UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Karl Tarbe

Integer programming model for automated valet parking

Master's Thesis (30 ECTS)

Supervisor: Dr. Dirk Oliver Theis

Tartu 2016

Integer programming model for automated valet parking

Abstract: As parking becomes a more and more complex problem with the number of cars and city density, more complex technological solutions can be used to rectify it. One of possible solutions for parking is automated valet parking, where cars are not driven to parking place by humans but are carried by specially designed robots. Such solution presents us many possible optimization problems, one of which is addressed in this work using integer programming models, that can be solved using off-the-shelf solvers. We look at a specific possible implementation of automated valet parking and successfully design an integer programming model to solve it. Existing theoretical results have shown that even simplified cases of the problem can be APX-hard [4]. Using Gurobi, optimal solution was found for sample cases. The model is compared to another integer programming model from Algorithms and Theory research group, which provides comparison and verification for the model performance.

Keywords: integer programming

CERCS: P170 Computer science, numerical analysis, systems, control

Täisarvulise planeerimise mudel automaatparklale

Lühikokkuvõte:

Kuna parkimine on autode hulga suurenemisega ja linnastumise tihenemisega üha keerulisem probleem, muutub selle kõrgtehnoloogiline lahendamine otstarbekaks. Üks pakutud lahendus on automaatparkla, kus autodega ei sõideta oma parkimiskohta, vaid autod toimetatakse parkimiskohta ja tagasi spetsiaalsete robotite poolt. Selline kõrgtehnoloogiline lahendus annab meile palju erinevaid optimiseerimise ülesandeid ja võimalusi, millest ühte konkreetset käsitleme käesolevas töös kasutades täisarvulise planeerimise mudeleid, mida saab lahendada juba eksisteerivate lahendajatega. Käesolevas töös käsitletakse ühte kindlat võimalikku automaatparkla implementatsiooni ja edukalt tuletatakse täisarvulise planeerimise mudel selle lahendamiseks. Varasemad teoreetilised tulemused on näidanud, et isegi lihtsustatud variandid sellest probleemist saavad olla APX-keerukusega [4]. Kasutades Gurobi lahendajat leiti optimaalne lahendus näidisjuhtude jaoks. Mudelit võrreldi teise täisarv planeerimise mudeliga algoritmide ja teooria teadusgrupist, mis andis kindlust ja võrdlusmaterjali mudeli toimimisele.

Võtmesõnad: täisarvuline planeerimine

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

List of Figures **5**

List of Tables **6**

1 Introduction **8**
1.1 The problem . . . 8
1.2 Physical parameters of the problem . . . 9
1.3 Formulation as a discrete problem . . . 10
1.3.1 Possible further simplifications . . . 12

2 Integer programming model **14**
2.1 Variables . . . 16
2.1.1 Node status variables . . . 16
2.1.2 Edge-occupied variables . . . 16
2.1.3 Decision variables . . . 16
2.2 Example configurations to illustrate the model . . . 17
2.2.1 Collision . . . 18
2.2.2 Collision with same destination . . . 19
2.2.3 Collision with special . . . 19
2.2.4 Orthogonal collision . . . 20
2.2.5 Collision with itself . . . 21
2.2.6 Simultaneous movement . . . 21
2.2.7 Simultaneous movement II . . . 22
2.2.8 Invalid simultaneous movement . . . 23
2.2.9 Lift and move . . . 23
2.2.10 Lift and move forced . . . 24
2.2.11 Persistence, no robots . . . 25
2.2.12 Continue together . . . 26
2.3 Constraints . . . 26
2.3.1 Helper functions and some notations . . . 26
2.3.2 Simple constraints . . . 27
2.3.3 Lifting and dropping constraints . . . 28
2.3.4 Moving constraints . . . 29
2.3.5 Node status constraints . . . 33
2.3.6 Edge occupied constraints . . . 34
2.4 Initial status and objective . . . 35
2.5 Size of the model . . . 37
2.5.1 Number of variables . . . 37
2.5.2 Number of equalities . . . 37

2.5.3 Number of inequalities 38

3 Implementation **39**
3.1 PANDA 39
3.2 Graphical visualization 40
3.3 Irreducible inconsistent subsystem 41
3.4 Tuning Gurobi parameters 43

4 Comparison **44**
4.1 Testing platform 44
4.2 Problem instances used for comparison 44
4.3 Results 44
4.4 Interpretation of results 45

5 Conclusions **47**

References **48**

A Example of movement **49**

B Code created for the integer programming model **56**

List of Figures

1 This diagram serves as an illustration for the interpretation of time it takes to change direction. The direction of movement can change instantaneously, when speed is 0. Change of direction is not explicitly shown on this graph. 10
2 This figure shows all the possible movements to an adjacent parking space. 11
3 This figure shows the movement of an empty robot in *North* or *South* direction with moving for 2 parking spaces. The total distance traveled is 12 meters, which is exactly the length of 2 parking spaces in that direction. 12
4 This figure shows a pair of initial and terminal statuses, which cannot be solved. Green robot can move under the cars, but there is no way how red car can exchange places with the blue car without colliding. 13

5 This diagram serves as an illustration for node statuses during movement. Each row represents a seperate timestep of the same nodes. Gray boxes represent parking spaces and the green circle represents a moving robot. Below each parking space is the node status. . . . 15
6 Initial node statuses for collision test. 18
7 Initial node statuses for collision test with same destination. 19
8 Initial node statuses for collision test with loaded robots 19
9 Initial node statuses for collision test with orthogonal directions . . 20
10 Green rectangles are robots and gray rectangles are parking spaces. One robot is moving away from a parking space and another is moving toward it from an orthogonal direction. This causes a collision. 20
11 Initial node statuses for collision test with itself. 21
12 Green rectangles are robots and gray rectangles are parking spaces. Both robots are moving in the same direction without a collision as long as the left robot is not moving faster than the right one. 21
13 Initial node statuses for simultaneous movement test. 22
14 Initial node statuses for simultaneous movement test with a car on the way and the last robot carrying a car. 22
15 Initial node statuses for simultaneous movement test with an invalid node status constraint. 23
16 Initial node statuses for the lift and move test. 24
17 Initial node statuses for the forced lift and move test. 24
18 Initial node statuses for the persistence test. 25
19 Initial node statuses for the continue test. 26
20 Illustration for the meanings of u', u, v, v'. The arrows show the direction d. 30
21 Early textual output of variables. On the left are the node statuses for a 2x2 grid and on the right are the decision variables. Note that the variables were different then. 41
22 Screenshot of the graphical visualization. Circles are the nodes, gray lines between them show edges. Black arrows show occupied edges. Inside the nodes, the upper text displays the node status and the lower text the current decision. In the lower right corner is a number, which show the current timestep. 42

List of Tables

1 We list different actions and the time they need to complete. 12

2 We list all possible statuses of parking spaces with their descriptions. Note that $\mathbf{i} \in \{0, \ldots, K-1\}$. So when there is a robot carrying car with label 2, the status should be **2r**, not **ir**. 14
3 The table explaining the meaning of decision variables. 17
4 Performance measurements of two models. The C column displays the number of car that were not delivered to their destinations by $t_{\max}$. 45

1 Introduction

It does not take a scientific paper to see that the number of cars on our streets has increased. This creates a need for more parking places. It is common for shopping centers and other public places with lots of guests to have large parking lots or even parking houses. However, finding a parking space still is a frustrating experience for the drivers. Some fancy places have valet parking. Valet parking is rare in Estonia, but it is quite common in North America. Valet parking means that the driver hand the car over to a valet and a valet will park and retrieve their car when requested.

One way to further optimize the parking problem - save time and require less parking space - is to use automated valet parking. Instead of a human valet, there is a parking robot that carries your car to a free parking space. In the system explored in this thesis, drivers will stop their cars on a special platform. A parking robot can freely move under those platforms and also can lift the platform up from the ground and carry it with the car. For a driver the experience should be very similar to regular valet parking: driver stops near the door and does not need to worry about parking, after visiting the venue the driver can notify the system, which will then bring the car back to them.

1.1 The problem

In an automated valet parking system, the software dictates which parking robot will service the incoming parking or car retrieval request, it will also need to do path-planning for the robots. Regular car parks have passageways, so that no car is parked in a way that would block other cars from entering or exiting. In such automated system we can park cars more densely, because blocking cars can be lifted and moved out of the way. It is important that the time taken to retrieve cars is minimized, otherwise driver's time is wasted waiting. In a car park without passageways, the parking robots have to be cooperative and cannot do path planning as independent agents like in [11]. There are car parks with different layouts, some have large areas connected by narrow corridors, which would break algorithms that work on a rectangular grid. Algorithms specially designed for a park layout or even car parks built to match the specifics of an algorithm are operational, but they need a lot of work and are generally more expensive. We want to test if a general purpose tool like integer programming can be used to optimize automated valet parking operations.

There are a lot of different optimization problems in an automated parking system described. We chose to tackle a sub-problem: given an initial configuration of robots and cars in the system and a terminal configuration, find the best sequence of actions for robots to transform the initial configuration into the ter-

minal configuration. Note that best in this case means mostly shortest time, but we might also want to give some optimization weights for different actions because they waste energy and moving parts wear.

The goal of this thesis was to model the problem as an integer programming model optimization problem. Unlike heuristic algorithms, integer programming can give us a strong guarantee that the solution found is in fact optimal.

1.2 Physical parameters of the problem

The parking robot can move either forwards, backwards or sideways. The length of the parking space is 6 meters, and it is 3 meters wide. The robots have smaller dimensions. Every action takes some time. Lifting a car takes 9 seconds, lowering it takes 3 seconds. The speed of the robot is 2 meters per second, when not carrying a car, and 1 meter per second, when loaded. Robots needs time to accelerate and decelerate. Changing the direction takes 3 seconds.

The previous paragraph describes the estimated times which were to be used as basis for building the model. In the reality they might be different, but this is an idealized model. After all, the actual robots are not yet built.

It is perfectly valid to interpret the time required for stopping and reversing as acceleration. It is said that changing direction takes 3 seconds. The interpretation we used, is that it takes 1.5 seconds to stop a robot moving at its maximum speed. And another 1.5 seconds to accelerate it to maximum speed in another direction. Maximum speed and acceleration depends on whether the robot is carrying a car or not. According to Newton's Second Law, acceleration of robot can be greater when the mass is lower. Meaning that, in this case, a heavy loaded robot can accelerate at half the rate as unladen one. It might help to see Figure 1.

It is assumed that car park is a grid of parking spaces. We define a set of directions as $\mathcal{D} = \{North, East, South, West\}$. A parking place in the $North$ - $South$ dimension is 6 meters long, and in the $East$ - $West$ dimension is 3 meters wide. Note that the directions defined here do not have to correspond to geographical directions.

After those definitions we can say that an empty robot can go to adjacent space in the $North$ or $South$ direction in 4.5 seconds. Empty robot can go to adjacent space in the shorter dimension in 3 seconds. A loaded robot needs 7.5 seconds to reach adjacent parking space in the longer dimension and 4.5 seconds for shorter dimension. Speed versus time graph of all of those movements is on Figure 2, the area under those lines is equal to the dimensions of the parking space.

When a robot decides to move many spaces in one direction the distance traveled is equal to the distances covered while: accelerating, decelerating and moving with full speed. For an empty robot the distance traveled while accelerating and decelerating sums to 3 meters. For each additional parking place traveled we need

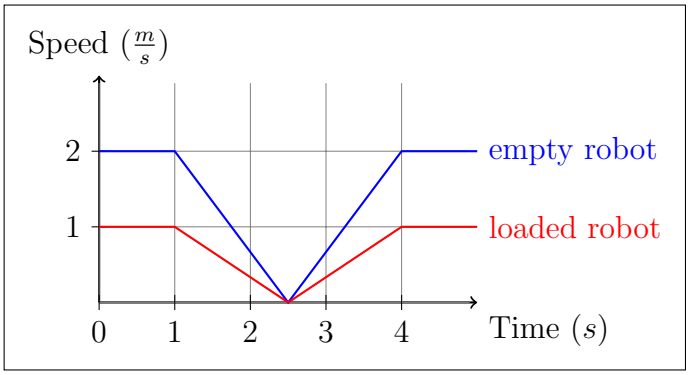


Figure 1: This diagram serves as an illustration for the interpretation of time it takes to change direction. The direction of movement can change instantaneously, when speed is 0. Change of direction is not explicitly shown on this graph.

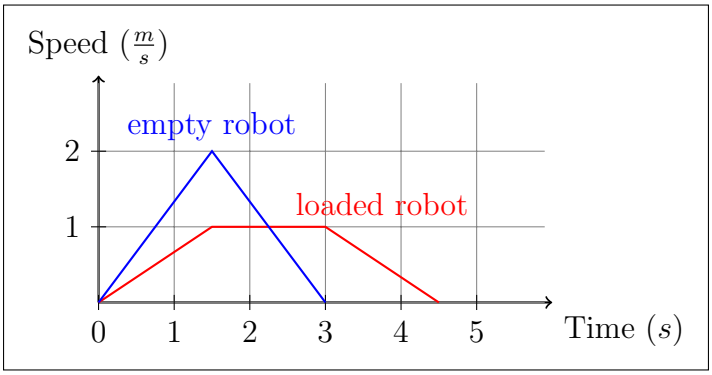
1.5 seconds for *East* or *West* direction. For *North* or *South* direction we first need to add 1.5 seconds for the first space and 3 seconds for each additional parking space. For a loaded robot the distance traveled while accelerating and decelerating sums to 1.5 meters. For the shorter dimension of the parking space, we need additional 1.5 seconds, to reach a full 3 meters. For additional spaces, we need to add 3 seconds for each additional space traveled. For the longer dimension, we need additional 4.5 seconds, and for each additional space we need to add 6 seconds to reach the desired distance.

As an example, let us consider that a loaded robot wants to move 3 parking spaces in the direction *North*. The total distance needed to cover is $3 \times 6 = 18$ meters. It takes $3 + 4.5 + 2 \times 6 = 19.5$ seconds. Lets verify that the distance is right: for 3 seconds we are accelerating and decelerating, with top speed 1 $\frac{m}{s}$, the average speed being 0.5 $\frac{m}{s}$. Other time we are moving with top speed which is 1 $\frac{m}{s}$. The distance traveled will be $3 * 0.5 + 4.5 \times 1 + 2(6 \times 1) = 18$ meters. An additional example is shown on Figure 3.

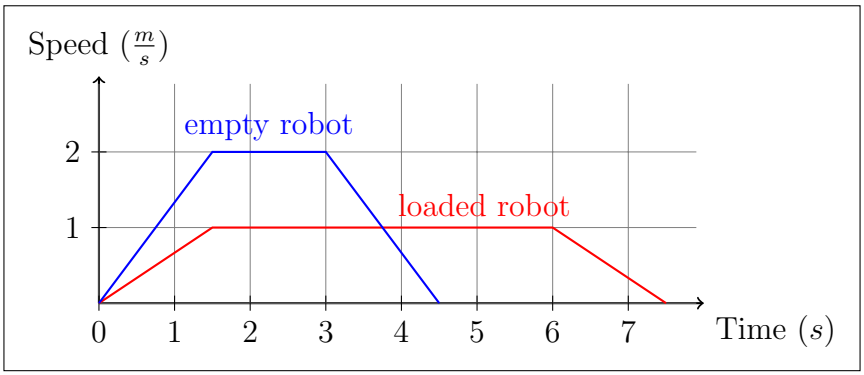
1.3 Formulation as a discrete problem

From the Table 1, it can be seen that all the times are multiples of 1.5 seconds. That means we can take 1.5 seconds as a our timestep. All the movements take a discrete number of timesteps and every movement ends up in the exact center of the parking place.

For the car park, we use graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where every vertex is a parking space and every edge (u, v) means that there is no obstacle between parking spaces u and v. The parking places have degree at most 4, with the edges corresponding to



(a) Moving in the *East* or *West* direction with parking space 3 meters wide.



