMove(Vehicle5, Location25, Location30) WaterStay(Vehicle5, Location30, Location30) Unload(Crane3, Box0, Vehicle5, Location30)

Figure 6.3: Illustration of the plan gained from the ideal individual, solving the first version of two boxes transportation problem in *water world*.

	min	Q1	median	Q3	max	mean	succ. rate
A	760	2402,5	3965	5050	5050	3658,76	60%
B	487	839,25	1314,5	2658,5	5050	1996,14	91%
C	5050	5050	5050	5050	5050	5050	0%
D	474	851,5	1080,5	1747,75	5050	1543,14	95%

Table 6.2: Statistical information about tests in the *water world* (number of generations is numbered from 0).

6.4.2 Big World environment test

The target of this experiment is to show that our algorithm can transfer two boxes across the *big world* — one of them is far away from the target location, the second one is buried. Moreover, we will compare the result with two independent runs, each transporting one of the boxes.

The goal of the algorithm is to transfer the box 0 (the only box of type 0) and the box 16 (the only box of type 1) to the location 44. For a better idea, box 0 is on the top of a pile at the topmost, leftmost location, box 1 lays in the bottom of a pile at the lowest, leftmost location and location 44 is the only location with three places at the figure 6.1.

Each test was launched 50 times. The algorithm was running with 1024 individuals in each of 4 independent subpopulations, for maximum of 8051 generations.

The algorithm found ideal individual in 50% cases. For illustration of the gained plan see figure 6.5. Minimum number of generation in which ideal individual was found was 3063. All statistical information are summed up in table 6.3. When both boxes are solved separately, the result is perceptibly improved. For example, if we consider only the worst run with the buried box, and add it to all individuals solving the distant box problem, 98% of them will transfer both boxes in time.

This implies that it could be useful to divide the given problem into several small ones and solve them using cooperatove coevolution.

	min	Q1	median	Q3	max	mean	succ. rate
two boxes	3063	6153,5	8007,5	8050	8050	6923,82	50%
distant box	879	2075,75	2686,5	3888,5	8050	3511.05	98%
buried box	65	80,25	91	105,75	137	137	100%
combination	1016	2212.75	2823.5	4025.5	8050	3648.05	98%

Table 6.3: Statistical information about tests in the *big world*.

