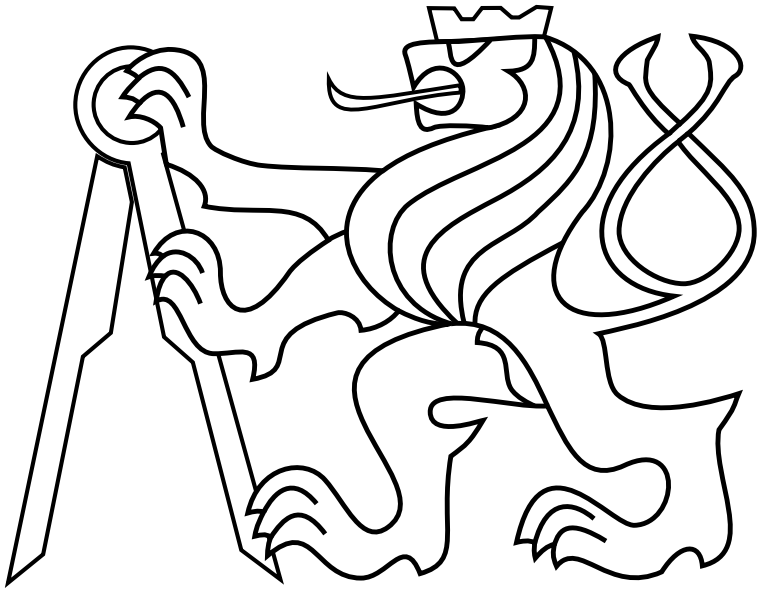
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

MASTER'S THESIS



Michal Němec

Planning for team of robots in cooperative wall building task

Department of Control Engineering

Thesis supervisor: **Ing. Robert Pěnička**

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

.....
Podpis autora práce

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Němec Michal** Personal ID number: **457202**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

Planning for team of robots in cooperative wall building task

Master's thesis title in Czech:

Plánování pro tým robotů v úloze kooperativní stavby zdi

Guidelines:

1. Get familiar with orienteering problem and algorithms used for cooperative task planning.
2. Formulate a variant of the orienteering problem for wall building task as integer linear programming.
3. Implement heuristic solution approach for planning the wall building.
4. Evaluate the proposed method and compare it with optimal solution.

Bibliography / sources:

[1] Roozbeh, Iman, Melih Özlen and John W. Hearne. "A heuristic scheme for the Cooperative Team Orienteering Problem with Time Windows." ArXiv abs/1608.05485 (2016).
[2] Pieter Vansteenwegen, Wouter Souffriau, Dirk Van Oudheusden. "The orienteering problem: A survey." European Journal of Operational Research, Volume 209, Issue 1, 2011, Pages 1-10.
[3] Aldy Gunawan, Hoong Chuin Lau, Pieter Vansteenwegen. "Orienteering Problem: A survey of recent variants, solution approaches and applications." European Journal of Operational Research, Volume 255, Issue 2, 2016, Pages 315-332.

Name and workplace of master's thesis supervisor:

Ing. Robert Pěnička, Multi-robot Systems, FEE

Name and workplace of second master's thesis supervisor or consultant:

Ing. Martin Saska, Dr. rer. nat., Multi-robot Systems, FEE

Date of master's thesis assignment: **16.01.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until:
by the end of summer semester 2020/2021

Ing. Robert Pěnička	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisor Ing. Robert Pěnička for his help and guidance over the writing of this thesis. I would also like to thank my family, partner and friends who always supported me.

Abstract

This thesis deals with the Wall Building Problem (WBP) by a group of robots. Problem is motivated by Mohamed Bin Zayed International Robotic Challenge 2020 competition, where one of the challenges was to build a wall using a group of robots while collecting maximum reward withing a specified time budget. The problem is first formalized as a variation of the Orienteering Problem (OP), where the main goal is to collect maximum reward given some problem space. Physically building the wall naturally introduces precedence constraints for each brick that are not generally present in OP and must also be added to the formulation. A metaheuristic approach as a variation of GRASP method is proposed. The method solves a specified Wall Building Problem as an iterative algorithm. In the later sections, we show how our method can be extended to account for the heterogeneous group of robots. The last sections are reserved for performance evaluation and comparison with optimal solutions obtained by the generic solver.

Keywords: *wall building problem, vehicle routing problem, metaheuristics, greedy randomized adaptive search procedure*

Abstrakt

V této práci je řešen problém stavění zdi (WBP) pomocí skupiny robotů. Práce je inspirovaná jednou ze soutěží Mohamed Bin Zayed International Robotic Challenge 2020. Cílem soutěže bylo postavit zeď pomocí skupiny robotů a přitom dosátnout největšího skóre v zadaném časovém limitu. Problém je nejprve definován jako úloha lineárního programování motivovaném tzv. Orienteering problémem (OP). OP má za cíl maximalizovat zisk na předem definované trati, po které se mohou roboti pohybovat. Při stavění zdi se jednotlivé cihly musí pokládat na sebe. Tato podmínka neni v OP běžně obsažena a metodu je nutné o ni rozšířít. Pro řešení problému je použit metaheuristický přístup ve formě iterační metody GRASP. V pozdějších sekcích je naše metoda rozšířena o plánování pro heterogenní skupinu robotů. Na závěr jsou porovnýny výsledky navrhnuté metody s optimálním řešením.

Klíčová slova: *problém stavění zdi, problém směrování vozidel, metaheuristika, greedy randomized adaptive search procedure*

Contents

1 Introduction **1**

1.1 Wall Building Problem 2

1.2 Orienteering problem 2

1.3 Metaheuristic 2

1.3.1 Greedy randomized adaptive search procedure (GRASP) 3

2 Wall Building Optimization Problem **5**

2.1 Problem formulation 5

2.2 Wall layers 8

2.3 Virtual nodes 9

2.4 Time constraint objective function 10

2.5 Geometry representation 12

2.5.1 Generating precedence rules 12

2.5.2 Generating concurrence rules 13

3 Heuristic approach **15**

3.1 Plan construction 17

3.2 Process edges 18

3.3 Process Nodes 18

3.4 Assign available nodes 19

3.5 Time update 21

3.6 Node placement 23

3.7 Optimization procedure 23

3.8 Local Search 25

3.9 Solution update 26

4 Heterogenous robot fleet **27**

5 Performance evaluation **31**
5.1 CPLEX 31
5.2 Datasets 31
5.3 Multithreaded performance 32
5.4 Weakly constraint performance 34
5.5 Highly constrained performance 35
5.6 Plan comparison 37
5.7 Robot utilization limits 39

6 Experiments **40**

7 Conclusion **43**

Appendix A List of abbreviations **47**

Appendix B CD content **47**

List of Figures

2.1 Simple wall 7
2.2 Concurrence and precedence rules 8
2.3 Wall demonstrating example where bricks spans more layers vertically. . . 9
2.4 Graph representation corresponding to Figure 2.3. 9
2.5 Start and end nodes 10
2.6 Plan solution with end time minimization 11
2.7 Plan solution with reward only 11
3.1 Resource state transition diagram. 21
3.2 Time update histogram and cumulative sum. 22
5.1 Concurrence and precedence rules 32
5.2 Multithread model 33
5.3 Solution of small problem 37
5.4 Solution of small problem 37
5.5 Solution of small problem 38
5.6 Solution of small problem 38
6.1 Brick reservoirs placement during the simulation. Wall is built in the center of coordinate system. 41
6.2 Wall used in the simulation 41
6.3 Wall being built by 2 UAVs. 41
6.4 Simulated camera feedback of the first UAV in Figure 6.3. 42
6.5 Simulated camera feedback of the second UAV in Figure 6.3. 42
6.6 Simulation plan 42

List of Algorithms

1 GRASP - general algorithm . . . 3
2 Construction of precedence set. . . . 12
3 Precedence brick comparison. . . . 13
4 Construction of concurrence set. . . . 13
5 Plan construction procedure. (function plan_construct) . . . 17
6 One iteration step of the construction (function iterate_step) . . . 17
7 Process constraints algorithms. (function process_edges) . . . 18
8 Find available nodes. (function find_available_nodes) . . . 18
9 Find available nodes. (function conccurence_rules_met) . . . 19
10 Greedy assignment procedure. (function assign_available_nodes) . . . 19
11 Find resources assignable to the node. (function get_available_resources) . 20
12 Assign given node to resources. (function assign_available_nodes) . . . 20
13 Find minimal time update procedure. (function update_time) . . . 22
14 Finish assigned node. (function place_assigned_nodes) . . . 23
15 GRASP - main algorithm . . . 24
16 GRASP - greedy randomized construction. (function greedy_randomized_construction) . . . 24
17 GRASP - local search. (function local_search) . . . 25
18 GRASP - update solution based on reward. (function update_solution) . . . 26
19 Find all assignable combination of node given available resources. (function get_available_resources) . . . 28
20 Greedy assignment procedure. (function assign_available_nodes) . . . 29
21 Assign given node to resources . . . 29

1 Introduction

Planning has always been one of the extensively researched areas, from task scheduling up to large-scale planning of manufacturing processes. Wall building task can be split into multiple areas, from physical safety precautions when automating the build process using heavy machinery [1], to a more abstract formulation where tasks are formulated decoupled from actual physical actions required to perform them.

Building process can have multiple criteria in which we build the wall, using general robot-human cooperation, or fully autonomous solution where one or group of robots cooperates in building task. This thesis focuses only on abstract planning algorithms using a fully autonomous group of robots where no additional human interaction is required. We want our solution to be in some sense optimal. One of the extensively researched areas is Vehicle Routing Problem (VRP), where the simple question is asked, "What are the optimal paths for each vehicle in a group given some set of customers?". Here we refer to customers as some abstract entity that each vehicle must visit to complete the task.

Since it was first proposed by Dantzig and Ramser [2], a lot of research has been devoted to finding exact/approximate solutions. Many variants of this problem have been explored, such as the Capacitated VRP (CVRP)[3], in which a homogeneous fleet of vehicles is available and the only constraint is the vehicle capacity. When fixed visitation time interval is required, VRP with Time Windows (VRPTW)[4] has been developed. A special case in which certain group members are required to collect the reward from each customer cooperatively is Cooperative Orienteering Problem (COP). The problem is extended by conditions where customers must be served within a specified time window (COPTW).

Real-life applications of VRP can become large such that exact methods [5] can not be used to obtain an optimal solution. Since the vehicle routing problem specified as linear programming optimization is NP-hard [6]. Exact algorithms can only be used when we are solving small instances of the problem. Many heuristic algorithms were developed [7, 8] to solve such problems. The problem becomes more complex once we introduce some problem-specific constraints that are not generally included in a formulation. Namely in wall building problem, we are forced to introduce precedence rules that ensure correct order in which bricks are built. The plan must ensure that bricks are placed on top of each other. Precedence constraints also arise when one action or multiple actions must finish before the start of another. Some applications VRP include the dial-a-ride problem (DARP) [9], airline scheduling [10], bus routing, or tractor semi-trailer problem (TSRP) [11]. The emphasis has been made on metaheuristics [12], which are methods used to find feasible solutions quickly. One of the widely known algorithms used is Clark and Wrights saving method [13], which in original or modified forms can be used to obtain optimal/sub-optimal solutions. In case of more complex constraints methods using genetic algorithms [14], [15] has also been explored.

1.1 Wall Building Problem

This work is motivated by Mohamed Bin Zayed International Robotic Challenge (MBZIRC) 2020 [1], where the main goal is to build a wall using the cooperation of terrestrial and unmanned aerial vehicles(UAVs). Competition has three challenges, from which second one is challenge of building wall using group of robots. The main objective is to maximize total score by building a wall from different types of brick. Wall is built vertically using the homogeneous group of robots, namely using three unmanned aerial vehicles (UAVs) and one unmanned ground vehicle (UGV). There are four different types of bricks(red, green, blue, orange) with different rewards for placing each of them. Each type of bricks has different dimensions, and the biggest one must be carried using two UAVs at the same time. Since it is competition, there is a time limit in which we need to place as many bricks as possible. Based on the competition rules we propose solution in a form of Orienteering problem.

1.2 Orienteering problem

The orienteering problem (OP) [16] is a routing problem, where we seek to find a path through specified nodes such that we collect maximum reward associated with each node. Requesting cooperation of multiple robots with collecting reward leads to the extension of OP to Cooperative Orienteering Problem (COP) [16]. If reward at each node can be collected only within a specified time window, we are talking about the Cooperative Orienteering Problem with time windows (COPTW)[17]. All variations of OP can be defined for a homogeneous or heterogeneous group of robots. The objective function for OP is commonly defined as a sum of rewards at each collected node using integer only optimization variables. WBP is formulated as mixed-integer linear programming optimization problem that is also NP-hard [18, 19]. Actual solutions are visualized using simple Gantt chart[20], showing associated actions of each robot. The reward is received for each successfully placed brick. Since it was a competition wall composed of bricks with different dimensions and requirements on the cooperation of the robot fleet. Proposed algorithm must be flexible to be able to incorporate various placement constraints that arise from the cooperation of UAVs, such as collision avoidance requirements.