(b) Moving in the *North* or *South* direction with parking space 6 meters long.

Figure 2: This figure shows all the possible movements to an adjacent parking space.

directions in $\mathcal{D}$.

We let K be the number of different labels we want to give to cars. Cars with the same label are indistinguishable in our model. Usually there are a lot of dormant cars and only the cars that have just arrived or requested need to be given distinctive labels. We do not label robots as they are interchangeable.

The problem is to find a suitable sequence of actions for robots, to reconfigure the given initial configuration into the terminal configuration. Both initial and terminal configurations are given by node statuses for every vertex.

Sometimes a given initial status with given terminal status is unsolvable no matter how many timesteps are considered. For a very simple example see Figure 4.

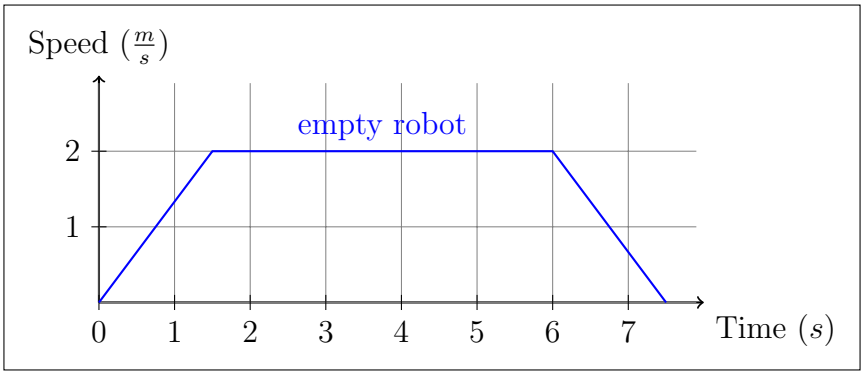


Figure 3: This figure shows the movement of an empty robot in *North* or *South* direction with moving for 2 parking spaces. The total distance traveled is 12 meters, which is exactly the length of 2 parking spaces in that direction.

Action	Time (s)
Lift car	9
Drop car	3
Accelerate to full speed	1.5
Decelerate from full speed to stop	1.5
Loaded robot moving to adjacent space (N or S)	7.5
Loaded robot moving to adjacent space (E or W)	4.5
Empty robot moving to adjacent space (N or S)	3
Empty robot moving to adjacent space (E or W)	4.5

Table 1: We list different actions and the time they need to complete.

1.3.1 Possible further simplifications

From theoretical viewpoint, there is an interesting and relevant result in [4]. For this abstraction, we further simplify our problem, and also forget about robots and the moving times. We just have a graph $\mathcal{G}$, initial configuration of cars, and terminal configuration of cars. Cars can move from one node u to another node v in an instant, the only constraint is that there must be path from u to v, where all intermediate nodes are unoccupied by other cars. The problem is still to reconfigure the cars, so that terminal configuration is obtained with minimum number of moves.

Theorem 1 from [4], states that for every connected graph, for every initial and terminal configuration with n cars, there exists a solution with at most n moves and for trees there is a linear time algorithm for finding the optimal moves.

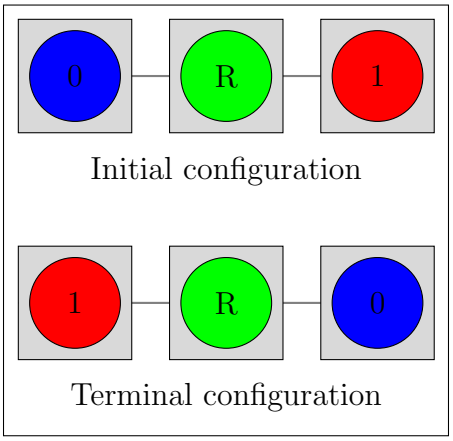


Figure 4: This figure shows a pair of initial and terminal statuses, which cannot be solved. Green robot can move under the cars, but there is no way how red car can exchange places with the blue car without colliding.

However, that applies to only unlabeled cars.

Maybe there are large firms, where everyone can just ride a random company car. But in general a user expects to receive a specific car. Therefore we still have to keep cars labeled.

According to theorem 3 from [4], with labeled cars on a graph, the reconfiguration problem is APX-hard. If $P \neq NP$, this implies that there is no polynomial-time approximation scheme for the problem.

2 Integer programming model

We will use the definitions $\mathcal{V}$ and $\mathcal{E}$ from page 10 and definition of $\mathcal{D}$ from page 9 quite extensively in this section. Every parking space has a status. All possible node statuses are described in Table 2. We denote the set of all node statuses as $\mathcal{W}$.

Status	Meaning
e	Parking space is empty.
r	There is a robot on the parking space.
i	There is a car with label i in the parking space.
ri	There is a car with label i and a robot is under it.
ir	There is a robot carrying the car with label i.
lft	There is a robot in the process of lifting a car.
drp	There is a robot in the process of dropping a car.

Table 2: We list all possible statuses of parking spaces with their descriptions. Note that $\mathbf{i} \in \{0, \ldots, K-1\}$. So when there is a robot carrying car with label 2, the status should be **2r**, not **ir**.

When an object moves from node u to an adjacent node v, the initial node status remains the same for the whole duration of the movement. Meaning that even though the moving object partly blocks both u and v, the node statuses for u and v will still be the old ones, until the moving object reaches the center of v, at which point node statuses are updated. As an example of this see Figure 5.

We have to fix the number of timesteps that the model has. We use $\mathcal{T} = \{0, \ldots, t_{\max}\}$ to denote the set of all timesteps in the model.

Now we define some subsets of node statuses. We start with $\mathcal{W}_{\text{lift}}$, the set of statuses where robot is under a car and decision to lift it can be made. Second is almost the opposite set $\mathcal{W}_{\text{drop}}$, which contains all the statuses, where robot is carrying a car and decision to drop it can be made. Third set is $\mathcal{W}_{\text{move}}$ and it consists of node statuses that contain something, that can move. The last set is $\mathcal{W}_{\text{mc}}$ that correspond to moving components. A robot can move and a robot carrying a car can move.

$$\mathcal{W}_{\text{lift}} = \{\mathbf{ri} \mid i \in \{0, \ldots, K-1\}\} \tag{1}$$
$$\mathcal{W}_{\text{drop}} = \{\mathbf{ir} \mid i \in \{0, \ldots, K-1\}\} \tag{2}$$
$$\mathcal{W}_{\text{move}} = \mathcal{W}_{\text{lift}} \cup \mathcal{W}_{\text{drop}} \cup \{\mathbf{r}\} \tag{3}$$
$$\mathcal{W}_{\text{mc}} = \mathcal{W}_{\text{drop}} \cup \{\mathbf{r}\} \tag{4}$$

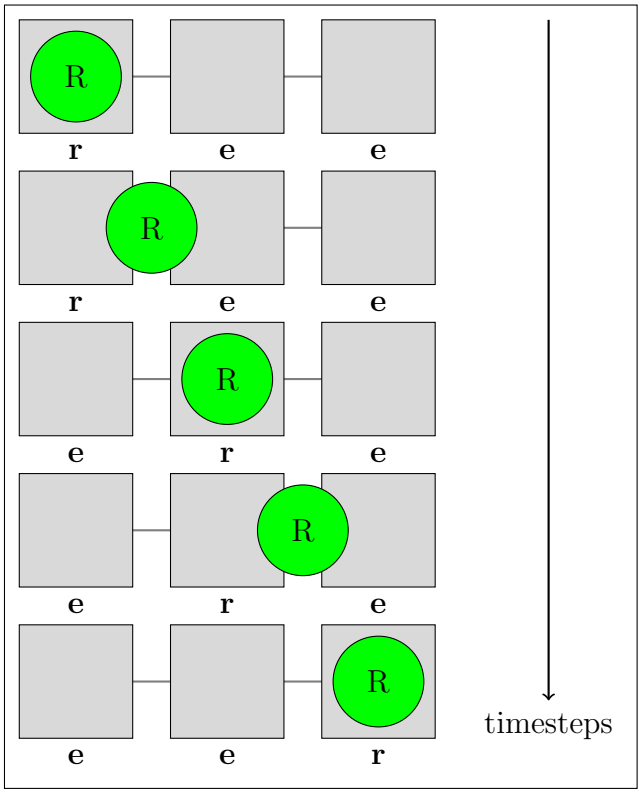


Figure 5: This diagram serves as an illustration for node statuses during movement. Each row represents a seperate timestep of the same nodes. Gray boxes represent parking spaces and the green circle represents a moving robot. Below each parking space is the node status.

Decelerating a moving robot takes 1 timestep. Therefore, the decision to continue moving in the same direction or to stop has to be made 1 timestep before arriving at an adjacent node. Since we change the node statuses only when we arrive at the center of parking place as illustrated on Figure 5, the decisions to continue or stop moving are also made on the same node. The object has not yet arrived at the other node, so it does not make sense to tie decision to the destination node.

Proposition 1. *All feasible solutions to the integer programming model correspond to feasible actions of robots and all feasible actions of the robots correspond to feasible solutions to the integer programming model.*

Proof. This will become clear from the explanations of variables in subsection 2.1 and explanations of constraints in subsection 2.3. □

2.1 Variables

The variables for the model are all binary, meaning their allowed values are from the set $\{0, 1\}$. There are three groups of variables:

1. node status variables,
2. edge-occupied variables,
3. decision variables.

For every timestep, there is another layer of the variables. Meaning that if we have n variables for timestep 0, and we have t timesteps, then in total there will be tn variables in the model. In the variable indexing, the timestep is always the last index.

2.1.1 Node status variables

As specified in subsection 1.3, there are many possible statuses for parking spots. Node status variables are meant to encode the node status. There is a variable for every combination of node, status and timestep. To be more accurate, the group of node status variables is defined as following.

$$\forall v \in \mathcal{V} \colon \forall w \in \mathcal{W} \colon \forall t \in \mathcal{T} \colon \qquad \text{nstat}_{v,w,t} \tag{5}$$

2.1.2 Edge-occupied variables

There are variables for each directed edge between parking spots. When a robot moves from vertex u to vertex v at timestep t, the edge variable $\text{occu}_{(u,v),t}$ is 1 to indicate that the edge is occupied. All edge variables are defined as:

$$\forall e \in \mathcal{E} \colon \forall t \in \mathcal{T} \colon \qquad \text{occu}_{e,t} \tag{6}$$

2.1.3 Decision variables

This is the most important group of variables, because they represent the decisions or actions that the model should find. Before we can give them, we have to introduce a little function $\text{dir} \colon \mathcal{E} \to \mathcal{D}$. Since we are operating on not just any graph, but on a grid, we can determine the direction of the edge and dir does

exactly that. The decision variables are the following.

$$\forall e = (u, v) \in \mathcal{E} \colon \forall w \in \mathcal{W}_{\text{mc}} \colon \forall t \in \mathcal{T} \colon \quad \text{go}_{u,w,\text{dir}(e),t} \tag{7}$$

$$\forall e = (u, v) \in \mathcal{E} \colon \forall w \in \mathcal{W}_{\text{mc}} \colon \forall t \in \mathcal{T} \colon \quad \text{cont}_{u,w,\text{dir}(e),t} \tag{8}$$

$$\forall e = (u, v) \in \mathcal{E} \colon \forall w \in \mathcal{W}_{\text{mc}} \colon \forall t \in \mathcal{T} \colon \quad \text{stop}_{u,w,\text{dir}(e),t} \tag{9}$$

$$\forall v \in \mathcal{V} \colon \forall w \in \mathcal{W}_{\text{lift}} \colon \forall t \in \mathcal{T} \colon \quad \text{lift}_{v,w,t} \tag{10}$$

$$\forall v \in \mathcal{V} \colon \forall w \in \mathcal{W}_{\text{drop}} \colon \forall t \in \mathcal{T} \colon \quad \text{drop}_{v,w,t} \tag{11}$$

Decision variable is set to 1, if and only if that action is taken at the specified node, with specified node status, and at specified timestep. For movement variables: $\text{go}_{v,w,d,t}$, $\text{cont}_{v,w,d,t}$ and $\text{stop}_{v,w,d,t}$ also the direction is specified. Decision variables already have self-describing names, but to clarify, we explain them one-by-one in Table 3.

Variable	Meaning
$\text{go}_{v,w,d,t}$	At timestep t on parking space v with node status w, a robot starts to accelerate in direction d.
$\text{cont}_{v,w,d,t}$	At timestep t on parking space v with node status w a robot that is about to reach the neighbouring node in direction d, does not slow down and continues moving in direction d.
$\text{stop}_{v,w,d,t}$	At timestep t on parking space v with node status w a robot that is about to reach the neighbouring node in direction d, starts to decelerate to stop in the neighbouring node.
$\text{lift}_{v,w,t}$	At timestep t on parking space v with node status w a robot starts to lift a car.
$\text{drop}_{v,w,t}$	At timestep t on parking space v with node status w a robot starts to drop a car.

Table 3: The table explaining the meaning of decision variables.

2.2 Example configurations to illustrate the model

Before diving into constraints, we give 12 small configurations which will illustrate the meaning of variables and our assumptions about the integer programming model.

As a side remark: complexity of the integer programming model is large enough to not fit into the working memory of humans. Humans can keep about 7 things in their working memory at once [9]. When developing the model a lot of time was consumed by manually altering the initial status of the model and then verifying if the feasible optimal solution to model was indeed a valid sequence of actions. To

cut down the time on manual rewriting of the initial status and manual verifying of solutions, these illustration configurations were also used for testing the model during development.

Testing was needed to avoid typos like writing v or even u instead of v'. Also there were some occurrences, where we made wrong assumptions, because there are so many cases to consider for a simple action.

Some of these illustrations or tests are meant to be infeasible. Others have expected outcomes, which are automatically verified. When a specific objective function is not mentioned, the objective function will be left empty for that test. By empty objective we mean that the objective function is $\min 0$.

The initial node statuses are fixed for every configuration. To easily display the initial statuses, we use images that depict the underlying graph of the problem and the node statuses at timestep 0. In the images, nodes are represented as circles. They gray lines between nodes show that there is an edge between those vertices. Inside the circles the upper text shows the node status and the lower text shows the identifier of the node.

2.2.1 Collision

The purpose of this test is to make sure head on collisions are infeasible in the model. To force a collision, we use the following constraints to fix the values of

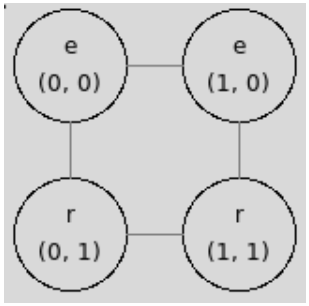


Figure 6: Initial node statuses for collision test.

two decision variables.

$$\mathrm{go}_{(0,1),\mathbf{r},East,0} = 1 \tag{12}$$

$$\mathrm{go}_{(1,1),\mathbf{r},West,0} = 1 \tag{13}$$

The desired output is infeasibility of the model.

2.2.2 Collision with same destination

Next test makes sure two robots cannot move into the same node at the same time. The following constraints oblige both of the robots to start moving to node

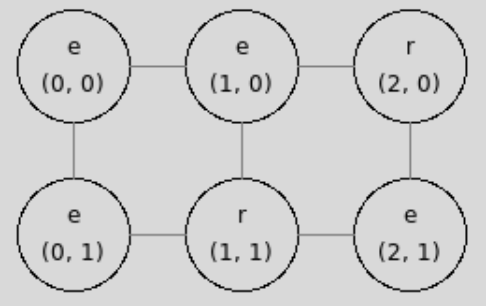


Figure 7: Initial node statuses for collision test with same destination.

$(1, 0)$.

$$\mathrm{go}_{(2,0),\mathbf{r},West,0} = 1 \tag{14}$$

$$\mathrm{go}_{(1,1),\mathbf{r},North,0} = 1 \tag{15}$$

As with the last test, this model should be infeasible.

