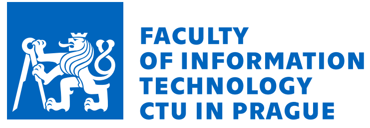


Assignment of bachelor's thesis

Title:	Compilation of Multi-Agent Collective Construction in the Minecraft Game
Student:	Martin Rameš
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2021/2022

Instructions

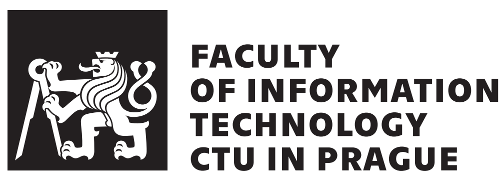
The aim of the work is to formulate and solve multi-agent collective construction as a problem in a selected formalism. The problem consists in planning actions for a team of agents who work together to construct a building, typically in the abstract blocks world. Constraint programming or integer programming are candidate for the target formalism. Specifically, the work will focus on the environment defined by the Minecraft game. The tasks of the student are as follows:

1. Study formalisms for problem modeling, such as constraint programming or integer programming, and related solving systems from the programming point of view.
2. Using the review, formulate the problem of multi-agent collective construction or its part in the selected formalism and solve it with an external solver.
3. Design a set of experiments and test your approach. Visualize the results in the Minecraft environment using a suitable interface.

–

[1] Edward Lam, Peter J. Stuckey, Sven Koenig, T. K. Satish Kumar: Exact Approaches to the Multi-agent Collective Construction Problem. CP 2020: 743-758
[2] Kirstin Petersen, Radhika Nagpal, Justin Werfel: TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction. Robotics: Science and Systems 2011
[3] Philippe Baptiste, Philippe Laborie, Claude Le Pape, Wim Nuijten: Constraint-Based Scheduling and Planning. Handbook of Constraint Programming 2006: 761-799
[4] Matthew Johnson, Katja Hofmann, Tim Hutton, David Bignell. The Malmo platform for artificial intelligence experimentation. IJCAI'16: Proceedings of the Twenty-Fifth International Joint Conference on Artificial IntelligenceJuly 2016 Pages 4246–4247

Electronically approved by Ing. Karel Klouda, Ph.D. on 28 January 2021 in Prague.



Bachelor's thesis

Compilation of Multi-Agent Collective Construction in the Minecraft Game

Martin Rameš

Department of Applied Mathematics
Supervisor: doc. RNDr. Pavel Surynek, Ph.D.

May 13, 2021

Acknowledgements

I hereby thank doc. RNDr. Pavel Surynek, Ph.D., for supervising the creation of this thesis. I also thank Ing. Miroslav Skrbek, Ph.D., for providing access to hardware necessary to perform the computationally intensive parts of this thesis (a powerful server from the Intelligent Embedded Systems Laboratory).

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Martin Rameš. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Rameš, Martin. *Compilation of Multi-Agent Collective Construction in the Minecraft Game*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato bakalářská práce zkoumá současné přístupy k přesným řešením problému multi-agentní kolektivní konstrukce, s důrazem na tří-rozměrné struktury, postavené agenty přenášejícími v mřížce bloky, za předpokladu přítomnosti gravitace. Zobecnění v současné době nejrychlejšího přesného modelu je navrženo, s použitím smíšeného celočíselného lineárního programování, k přizpůsobení se různému trvání kroků agentů. Uplatnění navrženého modelu je použito v kombinaci s řešičem k přesné optimalizaci stavebního plánování uživatelem navržených struktur. Výsledek je vizualizován v Minecraftu, za použití programu postaveného na Malmo API. Série experimetů je provedena na několika malých instancích, k naměření relativního snížení doby stavění vzhledem k jednokrokovým krokům agentů. Výsledky ukazují na výrazné snížení doby stavění při délkách kroků použitých pro vizualizaci v Minecraftu.

Klíčová slova multi-agentní stavění, multi-agentní plánování, smíšené celočíselné lineární programování, projekt Malmo, Minecraft

Abstract

This bachelor thesis studies current exact approaches to multi-agent collective construction problem, with emphasis on three-dimensional structures, built by agents repositioning passive blocks on a grid under the condition of gravity. A generalization of current fastest exact model is proposed, using mixed integer linear programming, to accommodate varying durations of agent actions. An application of the proposed model is used in conjunction with a solver to exactly optimize construction plan of user entered structures. The result is visualized in Minecraft, using program built upon project Malmo API. A series of experiments is performed on several small instances to measure relative construction makespan reduction, relative to the instances with one-timestep actions. Results show significant makespan reduction at action durations used for visualization in Minecraft.

Keywords multi-agent construction, multi-agent planning, mixed integer linear programming, Project Malmo, Minecraft

Contents

Introduction **1**

Goals **3**

1 Background **5**

1.1 Multi-agent Construction 5
1.2 TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction 5
1.3 Environmental Properties 6
1.4 Network Flow 7
1.4.1 Minimum Cost Flow Problem 7
1.5 Constraint Programming 7
1.5.1 Constraint Programming Example 8
1.5.2 Constraint-Based Scheduling and Planning 8
1.5.3 OR-Tools 9
1.6 Mixed Integer Linear Programming 10
1.6.1 MILP example 10
1.6.2 Gurobi Optimizer 11
1.7 Exact Approaches to the Multi-Agent Collective Construction Problem 12
1.7.1 Problem Definition 12
1.7.2 Mixed Integer Linear Programming Model 13
1.7.3 Constraint Programming Model 15
1.7.4 Experimental Results 19
1.8 Minecraft 21
1.8.1 World Generation 21
1.8.2 Player Actions 22
1.8.3 Project Malmo 22
1.8.3.1 Agent Commands 22

1.8.4 Agent Observations 24
1.8.5 Agent Rewards 25

2 Analysis and Design 27
2.1 Problem Instance Solver 28
2.1.1 Construction Fraction-Time Model 28
2.1.1.1 Incorporation of Minecraft and Malmo Control Scheme into the Model 28
2.1.1.2 Construction Fraction-Time MILP Model . . . 29
2.1.2 Problem Instance Assigner 34
2.1.3 Agent Instruction Assigner 34
2.2 Construction Visualizer 35
2.2.1 World Initializer 35
2.2.2 Agent Controller 35

3 Realisation 39
3.1 Implementation 39
3.2 Problem Instance Solver 40
3.2.1 Problem Instance Assigner 42
3.2.2 Agent Instruction Assigner 43
3.3 Construction Visualizer 45
3.3.1 World Initializer 45
3.3.2 Agent Controller 46
3.4 Implementation Challenges 49
3.4.1 Minecraft Local Server Agent Limit 49
3.4.2 Malmo Discrete Movement Commands Non-determinism 49
3.5 Experiments 49
3.5.1 Action Duration Experiment Description 49
3.5.2 Action Duration Experiment Setup 51
3.5.3 Action Duration Experiment Results 51
3.5.3.1 Influence of Action Type Durations on Action Type Count 53
3.5.4 Robot Count Experiment Description and Setup 55
3.5.5 Robot Count Experiment Results 56

Conclusion 61

Bibliography 63

A Acronyms 65

B Appendix 67
B.1 Example Problem Instance 67
B.1.1 Problem Instance Specification File 67

B.1.2 Construction Visualizer Controlled Minecraft Graphical Output 68
B.1.3 Agent Instructions File 71

C Contents of Enclosed CD **75**

List of Figures

1.1 Six problem instances solved in [4] 20
1.2 Example of default Minecraft world generation 21

2.1 Block diagram of the solution 27
2.2 Flowchart of suggested central agent controller 38

3.1 Implementation block diagram 40
3.2 Problem instance solver flowchart 41
3.3 Agent instruction assigner flowchart 44
3.4 Fifth timestep of mission to build a small cube 47
3.5 Three problem instances used for experiments 50
3.6 Makespan and sum-of-costs for instances 1, 2 and 3 53
3.7 Gurobi computational time for instances 1, 2 and 3 54
3.8 Relative instruction counts for the three instances 55
3.9 Cumulative relative instruction execution time for the three instances 56
3.10 Instance 3, used again for experiment with robot count 57
3.11 Makespan and sum-of-costs for decreasing maximum agent count . 57
3.12 Computational runtime for decreasing maximum agent count . . . 58
3.13 Makespan and relative time spent on the grid for decreasing maximum agent count 59

List of Tables

1.1 Blocks world actions 9
1.2 Best available solutions to six problem instances, data from [4] . . 20

3.1 Projection of fraction-time model actions to relative instructions . 43
3.2 Projection from relative instructions to Malmo commands 48
3.3 Action type duration for 1-timestep, 1-2-timestep and 1-2-3-timestep action sets 52
3.4 Experimental results 52

Introduction

Replacement of human labor with robots and automation in general are an undeniable part of modern age. While automation in general has already been used on larger scales during the first industrial revolution, machine-to-machine communication and internet of things have recently allowed for use of machines in professions, which could be previously done only by humans. This trend is sometimes called Fourth Industrial Revolution.

One of the industries, which is currently subjected to attempts at further automation, is construction industry. One of the approaches used for this purpose is multi-agent construction. This method allows, in many cases, for simultaneous construction of several parts of the goal structure and thus reduction in building time. Depending on the required conditions for the given multi-agent solution, the construction task could also be carried out in conditions dangerous to human health or in places normally inaccessible to humans (like the surface of other planetary bodies).

The main motivation behind this thesis is to further exact optimization of multi-agent construction, as this field of study has been relatively neglected in favour of sub-optimal heuristic approaches. The secondary motivation for this thesis is to popularize multi-agent construction. For this purpose, Minecraft, as a widely known sandbox game, was selected as the visualization tool.

This thesis focuses on formulation of multi-agent construction problem in a formalism that allows location of optimal solution, under specified criteria. The problem consists of planning a list of actions for a group of agents, which would lead them to construct a block structure. The solution is then to be visualized in a Minecraft world.

Goals

The main goal of this thesis is to create multi-agent construction system, able to convert user-entered structure to an optimal construction plan, and visualizing the result within Minecraft. To reach this goal, three sub-goals have to be performed.

Study of problem modeling formalisms has to be carried out. Constraint programming and integer programming are both the main candidates for use in the solution, along with their respective solvers. As the solving systems provide a black-box tool for solving problems described in their formalism, they are to be mainly studied from programming point of view.