1.3 Metaheuristic

Metaheuristic methods do not guarantee solutions to the optimization problems, but they are used to find improved solutions. They are relatively easy to implement and can find feasible solutions in a short time compared to the exact solution given by generic linear programming solver.

[1] MBZIRC 2020 `https://www.mbzirc.com/challenge/2020`

There are widely known algorithms for solving VRP, such as the Nearest neighbor algorithm or Clarke-Wright algorithm (CW). One of the popular metaheuristic widely used is Tabu search. First proposed by Fred Glover in 1986 [21] to improve local search by accepting non-improving moves within the neighborhood. Tabu search is a metaheuristic that aims to proceed from a local optimum by allowing non-improving moves. A few years later in 1989, GRASP was first introduced in [22] as an efficient probabilistic set covering heuristic.

There are variations of the classical algorithm, such as the Reactive GRASP. In this variation, the primary parameter that defines the restrictiveness of the Restricted Candidate List, see Section 1.3.1, during the construction phase is self-adjusted according to the quality of the solutions previously found. There are also techniques for search speed-up, such as cost perturbations, bias functions, memorization and learning, and local search on partially constructed solutions.

1.3.1 Greedy randomized adaptive search procedure (GRASP)

This thesis presents the GRASP meta-heuristic approach [23] applied to WBP. The GRASP can be generally defined as seen in 1. Algorithms use pseudorandom number generators (PRNG) with initial seed that are used throughout the procedure.

Algorithm 1: GRASP - general algorithm

Result:

```
1 while stopping criterion not met do
2 |   restart random number generator with different seed;
3 |   greedy_solution = greedy_randomized_construction();
4 |   refined_solution = local_search(greedy_solution);
5 |   update_solution(best_solution, refined_solution);
```

Output: best_solution

When PRNG is initialized, iteration starts with greedy randomized construction of a feasible solution. Generally, the solution is generated by incrementally adding elements from a Restricted Candidate List (RCL). RCL elements are generated based on problem space and generally represents elements that incrementally improve a partially built solution. Elements are then selected at random to form a feasible solution. Local search continues with improving the constructed solution. A simple implementation would consist of a refining solution based on comparison with the close neighborhood of the solution. The refined solution is compared with the actual best solution found. The metric on which solutions are compared also comes from the problem specification.

In our application algorithm starts from the initial configuration of all robots and state of all bricks (either placed, not placed, or carried by the robot). The greedy heuristic is used to find an initial feasible solution during which we save the state of a partial solution that is used in local search later. We re-optimize saved states with a random set of next possible permutations. All feasible solutions are compared by the optimization criterion,

and the best one is selected. The algorithm iteratively runs until the stopping criterion is met.

2 Wall Building Optimization Problem

In this section, we describe how wall building problem can be formalized in terms of mixed-integer linear programming. From the competition perspective, we choose to explore Orienteering problem variation. Wall building can be imagined as an orienteering problem where we try to maximize the maximum number of placed bricks with maximum reward associated with each brick. The main difference from COP is additional precedence and concurrence constraints.

2.1 Problem formulation

To fully describe the optimization problem, we introduce number of variables required, as seen in Table 1.

Table 1: Overview of basic variables used in problem definition.

Name	Description
x_{ij}^r	bool if used edge between nodes v_i and v_j for robot r
z_{ij}	bool if edge between nodes v_i and v_j is used
y_i	wether node i was visited
s_i	visit time of node v_i
p_i	reward of node v_i
t_i	duration of node v_i
β^r	start node of robot r
R	number robots
V_f	all non-termitationg nodes
e_{ij}	edge between v_i and v_j
v_N	terminationg node
ρ_i	number of robots required in node v_i
M	max time $M = \mathrm{T}_{\max} + \max t_i + \max \lvert e_{ij} \rvert$
N	number of nodes $N = \lvert V \rvert$
$\mathbf{V} = \{v_i\}$	the set of all nodes
$\mathbf{E} = \{e_{ij}\}$	the set of all edges between nodes v_i and v_j
$\mathbf{V_f}$	the set of non-terminating nodes

Optimization problem is then formulated as:

$$\text{maximize} \sum_{v_i \in V} p_i y_i, \tag{2.1}$$

$$s.t. \sum_{v_i \in V} x^r_{\beta^r i} = 1\,,\ \forall r \in (1, \ldots, R)\,, \tag{2.2}$$

$$\sum_{r \in (1,\ldots,R)} \sum_{v_i \in V} x^r_{iN} = R\,, \tag{2.3}$$

$$s_{\beta^r} = 0\,,\ \forall r \in (1, \ldots, R)\,, \tag{2.4}$$

$$y_{\beta^r} = 1\,,\ \forall r \in (1, \ldots, R)\,, \tag{2.5}$$

$$y_N = 1\,, \tag{2.6}$$

$$\sum_{v_i \in V \setminus \{v_c\}} x^r_{ic} = \sum_{v_i \in V \setminus \{v_c\}} x^r_{ci}\,,\ \forall v_c \in V_f\,,\ \forall r \in (1, \ldots, R)\,, \tag{2.7}$$

$$\sum_{r \in (1,\ldots,R)} \sum_{v_i \in V} x^r_{ci} = \rho_c y_c\,,\ \forall v_c \in V\,, \tag{2.8}$$

$$\sum_{r \in (1,\ldots,R)} x^r_{ij} \leq R z_{ij}\,,\ \forall e_{ij} \in E\,, \tag{2.9}$$

$$s_i + t_i + |e_{ij}| \leq s_j + M(1 - z_{ij})\,,\ \forall e_{ij} \in E\,, \tag{2.10}$$

where (2.1) is objective function that maximizes collected reward. Objective function is then subjected to multiple constraints. Constraint (2.2) ensures start of each robot in designated start position. Constraint (2.3) ensures that all nodes return to terminating node. Constraint (2.4) ensures start nodes to have initial time set to zero. Constraint (2.5) ensures that starting nodes are already visited. Constraint (2.6) marks end node to be visited. Constraint (2.7) conserves flow between edges, forcing incoming and outgoing edges to be visited. Constraint (2.8) introduces cooperative properties. Constraint (2.9) together with (2.8) ensures that collected reward at each node is dependent on resource requirement, and traveling group members through an edge is bounded by maximum number of robots. Final classical OP constraint (2.10) ensures start time continuity between used edges. Sufficient constant M is chosen to be $M = \mathrm{T}_{\max} + \max t_i + \max |e_{ij}|$. WBP introduces precedence constraints set $\mathbf{\Pi} = \{\pi_i \,|\, \pi_i = (v_b, v_a)\}$, with node before v_b and node after v_a. Generation of the precedence set is discussed in Section 2.5.1. Constraints

$$y_b \geq y_a\,,\ \forall \pi_i \in \Pi\,, \tag{2.11}$$

$$s_a \geq s_b + y_b t_b\,,\ \forall \pi_i \in \Pi\,, \tag{2.12}$$

enforce order in which bricks are placed. Equation 2.11 ensures that node v_a can be placed only after v_b is placed.

Equation 2.11 ensures starting time s_a of placing node v_a starts only after node v_b is placed at time $s_b + t_b$. Concurrence constraints restricts placement of multiple bricks that are close to each other.

$$s_b \geq s_a + y_a t_a \vee s_a \geq s_b + y_b t_b \,, \;\; \forall \gamma_i \in \Gamma \,, \tag{2.13}$$

where $\mathbf{\Gamma} = \{\gamma_i \,|\, \gamma_i = (v_a, v_b)\}$ is the set of concurrence rules, with nodes v_b and v_a that can not be built simultaneously.

Wall building is optimized on some time interval $\langle 0, T_{max} \rangle$. Constraint is generally defined from OP as:

$$0 \leq s_i \leq T_{max} \,, \;\; \forall v_i \in V \,. \tag{2.14}$$

Constraint visual representation

We can visualize defined optimization problem using a simple two-layer wall, as shown in Figure 2.1.

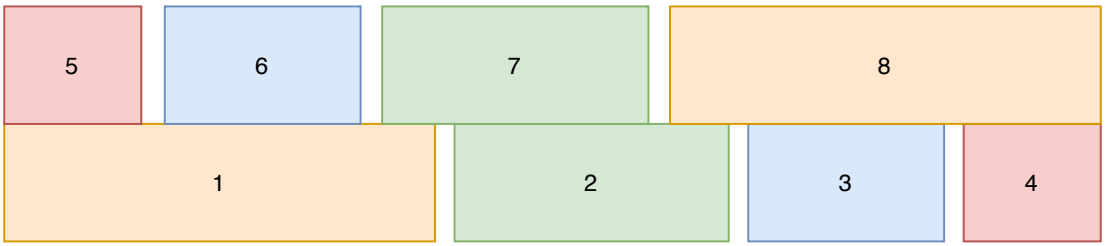


Figure 2.1: Simple wall used for demonstration of the problem, and used for visualization reference.

Each brick represents node $v_i \in V$. Edges e_{ij} are omitted from visualization because they would form a fully connected graph. Precedence and concurrence rules extracted from the wall layout are seen in Figure 2.2.

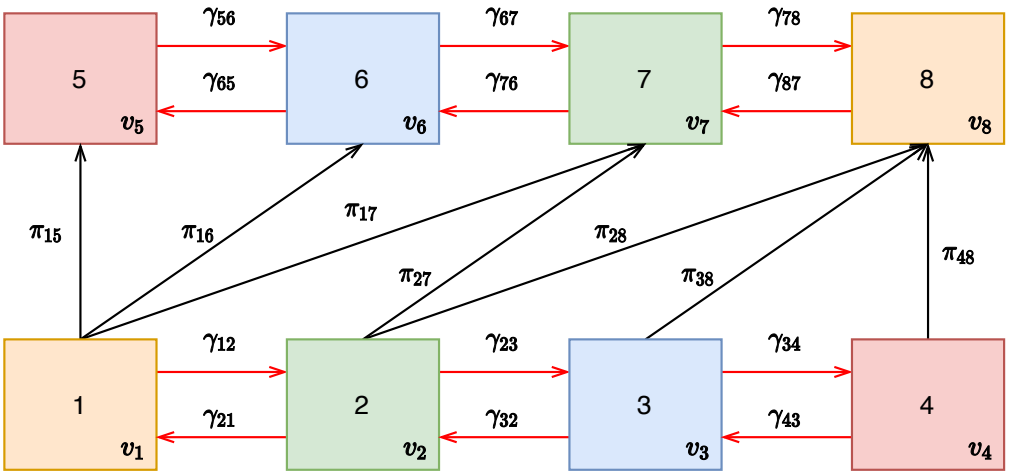


Figure 2.2: Concurrence (red) and precedence (black) rules visualization generated from Figure 2.1. Precedence rules π_{ij} correspond to $\pi_k \in \{\pi_{15}, \pi_{16}, ...\}$. Concurrence rules γ_{ij} correspond to $\gamma_k \in \{\gamma_{12}, \gamma_{21}, ...\}$.

For clarity, we marked them with two indexes as π_{ij} and γ_{ij}, but they correspond to single elements π_k and γ_k as seen in the definition of MILP.

2.2 Wall layers

We define layers of bricks as set of bricks that are located only next to each other as there are no physical restrictions for them to be built simultaneously. From the visual representation of the wall in Figure 2.1, we can intuitively see two layers, where the first layer is composed of bricks $\{1, 2, 3, 4\}$ and second layers $\{5, 6, 7, 8\}$.

In the further section, we discuss weakly and highly constrained WBP for which we define useful boolean variable as:

$$L_{first} = \begin{cases} \text{true}, & \text{bricks layers are fully connected by precedence rules}\,, \\ \text{false}, & \text{bricks layers are connected only by their position}\,. \end{cases} \tag{2.15}$$

Example of $L_{first} = false$ has been demonstrated in Figure 2.2. In general there might not exists straightforward distinction between layers as shown in Figure 2.3 and 2.4.

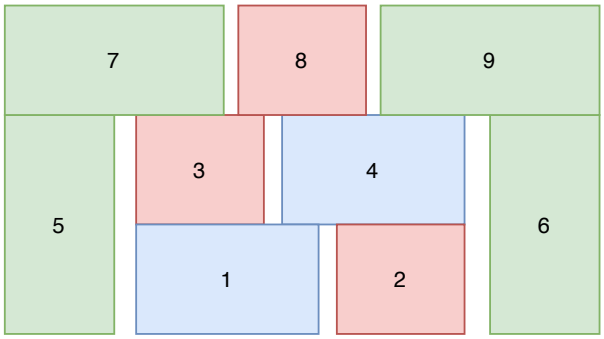


Figure 2.3: Wall demonstrating example where bricks spans more layers vertically.