FindCar(Vehicle9, Location40) FindOnPlace(Box16, Type1, Location38) BaseGroundMove2(Vehicle9, Location40, Location39) BaseGroundMove(Vehicle9, Location39, Location38) Take(Crane10, Box15, Place13, Location38) Put(Crane10, Box15, Place13, Location38) Take(Crane10, Box15, Place13, Location38) Put(Crane10, Box15, Place13, Location38) FindEmptyCr(Crane11, Location39, Location38) Take(Crane10, Box15, Place13, Location38) Load(Crane10, Box15, Vehicle9, Location38) BaseGroundMove(Vehicle9, Location38, Location39) Unload(Crane11, Box15, Vehicle9, Location39) Put(Crane11, Box15, Place14, Location39) BaseGroundMove2(Vehicle9, Location39, Location38) IsOnTop(Box16, Location38) Take(Crane10, Box16, Place13, Location38) Load(Crane10, Box16, Vehicle9, Location38) BaseGroundMove3(Vehicle9, Location38, Location39) BaseGroundMove(Vehicle9, Location39, Location40) BaseGroundMove2(Vehicle9, Location40, Location41) BaseGroundMove3(Vehicle9, Location41, Location42) Unload(Crane14, Box16, Vehicle9, Location42) FindNewCar(Vehicle10, Vehicle9, Location43) BaseGroundMove2(Vehicle9, Location42, Location31) BaseGroundMove3(Vehicle10, Location43, Location42) Load(Crane14, Box16, Vehicle10, Location42) BaseGroundMove(Vehicle10, Location42, Location43) BaseGroundMove3(Vehicle10, Location43, Location44) Unload(Crane16, Box16, Vehicle10, Location44) Put(Crane16, Box16, Place18, Location44) FindOnPlace(Box0, Type0, Location0) FindCar(Vehicle0, Location0) GroundStay(Vehicle0, Location0, Location0) IsOnTop(Box0, Location0) Take(Crane0, Box0, Place0, Location0) Load(Crane0, Box0, Vehicle0, Location0) BaseGroundMove(Vehicle1, Location1, Location3) BaseGroundMove3(Vehicle0, Location0, Location1) BaseGroundMove3(Vehicle1, Location3, Location4) BaseGroundMove2(Vehicle0, Location1, Location3) BaseGroundMove(Vehicle3, Location5, Location6) BaseGroundMove3(Vehicle1, Location4, Location5) BaseGroundMove3(Vehicle0, Location3, Location4) Unload(Crane2, Box0, Vehicle0, Location4) Put(Crane2, Box0, Place4, Location4) FindNewCar(Vehicle1, Vehicle0, Location5) BaseGroundMove2(Vehicle0, Location4, Location3) BaseGroundMove3(Vehicle1, Location5, Location4) Take(Crane2, Box0, Place4, Location4) Load(Crane2, Box0, Vehicle1, Location4) BaseGroundMove3(Vehicle1, Location4, Location5) Unload(Crane3, Box0, Vehicle1, Location5) FindNewCar(Vehicle3, Vehicle1, Location6) BaseGroundMove(Vehicle1, Location5, Location4) BaseGroundMove(Vehicle3, Location6, Location5) Load(Crane3, Box0, Vehicle3, Location5) BaseGroundMove2(Vehicle3, Location5, Location6) BaseGroundMove3(Vehicle3, Location6, Location7) BaseGroundMove2(Vehicle3, Location7, Location8) BaseGroundMove(Vehicle16, Location14, Location17) BaseGroundMove2(Vehicle3, Location8, Location13) BaseGroundMove(Vehicle3, Location13, Location14) BaseGroundMove2(Vehicle16, Location17, Location18) BaseGroundMove(Vehicle3, Location14, Location17) GroundStay(Vehicle3, Location17, Location17) BaseGroundMove2(Vehicle16, Location18, Location19) BaseGroundMove(Vehicle3, Location17, Location18) GroundStay(Vehicle12, Location26, Location26) BaseGroundMove(Vehicle12, Location26, Location27) BaseGroundMove2(Vehicle3, Location18, Location26) BaseGroundMove3(Vehicle15, Location33, Location34) BaseGroundMove(Vehicle3, Location26, Location33) BaseGroundMove2(Vehicle15, Location34, Location35) BaseGroundMove(Vehicle3, Location33, Location34) BaseGroundMove3(Vehicle10, Location44, Location45) BaseGroundMove3(Vehicle3, Location34, Location44) Unload(Crane16, Box0, Vehicle3, Location44)

Figure 6.5: Illustration of a plan gained from the ideal individual.

6.4.3 Influence of separate subpopulations

The target of this experiment is to show, whether the division of the algorithm into 4 smaller subpopulation influence the efficiency.

The goal of the algorithm is to transfer the box 0 to the location 32. For a better idea, box 0 is on the bottom of a pile at the topmost, rightmost location, and location 33 is the bottom leftomost location 6.2.

Each test was launched 100 times. The algorithm was running for maximum of 5051 generations, with 1024 individuals in each of 4 independent subpopulations, or it was running with 4096 individuals in 1 subpopulation.

As you can see in table 6.4, 80% of runs with 4 subpopulations found the solution, but only 57% of runs with one subpopulation found it. Approximately from the generation of 2500 on, 4 subpopulations are significantly better than one (cf. figure 6.6). It follows that implementation of 4 independent subpopulations improved our algorithm.

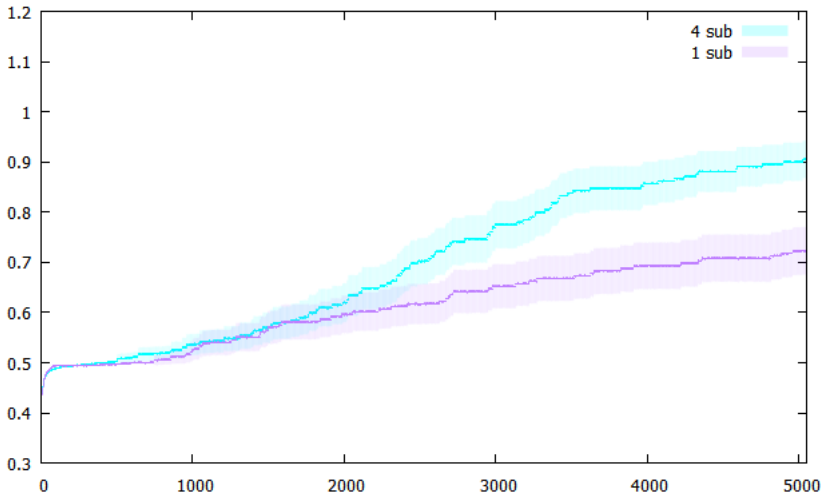

Figure 6.6: Comparison of the run of the algorithm with 1 subpopulation of 4096 individuals and 4 subpopulations with 1024 individuals each.

	min	Q1	median	Q3	max	mean	succ. rate
1 sub	749	2699,5	5050	5050	5050	3959,64	57%
4 sub	506	2113,25	2972,5	4398,25	5050	3137,21	80%

Table 6.4: Statistical information comparing 1 subpopulation of 4096 individuals to 4 subpopulations with 1024 individuals each.