Information gathered by the review is then to be used to formulate the multi-agent construction problem in selected formalism and solve it with external solver.

A set of experiments is then to be carried out on the solution, testing the approach. Visualization is to be carried out within Minecraft environment, using suitable interface.

Chapter 1

Background

1.1 Multi-agent Construction

Multi-agent collective construction problem has multiple variants. Besides the most obvious division into 2D and 3D version of the problem, the shape of construction blocks and their complexity (as well as the complexity of the agents performing the construction) can greatly differ. To name a few examples, quadrotors have been proposed to build structures from beams and columns[1], a construction model has been made for team of heterogeneous robots to build from multiple materials as passive building blocks (foam blocks and bean bags have been used for experiments, manipulated by four-wheeled robot with a manipulator arm) [2].

This thesis focuses on the 3D variant of the problem, where agents move passive blocks of uniform size, under the condition of gravity. The TERMES system, with its termite-like robots building from specialized blocks, is one of the most prominent examples of multi-agent system addressing this problem.[3] To be precise, this thesis focuses on exact optimization of the construction duration. It should be noted, that there are multiple heuristic approaches to multi-agent construction, which provide solutions without proof of optimality. They are mentioned for completeness and will not be further specified, as this thesis seeks only the optimal solutions. There are already models providing optimal solutions, their specification is provided in section 1.7 (named Exact Approaches to the Multi-Agent Collective Construction Problem). [4]

1.2 TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction

Multi-agent construction system TERMES proposes to use a set of simple agents for autonomous assembly of 3D structures under the condition of gravity. Inspired by social insects (termites in particular), the model suggests to

use a set of low-cost robots that can pick up, transport and place specialized passive building blocks. These blocks have attachment points for being picked up by the robots. The blocks also possess specialized shape and a set of magnets, both of which allow them to slide into place after delivery.[3]

TERMES robots have two pairs of specialized wheels and can climb up (and down) a difference of one building block. They can carry only one block at a time. Robots can be programmed using high-level primitives to execute actions, which are block attachment, block pick up, turning left or right by 90° and moving forward by one block. Different action types have different (mean) execution times (for instance, block pick up requires on average 15 seconds, block delivery/attachment requires on average 24 seconds). [3] TERMES movement style contains all actions of Malmo Discrete Movement Commands and can thereby be simulated using Malmo Platform (if given proper interface). [5]

1.3 Environmental Properties

Environments of the multi-agent systems are divided into different types, based on their properties [6]:

- *Accessible* environment allows agents to access its complete state through the sensory apparatus.
- *Deterministic* environment has the next state completely determined by its current state and actions of the agents.
- *Episodic* environment divides agent's experience into episodes of sensory perception followed by agent actions. Episodes provide independent units, where quality of agent action depends only on the current episode.
- *Static* environment does not change without agent performing an action. If the environment changes in time, without the action of an agent, it is considered to be *dynamic*. In *semidynamic* environments, only performance score of the agent changes in time without agent action.
- *Discrete* environment provides only distinct, clearly defined sensory perceptions and agent actions.

Various types of environments require different agent programs to solve assigned problem efficiently. In accessible deterministic environments, the agent systems can function without uncertainty, which can simplify the approach (depending on the complexity of the environment). [6]

1.4 Network Flow

Network flow is, in the most general sense, a field of study, which focuses on optimizing the movement of entities between points connected by an underlying network. The field has a long history (reaching back to 19. century), in which many different network related problems have been studied. Shortest path, maximum flow and minimum cost flow are all examples of such problems. Although the field is relatively old, it provides multiple algorithms with polynomial worst-case complexities. Those algorithms can be applied to ever growing set of problems, partially thanks to the still growing computational power of the modern day. [7]

1.4.1 Minimum Cost Flow Problem

Minimum cost flow problem is defined as determining a least cost shipment of commodity through a network in order to satisfy demands at certain nodes from supplies available at other nodes. [7] The problem can be alternatively defined using mathematical programming (definition from [7]):

Let $G = (\mathcal{N}, \mathcal{E})$ be a directed network with a node set $\mathcal{N}$ and directed edge set $\mathcal{E}$. Each edge $(i, j) \in \mathcal{E}$ has an associated cost per unit flow c_{ij}, unit flow upper bound u_{ij} and unit flow lower bound l_{ij}. When lower bound is not specified, it is implicitly considered to be $l_{ij} = 0$. Each node $i \in \mathcal{N}$ also has an associated unit supply/demand $b(i)$ ($b(i) > 0$ for supply, $b(i) < 0$ for demand, $b(i) = 0$ for transshipment nodes). The decision variables are edge flows x_{ij} for every edge $(i, j) \in \mathcal{E}$. The objective function is 1.1, with constraints 1.2, 1.3 and 1.4.

$$\min \sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij} \tag{1.1}$$

$$\sum_{\{j:(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j:(j,i)\in\mathcal{A}\}} x_{ji} = b(i), \forall i \in \mathcal{N} \tag{1.2}$$

$$l_{ij} \leq x_{ij} \leq u_{ij}, \forall i, j : (i, j) \in \mathcal{E} \tag{1.3}$$

$$\sum_{i=1}^{n} b(i) = 0 \tag{1.4}$$

1.5 Constraint Programming

"*Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, and operations research.*" [8] It allows to declaratively define the required solution using constraints for a set of decision variables. The user is typically also required to set a search strategy, which usually

uses standard methods like constant propagation. The use of problem specific heuristics is also possible. [8]

Constraint programming (CP) spans a wide area of research, from theoretical to applied. Further text will be focused on one field of constraint programming, which is quite relevant to the focus of this thesis - constraint-based scheduling and planning. Creation of movement instructions for the Minecraft agents can be viewed as a planning problem, where agents are unary resources and actions are possible agent moves. [8]

1.5.1 Constraint Programming Example

An excellent example is described in [8] and it is summarized in the following subsection. Let there be a blocks world with the following propositions:

- $clear(X)$ – no block is on block X
- $onT(X)$ – block X is on the table
- $on(X, Y)$ – block X is on block Y

The planning problem starts with the initial state $[on(C, A) \wedge onT(A) \wedge onT(B)]$ and its goal state is $[on(A, B) \wedge on(B, C)]$. Allowed actions (operators of the planning domain) are specified in table 1.1.

One way to solve given problem is to create a planning graph and search it for a plan, which achieves the goal and starts at initial state, while remaining 'mutex-free'. Mutex-free means, that there are no mutually exclusive (mutex) propositions or actions within the plan. Two actions are mutex, if they are to be executed in the same timestep and either action deletes a precondition or add-effect of the other. Two propositions (partial world states) are mutex, if all possible actions for reaching one of them are mutex with all possible actions to reach the second one. Furthermore, 'persistence' action, which keeps proposition unchanged from one timestep to the next, is added. For three timesteps deep graph, the solution is found – sequence $BT(C, A)$, $TB(B, C)$, $TB(A, B)$.

1.5.2 Constraint-Based Scheduling and Planning

The following five paragraphs summarize information about constraint-based scheduling and planning described by [8]:

Constraint-based scheduling requires time-based allocation of scarce resources to a specified set of activities, while satisfying given constraints. Constraint-based planning works with similar premise, but its requirements do not specify the full set of activities to use, so decision, which activities to schedule, is also part of the planning process.

There are three main types of scheduling – non-preemptive scheduling, preemptive scheduling and elastic scheduling. In non-preemptive scheduling,

BB(X, Y, Z)	Move X from atop Y to atop Z
preconditions:	$clear(X) \wedge clear(Z) \wedge on(X,Y)$
add-list:	$clear(Y) \wedge on(X,Z)$
delete-list:	$clear(Z) \wedge on(X,Y)$
TB(X, Y)	Move X from the table to atop Y
preconditions:	$clear(X) \wedge clear(Y) \wedge onT(X)$
add-list:	$on(X,Y)$
delete-list:	$clear(Y) \wedge onT(X)$
BT(X, Y)	Move X from atop Y to the table
preconditions:	$clear(X) \wedge on(X,Y)$
add-list:	$clear(Y) \wedge onT(X)$
delete-list:	$on(X,Y)$

Table 1.1: Blocks world actions

the activity has a start time, the end time and cannot be interrupted. Preemptive scheduling is similar, but allows activities to be interrupted (i.e. for some activity to execute during a pause of another activity execution). Elastic scheduling allows to assign anywhere from 0 to full resource capacity to an activity, anytime during its execution, as long as sum over time of the provided capacity equals to energy, required by the activity.

Resources can also be divided into main categories. Unary resources can be used by at most one activity at a time. Cumulative resources can execute as many activities as necessary, provided the resource capacity is not exceeded. Reservoirs are like cumulative resources, but their capacity is produced (and consumed) by activities. State resources have infinite capacity and a state, that can change over time (can be used to force two activities to execute at different times).

There are two main approaches for using CP in planning. First refines a partial plan, made of temporal activity network. Second compiles the problem to Constraint Satisfaction Problem (CSP) and solves it using CSP or SAT solver.

When a schedule or a plan is computed, some variables are more constrained than others and focus on them can lead quickly to a solution. Critical path is defined as a sequence of activities in chronological order, where each activity cannot start, before the previous ends, and the sum of processing times of sequence activities equals the makespan of the schedule.