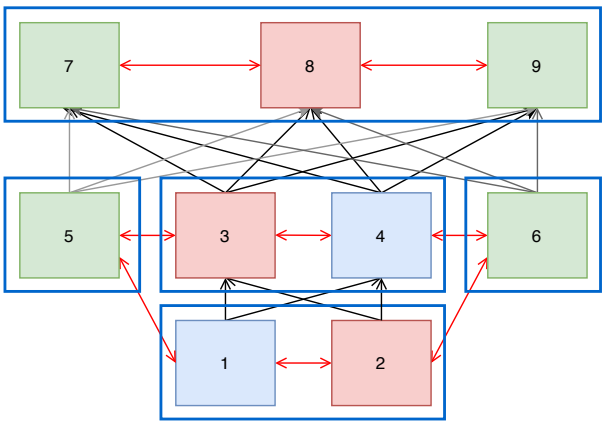


Figure 2.4: Graph representation corresponding to Figure 2.3.

Wall shown in Figure 2.3 has total of 5 layers, shown in blue boxes, 3 layers containing multiple bricks $\{1, 2\}$, $\{3, 4\}$, $\{7, 8, 9\}$ and remaining 2 layers with only contain one brick each, $\{5\}$ and $\{6\}$. Layers that contains only one brick must be separated due to the concurrence rules that are presented between two layers in the middle. We could possibly merge layers $\{5\}$, $\{6\}$, and $\{1, 2\}$ into one layer, but that would introduce more restriction on the planning algorithm. Black edges seen in Figure 2.4 corresponds to precedence rules. Red edges correspond to concurrence rules. The precedence graph is created using $L_{first} = true$, and concurrence created by taking into account only bricks directly next to each other side by side.

2.3 Virtual nodes

For implementation purposes, we introduce virtual nodes into the problem specification. When implementing MILP we need to specify the starting node of each robot, as shown in Figure 2.5. Virtual nodes are added to the problem in order to specify the starting point of each robot. Edges shown as dotted lines are not precedence nor concurrent. From MILP formulation, they would be contained inside.

In a case where robots are in the starting position, we need to create additional nodes apart from nodes representing bricks. Starting nodes are then connected with a set of starting bricks.

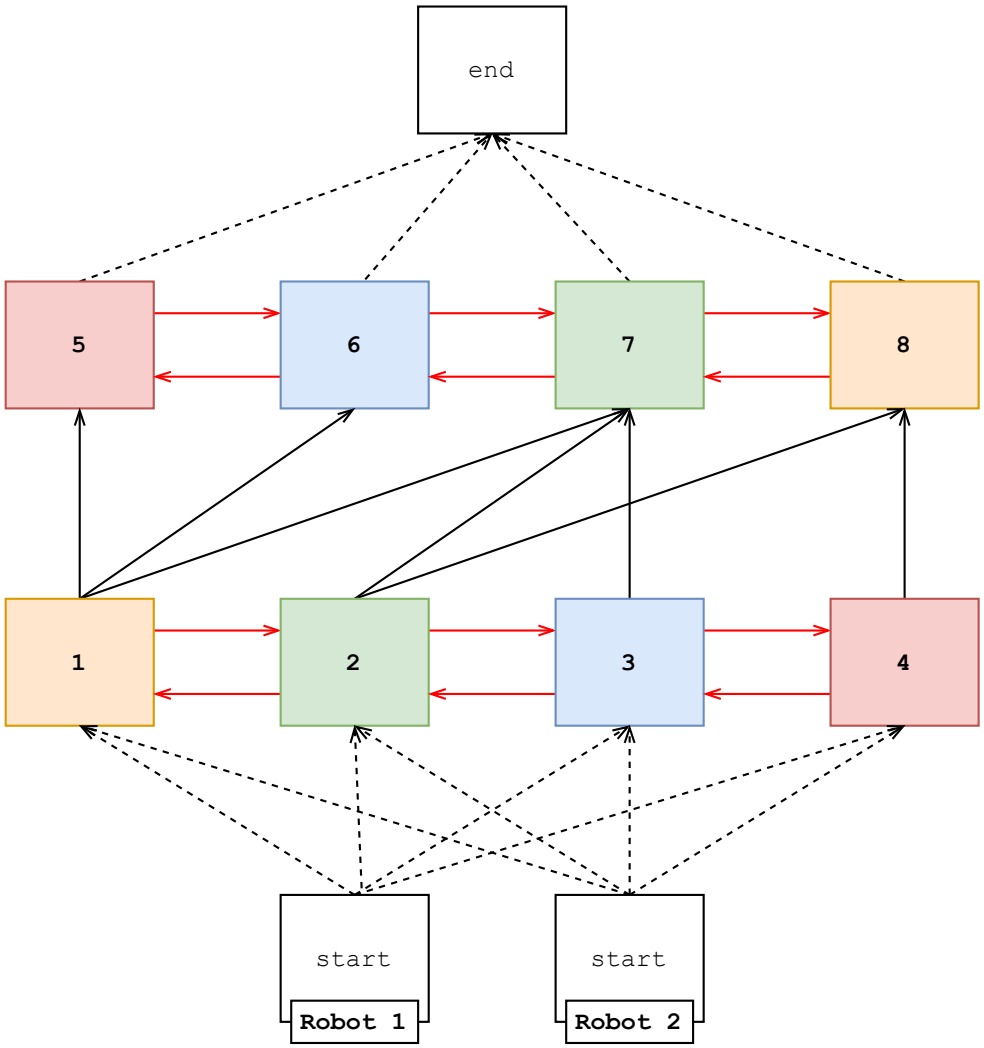


Figure 2.5: Visualization of virtual nodes.

Starting brick can be any brick that has no precedence or active concurrence rules applied to it; see Figure 2.5.

2.4 Time constraint objective function

Algorithm proposed here is a fast iterative method that can be used to obtain feasible solutions.

In case where T_{max} is greater then total time needed to build whole wall plan starts to contain gaps where no robot is utilized, see Figure 2.7. Time after which full wall is

built, further referred as T'. For time $t > T'$ objective function

$$\text{maximize} \sum_{v_i \in V} p_i y_i \xrightarrow{t>T'} constant \tag{2.16}$$

starts to be constant since no further bricks can be placed. We propose modified objective function which includes minimization of the time placement with respect to terminal nodes.

$$\max \sum_{v_i \in V} p_i y_i \ \wedge \ \min s_N \tag{2.17}$$

Both conditions can be merged into one as

$$\text{maximize} \sum_{v_i \in V} p_i y_i - \frac{1}{W} s_N \,, \tag{2.18}$$

where W is normalization constant which should be chosen such that sum of rewards is always greater than the sum of starting placement times. Rewards p_i are always integer values $p_i \geq 1$. In that case, we know the minimal possible value of the collected reward is zero or greater than one. For known T_{max} we can easily choose $W = T_{max}$, restricting time optimization part to evaluate on interval $\langle 0, 1 \rangle$. If our goal is to build the whole wall, we can discard part for collecting reward and use only the time optimization part.

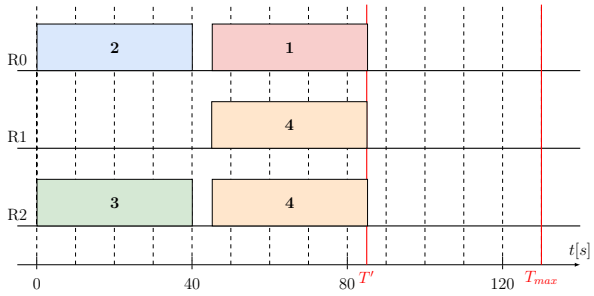


Figure 2.6: Plan shows how minimization term changes final solution for $T_{max} = 130s$ and $T_{max} > T'$.

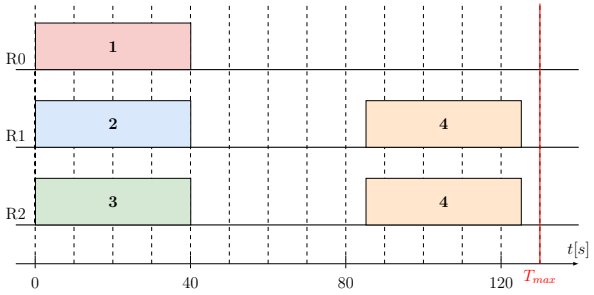


Figure 2.7: Optimal plan solution for $T_{max} = 130s$ without end time optimization criterion.

Figures 2.6 and 2.7 shows how resulting plan changes when we include end time constraint to the objective function. Times s_i have a certain degree of freedom to fill empty space. This empty space can differ based on used solver, but we can clearly see that time component forces plan to be tightly packed within the requested time limit.

2.5 Geometry representation

We introduce set of bricks $\mathbf{B} = \{b_i\}$ (using simplest cuboid shape) defined by their center coordinates $\vec{r}_i = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}$, reward ρ_i, their type and orientation around z-axis φ_i (assuming z axis is normal to the ground). Brick type defines their width, height and depth. Equivalent representation would be to represent brick by 8 vertices for each corner.

Assuming rotation only around z-axis we can always define up, down and side faces of the cuboid given only corner coordination. Graph representation requires knowledge of the relative position of each brick to others.

2.5.1 Generating precedence rules

Precedence set is obtained by comparing top-bottom facing faces of two bricks, described in Algorithm 2. We define the brick set to correspond to the graph node set directly. Precedence rules are generated using Algorithm 2 which performs each to each comparison of bricks and returns precedence rules set.

Algorithm 2: Construction of precedence set.

Input: Set of all bricks B
Result: Π, precedence rules set

```
Π = {};
N = |B|;
i = 1;
while i ≤ N do
    j = i;
    while j ≤ N do
        (b_bottom, b_top) = sort_by_z(b_i, b_j);
        if precedence_check(b_bottom, b_top) then
            Π = Π ∪ {(v_bottom, v_top)};
        j = j + 1;
    i = i + 1;
```

Output: Π

First, we compare z components to check if they are in different heights. Then we use a ray casting algorithm, for coordinates in 2D xy-plane is used to determine if brick lies on top of the other(see Alg. 3).

In our test examples, we assume only simple orientation if the bricks around their z-axis. In the case of general relative positions, we would need to perform sample each face of the brick and then perform the same xy-axis projection to other bricks to find out if

brick lies next to or above the other bricks.

Algorithm 3: Precedence brick comparison.

Input: Bricks b_i, b_j
Result: Boolean
valid = False;
z_i = top_face_z(b_i);
z_j = bottom_face_z(b_j);
if $z_i < z_j$ **then**
  if *is bottom face xy-corners are inside top face* **then**
    valid = True;
Output: valid

In the case of using fully connected layers, as discussed in Section 2.2, we can simplify the process by only determining the relative position of each layer. Then we just create precedence connection between bricks of they layers on top of each other.

2.5.2 Generating concurrence rules

Given the center of each bricks, concurrence rule is determined by the distance of their center positions described by Algorithm 4. Concurrence constraints are generally needed to avoid collisions between participating robots. When we build the wall using UAV's we must ensure non-collision trajectories generated. Despite the fact that collision avoidance is not a main concern in this work, having it as an additional parameter helps to create a plan that can be executed without additional collision avoidance systems in action.

Algorithm 4: Construction of concurrence set.

Input: Set of all bricks B
Result: Γ, concurrence rules set
$\Pi = \{\}$;
$N = |B|$;
$i = 1$;
while $i \leq N$ **do**
  $j = i$;
  while $j \leq N$ **do**
    $d_{ij} = center\_distance(b_i, b_j)$;
    if $d_{ij} \leq d_{min}$ **then**
      $\Gamma = \Gamma \cup \{(b_i, b_j), (b_j, b_i)\}$;
    $j = j + 1$;
  $i = i + 1$;
Output: Γ

Placement distance d_{min}, seen in Alg. 4, is set to

$$d_{min} = d_{place} \min_{b_i, b_j \in B} \{||\vec{r}_i - \vec{r}_j||\}\,,\ d_{place} \in \mathbb{R}\,, \tag{2.19}$$

where d_{place} is chosen constant. Vectors $\vec{r}_i$, $\vec{r}_j$ corresponds to center of the bricks b_i, b_j. We chose to create concurrence rules based only on relative distances between the bricks but we can easily add other empirically generated rules such as adding concurrent rule for bricks that lies next to each other without relaying on their real relative position.