6.4.4 Influence of a problem assignment

The target of this experiment is to show how important the correct domain assignment is, and how it influences the efficiency.

The goal of the algorithm is to transfer the box 0 to the location 32. For a better idea, box 0 is on the bottom of a pile at the topmost, rightmost location, and location 33 is the bottom leftomost location 6.2.

Each test was launched 100 times. The algorithm was running with 1024 individuals in each of 4 independent subpopulations, for maximum of 5051 generations. Tests differ by the number of movement operators. The first test runs with one stay operator and one move operaor of each type — ground and water. The second has one more move operator of each type. The third one has even one more move operator of each type, and finally the forth test has four move operators of each type. One may notice, that the third test is actually the same test as the test with four subpopulations in 6.4.3.

The test with one move operator is significantly worse that the others. The test with two moves is arguably worse than the other two (cf. figure 6.7). The test with three moves finds the optimal individual in 80% of cases, the test with four moves finds it in 85% case. The superiority between these two tests cannot be decided according to our quality measure.

There are two basic results of this measurement. The first one is, that we have three moves operators in our domain description. The second one is a warning — the efficiency of the algorithm is strongly dependent on the encoding of the domain.

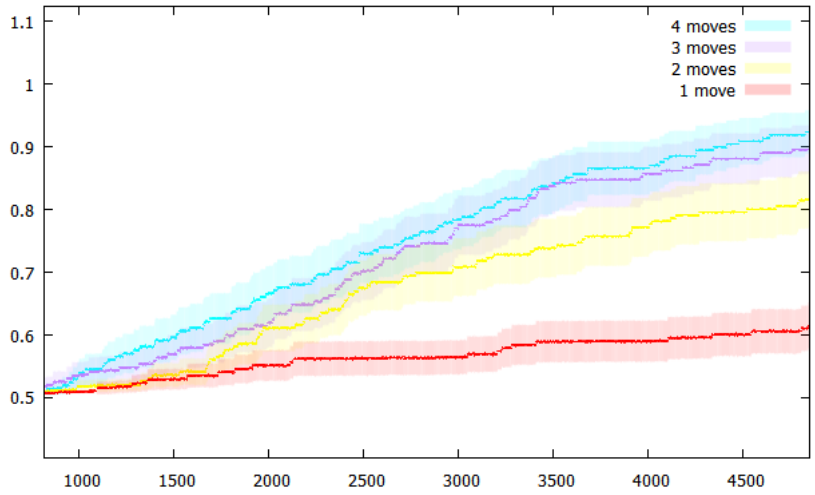

Figure 6.7: Comparison of four tests that differ in the number of move operators.

	min	Q1	median	Q3	max	mean	succ. rate
1 move	1093	5050	5050	5050	5050	4628,67	19%
2 moves	988	2365,5	3907	5050	5050	3629,69	63%
3 moves	506	2113,25	2972,5	4398,25	5050	3137,21	80%
4 moves	829	1820,75	2777,5	4091,75	5050	2941,83	85%

Table 6.5: Statistical information comparing our tests that differ in the number of move operators.

6.4.5 Example of a bad problem

The target of this experiment is to show that some problems that cannot be solved within our domain with our algorithm exist.

The test was launched 100 times. The algorithm was running with 1024 individuals in each of 4 independent subpopulations, for maximum of 5051 generations. Place 1 was moved to location 34.

The goal of the algorithm is to transfer the box 0 to the location 32. For a better idea, box 0 is on the bottom of a pile at the topmost, rightmost location, and location 33 is the bottom leftomost location 6.2.

Not only that this test never finishes its task (see table 6.6), but it has no signs of improvement through generations (cf. figure 6.8). This test allowed us to gain insight into weak points of our algorithm that are summarized in the next chapter.