2.2.3 Collision with special

Last two tests made sure that robots cannot collide, but another test is needed for robots carrying cars with different labels. We use the following constraints,

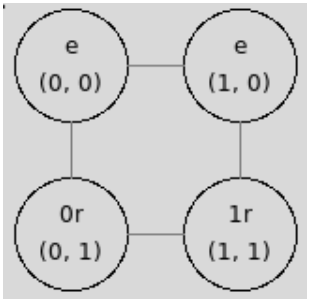


Figure 8: Initial node statuses for collision test with loaded robots

which are very similar to constraints used in the head-on collision test.

$$\mathrm{go}_{(0,1),\mathbf{0r},East,0} = 1 \tag{16}$$

$$\mathrm{go}_{(1,1),\mathbf{1r},West,0} = 1 \tag{17}$$

As with other collision tests, the model should be infeasible.

2.2.4 Orthogonal collision

We also have a test for collision with orthogonal directions. We use the following

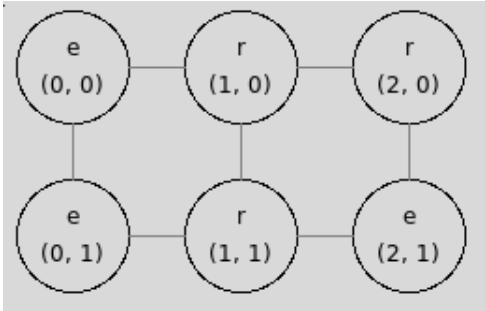


Figure 9: Initial node statuses for collision test with orthogonal directions

constraints to fix the movements.

$$\mathrm{go}_{(1,0),\mathbf{r},West,0} = 1 \tag{18}$$

$$\mathrm{go}_{(1,1),\mathbf{r},North,0} = 1 \tag{19}$$

Here robot on $(1, 0)$ is moving away from it, and robot on $(1, 1)$ is trying to move into $(1, 0)$. This kind of collision is explained with Figure 10. Because of the collision, the model should be infeasible.

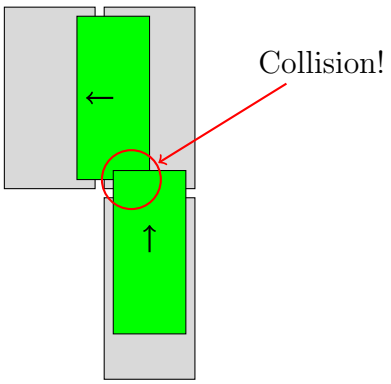


Figure 10: Green rectangles are robots and gray rectangles are parking spaces. One robot is moving away from a parking space and another is moving toward it from an orthogonal direction. This causes a collision.

2.2.5 Collision with itself

This is a test to see that edge constraints make edges occupied for the right number of timesteps. The initial conditions are forced with the following constraints.

Figure 11: Initial node statuses for collision test with itself.

$$\text{go}_{(2,0),\mathbf{r},West,0} = 1 \tag{20}$$

$$\text{go}_{(1,0),\mathbf{r},East,2} = 1 \tag{21}$$

When the edges are occupied for longer than actually needed, the model will become infeasible, because the first movement marks edge $((2,0),(1,0))$ as occupied, and the second movement marks edge $((1,0),(2,0))$ as occupied. And they cannot be both occupied at the same time because that would be a collision.

This model should be feasible.

2.2.6 Simultaneous movement

With this test, we make sure that robots can move in the same direction side by side, for explanation see Figure 12.

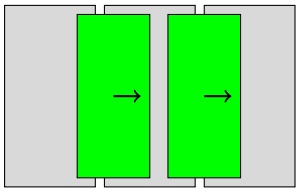


Figure 12: Green rectangles are robots and gray rectangles are parking spaces. Both robots are moving in the same direction without a collision as long as the left robot is not moving faster than the right one.

Both robots are forced to move at the start by following constraints.

$$\text{go}_{(1,0),\mathbf{r},West,0} = 1 \tag{22}$$

$$\text{go}_{(2,0),\mathbf{r},West,0} = 1 \tag{23}$$

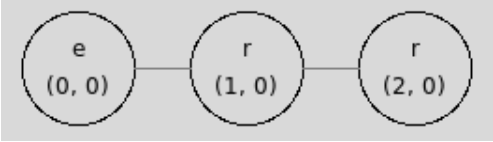


Figure 13: Initial node statuses for simultaneous movement test.

The model should be feasible and a feasible solution is also verified by following checks, which should hold for all feasible solutions.

$$\text{nstat}_{(0,0),\mathbf{r},2} \stackrel{?}{=} 1 \tag{24}$$

$$\text{nstat}_{(1,0),\mathbf{r},2} \stackrel{?}{=} 1 \tag{25}$$

$$\text{nstat}_{(2,0),\mathbf{e},2} \stackrel{?}{=} 1 \tag{26}$$

2.2.7 Simultaneous movement II

This is quite similar to last test, but now we have other than empty nodes in the way and one of the robots is also carrying a car. The following constraints force all the robots to start moving to $West$.

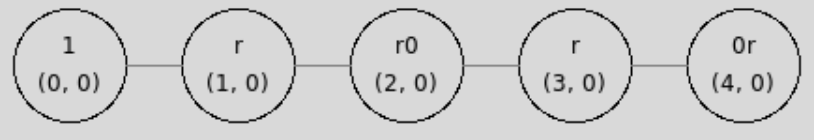


Figure 14: Initial node statuses for simultaneous movement test with a car on the way and the last robot carrying a car.

$$\text{go}_{(1,0),\mathbf{r},West,0} = 1 \tag{27}$$

$$\text{go}_{(2,0),\mathbf{r},West,0} = 1 \tag{28}$$

$$\text{go}_{(3,0),\mathbf{r},West,0} = 1 \tag{29}$$

$$\text{go}_{(4,0),\mathbf{0r},West,0} = 1 \tag{30}$$

The model should be feasible and optimal solution is verified by following checks, which verify that node statuses after movement are correct.

$$\text{nstat}_{(1,0),\mathbf{r1},2} \stackrel{?}{=} 1 \tag{31}$$

$$\text{nstat}_{(1,0),\mathbf{r},2} \stackrel{?}{=} 1 \tag{32}$$

$$\text{nstat}_{(2,0),\mathbf{r0},2} \stackrel{?}{=} 1 \tag{33}$$

$$\text{nstat}_{(3,0),\mathbf{0r},3} \stackrel{?}{=} 1 \tag{34}$$

$$\text{nstat}_{(4,0),\mathbf{e},3} \stackrel{?}{=} 1 \tag{35}$$

2.2.8 Invalid simultaneous movement

This test checks if node statuses are handled correctly when robots move side-by-side. We force simultaneous movement and an invalid node status after the

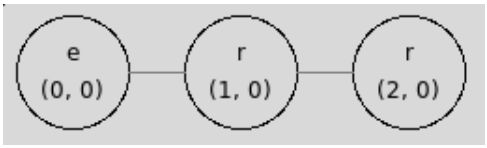


Figure 15: Initial node statuses for simultaneous movement test with an invalid node status constraint.

movement with these constraints.

$$\text{go}_{(1,0),\mathbf{r},West,0} = 1 \tag{36}$$

$$\text{go}_{(2,0),\mathbf{r},West,0} = 1 \tag{37}$$

$$\text{nstat}_{(1,0),\mathbf{r0},2} = 1 \tag{38}$$

The status of node $(1, 0)$ after the movement, at timestep 2 should be $\mathbf{r}$. However, we give a constraint which should make the model infeasible.

2.2.9 Lift and move

This test checks, if the model can find a correct solution to a simple problem. This time instead of adding constraints to force some actions, we define the objective function. The objective is to get a robot and a car labeled 0 to node $(0, 0)$ as soon as possible.

$$\min \sum_{t \in \mathcal{T}} -\text{nstat}_{(0,0),\mathbf{r0},t} \tag{39}$$

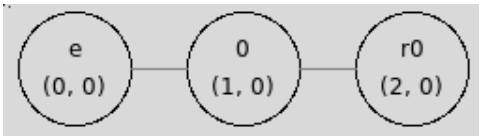


Figure 16: Initial node statuses for the lift and move test.

Correct solution would be for the robot to move into the middle node $(1, 0)$, lift the car from there and carry it to $(0, 0)$ and finally drop it there. The model should be feasible and solution is checked against the following conditions.

$$\text{nstat}_{(0,0),\mathbf{e},2} \stackrel{?}{=} 1 \tag{40}$$

$$\text{nstat}_{(1,0),\mathbf{r0},2} \stackrel{?}{=} 1 \tag{41}$$

$$\text{nstat}_{(2,0),\mathbf{0},2} \stackrel{?}{=} 1 \tag{42}$$

$$\text{nstat}_{(1,0),\mathbf{0r},8} \stackrel{?}{=} 1 \tag{43}$$

$$\text{nstat}_{(0,0),\mathbf{0r},11} \stackrel{?}{=} 1 \tag{44}$$

$$\text{nstat}_{(0,0),\mathbf{r0},13} \stackrel{?}{=} 1 \tag{45}$$

These conditions just make sure that node statuses at various timesteps are correct.

2.2.10 Lift and move forced

This test is almost the same as previous, but now the decisions are forced with the following constraints.

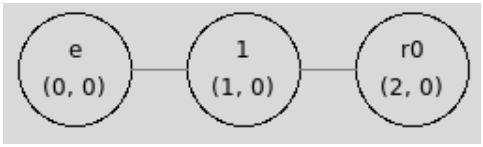


Figure 17: Initial node statuses for the forced lift and move test.

$$\text{go}_{(2,0),\mathbf{r},West,0} = 1 \tag{46}$$

$$\text{lift}_{(1,0),\mathbf{r1},2} = 1 \tag{47}$$

$$\text{go}_{(1,0),\mathbf{1r},West,8} = 1 \tag{48}$$

$$\text{nstat}_{(0,0),\mathbf{e},8} = 1 \tag{49}$$

The objective is now to issue as many lifts at node $(0,0)$ for car with label 1 as possible.

$$\min \sum_{t \in \mathcal{T}} -\text{lift}_{(0,0),\mathbf{r1},t} \tag{50}$$

The solution is checked against these conditions.

$$\text{nstat}_{(0,0),\mathbf{e},2} \stackrel{?}{=} 1 \tag{51}$$

$$\text{nstat}_{(1,0),\mathbf{r1},2} \stackrel{?}{=} 1 \tag{52}$$

$$\text{nstat}_{(2,0),\mathbf{0},2} \stackrel{?}{=} 1 \tag{53}$$

$$\text{nstat}_{(1,0),\mathbf{1r},8} \stackrel{?}{=} 1 \tag{54}$$

$$\text{nstat}_{(0,0),\mathbf{1r},11} \stackrel{?}{=} 1 \tag{55}$$

When the previous test failed, the model did found some infeasible movement to obtain a better objective value. This test was added to see, if a legal sequence of decisions was indeed feasible and we have not accidentally over constrained the model.

2.2.11 Persistence, no robots

This test is to make sure that node statuses stay the same when there are no robots. We also add one odd constraint.

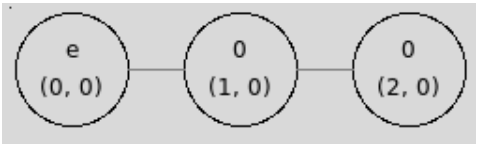


Figure 18: Initial node statuses for the persistence test.

$$\text{nstat}_{(0,0),\mathbf{r},20} = 1 \tag{56}$$

And the objective function used encourages the model to occupy edges and make robots appear.

$$\min \sum_{t \in \mathcal{T}} \left(-4\text{nstat}_{(0,0),\mathbf{r},t} - 4\text{nstat}_{(1,0),\mathbf{r},t} - \sum_{e \in \mathcal{E}} \text{occu}_{e,t} \right) \tag{57}$$

Of course node statuses should not change by themselves, and the model should be infeasible.

2.2.12 Continue together

This tests makes sure that robots can continue moving in one direction, even when moving simultaneously. Actually it tests a single robot continuing and at the same time also 2 robots continuing simultaneously.

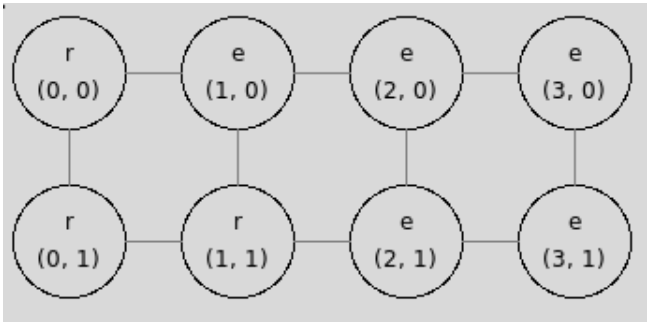


Figure 19: Initial node statuses for the continue test.

The objective is for robots to go to the right side.

$$\min \sum_{t \in \mathcal{T}} -\text{nstat}_{(3,0),\mathbf{r},t} - \text{nstat}_{(3,1),\mathbf{r},t} - \text{nstat}_{(2,1),\mathbf{r},t} \tag{58}$$

The model should be feasible and the optimal solution is verified by hand. Valid solution would have robots in the east at the last timestep. And robots do not stop at intermediate nodes.

2.3 Constraints

In this section we give the constraints of the integer programming model. Some of these constraints are redundant, but they might help the solver and they also provide a meaningful bound for some abbreviations.

2.3.1 Helper functions and some notations

The first helper function used is $\text{edg}\colon \mathcal{V} \times \mathcal{D} \to \mathcal{V}$. Let $\text{edg}(v, d)$ denote the node adjacent to v that is in the direction d from node v.

For directions we use $d + 1$ to denote the next direction from direction d in clockwise manner. Logically $d - 1$ is used to mark the next direction from d in counterclockwise manner. For an example: $North + 1 = East$.

Some decisions take a specified amount of time. We want to disallow making such decisions, which lead to actions that cannot be completed in the time frame

of the model. For this purpose we want to split $\mathcal{T}$ into two disjoint subsets. Let $\mathcal{T}_i = \{t \mid (t+i) \in \mathcal{T}\}$ and $\overline{\mathcal{T}}_i = \mathcal{T} \setminus \mathcal{T}_i$.

The decisions to stop moving or continue moving are mostly used together. Either one of these implies that node status is about to change. For that reason we use an abbreviation $\cos_{u,w,d,t} = \text{cont}_{u,w,d,t} + \text{stop}_{u,w,d,t}$.

Sometimes we need to denote the set of neighbours for a vertex. We use $\mathcal{N}(v)$ to denote the neighbouring vertices of node v.

Depending on the direction and whether the robot is carrying a car, the movement takes different number of timesteps. For this, we introduce a new function $\text{getMT}\colon \mathcal{W}_{\text{mc}} \times \mathcal{D} \to \mathbb{N}$. Function $\text{getMT}(w, d)$ returns the number of timesteps the movement of w takes in direction d.