3 Heuristic approach

Our method was inspired by the graphical representation of precedence and concurrence rules as shown in Figure 2.2. In this section, we primarily focus on the description of the method applied to the problem with a homogeneous group of cooperating robots.

We start by identifying all necessary variables needed for full description of the state of the problem. Iterative plan construction algorithms are presented that are used to obtain feasible solution using greedy heuristic. Each iteration is split into multiple sub-problem as finding all available brick that can be picked by a robots at specified times while meeting all constraints inherited from precedence and concurrence rules. Construction is then applied in metaheuristic GRASP algorithms that is used to refine our feasible solution.

The geometric representation of the wall is firstly converted to a special graph representation. Wall bricks are represented by nodes $v_i \in \mathbf{V}$, same as defined in MILP formulation. For each constraint type we define standalone set of edges $e_{ij}^k \in E^k$. Wall is then represented by multigraph $V, E = \{\bigcup_k E^k\}$. We consider E^k sets representing precedence E^p (corresponds to set Π), concurrence rules E^c (corresponds to set Γ).

Edges

The Wall Building Problem is time-dependent, meaning each node of a graph is visible only when all precedence edges are available, which depends on the time of the placement of the respected node. We assign each edge e_{ij}^k with:

- τ_{ij}^k visitation time and
- ξ_{ij}^k variable which is true if edge was visited, false otherwise.

Visitation time is a variable that corresponds to the time in which the end node v_j of edge e_{ij} can be visited.

Algorithm performs forward-time evolution on interval $t \in \langle O, T_{max} \rangle$ from initial configuration to obtain feasible solution.

Nodes

Bricks are interchangeably referenced as nodes. Individual node v_i has associated parameters :

- ρ_i number of required resources to process node v_i,

- ϵ_i time it takes to go from placed brick to its reservoir,
- p_i node reward, and
- t_i time it takes to process node (place corresponding brick).

Resources

Group of robots is interchangeably referenced as *Resources* Φ. Each resource ϕ_i has parameters associated with it

- η_i node $\eta_i \in V$ assigned to the resource ϕ_i,
- σ_i^s start time when node processing began (brick is picked),
- σ_i^e end time when node processing is done (brick is placed),
- σ_i^a time in which resources is available to process new node (brick is placed),
- λ_i resource state.

The state λ_i can be written

$$\lambda_i = \begin{cases} \text{WORKING}, & \text{for } \sigma_i^s \leq t < \sigma_i^e\,, \\ \text{RESTING}, & \text{for } \sigma_i^e \leq t < \sigma_i^a\,, \\ \text{IDLE}, & \text{for } t \geq \sigma_i^a\,. \end{cases} \tag{3.1}$$

Available time σ_i^a is introduced for handling resource resting time when brick is placed and we must return to brick reservoir.

Plan configuration

We define plan configuration as collective state of sets Z_u^e, Z_v^e, Z_u^n, Z_v^n, Ω, state of all resources ϕ_i, edges e_{ij}, nodes v_i, actual time t. Plan reward θ_i is defined based on actual used objective function.

Algorithms additional uses sets to keep track of already processed nodes and edges

- Z_u^e unvisited edges
- Z_v^e visited edges

- Z_u^n unvisited nodes (bricks that has been visited at least once, but precedence rules are not met)
- Z_v^n available nodes (bricks that can be placed)
- Ω already processed nodes (placed bricks)

How these sets are operated is seen in the description of the forward-pass evolution.

3.1 Plan construction

The plan construction is essentially iterative procedure, as shown in Algorithm 5, in which we simulate time forward wall building procedure.

Algorithm 5: Plan construction procedure. (function plan_construct)

Input: Starting plan configuration *plan*.
Result: Feasible time plan for each robot

```
1 while stopping criterion not met do
2 |  iterate_step(plan);
3 return plan;
```

One iteration step shown in Algorithm 6 checks all visible edges Z_v^e, edges that are collectively visible from each robot's point of view.

Algorithm 6: One iteration step of the construction (function iterate_step)

Input: Plan configuration on which iteration is performed

```
1 process_edges();
2 find_available_nodes();
3 minimum_required = assign_available_nodes();
4 if no nodes were assinged then
5 |  update_time(minimum_required);
6 |  if t < T_max then
7 |  |  place_assigned_nodes();
8 |  else
9 |  |  signalize stop;
```

Iteration is decomposed to multiple sub-procedures. First we process all edges and update their state based on actual time t. Based on the state of the edges we are able to find all node candidates that can be assigned to the resources. Then assignment is performed using greedy heuristic. When no nodes were assigned, we update time t. After the update we check if time constraints are met and if not, placement of the nodes is performed.

3.2 Process edges

Although Algorithm 7 finds all available edges in the graph that are visible in the simulation's current time.

Algorithm 7: Process constraints algorithms. (function process_edges)

Result: Set of possible node candidates Z_u^n

```
foreach e_ij^p ∈ Z_v^e do
    if t ≥ τ_ij then
        ξ_ij = true;
        Z_u^n = Z_u^n ∪ {v_j}
```

For each precedence edge e_{ij}, from node v_i to node v_j, we add end node v_j to set of unvisited nodes Z_u^n. If node v_j is already in the set, we skip this step. After procedure finishes, set Z_u^n contains all possible candidates on nodes that might be assignable to the resources. The next step is to actually check for each candidate if all constraints are met.

3.3 Process Nodes

One of the main parts of our method is described by Alg. 8. We construct a list of all nodes that can be assigned to resources by first checking if all precedence rules are met. Precedence rules are met when all precedence edges e_{ij}^p, associated to node v_j, has been visited.

Algorithm 8: Find available nodes. (function find_available_nodes)

```
foreach v_i ∈ Z_u^n do
    precedence_visited = 0;
    foreach outgoing e_ij^p ∈ Z_v^e do
        if ξ_ij^p then
            precedence_visited = precedence_visited + 1;
    if precedence_visited = || {., e_ij^p, .} || then
        if conccurence_rules_met(v_i) then
            Z_v^n = Z_v^n ∪ {v_i};
            Z_u^n = Z_u^n \ {v_i};
```

When precedence rules are met, we perform the check for concurrence rules, described in Algorithm 9. This can be implemented using a fixed lookup table. If no constraints are violated, we move node v_i from the unvisited set Z_u^n to visited set Z_v^n. The visited set is

used in decision which nodes are assigned to the resources.

Algorithm 9: Find available nodes. (function conccurence_rules_met)

Input: Node v_i
Result: Boolean values if concurrence constraints are met. (function concurrence_rules_met)

```
foreach ingoing e_ij^c do
    if ξ_ij^c then
        return false;
return true;
```

3.4 Assign available nodes

Algorithm 10 performs the Greedy assignment of the nodes based on their reward. Given available nodes and available resources, it assigns the maximum number of resources with maximum reward. Algorithm 10 shows how we implemented greedy heuristic in our method. In the case of multiple nodes are having the same effective reward, we pick choose between them randomly.

Algorithm 10: Greedy assignment procedure. (function assign_available_nodes)

Result: In case of no resources assigment, returns number of needed resources available, else returns 0.

```
max_reward = 0;
max_reward_nodes = {};
minimum_resources_required = ∞;
foreach v_i ∈ Z_v^n do
    if ρ_i < minimum_resources_required then
        minimum_resources_required = ρ_i;
    r_a = get_available_resources(v_i);
    if |r_a| ≥ ρ_i then
        if p_i > max_reward then
            max_reward = p_i;
            reward_nodes = {};
        if p_i = max_reward then
            max_reward_nodes = max_reward_nodes ∪ {v_i} ;
if max_reward_nodes is not empty then
    v_p = pick random node from max_reward_nodes;
    assign node v_p to available resources;
    return 0;
return minimum_resources_required;
```

Random pick from maximum reward set of nodes is implemented using integer uni-

form distribution $\mathcal{U}(1, |\text{max reward nodes}|)$ where we chose element on the position obtained from one random choice from the distribution. Given node v_i, available resources are obtained based on their actual state, as shown in Algorithm 11. Under the assumption of all resources to be equivalent, we can stop iteration after necessary number of needed resource ρ_i is found.

Algorithm 11: Find resources assignable to the node. (function get_available_resources)

Input: Assignment node v_i

$\Phi_a = \{\}$;
forall *resources* ϕ_k *in* Φ_a **do**
  if λ_k = *IDLE* **then**
    $\Phi_a = \Phi_a \cup \{\phi_k\}$;
    if $|\Phi_a| = \rho_i$ **then**
      return Φ_a;
return Φ_a;

Assignment to the resources, used in Algorithm 6 is then performed for selected node v_i, used in Algorithm 12.

Algorithm 12: Assign given node to resources. (function assign_available_nodes)

Input: Assignment node v_i to selected resources Φ_a

forall ϕ_k *in* Φ_a **do**
  $\sigma_k^s = t$;
  $\sigma_k^e = t + t_i$;
  $\sigma_k^a = t + t_i + \epsilon_i$;
  $\eta_k = v_i$;
  foreach *outgoing* e_{ik}^c **do**
    ξ_{ik}^c = true;

During node assignment, we also keep the reference of a node with the minimum number of resources required to process it. When there is not enough resources to process any node, we return this number for performing a time update procedure.

Resource transition diagram in Figure 3.1 shows how each state of node v_i, precedence, concurrence edges, and internal resource variables change during assignment and placement of a brick. Resource ϕ_i gets assigned node v_j at time $t = 0$. The diagram shows how resource state λ_i propagates with different simulation times. We can see how states of outgoing edges e_{jk} changes with time.

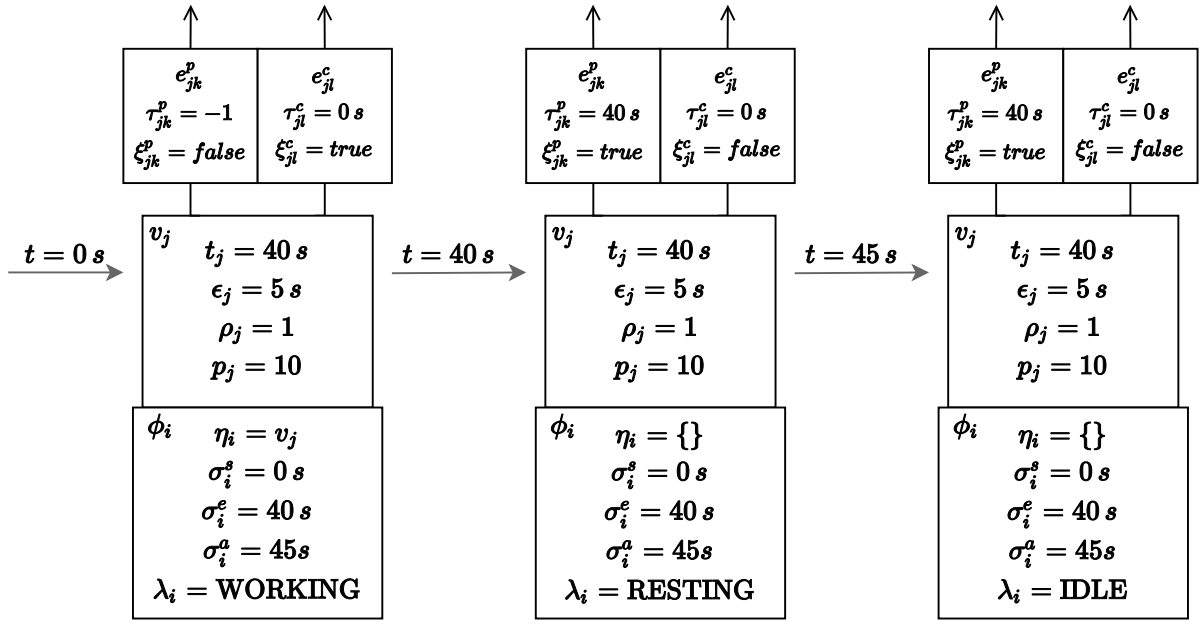


Figure 3.1: Resource state transition diagram.

Time updates are generally not directly related to one resource σ^e_i, but it is determined by finding minimum timestep required to process the next available node in unvisited nodes (Z^n_u) set.

3.5 Time update

When we exhaust all possible resource assignment at some time t, we need to compute minimal time needed to free needed resources. The processs described in Algorithm 13. The algorithm finds the closest possible time where the state of the search increments number of available resources, or state of any resource is changed. We create a histogram of the times where robots are placing bricks and when they are available to pick the next one.

Algorithm 13: Find minimal time update procedure. (function update_time)

Input: number of resources required ρ
Result: Minimal time needed to ensure avaibility number of ρ resources

```
initialize histogram of available resources;
foreach φ_i do
    add entry (ε_i, φ_i);
time = 0;
cumulative_sum = 0;
foreach histogram entry (ε_k, Φ_k = {φ_l, ...}) do
    time = σ^a_k;
    cumulative_sum = cumulative_sum + |Φ_k|;
    if cumulative_sum ≥ ρ then
        break;
return time
```

Performed cumulative search is visualized in Figure 3.2. Based on search we find the closest time when the requested number of robots ρ is available. Iterations are done in ascending order of time entries.