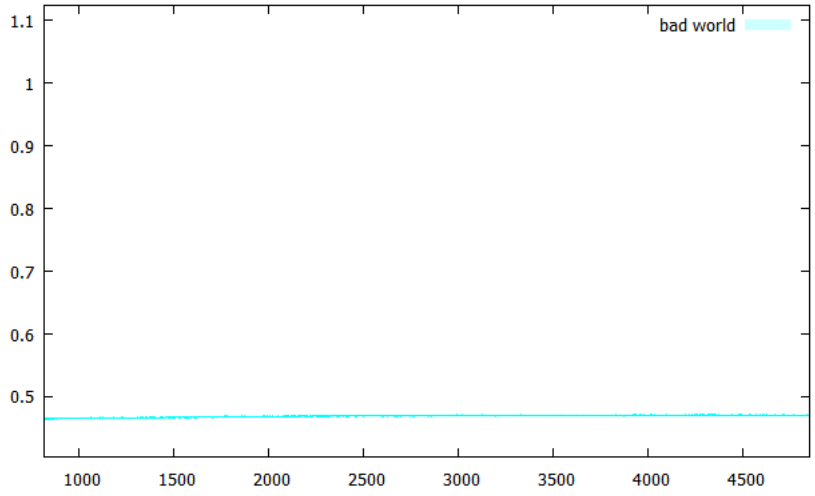

Figure 6.8: Illustration of the progress of tests trying to solve the bad world.

min	Q1	median	Q3	max	mean	succ. rate
5050	5050	5050	5050	5050	5050	0%

Table 6.6: Statistical information about the progress of the algorithm over the bad problem.

7. Conclusions

In this thesis we have proposed and designed an algorithm solving the domain-independent partial order simple task network planning problem. The algorithm gets problem description and domain specification as an input. Its output is a sequence of actions solving a given problem, when they are applied on the initial state of the world. The algorithm is using the tree-based genetic programming technique. A simple task network is encoded as a tree, and the plan is given by pseudo-terminals of the tree in a given order. Nodes of the tree are typed. Local search and utilization of parameter passing are used to avoid uninteresting parts of the search space. A distance heuristic was defined to help the algorithm to overpass bigger distances. Antibloat penalization is added to the fitness function to maintain the size of individuals. Fitness of an individual depends on the fitness and number of equivalent individuals in the population. The implementation is written in java.

Our algorithm was verified with practical experiments. The algorithm is able to solve nontrivial medium size problems. The partial order is working, but in our case, it is better to use ordered methods. We have proved, that division of population into 4 subpopulations was useful. We also showed that the efficiecncy of the algorithm strongly depends on the encoding of the domain. Experiments also indicate that a division of the problem into smaller parts, that are solved separetly, could be a great efficiency improvement in the future. Unfortunatelly, experimetns also proved, that there are problems, that cannot be solved by our algorithm.

Our algorithm has problems, when it is not guided by fitness for longer time. That is the reason why it cannot deal with situation as followed:

1. A box is on the crane, transshipement should be done here and

no place is available to store the box;

2. Several locations are connected to the line. Only one location in the middle has the crane, but not the others. The first car on the first location has a loaded box and should transport it to the last location. The second car occupies the last location.

The first situation can be solved by summoning another car and transporting the box elsewhere. The second situation is solved when the box is transshipped to the second car at the only location with the crane. The problem is that in both situations, this must be done ahead. For example, the box must be deported before the transshipement starts. However, if the world looks like at figure 7.1, a deportation of the box must be done before the red car moves, and yet, the yellow car must travel back to get out of the way. But the fitness supports movement to the target, not the waiting. Waiting can be set as an actual problem, but most of the individuals would not do it, or they would do it too late. To sum it up, our algorithm has problems with situations that have only one correct solution which is not directly supported by the fitness. One way how to improve this is a change of the fitness. Another way is dividing the problem into separate parts and coevolve them together, since it is clear from the experiments that evolving a small solution apart has better results than evolving them at once.

There are many ways how to improve our algorithm, let us mention the most interesting ones. A smart antibloat operator could be added. It should be able to reduce the size of the tree by deleting unnecessary subtrees. For example, *movement* from A to B followed by *movement* back from B to A can be deleted. Of course, improvements of the fitness, division of the problem into separate parts and their coevolution, or any other enhancement of the algorithm that would improve behavior during the above-mentioned situations could be implemented. Other thing to do is extending

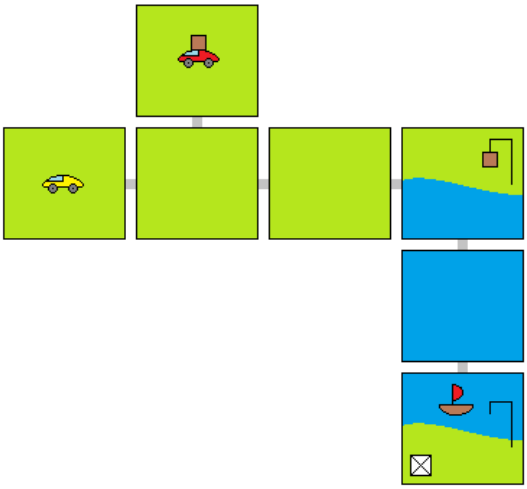

Figure 7.1: Illustration of the problematic setting of the world.

the algorithm to the full hierarchical task network. The parser of the problem description in our software implementation can be also improved to be more robust.