For working with node statuses related to moving there is a notion of moving component, which is the status, that can move. First is when node status is w, we need to get the status that can move away from w, we define $\text{getMC}\colon \mathcal{W}_{\text{move}} \to \mathcal{W}_{\text{mc}}$ as:

$$\forall i \in \{0, \ldots, K-1\}\colon \qquad \text{getMC}(\mathbf{ri}) = \mathbf{r} \tag{59}$$
$$\forall i \in \{0, \ldots, K-1\}\colon \qquad \text{getMC}(\mathbf{ir}) = \mathbf{ir} \tag{60}$$
$$\text{getMC}(\mathbf{r}) = \mathbf{r} \tag{61}$$

Next we want to know what remained in the node, when moving component was removed, for that we define function $\text{remMC}\colon \colon \mathcal{W}_{\text{move}} \to \mathcal{W}$ as:

$$\forall i \in \{0, \ldots, K-1\}\colon \qquad \text{remMC}(\mathbf{ri}) = \mathbf{i} \tag{62}$$
$$\forall i \in \{0, \ldots, K-1\}\colon \qquad \text{remMC}(\mathbf{ir}) = \mathbf{e} \tag{63}$$
$$\text{remMC}(\mathbf{r}) = \mathbf{e} \tag{64}$$

And sometimes when a moving component comes into node with some status, we want to know what will the new status will be. We define a partial function $\text{addMC}\colon \mathcal{W} \times \mathcal{W}_{\text{mc}} \to \mathcal{W}$ as:

$$\forall i \in \{0, \ldots, K-1\}\colon \qquad \text{addMC}(\mathbf{e}, \mathbf{ir}) = \mathbf{ir} \tag{65}$$
$$\forall i \in \{0, \ldots, K-1\}\colon \qquad \text{addMC}(\mathbf{i}, \mathbf{r}) = \mathbf{ri} \tag{66}$$
$$\text{addMC}(\mathbf{e}, \mathbf{r}) = \mathbf{r} \tag{67}$$

2.3.2 Simple constraints

First constraint is to make sure that at every timestep each node has exactly one status variable set.

$$\forall v \in \mathcal{V}\colon \forall t \in \mathcal{T}\colon \qquad \sum_{w \in \mathcal{W}} \text{nstat}_{v,w,t} = 1 \tag{68}$$

When a directed edge is used, its opposite cannot be used at the same time.

$$\forall (u,v) \in \mathcal{E} \colon \forall t \in \mathcal{T} \colon \qquad \mathrm{occu}_{(u,v),t} + \mathrm{occu}_{(v,u),t} \leq 1 \tag{69}$$

No more than one moving thing can arrive at the same node at the same time.

$$\forall v \in \mathcal{V} \colon \forall t \in \mathcal{T} \colon \qquad \sum_{u \in \mathcal{N}(v)} \mathrm{occu}_{(u,v),t} \leq 1 \tag{70}$$

We also want to forbid movements in the orthogonal directions.

$$\begin{aligned} \forall v \in \mathcal{V} \colon \forall d \in \mathcal{D} \colon \forall t \in \mathcal{T} \colon \quad & \mathrm{occu}_{(\mathrm{edg}(v,d),v),t} \\ & + \mathrm{occu}_{(v,\mathrm{edg}(v,d-1)),t} + \mathrm{occu}_{(v,\mathrm{edg}(v,d+1)),t} \leq 1 \end{aligned} \tag{71}$$

Note that there are vertices such that they do not have an edge between them, or they do not even have a neighbour in some directions. When that happens, the invalid $\mathrm{occu}_{e,t}$ variables are not added to the sums in constraints (70) and (71).

It does not make sense to allow more than 1 decision at the same time, at the same location. Only one robot is at one node at one time and it cannot make two actions at the same time.

$$\begin{aligned} \forall v \in \mathcal{V} \colon \forall t \in \mathcal{T} \colon \quad & \sum_{d \in \mathcal{D}, w \in \mathcal{W}_{\mathrm{mc}}} (\mathrm{go}_{v,w,d,t} + \mathrm{stop}_{v,w,d,t} + \mathrm{cont}_{v,w,d,t}) \\ & + \sum_{w \in \mathcal{W}_{\mathrm{lift}}} \mathrm{lift}_{v,w,t} + \sum_{w \in \mathcal{W}_{\mathrm{drop}}} \mathrm{drop}_{v,w,t} \leq 1 \end{aligned} \tag{72}$$

2.3.3 Lifting and dropping constraints

These constraints describe the lifting and dropping of cars. In fact they are quite similar, they indeed are opposite actions. We will start with lifting constraints. Instead of using $\mathcal{T}$, like we did in subsubsection 2.3.2, we use $\mathcal{T}_6$, because lifting takes 6 timesteps. First constraint type is to make sure that the node status is correct, when a decision to lift is made.

$$\forall v \in \mathcal{V} \colon \forall w \in \mathcal{W}_{\mathrm{lift}} \colon \forall t \in \mathcal{T}_6 \colon \qquad \mathrm{lift}_{v,w,t} - \mathrm{nstat}_{v,w,t} \leq 0 \tag{73}$$

While the lifting is in progress, we want the node status be fixed to **lft**.

$$\forall v \in \mathcal{V} \colon \forall w \in \mathcal{W}_{\mathrm{lift}} \colon \forall t \in \mathcal{T}_6 \colon \forall i \in \{1, \ldots, 5\} \colon \quad \mathrm{lift}_{v,w,t} - \mathrm{nstat}_{v,\mathbf{lft},t+i} \leq 0 \tag{74}$$

And at the end of lifting, the node status should correspond to the lifted thing. For that we have a simple bijective helper function $f \colon \mathcal{W}_{\mathrm{lift}} \to \mathcal{W}_{\mathrm{drop}}$. It is defined as: $\forall i \in \{0, \ldots, K-1\} \colon f(\mathbf{ri}) = \mathbf{ir}$. In words, the function f determines the node

status after lifting. By using f we can now give the constraints, which determine the node status after lifting.

$$\forall v \in \mathcal{V}\colon \forall w \in \mathcal{W}_{\text{lift}}\colon \forall t \in \mathcal{T}_6\colon \qquad \text{lift}_{v,w,t} - \text{nstat}_{v,f(w),t+6} \leq 0 \tag{75}$$

If the lifting cannot be completed, we just make sure that the decision is never made. As a reminder $\overline{\mathcal{T}}_6$ is the set timesteps for which adding 6 more timesteps would be more that $t_{\max}$, and therefore be outside the scope of the model.

$$\forall v \in \mathcal{V}\colon \forall w \in \mathcal{W}_{\text{lift}}\colon \forall t \in \overline{\mathcal{T}}_6\colon \qquad \text{lift}_{v,w,t} = 0 \tag{76}$$

For dropping we have almost the same constraints. First make sure that the node status is correct, when decision is made.

$$\forall v \in \mathcal{V}\colon \forall w \in \mathcal{W}_{\text{drop}}\colon \forall t \in \mathcal{T}_2\colon \qquad \text{drop}_{v,w,t} - \text{nstat}_{v,w,t} \leq 0 \tag{77}$$

Dropping takes less time than lifting, therefore the constraints for intermediate node state are simpler than for lifting.

$$\forall v \in \mathcal{V}\colon \forall w \in \mathcal{W}_{\text{drop}}\colon \forall t \in \mathcal{T}_2\colon \qquad \text{drop}_{v,w,t} - \text{nstat}_{v,\mathbf{drp},t+1} \leq 0 \tag{78}$$

To get the correct node status at the end of the drop, we can use the inverse of the previous helper function f.

$$\forall v \in \mathcal{V}\colon \forall w \in \mathcal{W}_{\text{drop}}\colon \forall t \in \mathcal{T}_2\colon \qquad \text{drop}_{v,w,t} - \text{nstat}_{v,f^{-1}(w),t+2} \leq 0 \tag{79}$$

As with lifting, if dropping action cannot be completed in the time frame of the model, disable the decision.

$$\forall v \in \mathcal{V}\colon \forall w \in \mathcal{W}_{\text{drop}}\colon \forall t \in \overline{\mathcal{T}}_2\colon \qquad \text{drop}_{v,w,t} = 0 \tag{80}$$

2.3.4 Moving constraints

While constructing movement constraints, we need some abbreviations and cannot write all the quantifiers before every constraint. Therefore, the constraints in this section are for all combinations of $u \in \mathcal{V}$, $w \in \mathcal{W}_{\text{mc}}$, $d \in \mathcal{D}$ and $t \in \mathcal{T}$.

Now let $\delta = \text{getMT}(w, d)$, and $\delta' = \delta - 1$. When movement starts at time t, the node status will change at time $t + \delta$, and at time $t + \delta'$ a decision has to be made to continue moving in the same direction or stop. Let $v = \text{edg}(u, d)$ be the destination node and $e = (u, v)$ the edge that is used for movement. Later we also need $u' = \text{edg}(u, d + 2)$[1], which is the node before u and $v' = \text{edg}(v, d)$, which is the node after v. It might be easier to look at an example on Figure 20.

[1] $d + 2$ means the opposite direction from d

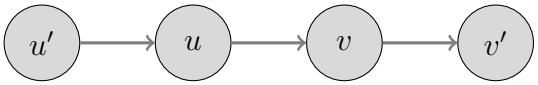


Figure 20: Illustration for the meanings of u', u, v, v'. The arrows show the direction d.

Also we need the set of node statuses where the moving component w could come from.

$$\text{uWhat} = \{w_m \mid w_m \in \mathcal{W}_{\text{move}} \wedge \text{getMC}(w_m) = w\} \tag{81}$$

When $e \notin \mathcal{E}$ we skip the constraints, because we do not have movement variables for such a combination of starting vertex u and direction d. By skipping the constraints we mean that we do not add the remainder of the constraints in this section for current values of (u, w, d, t) and we immediately go to next combination of (u, w, d, t). Also when $t + \delta' \notin \mathcal{T}$ we disable the start of movement, because it cannot be finished.

$$\text{if } t + \delta' \notin \mathcal{T}\text{:} \qquad \text{go}_{u,w,d,t} = 0 \tag{82}$$

In addition to disabling go, we skip the rest of the constraints.

As with other decisions the first group of constraints make sure that node status is correct at the time of the decision. In this case it has to be one of the possible node statuses in uWhat.

$$\text{go}_{u,w,d,t} - \sum_{w_u \in \text{uWhat}} \text{nstat}_{u,w_u,t} \leq 0 \tag{83}$$

If we have the edge $(u', u) \in \mathcal{E}$, that means that an already moving object could continue moving from u' and is in other ways equivalent to a moving object, that starts accelerating from u. So we define an abbreviation.

$$\text{goOrCont} = \begin{cases} \text{go}_{u,w,d,t} & \text{if } (u', u) \notin \mathcal{E} \\ \text{go}_{u,w,d,t} + \text{cont}_{u',w,d,t} & \text{if } (u', u) \in \mathcal{E} \end{cases}$$

We also give a constraint to make sure that these things do not happen at the same time.

$$\text{goOrCont} \leq 1 \tag{84}$$

When there is movement, it has to be continued or stopped.

$$\text{goOrCont} - \text{stop}_{u,w,d,t+\delta'} - \text{cont}_{u,w,d,t+\delta'} = 0 \tag{85}$$

Now again we have to check if movement can actually be completed. Earlier we checked if $t + \delta' \notin \mathcal{T}$ and skipped the rest of the constraints. However now we check for $t + \delta \not\in \mathcal{T}$. When that happens, we again disable the movement like in constraint (82).

$$\text{if } t + \delta' \notin \mathcal{T}: \qquad \text{go}_{u,w,d,t} = 0 \tag{86}$$

And skip the rest of the constraints. At first this seems redundant, but we need to do it like this, because otherwise constraint (85) would not exist for cont and stop variables at the last timestep and no constraint would stop them from obtaining value 1.

For the duration of movement the edge e must be occupied and the node status should remain same. However from w, we do not know the actual status of u, therefore we have to go through all possibilities.

$$\forall i \in \{0, \ldots, \delta'\}: \ \text{goOrCont} - \text{occu}_{e,t+i} \leq 0 \tag{87}$$

$$\forall i \in \{1, \ldots, \delta'\}: \forall w_u \in \text{uWhat}: \ \text{goOrCont} + \text{nstat}_{u,w_u,t} - \text{nstat}_{u,w_u,t+i} \leq 1 \tag{88}$$

There is a simple check: if edge $(v, v') \notin \mathcal{E}$, we can disable continue at u, because after reaching v the robot cannot continue to v'. Also when t is so small, that a previous continue or go decision could not be made, we disable both the stop and continue at time t.

$$\text{if } (v, v') \notin \mathcal{E}: \qquad \text{cont}_{u,w,d,t+\delta'} = 0 \tag{89}$$

$$\text{if } t < \delta': \qquad \text{cont}_{u,w,d,t} = 0 \tag{90}$$

$$\text{if } t < \delta': \qquad \text{stop}_{u,w,d,t} = 0 \tag{91}$$

Specifying node statuses for u and v after the movement would be simpler, if things could not move simultaneously side-by-side as on Figure 12. However robots can move side-by-side, at the same direction. For that reason we need to define additional set of node statuses and corresponding variables. When the robot moves away from u, another thing can at the same time come from u' into u. For that purpose, we define the set of statuses, that can move behind w as $\mathcal{B}$. At the same time another moving object can disappear from v to v'. Let $\mathcal{A}$ denote the set of things that can move ahead of w.

$$\mathcal{B} = \begin{cases} \mathcal{W}_{\text{mc}} & \text{if } w = \mathbf{r} \\ \mathcal{W}_{\text{drop}} & \text{if } w \neq \mathbf{r} \end{cases} \qquad \mathcal{A} = \begin{cases} \{\mathbf{r}\} & \text{if } w = \mathbf{r} \\ \mathcal{W}_{\text{mc}} & \text{if } w \neq \mathbf{r} \end{cases} \tag{92}$$

Now we construct the set of variables, which imply that some additional object is coming into u. Also now is the time to use the abbreviation cos, as a reminder it

was defined as $\text{cos}_{u,w,d,t} = \text{cont}_{u,w,d,t} + \text{stop}_{u,w,d,t}$.

$$\text{uMore} = \begin{cases} 0 & \text{if } (u', u) \notin \mathcal{E} \\ \sum_{w_{u'} \in \mathcal{B}} \text{cos}_{u',w_{u'},d,t+\delta'} & \text{if } (u', u) \in \mathcal{E} \end{cases} \tag{93}$$

Now we can give the general constraint, which says something about u status at the end of movement. When w moves away from u, its status should be the beginning status minus the w, or something else must have come into u. To get the beginning status without w we can use the helper function remMC, which removes a moving component from node status.

$$\text{cos}_{u,w,d,t+\delta'} - \sum_{w_u \in \text{uWhat}} \text{nstat}_{u,\text{remMC}(w_u),t+\delta} - \text{uMore} \leq 0 \tag{94}$$

The last constraint was a general one, that did not exactly specify the status of u after movement. To actually specify the status of u after movement, we need to go over all possible $w_u \in \text{uWhat}$. When u status is about to change by continuing or stopping, and the u status was w_u, the new status should be $\text{remMC}(w_u)$[2] or something else moved at the same time into u.

$$\begin{aligned} \forall w_u \in \text{uWhat}\colon \quad \text{cos}_{u,w,d,t+\delta'} - \text{uMore} + \text{nstat}_{u,w_u,t+\delta'} \\ - \text{nstat}_{u,\text{remMC}(w_u),t+\delta'} \leq 1 \end{aligned} \tag{95}$$

Only when there the edge $(u\,u)$ exists in $\mathcal{E}$, the following group of constraints is added. These make sure that if $w_b \in \mathcal{B}$ moved from u' to u the new status of u will be what would be left in u plus the thing that moved in.

$$\begin{aligned} \forall w_u \in \text{uWhat}\colon \forall w_b \in \mathcal{B}\colon \quad \text{cont}_{u',w_b,d,t+\delta'} + \text{stop}_{u',w_b,d,t+\delta'} \\ + \text{nstat}_{u,w_u,t+\delta'} - \text{nstat}_{u,\text{addMC}(\text{remMC}(w_u),w_b),t+\delta} \leq 1 \end{aligned} \tag{96}$$

There are some subtle details hidden here: when the thing left in u is a car and a robot carrying another car comes. In that case the function addMC is undefined, and the constraint (96) for that combination of w_u and w_b is not added.