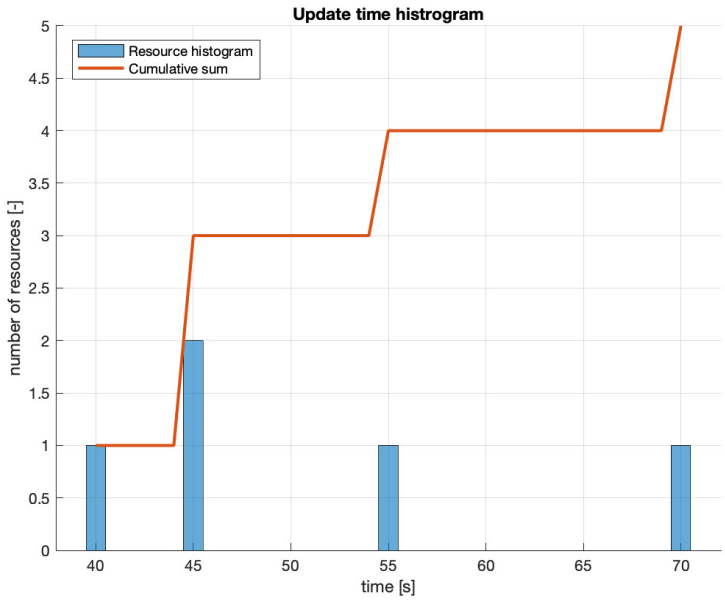


Figure 3.2: Time update histogram and cumulative sum.

3.6 Node placement

When no nodes are assigned and time is updated, we perform check for all active resources, as described Algorithm 14. We check if simulation time t is greater than the end time of the resource σ_i^e. If true, we add the node to the plan that has been assigned to the respected resource. Node is inserted from time σ_i^s to σ_i^e.

Algorithm 14: Finish assigned node. (function place_assigned_nodes)

```
1  foreach φ_i do
2      if has assigned node η_i and t ≥ σ_i^a then
3          add node to current plan, placed at time σ_i^s to σ_i^e;
4          v_k = η_i;
5          foreach outgoing e_kj^p do
6              τ_kj^p = σ_i^e;
7              Z_u^e = Z_u^e ∪ {e_kj^p}
8          foreach outgoing e_jk^c do
9              ξ_jk^c = false;
10         η_i = {};
```

When a node is placed, we iterate all outgoing precedence edges and add them to unvisited edge set Z_u^e from where they are processed in the next iteration. Concurrence edges e_{kj}^c are disabled by explicitly setting their visited flag ξ_{ik}^c to false.

3.7 Optimization procedure

Our heuristic greedy search can be used in the GRASP optimization algorithm; see Algorithm 15. Stopping criterion is defined using K_{max}, maximum number of iterations, and $K_{max\,not\,improved}$, maximum number of iteration where best solution did not improve.

The GRASP optimization procedure can be used to improved already found solutions. Mersenne twister [24] pseudorandom number generator is used throughout our implemen-

tation.

Algorithm 15: GRASP - main algorithm.

Result:

$k_{iter} = 0$;
$k_{not\,improved} = 0$;
best_solution = initial_solution;
while $k_{iter} < K_{max} \wedge k_{not\,improved} < K_{max\,not\,improved}$ **do**
    restart random number generator with different seed;
    greedy_solution = greedy_randomized_construction(initial_solution);
    solution = local_search(greedy_solution);
    update_solution(best_solution, solution);
    if *best_solution did not improve* **then**
        $k_{not\,improved} = k_{not\,improved} + 1$;
    $k_{iter} = k_{iter} + 1$;

Output: best_solution

At each iteration, see Alg. 16, we construct plan that satisfies all constraints. During the creation, we specify **snapshots**. We define snapshot as a partially built plan with a complete state of the graph, states of all resources, nodes, and edges. A partial plan can be saved and later used as an initial solution of the construction procedure shown in Alg. 16.

Algorithm 16: GRASP - greedy randomized construction.
(function greedy_randomized_construction)

Input: initial_partial_solution, global $B^{\star}_{max}$
Result: Feasible solution that satisfies all constraints

solution = initial_partial_solution;
snapshots = {};
placed_node = 0;
Δ_s = generate_snapshot_positions($B^{\star}_{max}$);
while *is not finished* **do**
    iterate_step(solution);
    if *solution reward increased* **then**
        placed_nodes = placed_node + 1;
        if *placed_nodes* $\in \Delta_s$ **then**
            snapshots = snapshots $\bigcup$ solution;

Output: {solution, snapshots}

Initial solution is special type of partial solution where we specify already known state of the wall that is physically built. It can contain state of all robots and state of the bricks that are already placed or are carried by robots. Randomized construction returns a feasible solution and state of the solution together with associated snapshots. Iteration step described by Algorithm 6 is used.

If the total number of bricks is B_{max}, the naive method would be to choose snapshots at each brick placement, but that can lead to memory problems when graph is huge. We propose an adaptive procedure using a previously known solution to restrict number of snapshots created. Each GRASP iteration produces feasible solution from which we can estimate number of bricks B_{est}, such that $B_{est} \leq B_{max}$, where B_{est} is computed as number of bricks that were placed in the found solution.

Snapshots

During construction of the feasible solution, we propose an adaptive procedure that generates a set of placement points where a snapshot of the partially built plan is performed. Algorithm 16 is expanded with global state $B^{\star}_{max}$, the maximum number of bricks that were placed in the best solution obtained by GRASP. This value is updated after the full iteration of the GRASP procedure. Before iteration starts we prepare a set of placing points by generating set Δ_s of uniformly chosen non-repeating integer numbers on the interval $\{1, B^{\star}_{max}\}$ as:

$$\Delta_s = \{\delta_s \,|\, \text{non-repeating}\, \delta_s \in \{1, B^{\star}_{max}\}\}\,. \tag{3.2}$$

Size of the set is set to be computed using snapshot coefficient $\Upsilon \in \langle 0, 1\rangle$ as $|\Delta_s| \;=\; \Upsilon\, B^{\star}_{max}$.

3.8 Local Search

We propose local search described in Alg. 17 as direct re-optimization of each snapshot saved during greedy construction, see Alg. 16.

Algorithm 17: GRASP - local search. (function local_search)

Input: found solution with set of snapshots

```
1 final_solutions = {};
2 foreach snapshot do
3 |   solution = plan_construct(snapshot);
4 |   final_solutions = final_solutions ∪ {solution};
5 best_found_solution = pick solution with best reward from final_solutions;
```

Output: best_found_solution

Re-optimization is achieved by applying snapshot as starting plan configuration in Algorithm 5. Solution obtained from the construction is then saved. After all snapshots are re-optimized we find and return solution with the best reward.

3.9 Solution update

Simple update procedure is proposed in Algorithm 18. When randomized construction is refined with the local search (see Alg. 15), we compare the actual known best solution with the found solution. In our implementation, we used direct reward comparison, but we are able to include other measures when rewards are the same.

Algorithm 18: GRASP - update solution based on reward. (function update_solution)

Result:
Input: found solution P_{found}, best solution P_{best}
1 **if** *found solution reward* > *best solution reward* **then**
2 | best solution = found solution

Solutions with the same reward but different state variables such as the utilization of each robot or end time of the plan T' can be easily added to update procedure.

4 Heterogenous robot fleet

In this section, we consider modifications of our proposed method to include the use of a heterogeneous group of robots. Cooperation situation requires us to model each robot separately with different parameters and behavior. Our proposed method in earlier section can be easily extended to account for such changes. We describe simple structural changes to variables used in our method to account for additional information needed in each iteration.

Nodes

Nodes no longer contain information about the time it takes to process them t_i. We also discard information about reservoir time ϵ_i. The number of robots ρ_i required to process node now becomes a set of types of the resources needed to be present at given node in order to process it. New variable node type n_i is introduced. In the case of previously shown examples, this variable would be used to represent different brick colors. Modified node must have these attributes

- ρ_i set of types of the resources in order to process node,
- p_i node reward,
- n_i node type.

In our test, we define node type as being RED, GREEN, BLUE or ORANGE. Resource mapping ρ_i contains information about all combinations in which node can be processed. For example, let us consider a group of 2 UVAs and one terrestrial robot. Assume that brick is heavy and must be carried by 2 UAV's simultaneously or can be carried by the terrestrial robot. Corresponding ρ_i would contain :

$$\rho_i = \left\{ \left\{ \nu^{uav}, \nu^{uav} \right\}, \left\{ \nu^{terrestrial} . \right\} \right\} \tag{4.1}$$

Resources

Resources must now provide variables that were removed from node in form of mapping from visiting node type n_i. Resource must have these attributes

- η_i current node assigned to the resource $\eta_i \in V$,
- σ_i^s start time when node processing began,

- σ_i^e end time when node processing is done,
- σ_i^a time in which resources is available to process new node,
- λ_i resource state,
- ν_i resource type,
- $t_i(n_i)$ time mapping corresponding to time it takes resource ϕ_i to process node v_i.

Time mapping in our example would have to specify the time for each type as

$$t_i = \begin{cases} t_i^{red}, & \text{for } n_i = RED \\ t_i^{green}, & \text{for } n_i = GREEN \\ t_i^{blue}, & \text{for } n_i = BLUE \\ t_i^{orange}, & \text{for } n_i = ORANGE \end{cases} . \tag{4.2}$$

Algorithm 19 is modification of Algorithm 11 to cover all possible combinations in ρ_i set. Instead of choosing an arbitrary resource that is in the IDLE state, we find some combination that would be required to process the node.

Algorithm 19: Find all assignable combination of node given available resources. (function get_available_resources)

Input: Assignment node v_i
Result: Set of all valid combinations of the resources that can be assigned to the node v_i at current iteration.

```
1  Φ_a = {};
2  foreach ϱ_k in ρ_i do
3  |   Φ_ϱk = {};
4  |   forall resources ϕ_j in Φ do
5  |   |   if λ_j = IDLE and ϕ_j ∈ ϱ_k then
6  |   |   |   Φ_ϱk = Φ_ϱk ∪ {ϕ_j};
7  |   |   |   if |Φ_ϱk| = |ϱ_k| then
8  |   |   |   |   Φ_a = Φ_a ∪ {Φ_ϱk};
9  |   |   |   |   break;
10 return Φ_a;
```

Algorithm 10 previously returned first available resources with state $\lambda_i = IDLE$ until requested number was met, now returns subset of all possible combination Φ_a in which node must be assigned. Algorithm 20 is changed to account for different return value from assignment algorithm. There are two changes from the original algorithm. After we find all possible combinations that can be assigned given the actual state we need to choose

only one combination. We assume that reward is the same for all combinations ϱ_k, so we implement procedure **pick_possible_resources** as a random choice of given possibilities, but there is no obstacle in implementing this procedure to account for other empirical conditions if needed.

In case of multiple nodes having the same reward, we now store information both about node v_i and its assigned resources that we picked earlier. The previous implementation does not require to store picked resource information since algorithm **get_available_resources** always return the same result when called later.

Algorithm 20: Greedy assignment procedure. (function assign_available_nodes)

Result:

max_reward = 0;
max_reward_nodes = {};
minimum_resources_required = ∞;
foreach $v_i \in Z_v^n$ **do**
  if *min*$\{|\varrho_k| \mid \varrho_k \in \rho_i\} <$ *minimum resources required* **then**
    minimum_resources_required = ρ_i;
  Ψ_a = get_available_resources(v_i);
  r_p = pick_possible_resources(r_a);
  if $|r_p| \geq \rho_i$ **then**
    if $p_i >$ *max_ reward* **then**
      max_reward = p_i;
      reward_nodes = {};
    if $p_i =$ *max reward* **then**
      max_reward_nodes = max_reward_nodes $\cup \{\{v_i, r_p\}\}$;
if *max_ reward_ nodes is not empty* **then**
  v_p = pick random node from max_reward_nodes;
  assign node v_p to available resources;
  return 0;
return minimum_resources_required;

Node assignment Algorithm 12 is extended using time mapping as shown in Algorithm 21.

Algorithm 21: Assign given node to resources

Input: Assignment node v_i to selected resources Φ_a

forall ϕ_k *in* Φ_a **do**
  $\sigma_k^s = t$;
  $\sigma_k^e = t + t_i(n_i)$;
  $\sigma_k^a = t + t_i(n_i) + \epsilon_i$;
  $\eta_k = v_i$;
  foreach *outgoing* e_{ik}^c **do**
    ξ_{ij}^c = true;

Modifications presented in this section shows how easy it is to extend original method to incorporate different types of constraints. Such flexibility is required when implementing these types of algorithms in competition environment where modification using linear programming formulation might require to be entirely different.