Bibliography

[1] Codemodel java library. available online at https://codemodel.java.net/. 5.2

[2] E. Besada-Portas, L. de la Torre, J. M. de la Cruz, and B. de Andrés-Toro. Evolutionary trajectory planner for multiple UAVs in realistic scenarios. *IEEE Transactions on Robotics*, 26(4):619–634, Aug 2010. 1

[3] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, 2004. 3.1.3, 3.2.2, 3.2.3, 6.1.1

[4] Terry Jones. Crossover, macromutation, and population-based search. Technical report, Santa Fe Institute, 1995. 2.1.2

[5] Steven Julious, Say-Beng Tan, and David Machin. *An Introduction to Statistics in Early Phase Trials*. Wiley, 2010. 6.2

[6] Ion Muslea. SINERGY: A linear planner based on genetic programming. In *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24-26, 1997, Proceedings*, pages 312–324, 1997. 1

[7] Dana Nau, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003. 1

[8] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza). 2.2, 2.2.1

[9] Lea Ruscio, John Levine, and John K. Kingston. Applying genetic algorithms to hierarchical task network planning. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000).*, 2000. 1

[10] J.E. Smith and Agoston E. Eiben. *Introduction to Evolutionary Computing (Natural Computing Series).* Springer International Publishing, 2008. 2, 2.1.1

[11] David R. White. Software review: the ECJ toolkit. *Genetic Programming and Evolvable Machines*, 13(1):65–67, 2012. 5.1

[12] Wikipedia. Standard error — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-July-2016]. 6.2

List of Figures

List of Tables

List of Algorithms

Appendix A: The Big World Initialization

HTNSettings.params

define parameters

6
Type
4
Vehicle
17
Box
21
Crane
21
Location
50
Place
21

initWorld

AdjacentLoc 0 1
AdjacentLoc 1 0
AdjacentLoc 1 2
AdjacentLoc 1 3
AdjacentLoc 2 1
AdjacentLoc 3 1
AdjacentLoc 3 4
AdjacentLoc 4 3
AdjacentLoc 4 5
AdjacentLoc 5 4
AdjacentLoc 5 6
AdjacentLoc 6 5
AdjacentLoc 6 7
AdjacentLoc 6 11
AdjacentLoc 7 6
AdjacentLoc 7 8
AdjacentLoc 7 12
AdjacentLoc 8 7
AdjacentLoc 8 9
AdjacentLoc 8 13
AdjacentLoc 9 8
AdjacentLoc 9 14
AdjacentLoc 10 11
AdjacentLoc 11 6
AdjacentLoc 11 10
AdjacentLoc 11 12
AdjacentLoc 12 7
AdjacentLoc 12 11
AdjacentLoc 12 13
AdjacentLoc 13 8
AdjacentLoc 13 12
AdjacentLoc 13 14

AdjacentLoc 13 16
AdjacentLoc 14 9
AdjacentLoc 14 13
AdjacentLoc 14 15
AdjacentLoc 14 17
AdjacentLoc 15 14
AdjacentLoc 16 13
AdjacentLoc 16 17
AdjacentLoc 17 14
AdjacentLoc 17 16
AdjacentLoc 17 18
AdjacentLoc 18 17
AdjacentLoc 18 19
AdjacentLoc 18 26
AdjacentLoc 19 18
AdjacentLoc 19 20
AdjacentLoc 20 19
AdjacentLoc 21 22
AdjacentLoc 21 28
AdjacentLoc 21 47
AdjacentLoc 22 21
AdjacentLoc 22 23
AdjacentLoc 22 29
AdjacentLoc 23 22
AdjacentLoc 23 24
AdjacentLoc 23 30
AdjacentLoc 24 23
AdjacentLoc 24 25
AdjacentLoc 24 31
AdjacentLoc 25 24
AdjacentLoc 25 26
AdjacentLoc 25 32
AdjacentLoc 26 18
AdjacentLoc 26 25
AdjacentLoc 26 27
AdjacentLoc 26 33
AdjacentLoc 27 26
AdjacentLoc 27 34
AdjacentLoc 28 21
AdjacentLoc 28 29
AdjacentLoc 28 36
AdjacentLoc 29 22
AdjacentLoc 29 28
AdjacentLoc 29 30
AdjacentLoc 29 40
AdjacentLoc 30 23
AdjacentLoc 30 29
AdjacentLoc 30 31
AdjacentLoc 31 24
AdjacentLoc 31 30
AdjacentLoc 31 32
AdjacentLoc 31 42
AdjacentLoc 32 25
AdjacentLoc 32 31
AdjacentLoc 32 33
AdjacentLoc 33 26
AdjacentLoc 33 32
AdjacentLoc 33 34
AdjacentLoc 34 27
AdjacentLoc 34 33
AdjacentLoc 34 35
AdjacentLoc 34 44