Now we want to say something about v status at the end of movement. For that we use the set of statuses that are possible for v after w has moved into it. It turns out that the set is exactly uWhat, which contained all the possible node status from where movement could originate from. Movement implies that at the end v has one of statuses in uWhat.

$$\text{cos}_{u,w,d,t+\delta'} - \sum_{w_v \in \text{uWhat}} \text{nstat}_{v,w_v,t+\delta} \leq 0 \tag{97}$$

[2] w_u without the moveable component

Similarly to uMore we construct the set of variables, which imply that something is moving away from v.

$$\text{vLess} = \begin{cases} 0 & \text{if } (v, v') \notin \mathcal{E} \\ \sum_{w_v \in \mathcal{A}} \cos_{v,w_v,d,t+\delta'} & \text{if } (v, v') \in \mathcal{E} \end{cases} \tag{98}$$

Now we go over all possible status that v could have at the end of the movement and specify the actual status. If the end status of v is w_v then status of v was $\text{remMC}(w_v)$ or something moved away from v.

$$\begin{aligned} \forall w_v \in \text{uWhat}: \quad & \cos_{u,w,d,t+\delta'} - \text{nstat}_{v,\text{remMC}(w_v),t+\delta'} \\ & + \text{nstat}_{v,w_v,t+\delta} - \text{vLess} \leq 1 \end{aligned} \tag{99}$$

2.3.5 Node status constraints

Purpose of node status constraints is to make sure that node statuses remain unchanged when no movement occurs. As with moving constraints, we declare that the remainder of the section is applied to all possible combinations of $u \in \mathcal{V}$, $w \in \mathcal{W}$ and $t \in \mathcal{T}_1$.

The basic idea is to list all variables that could change the status of u between timesteps t and $t+1$. We define a lot of sets and then finally use the union of them. First are the lift and drop variables for the current timestep t.

$$\mathcal{A}_{\text{lift}} = \begin{cases} \{\text{lift}_{u,w,t}\} & \text{if } w \in \mathcal{W}_{\text{lift}} \\ \emptyset & \text{if } w \notin \mathcal{W}_{\text{lift}} \end{cases} \quad \mathcal{A}_{\text{drop}} = \begin{cases} \{\text{drop}_{u,w,t}\} & \text{if } w \in \mathcal{W}_{\text{drop}} \\ \emptyset & \text{if } w \notin \mathcal{W}_{\text{drop}} \end{cases} \tag{100}$$

Next are the lift and drop variables that happened in the past.

$$\mathcal{B}_{\text{lift}} = \begin{cases} \{\text{lift}_{u,w_l,t-5} \mid w_l \in \mathcal{W}_{\text{lift}}\} & \text{if } w = \textbf{lft} \wedge t \in \mathcal{T}_{-5} \\ \emptyset & \text{otherwise} \end{cases} \tag{101}$$

$$\mathcal{B}_{\text{drop}} = \begin{cases} \{\text{drop}_{u,w_d,t-1} \mid w_d \in \mathcal{W}_{\text{drop}}\} & \text{if } w = \textbf{drp} \wedge t \in \mathcal{T}_{-1} \\ \emptyset & \text{otherwise} \end{cases} \tag{102}$$

Now there are the movement variables. Only continue and stop variables actually correspond to immediate node status changes.

$$\mathcal{A}_{\text{move}} = \begin{cases} \{\cos_{u,\text{getMC}(w),d,t} \mid d \in \mathcal{D} \wedge (u, \text{edg}(u,d)) \in \mathcal{E}\} & \text{if } w \in \mathcal{W}_{\text{move}} \\ \emptyset & \text{otherwise} \end{cases} \tag{103}$$

We use $\mathcal{B}_{\text{move}}$ to denote all the movements that could come into u. The expression $\text{addMC}(w, w_m) = \emptyset$ holds, when w_m could not be added to w.

$$\begin{aligned} \mathcal{B}_{\text{move}} = \{\cos_{\text{edg}(u,d),w_m,d+2,t} \mid & d \in \mathcal{D}, w_m \in \mathcal{W}_{\text{move}} \\ & \wedge \text{addMC}(w, w_m) \neq \emptyset \wedge (\text{edg}(u,d), u) \in \mathcal{E}\} \end{aligned} \tag{104}$$

Now that we have defined all the subpart we join them together.

$$\mathcal{B} = \mathcal{B}_{\text{lift}} \cup \mathcal{B}_{\text{drop}} \cup \mathcal{B}_{\text{move}} \tag{105}$$

$$\mathcal{A} = \mathcal{A}_{\text{lift}} \cup \mathcal{A}_{\text{drop}} \cup \mathcal{A}_{\text{move}} \tag{106}$$

Now we can finally give the constraints. The following constraints say that only one action from the set of node status changing actions can happen at once. There are two sets, because with simultaneous movement, it can happen that one action is for moving away and another for moving into u.

$$\sum_{b \in \mathcal{B}} b \leq 1 \tag{107}$$

$$\sum_{a \in \mathcal{A}} a \leq 1 \tag{108}$$

When node status for u is w, the node status has to remain same or some kind of action must have changed it.

$$\text{nstat}_{u,w,t} - \text{nstat}_{u,w,t+1} - \sum_{b \in \mathcal{B}} b - \sum_{a \in \mathcal{A}} a \leq 0 \tag{109}$$

If node status in the future is w and something came into the node, another thing must have left the node u.

$$\text{nstat}_{u,w,t+1} + \sum_{b \in \mathcal{B}} b - \sum_{a \in \mathcal{A}} a \leq 1 \tag{110}$$

2.3.6 Edge occupied constraints

These constrains are to make sure, that when no movement is using an edge the edge occupied variable will be 0. Constraints in this section are applied for all combinations of $e = (u, v) \in \mathcal{E}$ and $t \in \mathcal{T}$. Let $d = \text{dir}(e)$ denote the direction of edge e. First we collect all variables which imply that edge is occupied.

$$\mathcal{A} = \{\text{cos}_{u,w,d,t+\delta'} \mid w \in \mathcal{W}_{\text{mc}}, \delta' \in \{0, \ldots, \text{getMT}(d, w) - 1\} \wedge t + \delta' \in T\} \tag{111}$$

At most two of movement variables in $\mathcal{A}$ can be 1. The reasoning is that at time t an object moves away from u but at the end of the movement $(t + \delta')$ another object which entered u at time t could also leave it. This happens only when two robots are simultaneously moving side-by-side and continuing to move.

$$\mathcal{A} \leq 2 \tag{112}$$

We would want an equality constraint $\text{occu}_{e,t} - \sum_{a \in \mathcal{A}} a = 0$ which would tell us that, edge is occupied when movement implied it, and movement occurs, when

edge is occupied. However, $\text{occu}_{e,t}$ is a binary variable, but $\mathcal{A}$ is not and can also obtain value 2, therefore we need to simulate the equality constraint with two inequality constraints.

$$\text{occu}_{e,t} - \sum_{a \in \mathcal{A}} a \leq 0 \tag{113}$$

$$2\text{occu}_{e,t} - \sum_{a \in \mathcal{A}} a \geq 0 \tag{114}$$

2.4 Initial status and objective

Initial status will be fixed by adding constraints. Let the initial status of node v be denoted by $\bar{v}$, then we can add constraints to fix node status variables at the first timestep.

$$\forall v \in \mathcal{V}: \qquad \text{nstat}_{v,\bar{v},0} = 1 \tag{115}$$

Let $\hat{v}$ denote the end status of node v. There are different ways to make the model find a solution. One way is to add constraints for the last timestep.

$$\forall v \in \mathcal{V}: \qquad \text{nstat}_{v,\hat{v},t_{\max}} = 1 \tag{116}$$

We still have not defined the objective function to the model. We can define the objective function in a way that we do not need the constraints for the end status.

$$\min \sum_{v \in \mathcal{V}, t \in \mathcal{T}} -\text{nstat}_{v,\hat{v},t} \tag{117}$$

This objective means that for every node v, we want the status be $\hat{v}$ for as long as possible or in other words, as soon as possible.

Other possible objective functions can be considered. Maybe it does not make sense to add all node statuses to objective. If we ignore the empty nodes, we can decrease the number of variables in the objective.

$$\min \sum_{v \in \mathcal{V}, t \in \mathcal{T}, \hat{v} \neq \mathbf{e}} -\text{nstat}_{v,\hat{v},t} \tag{118}$$

However this is objective does not actually minimize the time needed to move everything to final positions. A robot might go to its destination instead of first moving the car, because that might get a better objective. An example can be found at Appendix A.

Sometimes the optimal solution consist of one robot waiting for other to move out of way. Instead of standing still, the waiting robot moves back and forth. This

is undesired behaviour of the model, because it wastes energy. Therefore another possible objective function would be to use the end status constraints and then minimize energy used.

$$\begin{aligned}\min \sum_{e=(u,v)\in\mathcal{E},w\in\mathcal{W}_{\text{mc}},t\in\mathcal{T}} & C_{1,w}\text{go}_{u,w,\text{dir}(e),t} + C_{2,w}\text{cont}_{u,w,\text{dir}(e),t} + C_{3,w}\text{stop}_{u,w,\text{dir}(e),t} \\ + \sum_{v\in\mathcal{V},w\in\mathcal{W}_{\text{lift}},t\in\mathcal{T}} & D_{1,w}\text{lift}_{v,w,t} + \sum_{v\in\mathcal{V},w\in\mathcal{W}_{\text{drop}},t\in\mathcal{T}} D_{2,w}\text{drop}_{v,w,t}\end{aligned} \tag{119}$$

Here C is $(3\times|\mathcal{W}_{\text{mc}}|)$-dimensional matrix that contains the amount of energy used by the movement. For lifting and dropping we have a similar matrix D. That way we can give different costs to different to every action and even to every labeled car. This would give optimal solutions energy-wise, but we are more concerned about time.

Therefore, a combination of both (119) and (118) might be better. After testing such a combination, it turned out that the problem of robot moving to destination before doing any work was still the case, however it removed the problem of robots doing unnecessary actions instead of waiting.

The next objective function was to minimize energy, but actions at later timesteps would cost more. And also give a penalty if at $t_{\max}$, the node status is different from terminal state $\hat{v}$. Let $g(t) = 1 + \frac{t}{t_{\max}}$.

$$\begin{aligned}\min \sum_{(u,v)\in\mathcal{E},w\in\mathcal{W}_{\text{mc}},t\in\mathcal{T},d=\text{dir}((u,v))} & (0.2g(t)\cdot\text{go}_{u,w,d,t} + 0.1g(t)\cdot\text{cont}_{u,w,d,t} \\ & +0.2g(t)\cdot\text{stop}_{u,w,d,t}) + \sum_{v\in\mathcal{V},w\in\mathcal{W}_{\text{lift}},t\in\mathcal{T}} 0.4g(t)\cdot\text{lift}_{v,w,t} \\ & + \sum_{v\in\mathcal{V},w\in\mathcal{W}_{\text{drop}},t\in\mathcal{T}} 0.1g(t)\cdot\text{drop}_{v,w,t} \\ & + \sum_{v\in\mathcal{V},w\in\mathcal{W},w\neq\hat{v}} 10\cdot\text{nstat}_{v,w,t_{\max}}\end{aligned} \tag{120}$$

In the end we used objective function (120) with a small modification to only give penalties to those node statuses, where the terminal status contains a labeled car.

It should also be mentioned that the model performance varied greatly with different objective functions. And also on the fact, if terminal configuration is set with constraints or not.

2.5 Size of the model

It is customary to give the total number of variables, equalities and inequalities for a integer programming model. In our case these numbers are dependant of the exact graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of different labels for cars K, and the number of timesteps $t_{\max}$. Because $|\mathcal{T}| = t_{\max} + 1$, we use $|\mathcal{T}|$ in further calculations. We use abbreviations $T = |\mathcal{T}|, V = |\mathcal{V}|$ and $E = |\mathcal{E}|$.

We first give the sizes of $\mathcal{W}, \mathcal{W}_{\text{lift}}, \mathcal{W}_{\text{drop}}, \mathcal{W}_{\text{move}}$ and $\mathcal{W}_{\text{mc}}$.

$$|\mathcal{W}| = 4 + 3K \tag{121}$$
$$|\mathcal{W}_{\text{lift}}| = K \tag{122}$$
$$|\mathcal{W}_{\text{drop}}| = K \tag{123}$$
$$|\mathcal{W}_{\text{move}}| = 1 + 2K \tag{124}$$
$$|\mathcal{W}_{\text{mc}}| = 1 + K \tag{125}$$

2.5.1 Number of variables

We start with node status variables.

$$VWT = V(4 + 3K)T = TV(3K + 4) \tag{126}$$

The number of edge variables is ET. The number of decision variables is:

$$\begin{aligned} &3E\mathcal{W}_{\text{mc}}\mathcal{T} + V|\mathcal{W}_{\text{lift}}|T + V|\mathcal{W}_{\text{drop}}|T \\ &\quad = T(3E|\mathcal{W}_{\text{mc}}| + V(|\mathcal{W}_{\text{lift}}| + |\mathcal{W}_{\text{drop}}|)) \\ &\quad = T(3E(1 + K) + 2VK) \end{aligned} \tag{127}$$

Total number of variables is:

$$\begin{aligned} &TV(3K + 4) + ET + T(3E(1 + K) + 2VK) \\ &\quad = T(V(3K + 4) + E + 3E(1 + K) + 2VK) \\ &\quad = T(5VK + 3EK + 4V + 4E) \end{aligned} \tag{128}$$

2.5.2 Number of equalities

We have only two groups of real equality constraints. The others are meant to fix a value of a variable, and therefore we do not count them here. Although those variables will be fixed by equality constraints, We still counted them in subsubsection 2.5.1. Total number of equality constraints is $TV + TE(1 + K) = T(EK + E + V)$.

2.5.3 Number of inequalities

There are lots of inequalities so we will not give an exact number, but an upper bound. From simple constraint and lifting constraints the number of inequalities is:

$$\begin{aligned} TE + TV + 4TV + TV + TVK + 5TVK + TVK + TVK + TVK + TVK \\ = 10TVK + 6TV + TE \end{aligned} \tag{129}$$

From moving constraints we get the following upper bound on number of inequalities:

$$\begin{aligned} TE(1+K)(1+1+5+5K+1+K+K^2+1+K) \\ = TE(1+K)(K^2+7K+9) \end{aligned} \tag{130}$$

For node status constraints the upper bound on inequalities is:

$$TV(4+3K)(1+1+1+1) = 12TVK + 16TV \tag{131}$$

For edge occupied constraint the upper bound is: $TE(1+1+1) = 3TE$.

In total the upper bound of inequalities is:

$$\begin{aligned} &10TVK + 6TV + TE + TE(1+K)(K^2+7K+9) + 12TVK + 16TV + 3TE \\ &\quad = 22TVK + 22TV + 4TE + (TE + TEK)(K^2+7K+9) \\ &\quad = 22TVK + 22TV + 4TE + TEK^2 + 7TEK + 9TE + TEK^3 + 7TEK^2 + 9TEK \\ &\quad = TEK^3 + 8TEK^2 + 22TVK + 16TEK + 22TV + 13TE \\ &\quad = T(EK^3 + 8EK^2 + 22VK + 16EK + 22V + 13E) \end{aligned} \tag{132}$$

3 Implementation

For implementing the model, there were at least two major decisions to be made.

- Which optimization software to use?
- What programming language to use for generating the model?

The author had previous experience with a commercial grade optimization solver called Gurobi [5]. It is advertised as a state-of-the-art mathematical programming solver and it has a free, full-featured academic license.

Other capable solvers for integer programming models according to [8, 10] are CPLEX [2] and XPRESS [1]. However, they are also both commercial like Gurobi and Gurobi was to easiest to get an academic licence for.

Gurobi has bindings for several languages. The ones considered were C++ and Python. It is well known that Python as an interpreted language has a constant runtime overhead. However, C++ has lots of unwanted complexity. The author chose to use Python, because generating the integer programming model would only take a fraction of the time to actually solve it. And the model solving is not hindered by the runtime system of Python. Because there exist lots of code that still uses version 2.X of Python, it should be explicitly mentioned that Python 3.X was used in thesis. To be precise, the exact version was Python 3.5.1.