1.5.3 OR-Tools

OR-Tools is an open source software suite for combinatorial optimization, developed by Google and provided under Apache 2.0 licence. The suite contains a constraint programming solver, unified interface to multiple linear programming and mixed integer solvers (such as Gurobi or CPLEX), bin packing,

knapsack, traveling salesman problem algorithms, vehicle routing problem algorithms and graph algorithms. OR-Tools can be used with multiple programming languages, namely Python, Java, C++ and C#.[9]

1.6 Mixed Integer Linear Programming

Integer programming refers to a class of optimization problems with constrains, where at least some variables must have integer values. 'Mixed' indicates, that some of the variables do not have to be integer. Special subclass of integer programming, using only linear objective function and linear inequalities for constraints, is called mixed integer linear programming (MILP). [10]

The search mechanism used to solve MILP problems is called *branch-and-bound* and uses a linear programming (LP) relaxation of the problems, along with branching on non-integer variables to find integer solution of the problem. Cutting planes method is used to remove subsets of non-integer solutions from the relaxation. [8]

1.6.1 MILP example

An excellent example, demonstrating workings of branch-and-bound algorithm, is part of Gurobi tutorial [11]: Let equation 1.5 be the objective function and 1.6, 1.7, 1.8 and 1.9 be the constraints of an example problem.

$$max(5.5x_1 + 2.1x_2) = z \tag{1.5}$$

$$-x_1 + x_2 \leq 2 \tag{1.6}$$

$$8x_1 + 2x_2 \leq 17 \tag{1.7}$$

$$x_1, x_2 \geq 0 \tag{1.8}$$

$$x_1, x_2 \in \mathbb{Z} \tag{1.9}$$

To solve the problem, LP relaxation is calculated first. Linear programming relaxation solves the problem with the last constraint (1.9) removed and produces upper bound for the objective function $UB_z = 14.08$ for variable values $x_1 = 1.3 \land x_2 = 3.3$. When the solution of the relaxed problem also meets the integer requirement of the last constraint, the search can end there. Otherwise, branch-and-bound algorithm is used.

Branch-and-bound algorithm first chooses one of variables with non-integer value (x_i with value v_i), which is part of integer constraint 1.9, adds a constraint $x_i \leq \lfloor v_i \rfloor \lor \lceil v_i \rceil \leq x_i$. This constraint divides the problem into two

sub-problems, while keeping all candidate integer solutions and removing the original relaxed problem solution. The procedure (computing LP relaxation and, for non-integer solutions choosing not-yet-selected variable x_i and adding described constraint) is then repeated, until integer solution is found (or the problem is proven to contain no integer solutions and therefore to be infeasible). Integer solutions (in this context – solutions satisfying 1.9 constraint) of the sub-problems provide lower bounds for the the final value of the objective function. When two integer solutions are found, the one with higher objective function value is propagated, until only one solution remains. Optimal solution is found, when lower bound equals upper bound of the solution.

In this example, x_1 is the first variable used by branch-and-bound algorithm. Thus, two sub-sets of candidate solutions are created (set $\mathcal{P}_1$ for $x_1 \leq \lfloor 1.3 \rfloor$ and set $\mathcal{P}_2$ for $\lceil 1.3 \rceil \leq x_1$). Solution of $\mathcal{P}_1$ has integer variable values ($x_1 = 1 \wedge x_2 = 3$) and therefore provides lower bound of the objective function ($z = 11.8$) for the following search. Maximization of the objective function for $\mathcal{P}_2$ leads to not-fully-integer variable values $x_1 = 2 \wedge x_2 = 0.5$ with the objective function $z = 12.05$. Since the objective function of the relaxed sub-problem (and therefore upper bound of the objective function) is higher then the lower bound discovered earlier, further branching is necessary. In this case, branching happens on x_2, leading to two sub-problems ($\mathcal{P}_3$ for $x_2 \leq \lfloor 0.5 \rfloor$ and $\mathcal{P}_4$ for $\lceil 0.5 \rceil \leq x_2$). Sub-problem $\mathcal{P}_4$, even with LP relaxation, contains no solutions, and is therefore no longer part of subsets with potential optimal solution. Objective function value for LP relaxation of $\mathcal{P}_3$ is $z = 11.6875$, which is lower than the lower bound for the optimal solution. $\mathcal{P}_3$ can therefore also be discarded, as it cannot contain the optimal solution. With this step, all sub-sets of candidate solutions have been explored. The optimal solution is $x_1 = 1 \wedge x_2 = 3$ with the objective function value $z = 11.8$.

1.6.2 Gurobi Optimizer

Gurobi Optimizer is a powerful mathematical programming solver. It is able to solve linear programming, quadratic programming, mixed integer linear programming, mixed integer quadratic programming, quadratically constrained programming and mixed integer quadratically constrained programming problems. Gurobi Optimizer also provides interfaces to multiple modeling and programming languages, namely C++, Java, .NET (C#), Python, C, MATLAB, R and modeling languages AIMMS, AMPL and MPL.[12] Gurobi Optimizer uses a variation of branch-and-bound algorithm, called branch-and-cut, for solving assigned MILP problems. [11] Aside from the main algorithm, numerous heuristics are used to reduce solving time. [13] For the purposes of this thesis, the most significant heuristics are network flow oriented, as they are attributed to significant computational speed advantage of a MILP model this thesis will use as a base for generalization [4].

1.7 Exact Approaches to the Multi-Agent Collective Construction Problem

The paper defines a multi-agent construction problem based on the TERMES system, using mixed integer linear programming model and constraint programming model. Both mixed integer linear programming and constraint programming are used for exact optimization of the construction task described by the models.[4]

1.7.1 Problem Definition

The following paragraph will be a direct quotation of problem definition, as it appeared in [4]:

> Consider a planning horizon of $T \in \mathbb{Z}_+$ timesteps, and let $\mathcal{T} = \{0, \ldots, T-1\}$ be the set of timesteps. The problem is stated on a three-dimensional grid that is divided into cells. Let the grid be $X \in \mathbb{Z}_+$ cells wide, $Y \in \mathbb{Z}_+$ cells deep and $Z \in \mathbb{Z}_+$ cells high. Let $\mathcal{X} = \{0, \ldots, X-1\}$, $\mathcal{Y} = \{0, \ldots, Y-1\}$ and $\mathcal{Z} = \{0, \ldots, Z-1\}$ be the sets of coordinates in the three dimensions. Define $\mathcal{C} = \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ as the set of all cells. Then, every cell $(x, y, z) \in \mathcal{C}$ is a triple of coordinates in the grid. Define the border cells $\mathcal{B} = \{(x, 0, 0) : x \in \mathcal{X}\} \cup \{(x, Y-1, 0) : x \in \mathcal{X}\} \cup \{(0, y, 0) : y \in \mathcal{Y}\} \cup \{(X-1, y, 0) : y \in \mathcal{Y}\}$ as the perimeter cells on the ground level. Define the position $\mathcal{P} = \mathcal{X} \times \mathcal{Y}$ as the projection of the cells onto the first two dimensions. That is, the positions lie on the two-dimensional grid corresponding to the top-down view of the three-dimensional grid. Define the neighbors of a position $(x, y) \in \mathcal{P}$ as the set of positions $\mathcal{N}_{(x,y)} = \{(x-1, y), (x+1, y), (x, y-1), (x, y+1)\} \cap \mathcal{P}$.

Consider a problem with $A \in \mathbb{Z}_+$ cell sized identical robots. Each robot can carry up to one block at a time. Blocks are the size of a cell. All agents must start and end off the grid. Robots enter the grid through a border cell (with or without holding a block). During every timestep $t \in \mathcal{T}$ can each robot perform six kinds of actions, as long as both the start and end positions of the action are 'valid' [4]:

- Enter grid at border cell
- Move to neighbor position
- Wait at current position for one timestep
- Leave grid at border cell
- Pick up block from neighbor cell

- Deliver block to neighbor cell

In case of pick-up-action and deliver-action, end position refers to the position of the affected block. Robot's end position for these actions matches robot start position. All start positions refer to robot position at the start of specific timestep and end positions for enter-, move-, wait- and leave-actions refer to robot positions at the end of the timestep. Robot start position is 'valid', if it matches end position of any of the robots for previous timestep, is located at $(x, y, z_b + 1)$ where z_b is position of highest block in column at position (x, y) ($z_b = 0$ iff there are no blocks at (x, y)). Robot end position is 'valid', if outside the grid, when robot started move-action at the border cell, or is at a position $(x_2, y_2, z_{b2} + 1)$ iff $(x_2, y_2) \in \mathcal{N}_{(x,y)}$ and $|z_{b2} - z_b| \leq d$, where $d = 0$ for pick up and deliver actions and $d = 1$ for move actions, and did not lead to vertex or edge collision with another robot (i.e. robots did not share a position at one timestep and did not exchange positions in one timestep). [4]

1.7.2 Mixed Integer Linear Programming Model

The following set (1 - 16) of equations makes up the original MILP model. This model will be, from this point onward, called 'constant-time model' (for its constant-time actions), whenever it would not be implicitly obvious, which model is meant. This entire subsection takes information from [4]. Variable $r_i \in \{0, 1\}, i = (t, x, y, z, a, x', y', z')$, is an indicator, which determines, if robot at timestep t performs action a from position (x, y, z) to position (x', y', z'). Variable $h_j \in \{0, 1\}, j = (t, x, y, z, z')$ is an indicator, which equals to 1 if and only if z is the height of the highest block on (x, y) at the start of timestep t, z' is the height of the highest block on (x, y) at the end of timestep t. $\mathcal{R}$ is the set of all agent actions, $\mathcal{H}$ is the set of all one-timestep block height transitions. Value of $\overline{z}_{(x,y)}$ equals to the desired height of structure being constructed, at specified (x, y) position at the end of the building process (timestep $T-1$). Set with index is used to denote subset, where every written value must match the set value at matching position and '*' is a wildcard for any value at matching position (example of usage - $\mathcal{H}_{*,x_1,y_1,*,*}$ means subset of $\mathcal{H}$, where $x = x_1 \land y = y_1$).

$$\min \sum_{i=(t,x,y,z,c,a,x',y',z')\in\mathcal{R}:(x,y,z)\neq(\mathrm{S},\mathrm{S},\mathrm{S})} r_i \tag{1.10}$$

First equation (1.10) denotes the objective function (minimizes the sum-of-costs), which is the total number of timesteps the robots spend on the grid (summed over all robots). The following equations (1.11 – 1.25) describe the constraints.

$$h_{t,x,y,z,z} = 1, \forall t \in \{0, \ldots, T-3\}, (x, y, z) \in \mathcal{B} \tag{1.11}$$

$$h_{0,x,y,0,0} = 1, \forall (x, y) \in \mathcal{P} \tag{1.12}$$

$$h_{T-2,x,y,\overline{z}_{(x,y)},\overline{z}_{(x,y)}} = 1, \forall (x,y) \in \mathcal{P} \tag{1.13}$$

$$\sum_{i \in \mathcal{H}_{t,x,y,*,z}} h_i = \sum_{\mathcal{H}_{t+1,x,y,z,*}} h_i, \forall t \in \{0,\dots,T-3\}, (x,y,z) \in \mathcal{C} \tag{1.14}$$

$$\sum_{i \in \mathcal{H}_{t,x,y,*,*}} h_i = 1, \forall t \in \{0,\dots,T-2\}, (x,y) \in \mathcal{P} \tag{1.15}$$