5 Performance evaluation

In this section we provide set of tests designed to compare performance in multi-threaded environment, Section 5.3, or different wall sizes, sections 5.4 and 5.5. In section 5.7 we discuss existence of the upper limit on how many robots can be utilized for given wall size and constraints. Generated solutions are visually compared in section 5.6.

We choose C++ as main language for direct compatibility with robotic system used in the robots. All tests are run on machine described in Table 2 together with compiler information.

Table 2: System information used to run all tests.

System	macOS 15.4.1
CPU	Intel Core i7-8850H@2.6GHz, 6 core
RAM	16GB DDR4@2400 MHz
Compiler	clang-1100.0.33.17
Compilation flags	-std=c++17 -03

5.1 CPLEX

Proposed method is compared with the CPLEX solver that uses exact algorithms to obtain the optimal solution. IBM ILOG CPLEX[2], short CPLEX, offers C++ library that solve linear programming (LP) and related problems. Specifically, it solves linearly constrained optimization problems where the objective to be optimized can be expressed as a linear function or a convex quadratic function. The variables in the model may be declared as continuous or further constrained to take only integer values (ILP) or mixed-integer linear programming (MILP). It implements multiple exact algorithms such as branch-and-cut, branch-and-bound or simplex search. Solver is available for free with academic license. We use version 12.10 to perform all benchmarks.

Throughout all tests we limit runtime of CPLEX solver to 10 minutes. Solver provides optimality gap information to quantify feasibility of a solution. GAP parameter is computed inside solver as

$$GAP = 100\frac{\text{best bound solution} - \text{best integer solution}}{\text{best integer solution}}\,[\%]\,, \tag{5.1}$$

where *best bound solution* is upper estimate of the best achievable reward *best integer solution* is best solution reward found during computation.

5.2 Datasets

We created a set of testing walls where we aim to test how well our algorithm runs compared to the exact method provided by CPLEX. Testing walls are created in different

[2]Official site `https://www.ibm.com/analytics/cplex-optimizer`

sizes of power 2, from 2^2 - 2^{10}. See partial datasets used in Fig. 5.1.

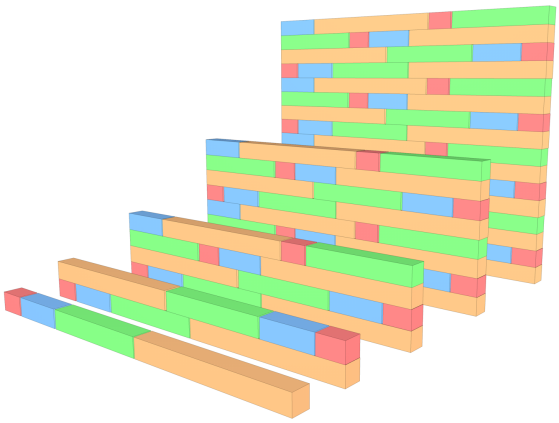


Figure 5.1: Visualization of benchmark datasets for wall with 4 up to 64 bricks. Datasets for more bricks are created in similar way.

Throughout benchmarks we use fixed placement duration of a bricks in Table 3, where ρ_i is number of robots required to place a brick, p_i is its reward, and t_i is the time it takes to place a brick to its position.

Table 3: Overview of used bricks throughout the benchmarks.

	Bricks			
parameter	red	green	blue	orange
ρ_i (required robots)	1	1	1	2
p_i (brick reward)	2	4	8	16
t_i (placement duration)	40 s	40 s	40 s	40 s

Together with group of 3 homogeneous robots, $R = 3$. Only parameter that needs to be specified is time it takes to go from wall to reservoir. Benchmarks use fixed value of reservoir time $\epsilon_i = 5\,s$. Walls are built from initial configuration where no bricks are placed, and robots are on their starting positions without holding any bricks.

5.3 Multithreaded performance

Our algorithm can be successfully parallelised since each iteration of GRASP is almost independent of the others, as shown in Fig. 5.2. By almost we are referring to the snapshot factor Υ, which is used with $B^{\star}_{max}$ generated at the end of GRASP iteration. In our implementation, we decided to have different factor Υ for each thread. Since there are hundreds or thousands iteration we let each thread to handle adaptive procedure independently.

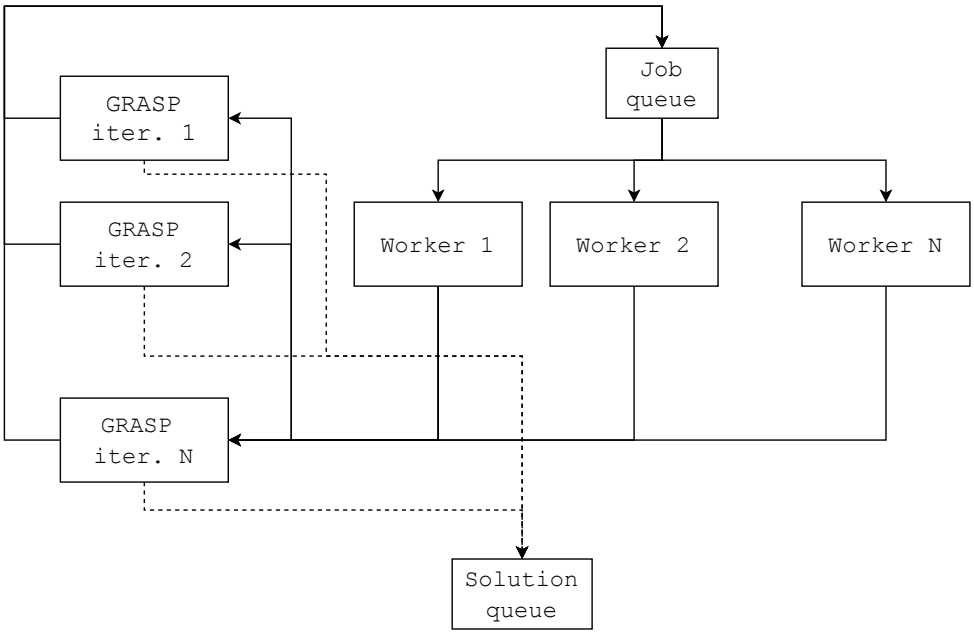


Figure 5.2: Multithreaded model used in our method.

Results from multithreaded benchmark are shown in Table 4. Data shows that our method scales well with number of available cores.

Table 4: Multithreading scaling comparison. Tables shows different runtimes of the algorithm with respect to number of thread used.

	number of threads						
Dataset	1	2	4	8	12	16	20
wall_4	291 ms	129 ms	68 ms	44 ms	43 ms	44 ms	38 ms
wall_8	376 ms	191 ms	100 ms	56 ms	55 ms	59 ms	49 ms
wall_16	572 ms	311 ms	173 ms	99 ms	88 ms	80 ms	113 ms
wall_32	1062 ms	507 ms	279 ms	178 ms	176 ms	173 ms	181 ms
wall_64	1.6 s	909 ms	512 ms	303 ms	309 ms	279 ms	251 ms
wall_128	12 s	7 s	4.7 s	2.8 s	2.5 s	2.5 s	2.3 s
wall_256	16.4 s	11.2 s	6.9 s	5.6 s	4.2 s	6.3 s	6.2 s
wall_512	37.6 s	23.2 s	14.4 s	12.1 s	9 s	9.3 s	9.9 s
wall_1024	103.6 s	56.3 s	36.2 s	22.8 s	18.4 s	19.3 s	19.6 s

We compare how our algorithm scales with additional cores. Benchmark was performed on machine with 6 physical cores with Intel's hyper-threading[3]. Each test was performed with $d_{min} = 80\,cm$, $L_p = true$, $T_{max} = 200\,s$, $R = 20$. This configuration allows to test all wall sizes within a reasonable time frame quickly. For small walls, we are able to find

[3] `https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`

solutions that build whole wall within time T_{max}. For huge walls all available robots are used, as discussed in Section 5.7. Results show that algorithm scales well with number of available cores. Best performance is as expected around number of 12 threads since that is number of physically available thread CPU can handle.

Other benchmarks are performed in the multithreaded version with 12 working threads. Multithreaded model is implemented using thread pool workers. GRASP iteration is further split into worker jobs. This model provides better core utilization instead from using one GRASP instance per thread. CPLEX solver is set to use same number of threads as our method.

5.4 Weakly constraint performance

We define weakly constrained as one where concurrence rules are excluded making feasible set of solution larger. Same effect can be achieved with choosing minimal concurrence distance as $d_{min} = 0$. To remove even more constraints, layer parameter L_{first}, discussed in Section 2.2, is set to $L_{first} = false$.

First test shown in Table 5 compares CPLEX method with our method where final time T' is determined by objective function minimizing end time. Due to hardware memory limitations we were not able to use snapshots local search for problems *wall_512*, *wall_1024*. Using $\Upsilon = 0$ leads to solution obtained only from randomized construction evaluated over multiple iterations.

Table 5: Benchmark results of our method with optimal solution obtained from CPLEX solver. Concurrence rules are excluded, $T_{max} = \infty$.

		CPLEX				Our method			
Dataset	max reward	collected reward	T'	runtime	GAP	collected reward	T'	runtime	Υ
wall_4	30	30	85 s	45 ms	0.00%	30	85 s	104 ms	0.1
wall_8	60	60	175 s	388 ms	0.00%	60	175 s	226 ms	0.1
wall_16	120	120	310 s	10 min	0.13%	120	310 s	240 ms	0.1
wall_32	240	240	665 s	10 min	0.15%	240	625 s	155 ms	0.1
wall_64	480	100	440 s	10 min	381.07%	480	1295 s	320 ms	0.1
wall_128	672	0	0	10 min	∞	672	2065 s	2.71 s	0.1
wall_256	1378	-	-	-	-	1378	4180 s	12.14 s	0.1
wall_512	2778	-	-	-	-	2778	8365 s	7.92 s	0.0
wall_1024	5556	-	-	-	-	5556	16735 s	5.22 s	0.0

Second test shown in (Tab. 8) compares optimal solution with our method in which T_{max} is fixed in interval $\langle 0, T' \rangle$ obtained from previous benchmark (Tab. 5).

Our method used *maximum iteration* set to 1000 and *did not improve* factor set to 100. Due to available hardware memory limitations we were not able to use snapshots local search for problems *wall_1024*. Using snapshot coefficient $\Upsilon = 0$ effectively disables snap-

shot creation. This leads to solution obtained only from randomized construction evaluated over multiple iterations.

Table 6: Benchmark results of our method with optimal solution obtained from CPLEX solver. T_{max} is chosen to be less then time in which wall is built.

			CPLEX			Our method		
Dataset	max reward	T_{max}	collected reward	runtime	GAP	collected reward	runtime	Υ
wall_4	30	45 s	24	12 ms	0.00%	24	116 ms	0.1
wall_8	60	130 s	40	216 ms	0.00%	40	114 ms	0.1
wall_16	120	220 s	96	12.37 s	0.00%	90	175 ms	0.1
wall_32	240	350 s	130	10 min	61.54%	112	237 ms	0.1
wall_64	480	800 s	74	10 min	548.65%	342	284 ms	0.1
wall_128	672	1600 s	92	10 min	630.43%	582	1.91 s	0.1
wall_256	1378	3000 s	0	10 min	∞	1110	8.33 s	0.1
wall_512	2778	5000 s	-	-	-	1888	37.63 s	0.1
wall_1024	5556	10000 s	-	-	-	3460	16.48 s	0.0

5.5 Highly constrained performance

When the problem is highly constrained, generic LP solvers like CPLEX can perform better than in weakly constrained problems. Table 7 shows how end time T' in which full wall can be built with concurrence placement rules applied with $d_{min} = 80\,cm$. To add more constrains, we enforce that full layers of bricks must be built before next one can start L_{first} is set to $L_{first} = true$. Optimization with end time was used so T' in case of CPLEX is optimal.

Number in dataset name corresponds to number of bricks in the wall. Benchmark shows that CPLEX solver wasn't able to compute the optimal solution ($GAP = 0.00\%$) in reasonable amount of time for wall with more than 16 bricks. Final times T' are used in next benchmark to determine some reasonable time T_{max} for which we perform the test under a time constraint.

Table 7: Benchmark results of our method with optimal solution obtained from CPLEX solver. Concurrence rules are excluded, $T_{max} = \infty$.