AdjacentLoc 35 34
AdjacentLoc 35 37
AdjacentLoc 36 28
AdjacentLoc 36 38
AdjacentLoc 37 35
AdjacentLoc 37 46
AdjacentLoc 38 36
AdjacentLoc 38 39
AdjacentLoc 39 38
AdjacentLoc 39 40
AdjacentLoc 40 29
AdjacentLoc 40 39
AdjacentLoc 40 41
AdjacentLoc 41 40
AdjacentLoc 41 42
AdjacentLoc 42 31
AdjacentLoc 42 41
AdjacentLoc 42 43
AdjacentLoc 43 42
AdjacentLoc 43 44
AdjacentLoc 44 34
AdjacentLoc 44 43
AdjacentLoc 44 45
AdjacentLoc 45 44
AdjacentLoc 45 46
AdjacentLoc 46 37
AdjacentLoc 46 45
AdjacentLoc 47 21
AdjacentLoc 47 48
AdjacentLoc 48 47
AdjacentLoc 48 49
AdjacentLoc 49 48
IsGround 0
IsGround 1
IsGround 2
IsGround 3
IsGround 4
IsGround 5
IsGround 6
IsGround 7
IsGround 8
IsGround 9
IsGround 10
IsGround 11
IsGround 12
IsGround 13
IsGround 14
IsGround 15
IsGround 16
IsGround 17
IsGround 18
IsGround 19
IsGround 20
IsGround 21
IsGround 22
IsGround 23
IsGround 24
IsGround 25
IsGround 26
IsGround 27
IsGround 28
IsGround 29

IsGround 30
IsGround 31
IsGround 32
IsGround 33
IsGround 34
IsGround 35
IsGround 36
IsGround 37
IsGround 38
IsGround 39
IsGround 40
IsGround 41
IsGround 42
IsGround 43
IsGround 44
IsGround 45
IsGround 46
IsGround 47
IsGround 48
IsGround 49
Car 0
Car 1
Car 2
Car 3
Car 4
Car 5
Car 6
Car 7
Car 8
Car 9
Car 10
Car 11
Car 12
Car 13
Car 14
Car 15
Car 16
StayVehicleAtLoc 0 0
OccupiedGr 0
EmptyVehicle 0
StayVehicleAtLoc 1 1
OccupiedGr 1
EmptyVehicle 1
StayVehicleAtLoc 2 2
OccupiedGr 2
EmptyVehicle 2
StayVehicleAtLoc 3 5
OccupiedGr 5
EmptyVehicle 3
StayVehicleAtLoc 14 9
OccupiedGr 9
EmptyVehicle 14
StayVehicleAtLoc 4 12
OccupiedGr 12
EmptyVehicle 4
StayVehicleAtLoc 16 14
OccupiedGr 14
EmptyVehicle 16
StayVehicleAtLoc 5 16
OccupiedGr 16
EmptyVehicle 5
StayVehicleAtLoc 11 23