Constraint 1.11 forbids placement of blocks within the border cells to avoid blocking agent access and exit routes. Constraint 1.12 ensures no blocks are placed before the start of the mission. Constraint 1.13 guarantees the completion of the structure. Constraint 1.14 flows grid structure heights from one timestep to the next. Constraint 1.15 disallows more than one height for each position in each timestep, making it impossible to use this model to create roofed and multi-floor structures.

$$\begin{aligned} &\sum_{i \in \mathcal{R}_{t,*,*,*,0,\mathrm{M},x,y,z}} r_i + \sum_{i \in \mathcal{R}_{t,x,y,z,1,\mathrm{D},*,*,*}} r_i \\ &\quad = \sum_{i \in \mathcal{R}_{t+1,x,y,z,0,\mathrm{M},*,*,*}} r_i + \sum_{i \in \mathcal{R}_{t+1,x,y,z,0,\mathrm{P},*,*,*}} r_i, \\ &\qquad \forall t \in \{0,\dots,T-3\}, (x,y,z) \in \mathcal{C} \end{aligned} \tag{1.16}$$

$$\begin{aligned} &\sum_{i \in \mathcal{R}_{t,*,*,*,1,\mathrm{M},x,y,z}} r_i + \sum_{i \in \mathcal{R}_{t,x,y,z,0,\mathrm{P},*,*,*}} r_i \\ &\quad = \sum_{i \in \mathcal{R}_{t+1,x,y,z,1,\mathrm{M},*,*,*}} r_i + \sum_{i \in \mathcal{R}_{t+1,x,y,z,1,\mathrm{D},*,*,*}} r_i, \\ &\qquad \forall t \in \{0,\dots,T-3\}, (x,y,z) \in \mathcal{C} \end{aligned} \tag{1.17}$$

$$\begin{aligned} &\sum_{i \in \mathcal{R}_{t,x,y,*,*,*,*,*,*}} r_i + \sum_{i \in \mathcal{R}_{t,*,*,*,*,\mathrm{P},x,y,*}} r_i + \sum_{i \in \mathcal{R}_{t,*,*,*,*,\mathrm{D},x,y,*}} r_i \leq 1, \\ &\qquad \forall t \in \{1,\dots,T-2\}, (x,y) \in \mathcal{P} \end{aligned} \tag{1.18}$$

$$\begin{aligned} &\sum_{i \in \mathcal{R}_{t,x,y,*,*,\mathrm{M},x',y',*}} r_i + \sum_{i \in \mathcal{R}_{t,x',y',*,*,\mathrm{M},x,y,*}} r_i \leq 1, \\ &\qquad \forall t \in \{1,\dots,T-2\}, (x,y) \in \mathcal{P}, (x',y') \in \mathcal{N}_{(x,y)} \end{aligned} \tag{1.19}$$

$$\sum_{i \in \mathcal{R}_{t,*,*,*,*,*,*,*,*}} r_i \leq A, \forall t \in \mathcal{T} \tag{1.20}$$

Constraints 1.16–1.20 govern the robot actions. Constraint 1.16 flows robot without block in and out of a cell, so robots located at a cell at the end of one timestep are starting their actions in the same location at the start of the next timestep. Constraint 1.17 does the same for robots carrying a block. Constraint 1.18 prevents multiple robots from standing in the same cell at the same time (vertex collisions). Constraint 1.19 prevents edge collisions (robots exchanging positions). Constraint 1.20 provides an upper limit to the number of robots.

$$\sum_{i \in \mathcal{H}_{t,x,y,z,*}} h_i \geq \sum_{i \in \mathcal{R}_{t,x,y,z,*,*,*,*,*}} r_i, \quad \forall t \in \{0, \ldots, T-2\}, (x,y,z) \in \mathcal{C} \tag{1.21}$$

$$h_{t,x,y,z+1,z} = \sum_{i \in \mathcal{R}_{t,*,*,*,0,\mathrm{P},x,y,z}} r_i, \quad \forall t \in \{0, \ldots, T-2\}, (x,y) \in \mathcal{P}, z \in \{0, \ldots, Z-2\} \tag{1.22}$$

$$h_{t,x,y,z,z+1} = \sum_{i \in \mathcal{R}_{t,*,*,*,1,\mathrm{D},x,y,z}} r_i, \quad \forall t \in \{0, \ldots, T-2\}, (x,y) \in \mathcal{P}, z \in \{0, \ldots, Z-2\} \tag{1.23}$$

Constraints 1.21–1.23 connect pillar heights with robot actions. Constraint 1.21 guarantees that when robot is at cell (x, y, z), then the height of the pillar of blocks at position (x, y) is z. Constraint 1.22 equates the decrease in pillar height to pick up actions. Constraint 1.23 equates the increase of pillar height to delivery actions. Constraints 1.24 and 1.25 specify the variable domains.

$$h_i \in \{0, 1\}, \forall i \in \mathcal{H} \tag{1.24}$$

$$r_i \in \{0, 1\}, \forall i \in \mathcal{R} \tag{1.25}$$

1.7.3 Constraint Programming Model

Constraint programming (CP) model uses similar structure to MILP model to eliminate robot symmetry. Main difference from MILP model is that CP model does not use vertical dimensions, block carrying state and action state of the network flow used by MILP model. Those parts of the MILP model are replaced with logical and ELEMENT constraints to better fit CP strengths.[4]

Each position $(x, y) \in \mathcal{P}$ is assigned an identifier $i = Y \cdot x + y$ and $\mathcal{I} \in \{0, \ldots, X \cdot Y - 1\}$ is the set of all position identifiers. Value of $\overline{z}_i$ equals desired height at position i. Value of $r_{t,i}$ is equal to action used at timestep

t and position i (M indicates robot moving/waiting action, B indicates block pick-up/delivery and U means robot is not present). $\mathcal{O} = \{-1, -2\}$ are start and end positions outside the grid. Let $\mathcal{E} = \mathcal{I} \cup \mathcal{O}$ be the set of all positions. $\mathcal{N}_i$ is a set of neighbors of position i on the grid (Manhattan distance 1 in (x, y) coordinates). $\mathcal{B}$ is set of border cells and $\overline{\mathcal{B}}$ are grid cells not in border. Robot position at next timestep is $n_{t,i} \in \mathcal{E}$, $b_{t,i} \in \mathcal{I}$ is a position of the picked up / delivered block. Variables $c_{t,i}, p_{t,i}, d_{t,i} \in \{0,1\}$ indicate that the robot is carrying a block, is picking up a block and that robot is delivering a block, respectively.[4]

$$\mathcal{N}_i^{\mathcal{E}} = \begin{cases} \mathcal{N}_i, & i \in \overline{\mathcal{B}} \\ \mathcal{N}_i \cup \mathcal{O}, & i \in \mathcal{B} \end{cases} \tag{1.26}$$

Following equations (1.27 – 1.64) and their descriptions define the CP model and use information from [4]:

$$\min \sum_{t \in \mathcal{T}} \sum_{i \in \mathcal{I}} (r_{t,i} \neq \mathrm{U}) \tag{1.27}$$

Equation 1.27 is the objective function and minimizes the number of occupied positions at all timesteps. Following are the constrain equations. Constraints 1.40, 1.42, 1.45 and 1.51–1.64 use Element global constraint.

$$h_{t,i} = 0, \forall t \in \mathcal{T}, i \in \mathcal{O} \tag{1.28}$$

$$h_{t,i} = 0, \forall t \in \mathcal{T}, i \in \mathcal{B} \tag{1.29}$$

$$h_{t,i} = 0, \forall t \in \{0, 1\}, i \in \mathcal{I} \tag{1.30}$$

$$h_{t,i} = \overline{z}_i, \forall t \in \{T-2, T-1\}, i \in \mathcal{I} \tag{1.31}$$

$$h_{t,i} - 1 \leq h_{t+1,i} \leq h_{t,i} + 1, \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \tag{1.32}$$

Constraint 1.28 sets unchangeable height to off-grid positions. Constraint 1.29 forbids placement of blocks within the border. Constraint 1.30 ensures no blocks are present at mission start. Constraint 1.31 makes sure the structure is completed before mission ends. Constraint 1.32 forces column heights to change by at most one block in a single timestep.

$$r_{t,i} = \mathrm{M}, \forall t \in \mathcal{T}, i \in \mathcal{O} \tag{1.33}$$

$$n_{t,i} = i, \forall t \in \mathcal{T}, i \in \mathcal{O} \tag{1.34}$$

$$c_{t,-1} = 1, \forall t \in \mathcal{T} \tag{1.35}$$

$$c_{t,-2} = 0, \forall t \in \mathcal{T} \tag{1.36}$$

Constraints 1.33–1.36 fixate the robot variables, when located outside the grid. Constraints 1.37 and 1.38 keep robots off the grid during the first and the last timestep. Constraint 1.39 dictates that the robot must stay at current position when picking up or delivering a block. Constraint 1.40 keeps the block-carrying state of the robot during movement. Constraint 1.41 changes block carrying state of the robot after block pick up or delivery.

$$r_{0,i} = \mathrm{U}, \forall i \in \mathcal{I} \tag{1.37}$$

$$r_{T-1,i} = \mathrm{U}, \forall i \in \mathcal{I} \tag{1.38}$$

$$(r_{t,i} = \mathrm{B}) \rightarrow (n_{t,i} = i), \forall t \in \mathcal{T}, i \in \mathcal{I} \tag{1.39}$$

$$(r_{t,i} = \mathrm{M}) \rightarrow (c_{t+1,n_{t,i}} = c_{t,i}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \tag{1.40}$$

$$(r_{t,i} = \mathrm{B}) \rightarrow (c_{t+1,i} = \neg c_{t,i}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \tag{1.41}$$