3.1 PANDA

Programmers are used to thinking in terms of if-clauses. However, in an integer programming model there are no ifs. At first it was hard to come up with constraints. We knew which variables were involved and what values were legal, but writing them as a constraint was still unintuitive. Finding the right constraint when there were five or more variables proved hard to do by hand. Fortunately there is a tool called PANDA [7] to help with finding constraints.

PANDA is short for Parallel AdjaceNcy Decomposition Algorithm and its internal algorithm is more or less the same as Fourier-Motzkin elimination algorithm, which is taught at the University of Tartu in the course MTAT.05.120 "Combinatorial Optimization".

PANDA takes an input file, which list the variables and feasible values of the variables. Then it processes this list of feasible values and outputs a complete system of equalities and inequalities that hold between the variables. As an example here is the contents of an valid input file.

```
Names:
occu incoming
```

```
Vertices:
0 0
1 1
1 2
```

And the corresponding output by PANDA is:

```
PANDA -- facet enumeration with double description
Inequalities:
-2occu +incoming <= 0
occu -incoming <= 0
occu <= 1
```

Writing input files for PANDA is a little tedious, because you have to list all the feasible assignments of values to variables. We wrote little programs with regular if-clauses to generate input files for PANDA. PANDA outputs lots of inequalities, but some of them are trivial and already covered by other constraints in the model. Therefore, human judgement was still needed to see which inequalities and equalities added something valuable to the integer programming model.

3.2 Graphical visualization

For testing the model and seeing the solutions found, we needed some kind of feedback. At first we had a textual output of the node status and decision variables for each timestep, it is depicted on Figure 21.

However, when the time came to find out, if edge occupied variables had reasonable values in the solution, the textual output was not good enough to display the directed edges well. Instead of improving the textual output, we chose to implement the visualization of solutions graphically.

Because the visualization was used mostly for debugging purposes, there was not a lot of effort and time put into writing it. To minimize the effort we wrote the graphical visualization using the de-facto standard Python graphical user interface package TkInter [3]. The graphical visualization shows the underlying graph of the problem one timestep at a time. There are keybindings to increase and decrease the visible timestep. For a given timestep, the occupied edges are highlighted. At each vertex, the node status is displayed and if one of the decision variables was set to 1, the decision is also displayed inside the vertex. A screenshot of the visualization graphical interface is on Figure 22.

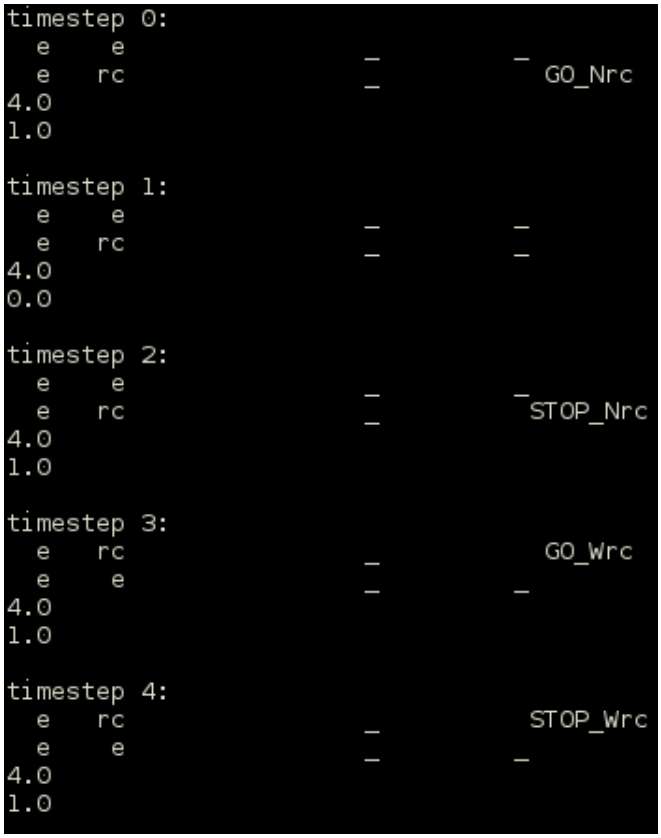


Figure 21: Early textual output of variables. On the left are the node statuses for a 2x2 grid and on the right are the decision variables. Note that the variables were different then.

3.3 Irreducible inconsistent subsystem

In the development phase errors were made. Sometimes a feasible movement were not feasible in the integer programming model. This would mean that all the constraints would need to be carefully verified again. Luckily, Gurobi can calculate an irreducible inconsistent subsystem of an infeasible model. Inconsistent subsystem of an integer program is a subset of constraints, that are inconsistent. Irreducible inconsistent subsystem is a minimal inconsistent subsystem. Basically we can ask Gurobi to find out which set of constraints make the model infeasible. After we get such a set, we only have to recheck the constraints that are in the irreducible inconsistent subsystem.

As an example here is the irreducible inconsistent subsystem given by Gurobi for the orthogonal collision illustration in subsubsection 2.2.4.

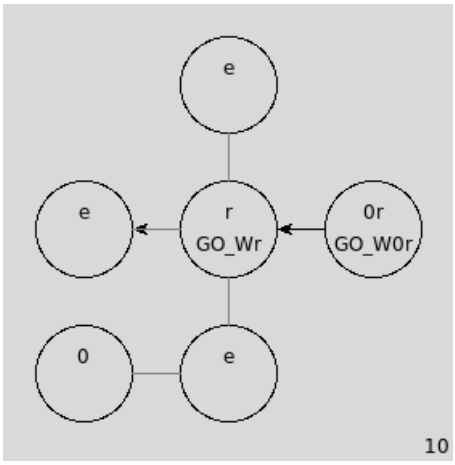


Figure 22: Screenshot of the graphical visualization. Circles are the nodes, gray lines between them show edges. Black arrows show occupied edges. Inside the nodes, the upper text displays the node status and the lower text the current decision. In the lower right corner is a number, which show the current timestep.

```
Minimize

Subject To
 R432: occu(((1,0),(0,0)),0) + occu(((1,0),(2,0)),0)
   + occu(((1,1),(1,0)),0) <= 1
 R5655: - occu(((1,0),(0,0)),0) + go((1,0),'r','W',0)
   + cont((2,0),'r','W',0) <= 0
 R5792: - occu(((1,1),(1,0)),0) + go((1,1),'r','N',0) <= 0
 R6328: cont((2,0),'r','W',0) = 0
 R8494: go((1,0),'r','W',0) = 1
 R8495: go((1,1),'r','N',0) = 1
Bounds
 occu(((1,0),(0,0)),0) free
 occu(((1,1),(1,0)),0) free
 go((1,0),'r','W',0) free
 go((1,1),'r','N',0) free
 cont((2,0),'r','W',0) free
Generals
 occu(((1,0),(0,0)),0) occu(((1,0),(2,0)),0) occu(((1,1),(1,0)),0)
 go((1,0),'r','W',0) go((1,1),'r','N',0) cont((2,0),'r','W',0)
End
```

3.4 Tuning Gurobi parameters

Gurobi might be the best mixed integer program solver around, but as the problems are NP-complete, it still takes a long time to find the optimal solution. Gurobi has a lot of parameters [6] that can be altered, which affect the total runtime. Finding a good set of parameters to tune is a tedious task, therefore Gurobi comes with an automatic tuning tool. The tuning tool has an API, but the easiest way is to run it from command line as a separate tool called *grbtune*.

The tuning tool will take as input integer programming models and then optimize them many times in a row to establish a baseline performance. After the baseline is established it will try different set of parameters and it then outputs those sets that improved on the baseline performance.

Different objective functions leaded to different parameters. When we used objective function (118) with the terminal configuration fixed with constraints (116), *grbtune* reported the following set of parameters.

- MIPFocus = 2
- Presolve = 2
- PrePasses = 3
- Cuts = 0

However, when I switched to objective function (120) and without the terminal configuration constraints, the only parameter that Gurobi tuning tool found was:

- AggFill = 5

Now we briefly explain the settings. In the first case MIPFocus is used to controls the focus of MIP solver. Focus 2 means to work on the best bound. Presolve controls the aggressiveness of pre-solving the model. Value of 2 means that pre-solving is aggressive. PrePasses is used to limit the number of pre-solving passes. Cuts = 0 turns off all cut generations. AggFill controls the fill level in presolve aggregations, with larger values generally lead to more presolved rows and columns, but at the same time leave more non-zeros in the constraint matrix [6].

4 Comparison

At the Algorithms and Theory research group in University of Tartu, another integer programming model was developed with the same problem in mind. The idea was that we come up with different models, and then can later compare them. We will denote the model described in this thesis as TH, and the other model as AT. The AT model used the same simplifications and assumptions described in subsection 1.3. The model has a more variables. With more variables it is possible use simpler constraints, which contain variables only from consecutive timesteps. To be more precise, constraints only use the variables from the current timestep t and from the next timestep $t + 1$. To

The objective function for the AT model is a mix between progressive energy minimization like (120) and minimizing the time when node statuses different. We estimate that there could be some problems similar to the problem depicted in Appendix A. Terminal state is fixed with a large cost, when it differs in the last timestep. The objective function only considers the location of cars and not the robots just like the model described in section 2.

4.1 Testing platform

The testing was done a machine with Intel(R) Core(TM) i7-4790K CPU. The machine has 16GB of operating memory. With hyper-threading the number of virtual CPUs available was 8. However, the solvers were instructed to use only one thread, because several instances were run in parallel.

4.2 Problem instances used for comparison

The underlying grid layout was taken from a real car park. Initial and terminal configurations were randomly generated.

4.3 Results

Both models were run against several problem instances. Each instance was run 3 times by each model. Number of timesteps for each problem instance was made to be small, but hopefully enough. The timelimit was passed down to Gurobi TimeLimit parameter. The time needed to generate the integer programming model from problem instance is not included in that time. The results show mean Gap found and the number of cars not in the terminal configuration at $t_{\max}$. The results are on Table 4.

			AT		TH	
Instance	tlimit	$t_{\max}$	Gap	C	Gap	C
marsi3a-1011	1h	50	64.53	0	96.23	2
marsi3a-1111	1h	70	97.38	2	95.74	3
marsi3a-2111	1h	50	92.70	1	95.83	3
marsi3b-1011	1h	50	94.15	1	94.66	2
marsi3b-1111	1h	50	74.64	0	92.92	1.66
marsi3b-2111	1h	50	68.70	0	95.68	2.66
marsi3c-1011	1.5h	100	96.45	1	95.16	2
marsi3c-1111	1.5h	120	96.01	1	95.16	2
marsi3c-2111	1h	40	43.84	0	91.81	2
marsi3d-1011	2h	150	95.85	1	95.41	2
marsi3d-1111	2h	150	98.45	2	95.61	3
marsi3d-2111	2h	150	97.39	1	95.21	3
marsi3e-1011	2h	150	98.72	2	94.95	2
marsi3e-1111	2h	150	98.18	2	94.76	3
marsi3e-2111	2h	150	97.02	1	95.17	3

Table 4: Performance measurements of two models. The C column displays the number of car that were not delivered to their destinations by $t_{\max}$.

4.4 Interpretation of results

It seems that AT model finds feasible solutions a lot more than TH model. However the Gap between incumbent solution and best bound seems to decrease better with TH model. One observation was that AT model's 3 runs had very similar bounds. For TH model, the 3 runs of the same problem instance had more variations.

We also look at a few lines from the Gurobi log for both models. We start with marsi3c-2111 instance. Gurobi writes the following lines, when solving AT model.

```
Optimize a model with 248610 rows, 70110 columns and 765808 nonzeros
Presolved: 47339 rows, 16854 columns, 199604 nonzeros
Explored 3255 nodes (1267231 simplex iterations) in 3600.01 seconds
```

The same lines for TH model are:

```
Optimize a model with 711882 rows, 64944 columns and 3479938 nonzeros
Presolved: 130560 rows, 38377 columns, 718910 nonzeros
Explored 588 nodes (984243 simplex iterations) in 3600.10 seconds
```

We can see, that AT model has more columns, but after pre-solving AT model has less than half of what is left in the TH model. We can also see that in AT model

3255 MIP nodes were explored, while the TH model only 588 were explored. From here we can conclude that AT model with easier constraints is better handled by Gurobi pre-solver.

The next problem instance is marsi3e-1111. For AT model Gurobi generated the following lines:

```
Optimize a model with 1505623 rows, 406492 columns and 4599392 nonzeros
Presolved: 539301 rows, 191201 columns, 2574442 nonzeros
Explored 282 nodes (322789 simplex iterations) in 7200.32 seconds
```

And the corresponding lines for TH are:

```
Optimize a model with 4351099 rows, 376141 columns and 21428351 nonzeros
Presolved: 1042718 rows, 293077 columns, 5719680 nonzeros
Explored 569 nodes (741744 simplex iterations) in 7207.95 seconds
```

Here we see similar size comparisons between initial model sizes and model sizes after pre-solve. However, in this case it turn out that Gurobi has managed to explore more nodes in the TH model, than in AT model. That might explain, why TH model has a better Gap in this problem instance.

5 Conclusions

Integer programming model for optimizing a specific automated valet parking system was designed. The model was tested with different problem instances. Some instances revealed problems with the objective function used. Several slightly different objective functions were proposed and field tested on problem instances.

Different integer programming models that are used to solve a same problem are known to have wildly different running times that are hard to predict. The performance of the current model was lower than the comparison that it was benchmarked against. One plausible way to try to improve it would be to come up with a novel way to encode various states and constraints needed and still represent the same automated valet parking problem.

To really solve the full automated valet parking problem, we need to deal with real-time requirements, which currently are infeasible for the integer programming models compared here. We believe that heuristic algorithms might be suitable for the task. Integer programming model could be used to evaluate the quality of heuristic solutions.

There is a lot of future work in this field, because existing solutions are not well suited for handling car parks with custom layouts and existing systems are too slow during peak periods.

References

[1] FICO Xpress Optimization Suite. http://www.fico.com/en/products/fico-xpress-optimization-suite, Last 2016.

[2] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, Last 2016.

[3] Tkinter. https://wiki.python.org/moin/TkInter, Last 2016.

[4] Gruia Calinescu, Adrian Dumitrescu, and János Pach. Reconfigurations in graphs and grids. *SIAM Journal on Discrete Mathematics*, 22(1):124–138, 2008.

[5] Inc. Gurobi Optimization. Gurobi optimizer reference manual. http://www.gurobi.com, 2016.

[6] Inc. Gurobi Optimization. Gurobi optimizer reference manual - parameters. https://www.gurobi.com/documentation/6.5/refman/parameters.html, 2016.

[7] Stefan Lörwald and Gerhard Reinelt. PANDA: a software for polyhedral transformations. *EURO Journal on Computational Optimization*, pages 1–12, 2015.

[8] Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, 2012.