OccupiedGr 23
EmptyVehicle 11
StayVehicleAtLoc 12 26
OccupiedGr 26
EmptyVehicle 12
StayVehicleAtLoc 15 33
OccupiedGr 33
EmptyVehicle 15
StayVehicleAtLoc 13 37
OccupiedGr 37
EmptyVehicle 13
StayVehicleAtLoc 9 40
OccupiedGr 40
EmptyVehicle 9
StayVehicleAtLoc 10 43
OccupiedGr 43
EmptyVehicle 10
StayVehicleAtLoc 8 47
OccupiedGr 47
EmptyVehicle 8
StayVehicleAtLoc 7 48
OccupiedGr 48
EmptyVehicle 7
StayVehicleAtLoc 6 49
OccupiedGr 49
EmptyVehicle 6
BuildCrAtLoc 0 0
BuildCrAtLoc 1 3
BuildCrAtLoc 2 4
BuildCrAtLoc 3 5
BuildCrAtLoc 4 7
BuildCrAtLoc 5 13
BuildCrAtLoc 6 29
BuildCrAtLoc 7 30
BuildCrAtLoc 8 31
BuildCrAtLoc 9 32
BuildCrAtLoc 10 38
BuildCrAtLoc 11 39
BuildCrAtLoc 12 40
BuildCrAtLoc 13 41
BuildCrAtLoc 14 42
BuildCrAtLoc 15 43
BuildCrAtLoc 16 44
BuildCrAtLoc 17 45
BuildCrAtLoc 18 46
BuildCrAtLoc 19 20
BuildCrAtLoc 20 2
LayPlAtLoc 0 0
LayPlAtLoc 1 0
LayPlAtLoc 2 2
LayPlAtLoc 3 7
LayPlAtLoc 4 4
LayPlAtLoc 5 5
LayPlAtLoc 6 5
LayPlAtLoc 7 13
LayPlAtLoc 8 13
LayPlAtLoc 9 30
LayPlAtLoc 10 31
LayPlAtLoc 11 32
LayPlAtLoc 12 32
LayPlAtLoc 13 38
LayPlAtLoc 14 39

LayPlAtLoc 15 41
LayPlAtLoc 16 44
LayPlAtLoc 17 44
LayPlAtLoc 18 44
LayPlAtLoc 19 46
LayPlAtLoc 20 46
EmptyCr 0
OnCr 6 1
EmptyCr 2
EmptyCr 3
OnCr 7 4
EmptyCr 5
EmptyCr 6
EmptyCr 7
OnCr 9 8
EmptyCr 9
EmptyCr 10
EmptyCr 11
EmptyCr 12
EmptyCr 13
EmptyCr 14
EmptyCr 15
EmptyCr 16
EmptyCr 17
EmptyCr 18
EmptyCr 19
EmptyCr 20
OnTop 0 0
On 0 1
OnPl 0 0
On 1 2
OnPl 1 0
On 2 20
OnPl 2 0
OnTop 3 1
On 3 20
OnPl 3 1
OnTop 4 2
On 4 5
OnPl 4 2
On 5 20
OnPl 5 2
OnTop 20 3
OnTop 20 4
OnTop 17 5
On 17 18
OnPl 17 5
On 18 19
OnPl 18 5
On 19 20
OnPl 19 5
OnTop 20 6
OnTop 8 7
On 8 20
OnPl 8 7
OnTop 20 8
OnTop 20 9
OnTop 10 10
On 10 20
OnPl 10 10
OnTop 20 11
OnTop 20 12

OnTop 15 13
On 15 16
OnPl 15 13
On 16 20
OnPl 16 13
OnTop 20 14
OnTop 14 15
On 14 20
OnPl 14 15
OnTop 20 16
OnTop 20 17
OnTop 20 18
OnTop 11 19
On 11 12
OnPl 11 19
On 12 13
OnPl 12 19
On 13 20
OnPl 13 19
OnTop 20 20

IsType 0 0
IsType 1 2
IsType 2 2
IsType 3 2
IsType 4 2
IsType 5 2
IsType 6 2
IsType 7 2
IsType 8 2
IsType 9 2
IsType 10 2
IsType 11 2
IsType 12 2
IsType 13 2
IsType 14 2
IsType 15 2
IsType 16 1
IsType 17 2
IsType 18 2
IsType 19 2

Appendix B: The Water World Initialization

HTNSettings.params

define parameters

6
Type
3
Vehicle
6
Box
6
Crane
6
Location
35
Place
5

initWorld

AdjacentLoc 0 1
AdjacentLoc 1 0
AdjacentLoc 1 2
AdjacentLoc 2 1
AdjacentLoc 2 3
AdjacentLoc 3 2
AdjacentLoc 0 4
AdjacentLoc 4 0
AdjacentLoc 1 5
AdjacentLoc 5 1
AdjacentLoc 4 5
AdjacentLoc 5 4
AdjacentLoc 4 7
AdjacentLoc 7 4
AdjacentLoc 5 8
AdjacentLoc 8 5
AdjacentLoc 6 7
AdjacentLoc 7 6
AdjacentLoc 7 8
AdjacentLoc 8 7
AdjacentLoc 8 9
AdjacentLoc 9 8
AdjacentLoc 9 10
AdjacentLoc 10 9
AdjacentLoc 9 14
AdjacentLoc 14 9
AdjacentLoc 11 12
AdjacentLoc 12 11
AdjacentLoc 12 13
AdjacentLoc 13 12
AdjacentLoc 13 14
AdjacentLoc 14 13