Constraint 1.42 states that, during each timestep, a position is either unoccupied or robot at the position must perform an action. Constraint 1.43 states, that robots in all on-grid non-border positions must have reached those positions from the neighboring positions in the previous timestep. Constraints 1.44 and 1.45 prevent robot vertex and edge collisions, respectively. Constraint 1.46 limits the number of robots on the grid.

$$(r_{t,i} = \mathrm{U}) \vee (r_{t+1,n_{t,i}} \neq \mathrm{U}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \tag{1.42}$$

$$(r_{t+1,i} = \mathrm{U}) \vee \bigvee_{j \in \mathcal{N}_i \cup \{i\}} (r_{t,j} \neq \mathrm{U} \wedge n_{t,j} = i), \forall t \in \{0, \ldots, T-2\}, i \in \overline{\mathcal{B}} \tag{1.43}$$

$$\sum_{j \in \mathcal{N}_i \cup \{i\}} (r_{t,j} = \mathrm{M} \wedge n_{t,j} = i) + (r_{t,i} = \mathrm{B}) + \sum_{j \in \mathcal{N}_i} (r_{t+1,j} = \mathrm{B} \wedge b_{t+1,j} = i) \leq 1,$$
$$\forall t \in \{1, \ldots, T-2\}, i \in \mathcal{I} \tag{1.44}$$

$$(r_{t,i} = \mathrm{M} \wedge n_{t,i} \neq i \wedge r_{t,n_{t,i}} = \mathrm{M}) \rightarrow (n_{t,n_{t,i}} \neq i), \forall t \in \{1, \ldots, T-2\}, i \in \mathcal{I} \tag{1.45}$$

$$\sum_{i \in \mathcal{I}} (r_{t,i} \neq \mathrm{U}) + \sum_{i \in \mathcal{B}} (r_{t-1,i} = \mathrm{M} \wedge n_{t-1,i} < 0) \leq A, \forall t \in \{1, \ldots, T-1\} \quad (1.46)$$

Constraints 1.47 and 1.48 govern block pick up and delivery (for pick up indicator, the agent must start carrying a block, for delivery agent must stop carrying a block). Constraint 1.49 requires the vertical robot position to change by at most one block in one timestep. Constraint 1.50 states, that height of the robot position must remain the same during wait actions.

$$p_{t,i} \leftrightarrow (r_{t,i} = \mathrm{B} \wedge c_{t+1,i} \wedge \neg c_{t,i}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.47)$$

$$d_{t,i} \leftrightarrow (r_{t,i} = \mathrm{B} \wedge \neg c_{t+1,i} \wedge c_{t,i}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.48)$$

$$(r_{t,i} = \mathrm{M}) \rightarrow (h_{t,i} - 1 \leq h_{t+1,n_{t,i}} \leq h_{t,i} + 1, \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.49)$$

$$(r_{t,i} = \mathrm{M} \wedge n_{t,i} = i) \rightarrow (h_{t+1,i} = h_{t,i}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.50)$$

Constraint 1.51 forces the height of the picked up block to match the vertical position of the robot performing the action. Constraint 1.52 decreases block column height by 1 after pick up. Constraints 1.53 and 1.54 do the equivalent of 1.51 and 1.52 for deliveries.

$$p_{t,i} \rightarrow (h_{t,b_{t,i}} = h_{t,i} + 1), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.51)$$

$$p_{t,i} \rightarrow (h_{t+1,b_{t,i}} = h_{t,b_{t,i}} - 1), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.52)$$

$$d_{t,i} \rightarrow (h_{t,b_{t,i}} = h_{t,i}), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.53)$$

$$d_{t,i} \rightarrow (h_{t+1,b_{t,i}} = h_{t,b_{t,i}} + 1), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.54)$$

Constraint 1.55 ensures each block column height reflects pick-up-actions and deliveries performed on in in the previous timestep. Constraints 1.56 and 1.57 improve the filtering.

$$h_{t+1,i} = h_{t,i} - \sum_{j \in \mathcal{N}_i} (p_{t,j} \wedge b_{t,j} = i) + \sum_{j \in \mathcal{N}_i} (d_{t,j} \wedge b_{t,j} = i), \\ \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.55)$$

$$h_{t+1,i} = h_{t,i} - 1 \rightarrow \bigvee_{j \in \mathcal{N}_i} (p_{t,j} \wedge b_{t,j} = i), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.56)$$

$$h_{t+1,i} = h_{t,i} + 1 \rightarrow \bigvee_{j \in \mathcal{N}_i} (d_{t,j} \wedge b_{t,j} = i), \forall t \in \{0, \ldots, T-2\}, i \in \mathcal{I} \quad (1.57)$$

Constraints 1.58–1.64 specify variable domains. Constraint 1.60 requires robot action end position to be the robot current position, its neighboring position or off the grid. Constraint 1.61 limits pick up and delivery end positions to the neighboring positions of the robot.

$$h_{t,i} \in \mathcal{Z}, \forall t \in \mathcal{T}, i \in \mathcal{E} \quad (1.58)$$

$$r_{t,i} \in \mathcal{K}, \forall t \in \mathcal{T}, i \in \mathcal{E} \quad (1.59)$$

$$n_{t,i} \in \mathcal{N}_i^{\mathcal{E}} \cup \{i\}, \forall t \in \mathcal{T}, i \in \mathcal{E} \quad (1.60)$$

$$b_{t,i} \in \mathcal{N}_i, \forall t \in \mathcal{T}, i \in \mathcal{I} \quad (1.61)$$

$$c_{t,i} \in \{0,1\}, \forall t \in \mathcal{T}, i \in \mathcal{E} \quad (1.62)$$

$$p_{t,i} \in \{0,1\}, \forall t \in \mathcal{T}, i \in \mathcal{I} \quad (1.63)$$

$$d_{t,i} \in \{0,1\}, \forall t \in \mathcal{T}, i \in \mathcal{I} \quad (1.64)$$

1.7.4 Experimental Results

This subsection takes information from [4]. Gurobi optimizer has been used to solve the problem described by MILP model. OR-Tools have been selected for problem in constraint programming model. Both solvers were run on six different problem instances (image 1.1). As can be seen in the table 1.2, for all but one problem instance was CP model solver unable to find optimal solution within given time limit (seven days). MILP model solver found optimal solution for all six problem instances. For the one problem instance, where OR-Tools found the optimal solution, MILP solver required 3 seconds, while CP solver required 1.2 hours.

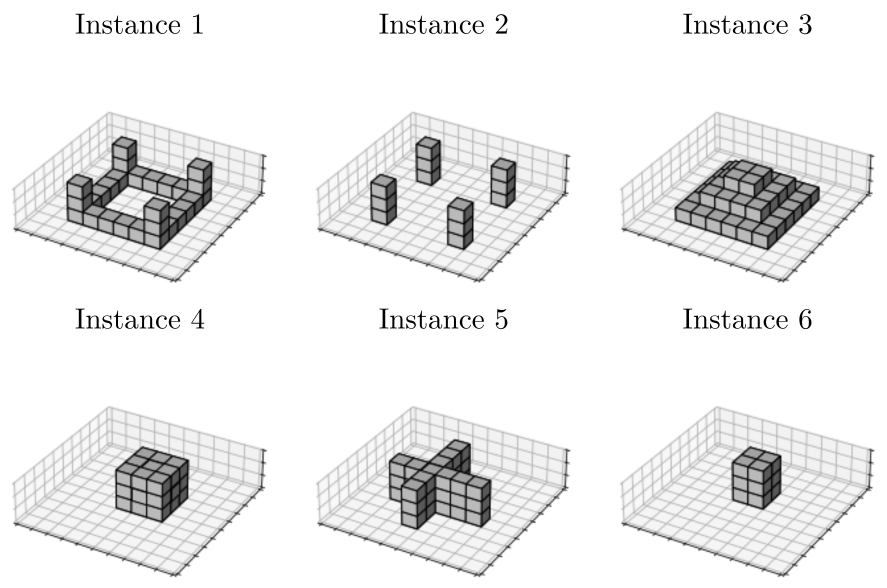


Figure 1.1: Six problem instances solved in [4]

Model	Instance	Run-time	Makespan	Sum-of-costs	Sum-of-costs LB	Robots
MILP	1	29s	11	176	176	34
	2	3s	11	128	128	28
	3	1.2hr	13	344	344	44
	4	5.5hr	17	429	429	42
	5	5.7d	17	368	368	37
	6	183s	15	234	234	27
CP	1	>7d	11	178	107	30
	2	1.2hr	11	128	128	28
	3	>7d	13	354	164	44
	4	>7d	17	452	189	50
	5	>7d	17	395	39	41
	6	>7d	15	245	154	28

Table 1.2: Best available solutions to six problem instances, data from [4]

Probable reason for significant MILP computational speed advantage is exploitation of two network flows, located within the problem. Constrain programming model lacked any similar exploitable structure.

Biggest obstacle, when scaling up the MILP model, is rapidly rising computational complexity, when the makespan increases. This also limits maximum height of the structure, as higher located blocks require time-expensive building of access ramps.