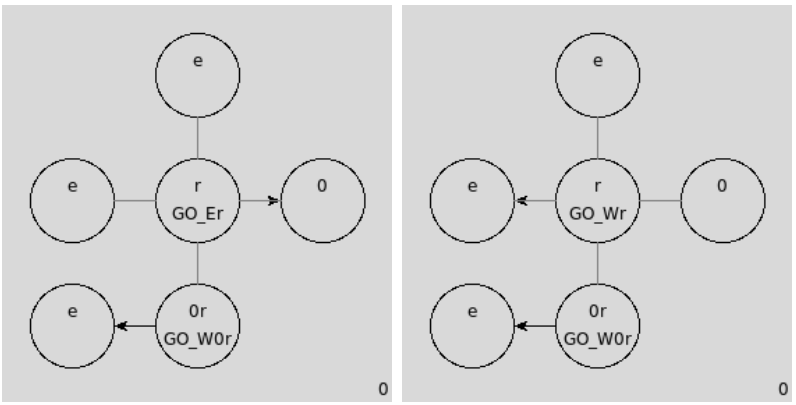
[9] George A Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

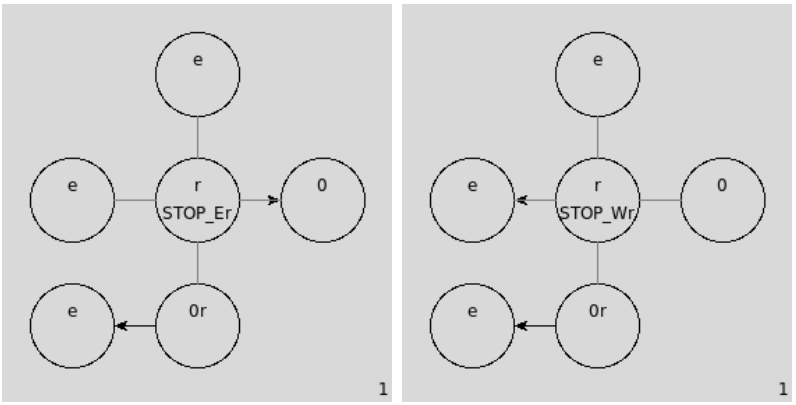
[10] Hans D. Mittlemann. The State-Of-The-Art in Optimization Software. http://plato.asu.edu/talks/ismp2015.pdf, July 2015. Talk given at *The International Symposium on Optimization*.

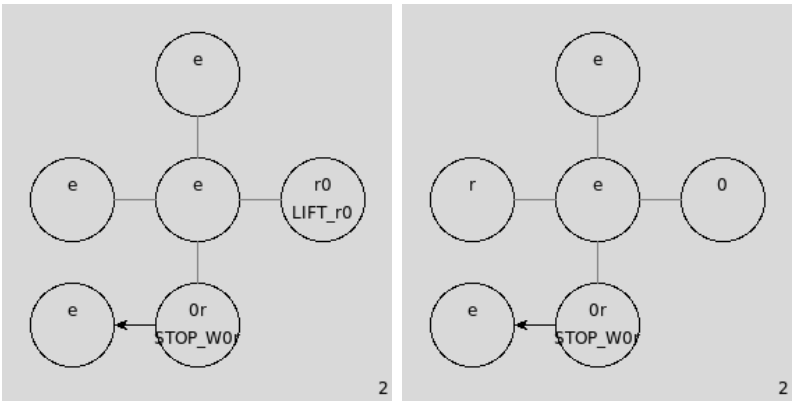
[11] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9, 2008.

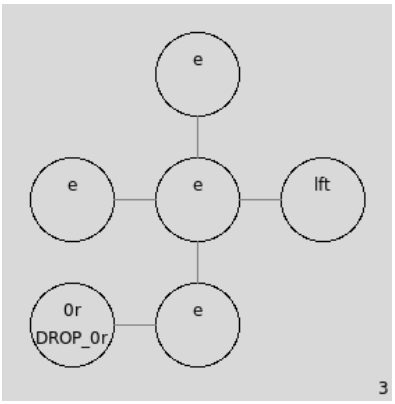
A Example of movement

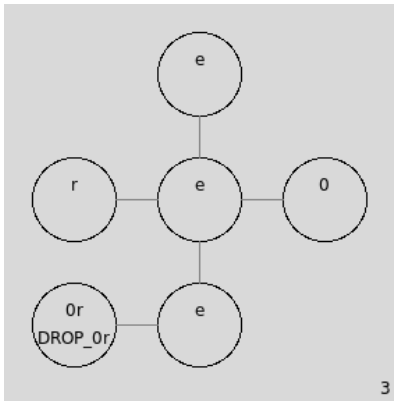
This is an example run of one problem. On the left is the correct solution, and the right is the solution obtained by using objective function (117). As a side note: the objective function used on the left, did not ignore the final position of robots. Otherwise a better solution can be found.