		CPLEX				Our method			
Dataset	max reward	collected reward	T'	runtime	GAP	collected reward	T'	runtime	Υ
wall_4	30	30	85 s	63 ms	0.00%	30	85 s	110 ms	0.1
wall_8	60	60	175 s	301 ms	0.00%	60	175 s	105 ms	0.1
wall_16	120	120	350 s	63.69 s	0.00%	120	350 s	196 ms	0.1
wall_32	240	240	700 s	10 min	0.05%	240	735 s	625 ms	0.1
wall_64	480	480	1445 s	10 min	0.09%	480	1465 s	2.01 s	0.1
wall_128	672	8	40 s	10 min	8311.81%	672	2295 s	7.34 s	0.1
wall_256	1378	0	0 s	10 min	∞	1378	4630 s	56.15 s	0.1
wall_512	2778	-	-	-	-	2778	9365 s	6.77 s	0.0
wall_1024	5556	-	-	-	-	5556	18740 s	11.82 s	0.0

Due to hardware memory limitations we were not able to use snapshots local search for problems *wall_ 512*, *wall_ 1024*. Using $\Upsilon = 0$ leads to solution obtained only from randomized construction evaluated over multiple iterations.

Second test shown in Table 8 compares optimal solution with our method in which T_{max} is fixed in interval $\langle 0, T' \rangle$ obtained from previous benchmark in Table 7. Our method used maximum iteration set to $K_{max} = 10000$ and did not improve factor set to $K_{max\,not\,improved} = 100$. Due to hardware memory limitations we were not able to use snapshots local search for datasets *wall_ 512* and *wall_ 1024*. Using $\Upsilon = 0$ leads to solution obtained only from randomized construction evaluated over multiple iterations.

Table 8: Benchmark results of our method with optimal solution obtained from CPLEX solver. T_{max} is chosen to be less then time in which wall is built, $T_{max} \approx T'/2$, completely to test collected reward criterion.

			CPLEX			Our method		
Dataset	max reward	T_{max}	collected reward	runtime	GAP	collected reward	runtime	Υ
wall_4	30	45 s	24	126 ms	0.00%	24	79 ms	0.1
wall_8	60	130 s	54	2.05 s	0.00%	54	100 ms	0.1
wall_16	120	220 s	84	14.86 s	0.00%	84	206 ms	0.1
wall_32	240	350 s	120	10 min	0.00%	120	305 ms	0.1
wall_64	480	800 s	180	10 min	166.67%	240	408 ms	0.1
wall_128	672	1600 s	40	10 min	1580.00%	488	3.41 s	0.1
wall_256	1378	3000 s	0	10 min	∞	916	21.24 s	0.1
wall_512	2778	5000 s	-	-	-	1486	5.61 s	0.0
wall_1024	5556	10000 s	-	-	-	3022	22.22 s	0.0

5.6 Plan comparison

Now we look into how plans actually compares between each solver. We will demonstrate found solution on dataset *wall_8*. Brick configuration is the same as in previous tests, see Tab. 3. From benchmark results in Table 7 we know that optimal time to build chosen dataset is $T' = 175\,s$. Optimal plan corresponding to this solution is visualized in Figure 5.3, and plan generated using our method in Figure 5.4. Highly constrained settings, $d_{min} = 80\,cm$, $L_{first} = true$ were used in comparison.

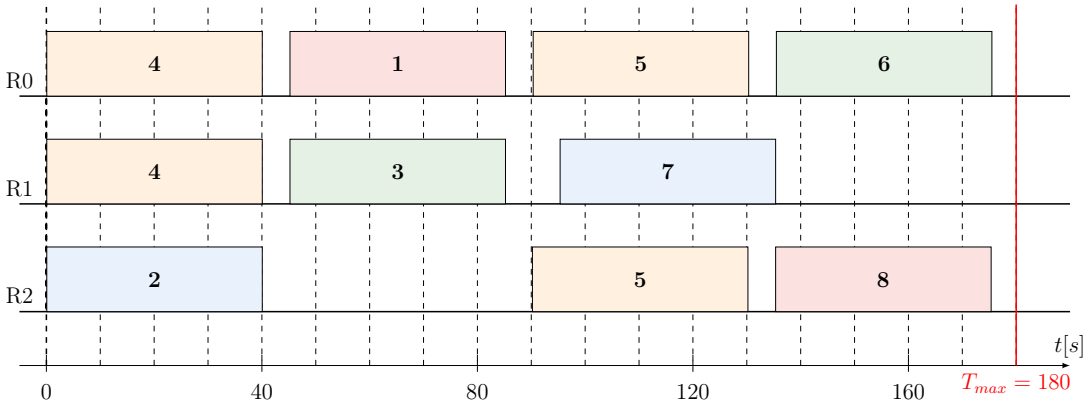


Figure 5.3: Optimal solution obtained using CPLEX solver, smallest time needed to build 8 brick wall with 3 robots is 175 seconds.

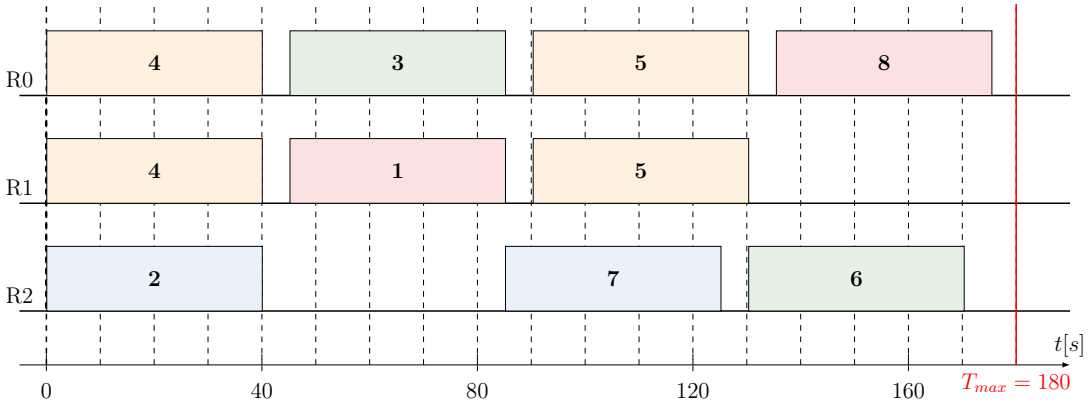


Figure 5.4: GRASP solution, all bricks were built. Final time is the same as optimal solution seen in Figure 5.3.

We can clearly see greedy decision made at time 85s where we start to built second layer immediately after first layer was built.

Up until now we tested performance when brick placement time $t_i = 40\,s$ is same

for all types of bricks. Next, we compare how our method holds when placement duration are different for each brick type. Test is run on dataset *wall_16* with brick configuration Tab. 9.

Table 9: Overview of brick configuration in benchmark with *wall_16*. Times t_i were chosen such that there is no close integer multiple of their times. This tests how solvers handlers non-aligned brick placement times σ_i^e.

	Bricks			
parameter	red	green	blue	orange
ρ_i	1	1	1	2
p_i	2	4	8	16
t_i	23 s	29 s	37 s	40 s

Using same method as before to obtain optimal time T' using CPLEX solver in Figure 5.5. That gives us $T' = 289\,s$ with maximum collected reward 120.

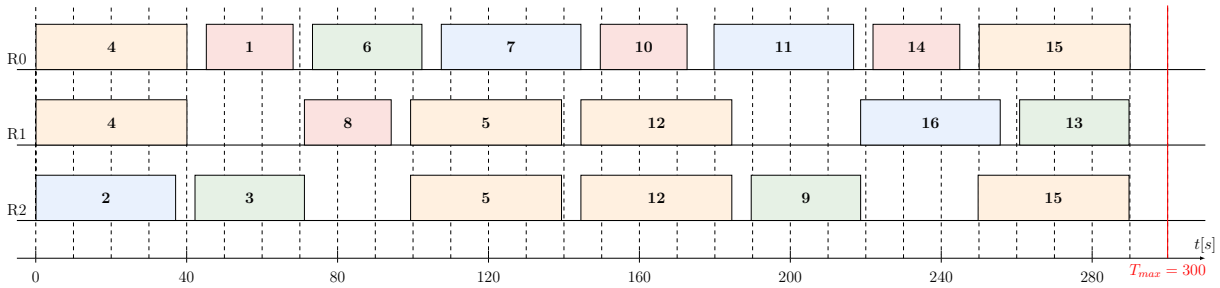


Figure 5.5: Optimal solution obtained using CPLEX solver, smallest time needed to build 16 brick wall with 3 robots is 289 seconds with total collected reward of 120.

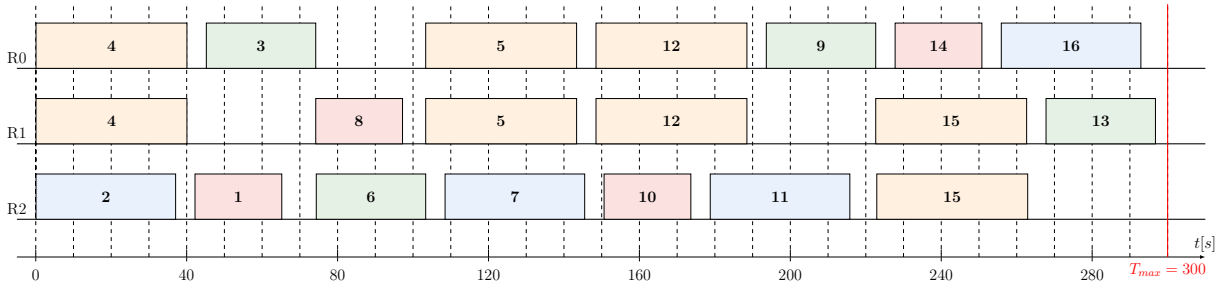


Figure 5.6: Solution obtained using our method, we were able to find full solution with $T' = 296\,s$. Total collected reward is 120.

For our solution, seen in Figure 5.6, we performed extensive search with snapshot parameter $\Upsilon = 1.0$, maximum iteration 1000. Search took 52 s in which we were able to find solution close to optimal one generated by exact method.

Visible blank spaces that occurs in all Figures 5.3, 5.4, 5.5 and 5.6 are caused by

$L_{first} = true$ parameter used in out tests. In our dataset *wall_8* contained 2 layers, each with 4 bricks. Dataset *wall_16* contained 4 layers, also with 4 brick each. From the generated plans we can clearly distinguish corresponding brick in each layer.

5.7 Robot utilization limits

When a feasible plan is created, we can compute utilization of each robot in plan execution time frame. Given set of all constraints specified for WBP we would intuitively expect that there exists the maximum number of robots R_{max}, after which adding more means that their utilization is equal to zero. We can demonstrate such effect on dataset *wall_16*. Table 10 shows results of maximum robots utilized during plan execution.

Table 10: Benchmark results of maximum robots utilization.

constrained	T_{max}	R	R_{used}
weakly	200 s	10	7
highly	200 s	10	5
weakly	500 s	10	9
highly	500 s	10	6

From the results, we clearly see that even if we use a group of 10 robots, we would not be able to utilize all of them. Interestingly, number R_{max} also depends on T_{max}. Such effect occurs when having multiple layers with a different number of bricks, and T_{max} effectively blocks layers where more robots could be utilized.

6 Experiments

The planning algorithm has been tested in simulation environment. A common framework used to develop robotic solutions in Robotic Operating System (ROS).

Robot operating system (ROS)

> *The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms* `http://www.ros.org/about-ros/`.

ROS can be described as general-purpose distributed system[25] in which modules are defined and interconnected between each other using a common interface. ROS modules can be implemented in various languages, from one of which is C++ that we used in our implementation. Multiple modules can run on one or multiple machines. Communication layers is provided by ROS. Distributed nature of whole system enables easy and fast development process of a complex solution such as cooperation between multiple robots that is required for wall building.

Simulation environment

To simulate a group of cooperating robots we used Gazebo simulator[4] in which one can implement all environmental aspects of given problem. Gazebo simulates multiple robots in a 3D environment, with dynamic interaction between objects.

We run simulation for two cooperating UAVs. Wall is built in the center of simulation space. Brick reservoirs are placed equidistantly from the center, as seen in Figure 6.1.