1.8 Minecraft

Minecraft[14] is a popular sandbox game, developed by Mojang Studios. The gameplay allows players to explore 3D procedurally generated terrain, which consists of different types of cubic blocks (1 meter in size; some types of the blocks only partially fill these cubes). [15] World creation allows generation of five types of worlds – Default, Amplified, Large Biomes, Superflat and Customized. [16]

Figure 1.2: Example of default Minecraft world generation

1.8.1 World Generation

Default world generates a complicated terrain, based on pseudo-randomly selected placement of biomes. En example of such generation can be seen in 1.2, with 'Savanna' biome in the foreground and 'Savanna M' in the background – indicated by significantly increased terrain height. It is mostly used for survival game play, which consists of exploration, resource gathering, combat with hostile mobile entities ('hostile mobs') and building. Amplified world is similar to the default one, but the terrain height of specific biomes is increased. Large Biomes world generation differs from the default one only in increasing the size of biomes in X and Z axis by 4. Both world generations are also used for survival gameplay.[16]

The remaining two world types are Superflat and Customized. Superflat generates a flat world with specified layers of blocks at the bottom. It is mostly used for building. Customized world allows for a large amount of customization of world generation constants. This world type can be used for experimentation with world generation.[16]

1.8.2 Player Actions

Available player actions depend on the current game mode. There are four game modes – adventure, survival, creative and spectator. The first three modes allow the player to move as a gravity-affected entity with collisions defined by a collision-box around it. Spectator game mode serves as a means for observation without affecting the environment. It allows three-axis collisionless movement.[17]

Survival game mode is based around resource gathering. Player only has access to picked up items and their respective amounts, inventory space for those items is limited. Player can place blocks, break them and combine items into other items and tools. Some blocks require tools to drop in a pickable form after being destroyed.[17]

Creative mode allows the same player actions as the survival mode, but placed blocks are not subtracted from player inventory and blocks directly broken by the player do not drop in a pickable form, regardless of tools used. Player has access to all types of game blocks.[17]

Adventure mode does not allow the player to directly place or destroy any block, unless tool with property tag explicitly allowing such a thing is used.[17]

1.8.3 Project Malmo

Malmo is an AI experimentation platform, built on top of the game Minecraft. The project platform can support research in robotics, computer vision, reinforcement learning, planning, multi-agent systems and similar research areas. [5] It is the only Minecraft game modification ('mod') that is officially supported by Mojang. [18]

Malmo provides abstraction layer and API on top of the Minecraft game for multiple programming languages [18]. The platform allows multiple environment observation schemes, multiple agent control schemes, multiple agent reward schemes (mostly for reinforcement learning), multiple mission ending schemes, as well as world generation presets and agent initial state management. [5]

1.8.3.1 Agent Commands

All agent commands have the form of `"command <values>"`, passed to the agent by `agent_host.sendCommand(commandString)`. Agents can be controlled by any of the following control schemes, including their combinations [19]:

- Continuous Movement Commands
- Absolute Movement Commands
- Discrete Movement Commands

- Inventory Commands
- Chat Commands
- Simple Craft Commands
- Mission Quit Commands – terminate the current mission
- Turn Based Commands
- Human Level Commands

Continuous movement commands that control smooth movement of the agent. Available commands are move, strafe, pitch, turn, jump, crouch, attack and use. These commands do not have any requirements for the agent, but their results depend on the environment around the agent. For instance, `"move 0.5"` will cause the agent to move forward at 50 % of normal walking speed ('normal' for the Minecraft player at 4.317 m/s [17]), but collisions with other entities, flowing water and other environmental effects can cause the agent to ultimately move at a different speed. [19]

Absolute movement commands directly set the agent's position and orientation, have no requirements and can even place the agent into an otherwise invalid position (i.e. inside a block). Available commands are tp (short for teleport to position), tpx, tpy, tpz (which change only one coordinate of the agent), setYaw and setPitch. [19]

Discrete movement commands move agent in discrete steps. For instance, `move 1` moves the agent one block forward, with no intermediate state (agent moves immediately to the new position). Unlike absolute and continuous movement commands, discrete movement commands require agent to stand in the center of block's top side (decimal part of x- and z-coordinate must be equal to 0.5). The use command also requires the agent to hold a block and look at block face, where new block can be added without intersecting any entities. Available commands are `move`, `turn` (turns by 90 ° increments), `movenorth`, `moveeast`, `movesouth`, `movewest` (all move agent in respective absolute direction), `jump` and `look` (moves agent view direction downward in increments of 45 °). [19]

Inventory commands move items within the inventory and remove items from the inventory. Available inventory commands are `selectInventoryItem`, `dropInventoryItem`, `discardCurrentItem` and `hotbar.<hotbarSlotIndex>`. None of the commands have any requirements. [19]

Chat commands send messages to Minecraft chat via `chat <chatMessage>` command. While primarily intended for broadcasting text messages to other agents, Malmo chat commands also allow to send Minecraft console commands. [19] These commands do not have any requirements to execute and can be used to teleport the agent to absolute or relative positions, change

agent direction, place and destroy blocks, place and remove items from the agent's inventory. [20]

Simple craft commands allow to combine resources within the agent's inventory to craft other items (according to Minecraft crafting specification and without any other requirements than the raw source ingredients - for example, the presence of a crafting table is not required). [19]

Turn based commands, unlike other command groups, do not control the agent directly, but set the execution of command groups named inside the `TurnBasedCommands` XML tag to turn-based. Turn-based execution means, that agents execute their assigned commands one-by-one and take turns doing so. [19]

Human level commands allowing controlling Minecraft at keyboard- and mouse-event basis. Due to their low-level approach, these commands have no execution requirements. [19]

1.8.4 Agent Observations

Agent observations, along with the `VideoProducer`, provide information about the state of the agent and the environment. The `VideoProducer` requests video frames to be sent. Returned frames can have either RGB or RGBD format, where RGBD adds depth information as its additional fourth channel. Observations return agent/environment state data in JSON format and are offered in the following variants [19]:

- Observation From Chat
- Observation From Discrete Cell
- Observation From Distance
- Observation From Full Inventory
- Observation From Full Stats
- Observation From Grid
- Observation From Hot-Bar
- Observation From Nearby Entities
- Observation From Ray
- Observation From Recent Commands
- Observation From Subgoal Position List
- Observation From Turn Scheduler

1.8.5 Agent Rewards

Agent rewards are mainly used for reinforcement learning tasks. Their purpose is to reward the agent for performing various actions. There are multiple reward-able action types [19]:

- Reward For Catching Mob
- Reward For Collecting Item
- Reward For Damaging Entity
- Reward For Discarding Item
- Reward For Mission End
- Reward For Reaching Position
- Reward For Sending Command
- Reward For Sending Matching Chat Message
- Reward For Structure Copying
- Reward For Time Taken
- Reward For Touching Block Type

Reward given to the agents is defined by a decimal number, which also determines the reward amount. Due to this thesis not being aimed at reinforcement learning, specific details of rewarded actions will not be included. See [19] for more information about specific rewarded actions.

Chapter 2

Analysis and Design

Suggested solution (image 2.1) consists of two main parts – problem instance solver and construction visualizer. Problem instance solver loads the problem instance and assigns it to an external solver. The solver computes the solution. To avoid agent symmetry, the solved model does not distinguish between agents. Therefore, once the solution is found, the solver passes it to agent instruction assigner, which assigns instructions from the solver to individual agents. Construction visualizer takes instructions for individual agents, creates world with the initial state of the building area (initial state of the problem instance) and makes the agents execute their assigned instructions in proper order.

Environment of the problem is considered deterministic, accessible, with known initial state (the initial state of the building area). In order to simplify the problem further, the environment is considered to be discrete and static. The environment is not episodic, because quality of some agent actions depends on previous / future agent actions (for instance, placing a block on second layer requires previously placed stepping block on the first layer, working as scaffolding for the agent).

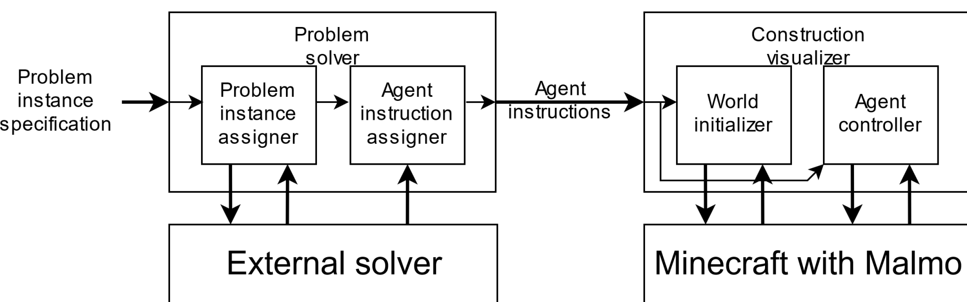


Figure 2.1: Block diagram of the solution

2.1 Problem Instance Solver

Problem instance solver acts as a tool for converting description of the construction goal (a structure made of blocks) to a set of instruction lists, each list assigned to a specific agent. Instruction execution will be considered deterministic to simplify the model. In real world scenario, such simplification would have to be based on very high agent reliability (especially in regards to timely and proper execution of assigned actions). In Minecraft (the intended visualization tool), high enough reliability can be achieved due to relatively simple set of rules the simulated world is governed by.

2.1.1 Construction Fraction-Time Model

The main aim of the construction model is to provide problem instance description for a MILP solver. MILP model has been selected due to significantly better computation speed of a blocks world construction model, compared to constraint programming (as shown in [4]). Due to high similarity between blocks world, described in [4], and Minecraft construction, this thesis uses a model based on the original model from cited white-paper.

2.1.1.1 Incorporation of Minecraft and Malmo Control Scheme into the Model

At this point, both Minecraft and Malmo are selected for the visualization. Malmo is selected, because it is the only officially supported modification of the game and because it provides interface for external Minecraft player control through Malmo API. Main candidates for control schemes are Discrete Movement Commands along with Absolute Movement Commands for entering/leaving the grid, or Chat Commands, which can serve the same purpose. Both are selected, because they allow to move agent in discrete and deterministic steps (through the problematic of command determinism ultimately proved to be a little bit more complicated than can be first gleaned from the command descriptions – more on that in the implementation section).

When compared to agent actions in the base MILP model (from [4]), Malmo control schemes require more than one command for accomplishing most of the actions (for example, entering the grid with a block first requires command to give the agent block to be carried, and then command to teleport the agent to the border cell). The exact number of commands required to perform one action depends on the interpretation and command scheme used. Some actions require fewer commands to be performed (move action to neighbor cell requires only one chat command, which performs agent movement and rotation simultaneously).

There are multiple approaches to performing agent actions with varying number of Malmo commands for each action type. Each action type has

its assigned duration, depending on how many Malmo commands are needed to execute it. The simplest approach to visualization would be to make one timestep the length of maximum action type duration. Then, in each timestep, Malmo commands belonging to actions of that timestep would be executed. The main disadvantage of this approach would be the unused time between actual action duration and maximum duration of all action types.

Another approach would be to wait for all actions in current timestep to finish and then start the next timestep. This can lower the waiting times, through the actual time saved depends on the maximum action duration for each timestep. Furthermore, the original MILP model would not take these time savings into account, so the solution might not be optimal time-duration-wise. Modification of the model is necessary to consistently target those time savings and get provably optimal solution of such strategy.

A more systematic approach is to allow different execution times for different action types. Action type is then defined as a set of actions, which can be executed using the same list of Malmo commands. Action type time is the (maximum) time duration of the command list execution. This approach is used for the proposed mathematical model of the construction problem, as it provides construction plans with the optimal building times, relative to action type execution time.