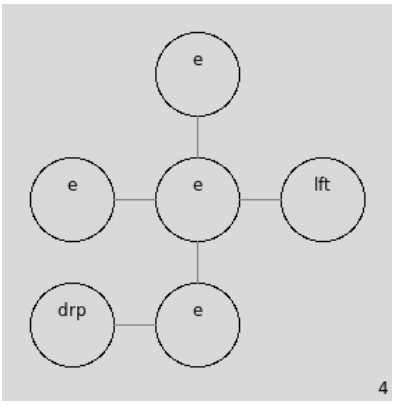


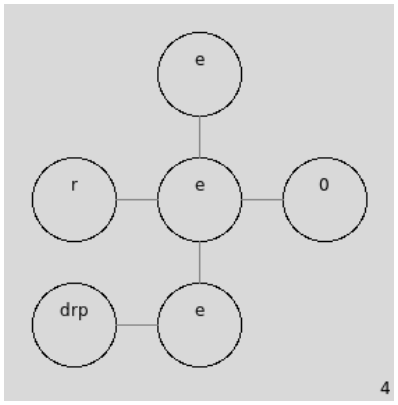


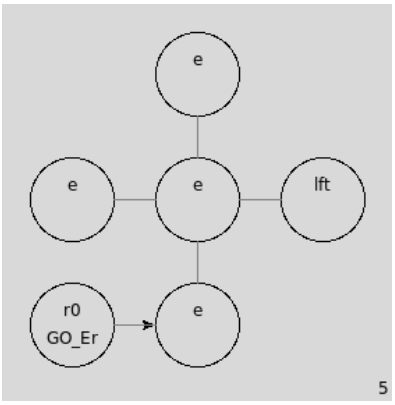


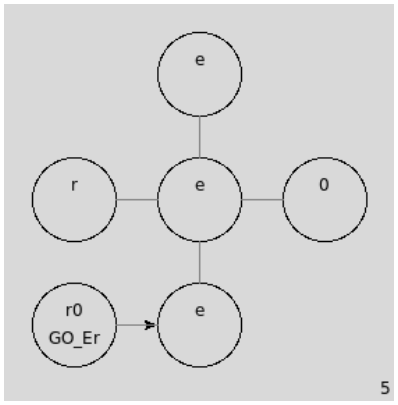

e
e
e
lft
0r
DROP_0r
e
3

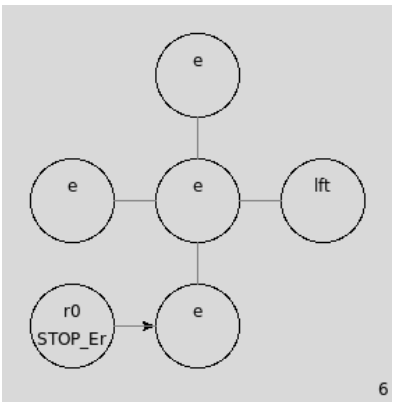

e
r
e
0
0r
DROP_0r
e
3

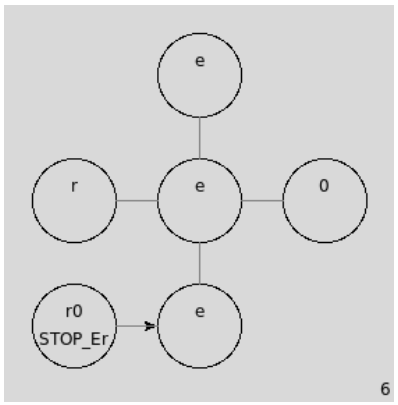

e
e
e
lft
drp
e
4

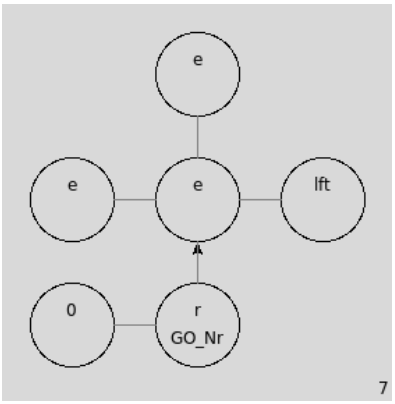

e
r
e
0
drp
e
4

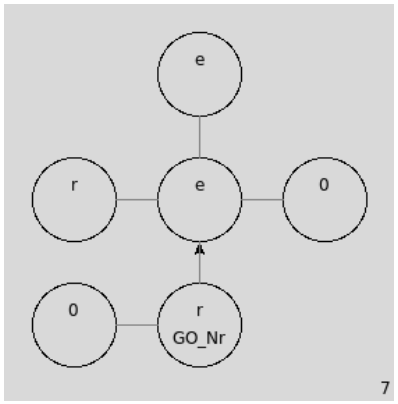

e
e
e
lft
r0
GO_Er
e
5

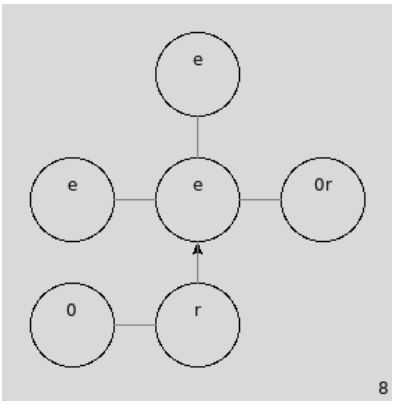

e
r
e
0
r0
GO_Er
e
5

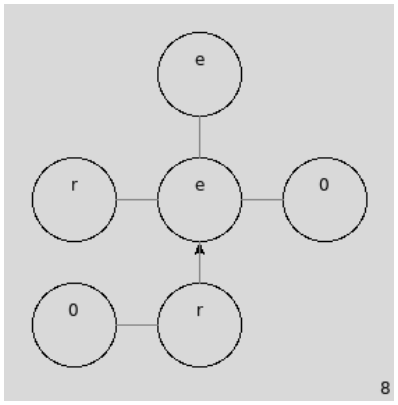

e
e
e
lft
r0
STOP_Er
e
6

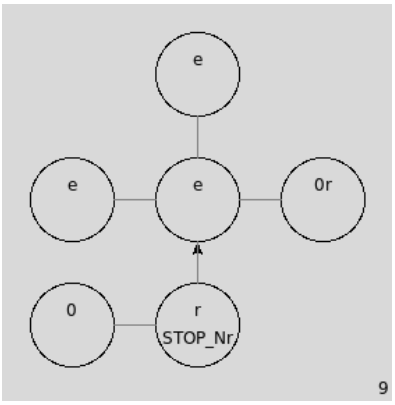

e
r
e
0
r0
STOP_Er
e
6

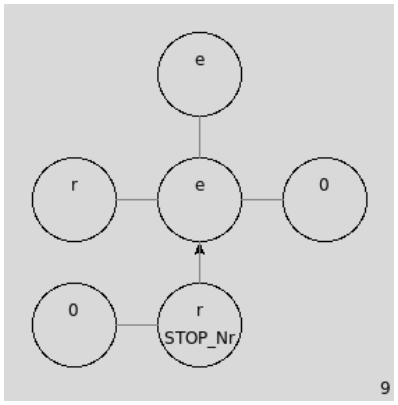

e
e
e
lft
0
r
GO_Nr
7

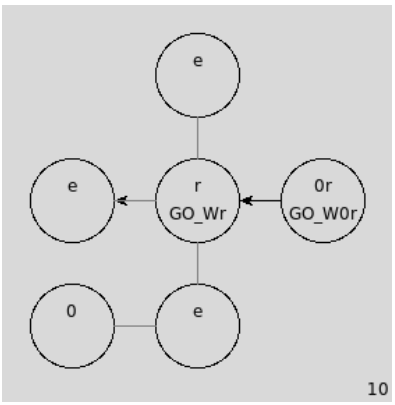

e
r
e
0
0
r
GO_Nr
7

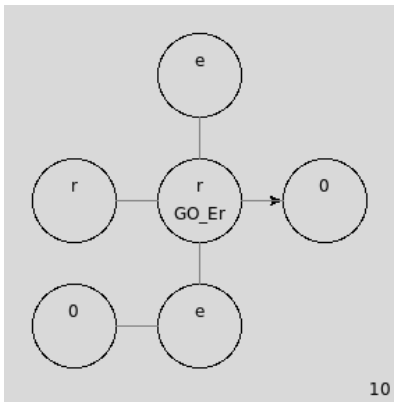

e
e
e
0r
0
r
8

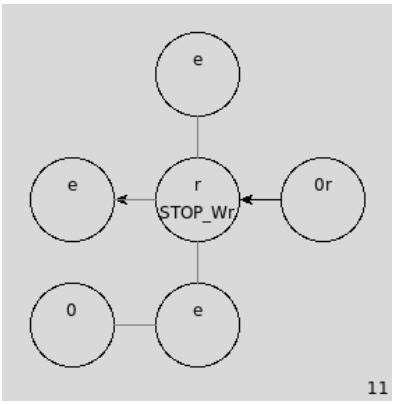

e
r
e
0
0
r
8

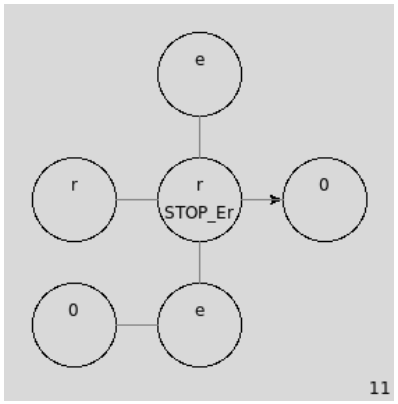

e
e
e
0r
0
r
STOP_Nr
9

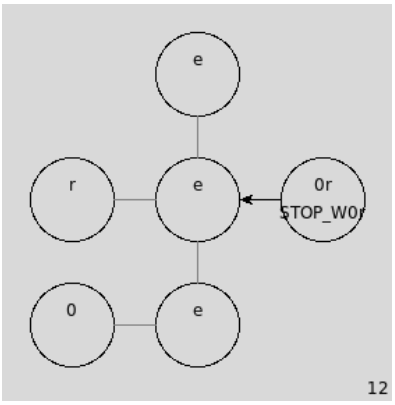

e
r
e
0
0
r
STOP_Nr
9

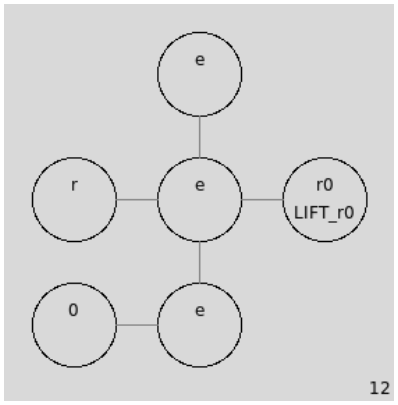

e
e
r
GO_Wr
0r
GO_W0r
0
e
10

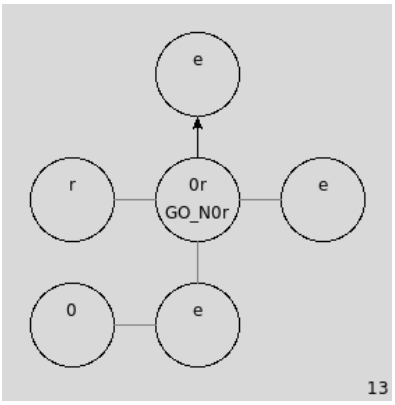

e
r
r
GO_Er
0
0
e
10

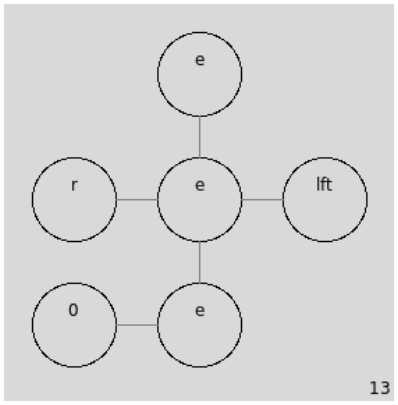

e
e
r
STOP_Wr
0r
0
e
11

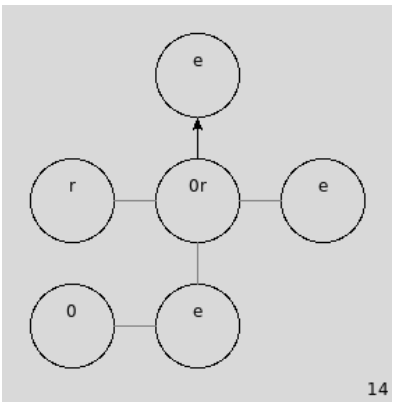

e
r
r
STOP_Er
0
0
e
11

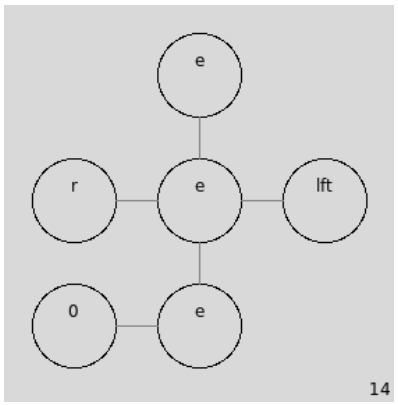

e
r
e
0r
STOP_W0r
0
e
12

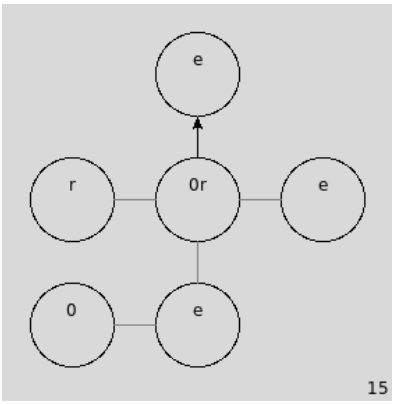

e
r
e
r0
LIFT_r0
0
e
12

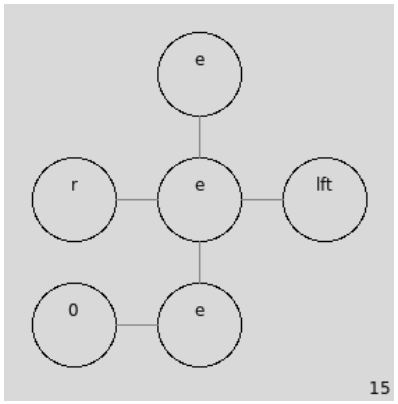

e
r
0r
GO_N0r
e
0
e
13

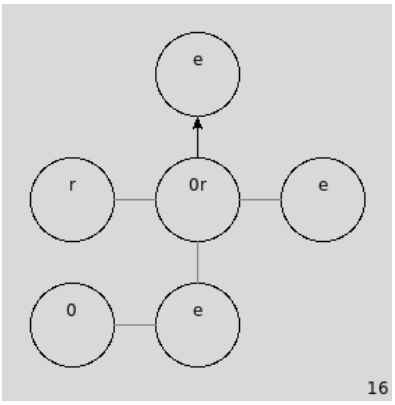

e
r
e
lft
0
e
13

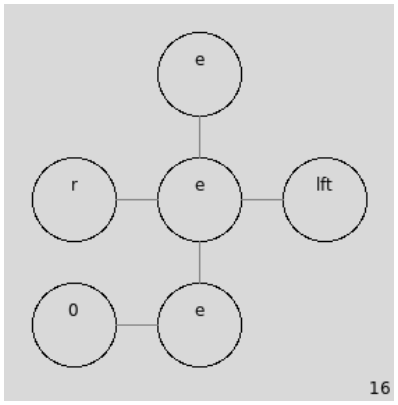

e
r
0r
e
0
e
14

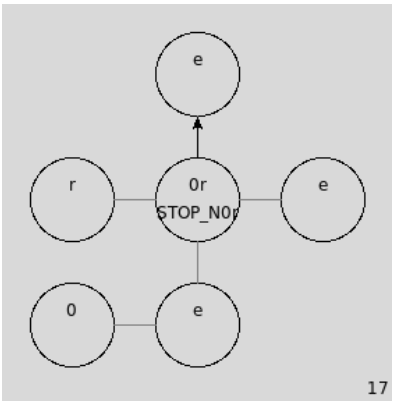

e
r
e
lft
0
e
14

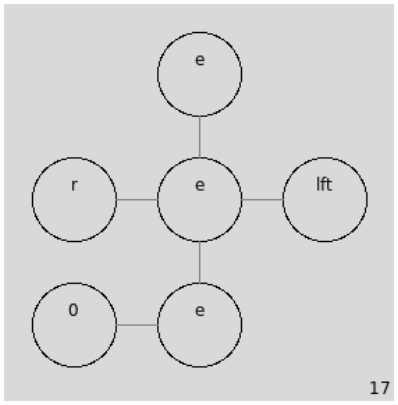

e
r
0r
e
0
e
15

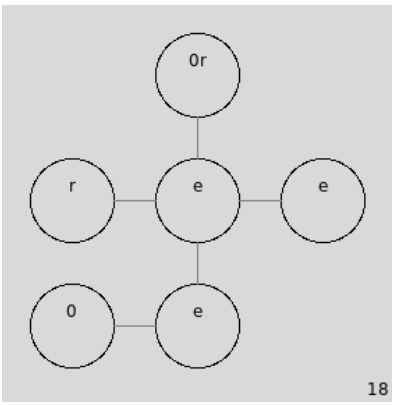

e
r
e
lft
0
e
15

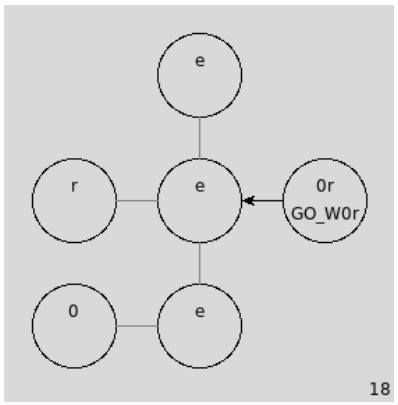

e
r
0r
e
0
e
16

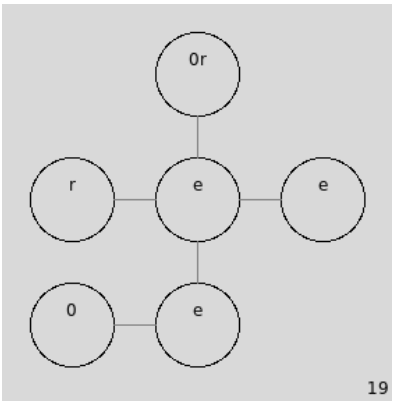

e
r
e
lft
0
e
16

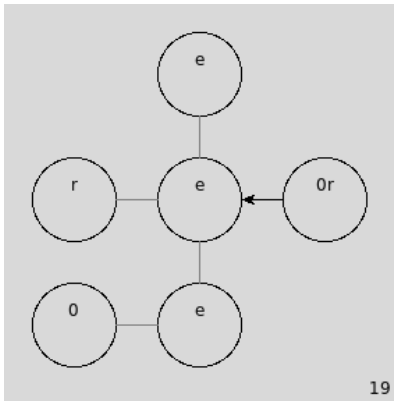

e
r
0r
STOP_N0r
e
0
e
17

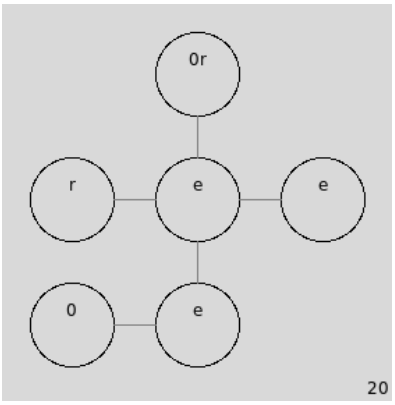

e
r
e
lft
0
e
17

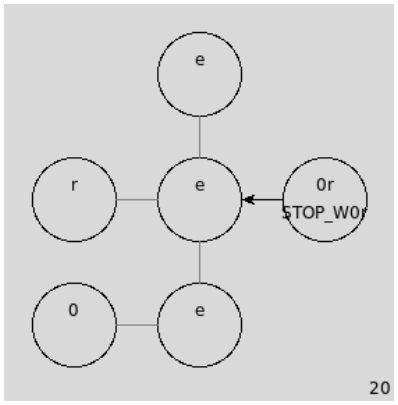

0r
r
e
e
0
e
18

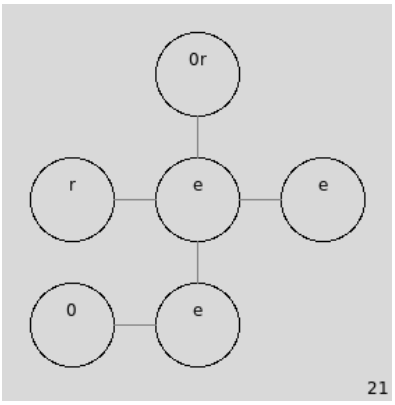

e
r
e
0r
GO_W0r
0
e
18

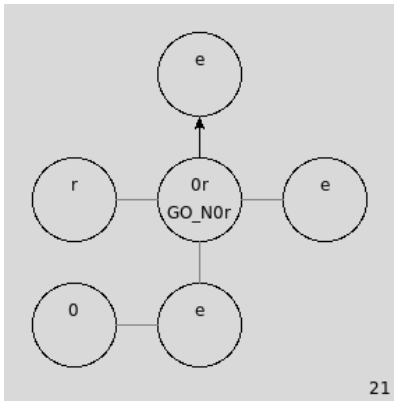

0r
r
e
e
0
e
19

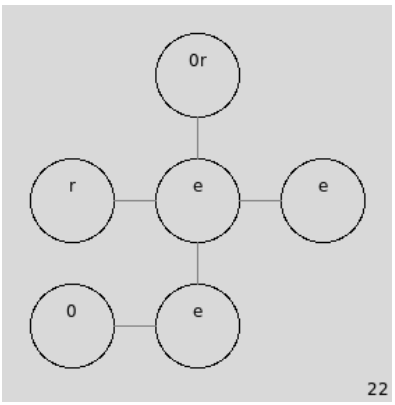

e
r
e
0r
0
e
19

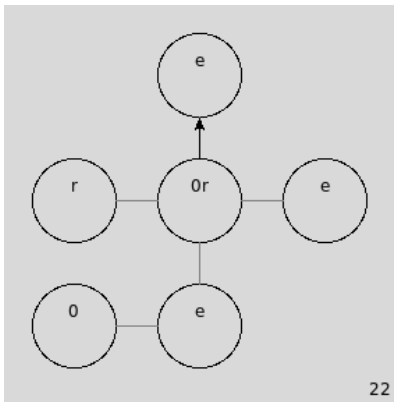

0r
r
e
e
0
e
20

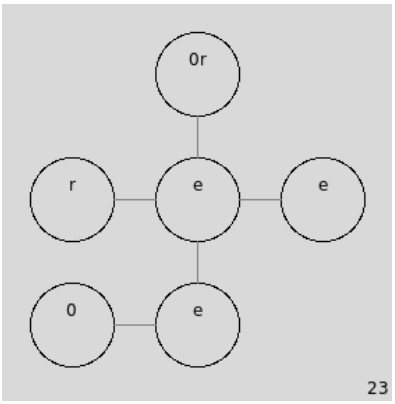

e
r
e
0r
STOP_W0r
0
e
20

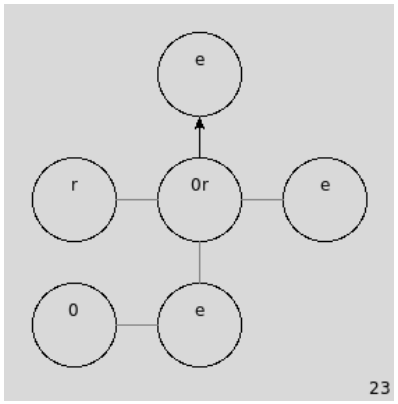

0r
r
e
e
0
e
21

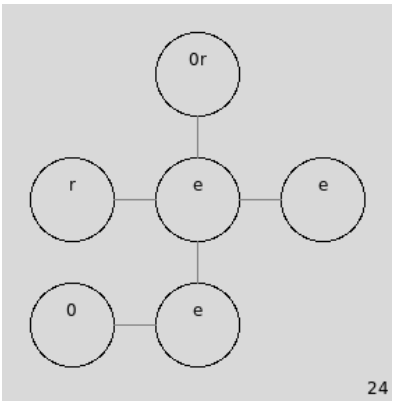

e
r
0r
GO_N0r
e
0
e
21

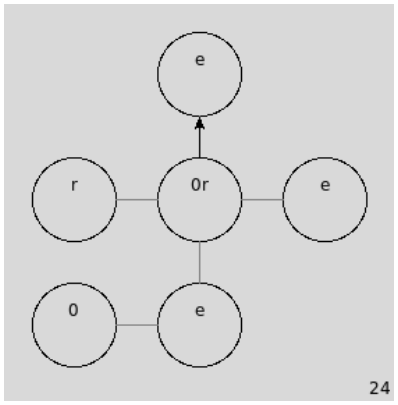

0r
r
e
e
0
e
22

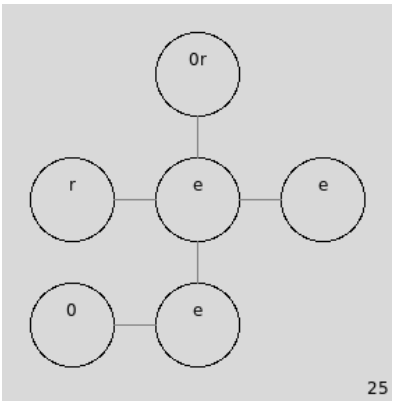

e
r
0r
e
0
e
22

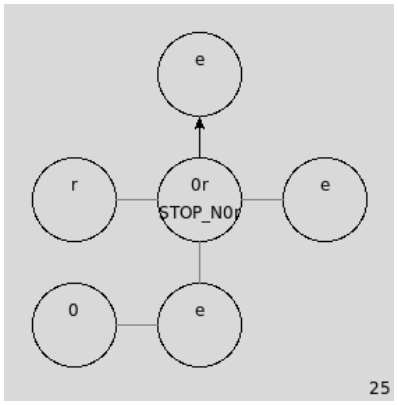

0r
r
e
e
0
e
23

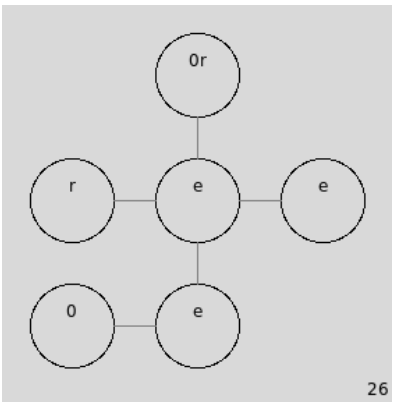

e
r
0r
e
0
e
23

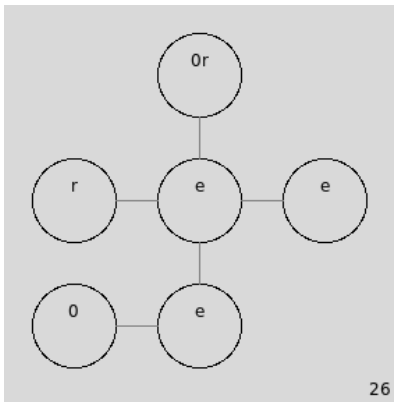

0r
r
e
e
0
e
24


e
r
0r
e
0
e
24


0r
r
e
e
0
e
25


e
r
0r
STOP_N0r
e
0
e
25


0r
r
e
e
0
e
26


0r
r
e
e
0
e
26

B Code created for the integer programming model

The code is available on github: `https://github.com/karulont/parking`.

Non-exclusive licence to reproduce thesis and make thesis public

I, Karl Tarbe (date of birth: 20th of February 1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Integer programming model for automated valet parking

supervised by Dr. Dirk Oliver Theis

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 19.05.2016