[4] `http://gazebosim.org/`

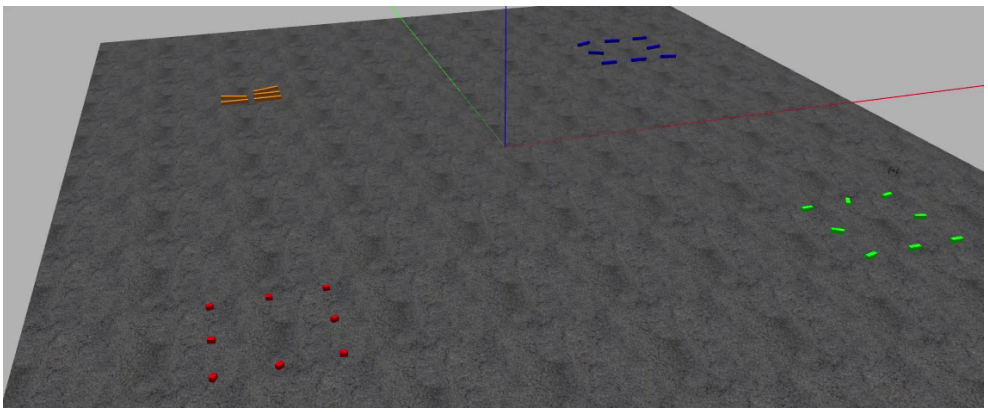


Figure 6.1: Brick reservoirs placement during the simulation. Wall is built in the center of coordinate system.

Simulation environment in Figure 6.3 shows cooperation of two UAVs (third one is waiting out of the camera frame). We are also able to simulate camera feedback as seen in Figures 6.4 and 6.5.

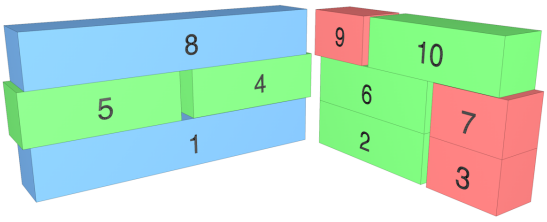


Figure 6.2: Visualization of wall used in the simulation.

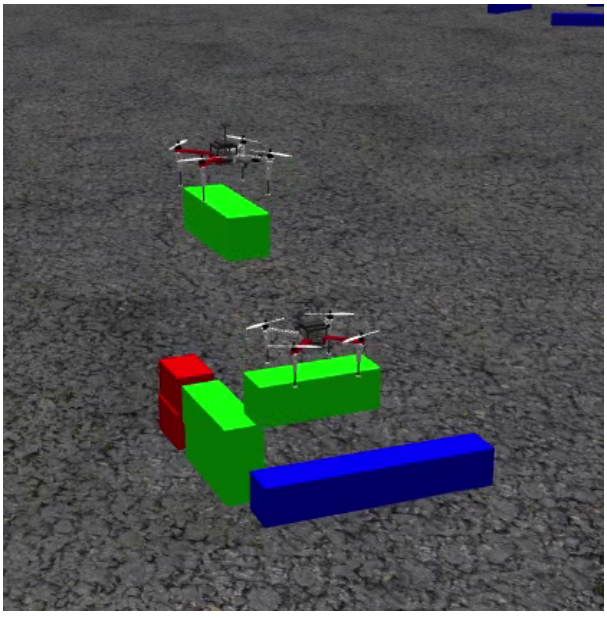


Figure 6.3: Wall being built by 2 UAVs.

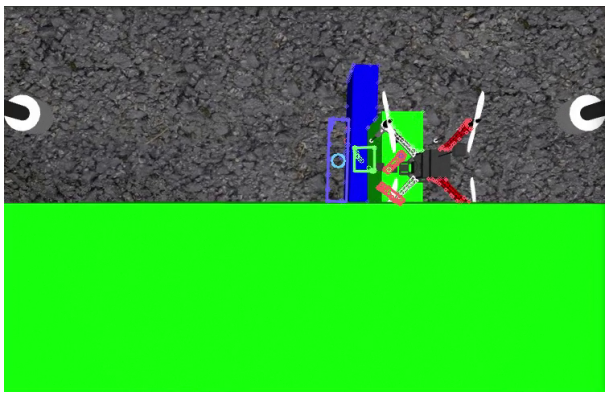


Figure 6.4: Simulated camera feedback of the first UAV in Figure 6.3.

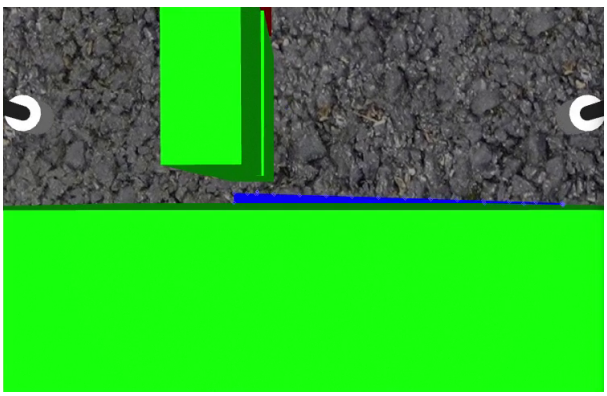


Figure 6.5: Simulated camera feedback of the second UAV in Figure 6.3.

Wall used in the simulation is shown in Figure 6.2. Executed plan is shown in Figure 6.6. Layers constraint was set as $L_{first} = true$ and no concurrence placement, $d_{min} = 0$, was used. Time constraint T_{max} was set to 180s with total collected reward of 28. In chosen time constraint wall is only partially built up to the second layers.

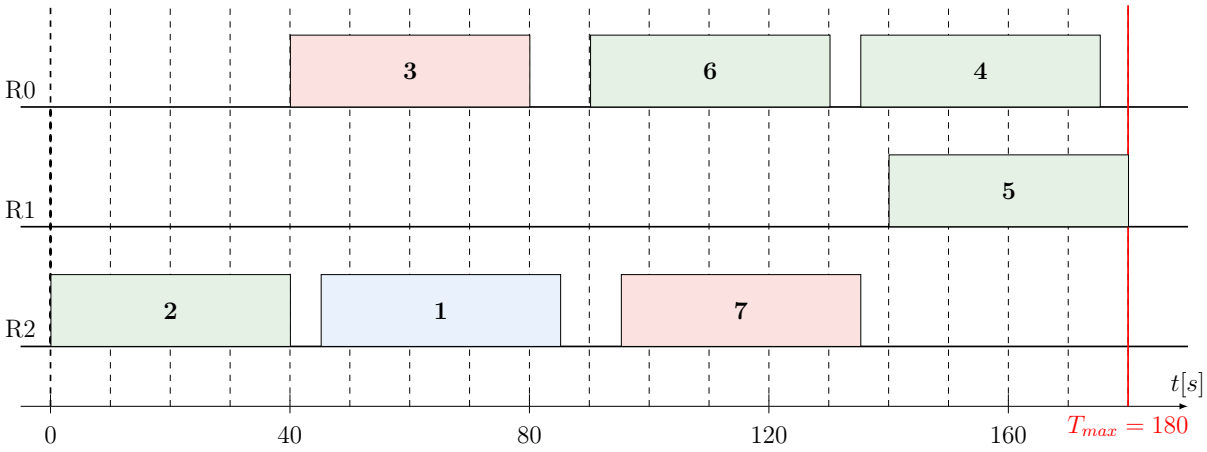


Figure 6.6: Plan used in simulation.

7 Conclusion

This thesis presented a self-contained method for wall building task. Works was motivated by MBZIRC 2020 competition where one of the challenges was to build wall using group of robots while collecting maximum reward withing specified time budget. We started by the description of various types of problems as variations of the Vehicle Routing Problem. A Wall building problem was formulated as a derivation of Cooperative Orienteering Problem, where optimization problems were formulated in mixed-integer linear programming. Wall was then transformed from geometric to a graph representation.

Graph representation was used as the main motivation of our heuristic approach. From metaheuristic algorithms, we chose the GRASP approach, which was applied in our implementation of the iterative solver. Proposed iterative algorithms searches problem graph while randomly picking incremental steps that build up feasible solution set. Local search further refines found solution. Refined solution is then compared to the best solution known at the iteration step and updated if needed. We showed algorithms applied on homogeneous group of robots. The method was further extended for handling a heterogeneous group of robots.

Benchmarks were designed to test how our solution compares with generic CPLEX solver in terms of collected reward and total runtime execution. In case of small problem size, see Tables 8 and 6, where optimal solution could be found, obtained results shows that our method is able to find solution with the same rewards as optimal one.

References

[1] Denis Chamberlain. *Automation and Robotics in Construction Xi*. Elsevier, 1994. ISBN: 978-0-444-82044-0.

[2] Dirk Briskorn, Andreas Drexl, and Sonke Hartmann. *Inventory-based dispatching of automated guided vehicles on container terminals*. 2007. ISBN: 978-3-540-49550-5.

[3] T. Carwalo, J. Thankappan, and V. Patil. Capacitated vehicle routing problem. In *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, pages 17–21, 2017.

[4] Nasser A. El-Sherbeny. Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods. *Journal of King Saud University - Science*, 22(3):123 – 131, 2010.

[5] Gilbert Laporte and Yves Nobert. Exact algorithms for the vehicle routing problem. *North-Holland Mathematics Studies*, 132:147–184, 1987.

[6] Suresh Nanda Kumar and Ramasamy Panneerselvam. A survey on the vehicle routing problem and its variants. *Intelligent Information Management*, 4, 2012.

[7] Anna Maria Sri Asih, Bertha Maya Sopha, and Gilang Kriptaniadewa. Comparison study of metaheuristics: Empirical application of delivery problems. *International Journal of Engineering Business Management*, 9(1847979017743603), 2017.

[8] Caroline Prodhon Nacima Labadie, Christian Prins. *Metaheuristics for Vehicle Routing Problems*. John Wiley & Sons, 2016. ISBN: 9781119136767.

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

[10] Guy Desaulniers, Jacques Desrosiers, Yvan Dumas, Marius Solomon, and François Soumis. Daily aircraft routing and scheduling. *Management Science*, 43:841–855, 1997.

[11] Hongqi Li, Yue Lu, Jun Zhang, and Tianyi Wang. Solving the tractor and semi-trailer routing problem based on a heuristic approach. *Mathematical Problems in Engineering*, 2012(182584), 2012.

[12] Michel Gendreau, Jean-Yves Potvin, Olli Braumlaysy, Geir Hasle, and Arne Lokketangen. *Metaheuristics for the Vehicle Routing Problem and Its Extensions: A Categorized Bibliography*. 2008. ISBN: 978-0-387-77778-8.

[13] Tantikorn Pichpibul and Ruengsak Kawtummachai. A heuristic approach based on clarke-wright algorithm for open vehicle routing problem. *The Scientific World Journal*, 2013(874349):11, 2013.

[14] Noraini Razali. An efficient genetic algorithm for large scale vehicle routing problem subject to precedence constraints. *Procedia - Social and Behavioral Sciences*, 195:1922–1931, 2015.

[15] R Jorgensen, Jesper Larsen, and K Bergvinsdottir. Solving the dial-a-ride problem using genetic algorithms. *Journal of the Operational Research Society*, 58(10):1321–1331, 2007.

[16] Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2):315–332, 2016.

[17] Martijn Merwe, James Minas, Melih Ozlen, and John Hearne. The cooperative orienteering problem with time windows. *School of Science, RMIT University, Melbourne, Australia*, 2014.

[18] Aykut Bulut and Ted K. Ralphs. On the complexity of inverse mixed integer linear optimization. *Department of Industrial and Systems Engineering, Lehigh University, USA*, 2015.

[19] C.H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences*, 28(2):244–259, 1984.

[20] Walter N. Polakov Clark, Wallace and Frank W. Trabold. *The Gantt Chart, A Working Tool of Management*. 1922. ISBN: 978-0-2437-5548-6.

[21] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[22] Thomas A Feo and Mauricio G.C Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67–71, 1989.

[23] Mauricio G. C. Resende and Celso C. Ribeiro. *Greedy Randomized Adaptive Search Procedures*. 2003. ISBN: 978-0-306-48056-0.

[24] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[25] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. *ICRA workshop on open source software*, 3:5, Jan 2009.

Appendix A List of abbreviations

In Table 11 are listed abbreviations used in this thesis.

Table 11: Lists of abbreviations

Abbreviation	Meaning
WBP	Wall Building Problem
OP	Orienteering Problem
MBZIRC	Mohamed Bin Zayed International Robotic Challenge
VRP	Vehicle Routing Problem
CVRP	Capacitated Vehicle Routing Problem
VRPTW	Vehicle Routing Problem with Time Windows
COP	Cooperative Orienteering Problem
COPTW	Cooperative Orienteering Problem with Time Windows
DARP	Dial-a-ride Problem
TSRP	Tractor Semi-trailer Problem
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
CW	Clarke-Wright algorithm
GRASP	Greedy Randomized Adapttive Search Procedure
PRNG	Pseudorandom Number Generator
RCL	Restricted Candidate List
LP	Linear Programming
ILP	Integer Linear Programming
MILP	Mixed-Integer Linear Programming
ROS	Robotic Operating System

Appendix B CD content

In Table 12 are listed names of all root directories saved on the CD.

Table 12: Lists of root directories

Directory	Description
code	implementation source code
thesis	thesis source files