2.1.1.2 Construction Fraction-Time MILP Model

Generalization of the existing MILP model, described in [4], is selected to exploit its inner network flow structure and computational speed advantage over CP model, while providing constraint-based planning features in form of action start and end times (from non-preemptive scheduling). The base model includes a timestep t in every action, which denotes in which timestep the action takes place. Generalization of the model for this bachelor thesis, proposes to replace the t with t_{start} and t_{end}, which would denote the action start timestep (inclusive) and action end timestep (exclusive), respectively. This notation mirrors non-preemptive scheduling notation, where action is executed within time interval $[t_{start}, t_{end})$.

Proposed model will be called '*fraction-time model*', because it, theoretically, allows any positive fraction $k/l; k, l \in \mathbb{Z}_+$ as the action time. The proposed fraction-time model only allows positive integer action times, but any set of fraction times can be turned to a set of integer times by multiplying them by the least common multiple of their denominators.

Before describing the proposed model, a few details regarding the notation are necessary to point out. The proposed model is using the notation of the base model, used in [4]. For set $\mathcal{U}$ of tuples $(u_1, u_2, \dots, u_n)$ notation $\mathcal{U}_{u_1, u_2, \dots, u_n}$ is shorthand for $\{(u'_1, u'_2, \dots, u'_n) : u'_1 = u_1 \wedge u'_2 = u_2 \wedge \dots \wedge u'_n = u_n\}$. Let $*$ denote a wildcard symbol for matching any value at position in the tuple it is used in (meaning $\mathcal{U}_{*, u_2, \dots, u_n}$ is shorthand for $\{(u'_1, u'_2, \dots, u'_n) : u'_2 =$

$u_2 \wedge \cdots \wedge u'_n = u_n\}$ and $\mathcal{U}_{u_1,u_2,*,\ldots,*}$ is shorthand for $\{(u'_1, u'_2, \ldots, u'_n) : u'_1 = u_1 \wedge u'_2 = u_2\})$.

Let $\mathcal{X} = \{0, \ldots, X-1\}$, $\mathcal{Y} = \{0, \ldots, Y-1\}$, $\mathcal{Z} = \{0, \ldots, Z-1\}$, where (X, Y) is the size of the building area and $Z - 1 = \max(\overline{z}_{(x,y)}), \forall (x,y) \in \mathcal{P}$ is equal to the height of the user entered structure. Let $\mathcal{C} = \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$ be the set of all positions within the grid, $\mathcal{P} = \mathcal{X} \times \mathcal{Y}$ is the projection of $\mathcal{C}$ into the first two dimensions, $\mathcal{B} = \{(x,0,0) : x \in \mathcal{X}\} \cup \{(x, Y-1, 0) : x \in \mathcal{X}\} \cup \{(0,y,0) : y \in \mathcal{Y}\} \cup \{(X-1, y, 0) : y \in \mathcal{Y}\}$ is the set of border cells at the perimeter of the building area. Let $\mathcal{K} = \{\mathrm{M}, \mathrm{P}, \mathrm{D}\}$ be a set of agent action distinguishers – M for move action (used for 'entry', 'leave', 'move_block', 'move_empty' and 'wait' action types), P for 'pick_up' action type and D for 'deliver' action type. Let $\mathcal{N}_{(x,y)} = \{(x-1,y), (x+1,y), (x,y-1), (x,y+1)\} \cap \mathcal{P}$ be the set of neighbor positions of (x,y) and $\mathcal{T} = \{0, \ldots, T-1\}$ be the the planning horizon of T timesteps.

The proposed model divides the set of agent actions into seven sets of agent action types, which in union give the new set of agent actions. Agent action types are derived from six different action subsets in the original model:

Set $\mathcal{R}_{entry} = \{(t_{start}, t_{end}, \mathrm{S}, \mathrm{S}, \mathrm{S}, c, \mathrm{M}, x, y, z)\} : t_{start} \in \{0, \ldots, T - T_{entry} - \min(T_{pick_up}, T_{deliver}) - T_{leave} - 1\} \wedge t_{end} = t_{start} + T_{entry} \wedge c \in \{0,1\} \wedge (x,y,z) \in \mathcal{B}$ of 'entry' type actions features all actions, where the agent enters from the starting position outside the grid to a border cell. Every such action has duration of $T_{entry} \in \mathbb{Z}_+$ timesteps.

Set of agent actions, where the agent moves to neighbor grid position is divided into two action types – 'move_block' for moving to neighbor position while carrying a block and 'move_empty' for moving to neighbor position while not carrying a block. This distinction is made for optional visualization of carrying a block above the agent's head, which uses an extra Malmo command. Set $\mathcal{R}_{move_empty} = \{(t_{start}, t_{end}, x, y, z, 0, \mathrm{M}, x', y', z') : t_{start} \in \{T_{entry}, \ldots, T - T_{move_empty} - T_{leave} - 1\} \wedge t_{end} = t_{start} + T_{move_empty} \wedge (x,y,z) \in \mathcal{C} \wedge (x', y') \in \mathcal{N}_{(x,y)} \wedge |z' - z| \leq 1\}$ is for 'move_empty' action type and set $\mathcal{R}_{move_block} = \{(t_{start}, t_{end}, x, y, z, 1, \mathrm{M}, x', y', z') : t_{start} \in \{T_{entry}, \ldots, T - T_{move_block} - T_{leave} - 1\} \wedge t_{end} = t_{start} + T_{move_block} \wedge (x,y,z) \in \mathcal{C} \wedge (x', y') \in \mathcal{N}_{(x,y)} \wedge |z' - z| \leq 1\}$ is for 'move_block' action type. Every 'move_empty' type action has duration of $T_{move_empty} \in \mathbb{Z}_+$ timesteps and every 'move_block' type action had duration of $T_{move_block} \in \mathbb{Z}_+$ timesteps.

Action type 'wait' with action set $\mathcal{R}_{wait} = \{(t_{start}, t_{end}, x, y, z, c, \mathrm{M}, x, y, z) : t_{start} \in \{T_{entry}, \ldots, T - T_{wait} - T_{leave} - 1\} \wedge T_{end} = T_{start} + T_{wait} \wedge c \in \{0,1\} \wedge (x,y,z) \in \mathcal{C}\}$ has duration of $T_{wait} = 1$ timestep. This fixed duration is chosen, because minimum waiting time of agents is generally not limited.

Action type 'leave' is for agents leaving the grid from a border cell (going to the end position $(\mathrm{E}, \mathrm{E}, \mathrm{E})$). The associated agent action set $\mathcal{R}_{leave} = \{(t_{start}, t_{end}, x, y, z, c, \mathrm{M}, \mathrm{E}, \mathrm{E}, \mathrm{E}) : t_{start} \in \{T_{entry} + T_{deliver}, \ldots, T - T_{leave} - 1\} \wedge t_{end} = t_{start} + T_{leave} \wedge c \in \{0,1\} \wedge (x,y,z) \in \mathcal{B}\}$ features actions with action type duration of $T_{leave} \in \mathbb{Z}_+$ timesteps.

Set $\mathcal{R}_{pick_up} = \{(t_{start}, t_{end}, x, y, z, 0, \mathrm{P}, x', y', z) : t_{start} \in \{T_{entry}, \ldots, T - T_{pick_up} - T_{leave} - 1\} \wedge t_{end} = t_{start} + T_{pick_up} \wedge (x, y) \in \mathcal{P} \wedge (x', y') \in \mathcal{N}_{(x,y)} \wedge z \in \{0, \ldots, Z-2\}\}$ of 'pick_up' type actions features all actions, where the agent picks up a block from a neighbor position. Every such action has duration of $T_{pick_up} \in \mathbb{Z}_+$ timesteps.

Set of 'deliver' type actions $\mathcal{R}_{deliver} = \{(t_{start}, t_{end}, x, y, z, 1, \mathrm{D}, x', y', z) : t_{start} \in \{T_{entry}, \ldots, T - T_{deliver} - T_{leave} - 1\} \wedge t_{end} = t_{start} + T_{deliver} \wedge (x, y) \in \mathcal{P} \wedge (x', y') \in \mathcal{N}_{(x,y)} \wedge z \in \{0, \ldots, Z-2\}\}$ features all actions, where the agent delivers a block to a neighbor position. Every such action has duration of $T_{deliver} \in \mathbb{Z}_+$ timesteps.

Action type times are constant within a problem instance (meaning T_{entry}, T_{leave}, $T_{deliver}$, T_{pick_up}, T_{move_block}, T_{move_empty} and T_{wait} must be set by user along with structure height-map, their omission will be considered to mean $T_{entry} = T_{leave} = T_{deliver} = T_{pick_up} = T_{move_block} = T_{move_empty} = T_{wait} = 1$.