AdjacentLoc 14 15
AdjacentLoc 15 14
AdjacentLoc 11 16
AdjacentLoc 16 11
AdjacentLoc 12 17
AdjacentLoc 17 12
AdjacentLoc 13 18
AdjacentLoc 18 13
AdjacentLoc 14 19
AdjacentLoc 19 14
AdjacentLoc 15 20
AdjacentLoc 20 15
AdjacentLoc 16 17
AdjacentLoc 17 16
AdjacentLoc 17 18
AdjacentLoc 18 17
AdjacentLoc 18 19
AdjacentLoc 19 18
AdjacentLoc 19 20
AdjacentLoc 20 19
AdjacentLoc 16 21
AdjacentLoc 21 16
AdjacentLoc 17 22
AdjacentLoc 22 17
AdjacentLoc 18 23
AdjacentLoc 23 18
AdjacentLoc 19 24
AdjacentLoc 24 19
AdjacentLoc 15 20
AdjacentLoc 20 15
AdjacentLoc 21 22
AdjacentLoc 22 21
AdjacentLoc 22 23
AdjacentLoc 23 22
AdjacentLoc 23 24
AdjacentLoc 24 23
AdjacentLoc 24 25
AdjacentLoc 25 24
AdjacentLoc 22 27
AdjacentLoc 27 22
AdjacentLoc 25 30
AdjacentLoc 30 25
AdjacentLoc 26 27
AdjacentLoc 27 26
AdjacentLoc 27 28
AdjacentLoc 28 27
AdjacentLoc 28 29
AdjacentLoc 29 28
AdjacentLoc 29 30
AdjacentLoc 30 29
AdjacentLoc 27 31
AdjacentLoc 31 27
AdjacentLoc 31 32
AdjacentLoc 32 31
AdjacentLoc 30 33
AdjacentLoc 33 30
AdjacentLoc 33 34
AdjacentLoc 34 33
IsWater 9
IsWater 11
IsWater 12
IsWater 13

IsWater 14
IsWater 15
IsWater 16
IsWater 17
IsWater 18
IsWater 19
IsWater 20
IsWater 21
IsWater 22
IsWater 23
IsWater 24
IsWater 25
IsWater 27
IsWater 30
IsGround 0
IsGround 1
IsGround 2
IsGround 3
IsGround 4
IsGround 5
IsGround 6
IsGround 7
IsGround 8
IsGround 9
IsGround 10
IsGround 26
IsGround 27
IsGround 28
IsGround 29
IsGround 30
IsGround 31
IsGround 32
IsGround 33
IsGround 34
StayVehicleAtLoc 0 32
Car 0
OccupiedGr 32
EmptyVehicle 0
StayVehicleAtLoc 1 27
Car 1
OccupiedGr 27
EmptyVehicle 1
StayVehicleAtLoc 2 5
Car 2
OccupiedGr 5
EmptyVehicle 2
StayVehicleAtLoc 3 6
Car 3
OccupiedGr 6
EmptyVehicle 3
StayVehicleAtLoc 4 16
Boat 4
OccupiedWa 16
EmptyVehicle 4
StayVehicleAtLoc 5 30
Boat 5
OccupiedWa 30
EmptyVehicle 5
BuildCrAtLoc 0 32
BuildCrAtLoc 1 27
BuildCrAtLoc 2 9
BuildCrAtLoc 3 30

BuildCrAtLoc 4 3
BuildCrAtLoc 5 34
LayPlAtLoc 0 3
LayPlAtLoc 1 3
LayPlAtLoc 2 32
LayPlAtLoc 3 34
LayPlAtLoc 4 30
EmptyCr 0
EmptyCr 1
EmptyCr 2
EmptyCr 3
EmptyCr 4
EmptyCr 5
OnTop 1 0
On 1 0
On 0 5
OnPl 1 0
OnPl 0 0
OnTop 5 1
OnTop 4 2
On 4 3
On 3 2
On 2 5
OnPl 4 2
OnPl 3 2
OnPl 2 2
OnTop 5 3
OnTop 5 4
IsType 0 1
IsType 1 2
IsType 2 2
IsType 3 0
IsType 4 2