The objective function minimizes sum-of-costs, which in this context means the sum of timesteps each robot spends on the grid. The proposed model counts 'entry' type action timesteps into the objective function, because the agent is considered to partially be on the grid, when the action starts (blocks border cell as part of action exclusion area).

$$\begin{aligned}
\min(T_{entry} &\sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{entry}} r_i \\
&+ T_{leave} \sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{leave}} r_i \\
&+ T_{deliver} \sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{deliver}} r_i \\
&+ T_{pick_up} \sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{pick_up}} r_i \\
&+ T_{move_block} \sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{move_block}} r_i \\
&+ T_{move_empty} \sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{move_empty}} r_i \\
&+ T_{wait} \sum_{i=(t_{start},t_{end},x,y,z,c,a,x',y',z')\in\mathcal{R}_{wait}} r_i)
\end{aligned} \tag{2.1}$$

$$h_{t,x,y,z,z} = 1, \forall t \in \{0, \ldots, T-1\}, (x, y, z) \in \mathcal{B} \tag{2.2}$$

$$h_{0,x,y,0,0} = 1, \forall (x, y) \in \mathcal{P} \tag{2.3}$$

$$h_{T-1,x,y,\overline{z}_{(x,y)},\overline{z}_{(x,y)}} = 1, \forall (x, y) \in \mathcal{P} \tag{2.4}$$

$$\sum_{i \in \mathcal{H}_{t,x,y,*,z}} h_i = \sum_{\mathcal{H}_{t+1,x,y,z,*}} h_i, \forall t \in \{0,\dots,T-2\}, (x,y,z) \in \mathcal{C} \tag{2.5}$$

$$\sum_{i \in \mathcal{H}_{t,x,y,*,*}} h_i = 1, \forall t \in \{0,\dots,T-1\}, (x,y) \in \mathcal{P} \tag{2.6}$$

Constraints 2.2 – 2.6 are the same as in the original MILP model to keep height information for every timestep. Constraint 2.2 forbids placement of blocks at border cells, constraint 2.3 starts the world devoid of blocks, constraint 2.4 ensures that user defined structure is finished at the end of construction, constraint 2.5 flows the column height from one timestep to the next (height of the block column at the end of the timestep must be equal to the height at which the column starts in the next timestep) and constraint 2.6 forces every position to have one height.

$$\begin{aligned} \sum_{i \in \mathcal{R}_{*,t,*,*,*,0,\mathrm{M},x,y,z}} r_i + \sum_{i \in \mathcal{R}_{*,t,x,y,z,1,\mathrm{D},*,*,*}} r_i \\ = \sum_{i \in \mathcal{R}_{t,*,x,y,z,0,\mathrm{M},*,*,*}} r_i + \sum_{i \in \mathcal{R}_{t,*,x,y,z,0,\mathrm{P},*,*,*}} r_i, \\ \forall t \in \{0,\dots,T-T_{leave}-1\}, (x,y,z) \in \mathcal{C} \end{aligned} \tag{2.7}$$

$$\begin{aligned} \sum_{i \in \mathcal{R}_{*,t,*,*,*,1,\mathrm{M},x,y,z}} r_i + \sum_{i \in \mathcal{R}_{*,t,x,y,z,0,\mathrm{P},*,*,*}} r_i \\ = \sum_{i \in \mathcal{R}_{t,*,x,y,z,1,\mathrm{M},*,*,*}} r_i + \sum_{i \in \mathcal{R}_{t,*,x,y,z,1,\mathrm{D},*,*,*}} r_i, \\ \forall t \in \{0,\dots,T-T_{leave}-1\}, (x,y,z) \in \mathcal{C} \end{aligned} \tag{2.8}$$

Constraints 2.7 and 2.8 flows the agents from one action to the next. Semi-closed interval of action execution ($[t_{start}, t_{end})$) is exploited for seamless transition between actions. Similarly to the base model, constraint 2.7 flows agents without block and ensures that the number of agents ending their action without block at position (x, y, z) at timestep t is the same as the number of agents without block starting their action at the same position and in the same timestep. Constraint 2.8 does the equivalent for agents carrying a block.

$$\begin{aligned} \sum_{i \in \mathcal{R}_{t_{start},t_{end},x,y,*,*,*,*,*,*}: t_{start} \le t < t_{end}} r_i + \sum_{i \in \mathcal{R}_{t_{start},t_{end},*,*,*,*,*,x,y,*}: t_{start} \le t < t_{end}} r_i \\ - \sum_{i \in \mathcal{R}_{t_{start},t_{end},x,y,*,*,*,x,y,*}: t_{start} \le t < t_{end}} r_i \le 1, \\ \forall t \in \{0,\dots,T-1\}, (x,y) \in \mathcal{P} \end{aligned} \tag{2.9}$$

Constraint 2.9 addresses vertex collision of agents during action execution. Since the only requirement for the agent is to perform the action a between t_{start} and t_{end}, exact position of the agent is unknown and both start and end position are made into exclusion zone, where no other action can take place (for timesteps within interval $[t_{start}, t_{end})$). To avoid agents removing blocks under moving agents, the constraint makes the whole block column at position (x, y) of both start and end of the action an exclusion zone. Constraint preventing edge collisions (constraint 1.19 in the base model) is no longer necessary, as two action can no longer share vertices while executing simultaneously, due to the exclusion zones.

$$\sum_{i \in \mathcal{R}_{t_{start}, t_{end}, *, *, *, *, *, *, *, *} : t_{start} \leq t < t_{end}} r_i \leq A, \forall t \in \mathcal{T} \tag{2.10}$$

Constraint 2.10 limits the number of agents on the grid.

$$\sum_{i \in \mathcal{H}_{t,x,y,z,*}} h_i \geq \sum_{i \in \mathcal{R}_{t_{start}, t_{end}, x, y, z, *, *, *, *, *} : t_{start} \leq t < t_{end}} r_i, \quad \forall t \in \{0, \ldots, T-1\}, (x, y, z) \in \mathcal{C} \tag{2.11}$$

$$h_{t,x,y,z+1,z} = \sum_{i \in \mathcal{R}_{*,t+1,*,*,*,0,\mathrm{P},x,y,z}} r_i, \quad \forall t \in \{0, \ldots, T-2\}, (x, y) \in \mathcal{P}, z \in \{0, \ldots, Z-2\} \tag{2.12}$$

$$h_{t,x,y,z,z+1} = \sum_{i \in \mathcal{R}_{*,t+1,*,*,*,1,\mathrm{D},x,y,z}} r_i, \quad \forall t \in \{0, \ldots, T-2\}, (x, y) \in \mathcal{P}, z \in \{0, \ldots, Z-2\} \tag{2.13}$$

$$h_i \in \{0, 1\}, \forall i \in \mathcal{H} \tag{2.14}$$

$$r_i \in \{0, 1\}, \forall i \in \mathcal{R} \tag{2.15}$$

Constraint 2.11 forces agents to always stand on the highest block in the column when performing their actions. Constraints 2.12 and 2.13 govern decreases and increases in block column height, respectively. Constraint 2.12 ties every decrease by one block to pick_up action. Constraint 2.13 does the equivalent for increase by one block and deliver action. Constraints 2.14 and 2.15 specify the variable domains.

2.1.2 Problem Instance Assigner

The purpose of the problem instance assigner is to load user input, create a construction problem instance, described by objective function 2.1 and constraints 2.2 – 2.15, and assign it to the external solver. User input must contain height-map of structure to be constructed (matrix $\underline{A}$, where $a_{m,n} = \overline{z}_{(m,n)}$). Values of X and Y are taken from the height-map size. Agent action type durations (T_{entry}, T_{leave}, $T_{deliver}$, T_{pick_up}, T_{move_block} and T_{move_empty}) may also be part of the user input. Absence of time T_i means, that $T_i = 1$, where i is any of the action type names. Last necessary value for the problem instance, is the planning horizon timestep limit T. This value can either be part of the user input, or is taken sequentially from $\mathbb{Z}_+$ (from lowest value, each time increasing by 1) to find minimum viable construction makespan, just as is done in the base model in [4]. For sequential makespan increases, multiple calls of problem instance assigner and solver may be necessary, before the solution is found. Agent instruction assigner is called only if a solution is found.

Sequential increasing of makespan allows simple proof of optimality – if the solution found by increasing the makespan by 1 did not have the optimal makespan, then a solution with lower makespan would have to exist. But since the algorithm went over all makespan values from minimum value to the current makespan and found all lower makespan solutions to be infeasible, solution with current makespan is optimal.

2.1.3 Agent Instruction Assigner

When the solver finds the (optimal) solution and closes the gap between solution lower and upper bounds, values of two sets of indicator variables are known – h_i, height indicators for the entire duration of the mission, and r_i, indicators, that some robot performed within $[t_{start}, t_{end})$ action a, the start of which was on coordinates $(x_{start}, y_{start}, z_{start})$ and action end was on $(x_{end}, y_{end}, z_{end})$ (and c indicates if the agent was carrying a block at action start). Since all the mentioned information can be stored along with the indicator variables, instruction sequence for single agent can be obtained using the constraints 2.7 and 2.8. The solution must satisfy these constraints, so the location of agent at the end of the action must be equal to the agent location at the start of the next action.

Agent instruction assigner first sorts the used actions (with $r_i = 1$) by t_{start}, so position of the agents can be tracked from their start positions $((x_{start}, y_{start}, z_{start}) = (\mathrm{S}, \mathrm{S}, \mathrm{S}))$ to their end positions $((x_{end}, y_{end}, z_{end}) = (\mathrm{E}, \mathrm{E}, \mathrm{E}))$ in chronological order. Empty agent object array is then created. Agent object tracks agent position, end time of last instruction assigned to the agent object and chronological list of instructions assigned to the agent. Then processes instructions one-by-one, for each one first searching agent object array for matching instruction end time and position. If no such agent exists, in-

struction must be of type ‘entry’ (starting at $(x_{start}, y_{start}, z_{start}) = (\mathrm{S}, \mathrm{S}, \mathrm{S})$), it is then assigned to any agent with current position equal to $(\mathrm{E}, \mathrm{E}, \mathrm{E})$ and last instruction end at most equal to new instruction start. Special case of wait instruction is used, if leave instruction end time $t_{leave_{end}}$ does not match entry instruction start time $t_{entry_{start}}$ – agent waits outside the grid for the duration of $t_{entry_{start}} - t_{leave_{end}}$. If no agent with matching position and last action end time exists within the agent object array, one is created (constraint 2.10 satisfaction guarantees that number of agent objects within the array will not exceed given limit). Once all instructions are assigned, instruction lists are passed to the output.

2.2 Construction Visualizer

Construction visualizer acts as a visualization tool for the output of the problem instance solver. Its work can be divided into two parts – preparing the initial state of the problem instance and executing the agent instructions to show the solution.

2.2.1 World Initializer

The purpose of world initializer is to create a Minecraft world, which reflects the initial state of the problem instance. Considering the initial state of the grid is a flat area without blocks, a superflat Minecraft world generator is selected, with at least one layer of blocks for the agents to stand on, which are not to be considered as part of the grid.

Another goal of the world initialization is the placement of agents into their starting positions outside the grid. This requires two conditions to be met – appropriate number of agents, as necessary for the mission, must be present within the world and the starting state of the agents must not interfere with the construction within the grid, while allowing agents to repeatedly enter it and move from it to border cells of the grid during the mission execution. Through the exact way to fulfil these conditions will be left on the implementation.

2.2.2 Agent Controller

Agent controller is responsible for instructing the assigned agents according to predetermined instruction list set. Each agent is assigned a list of instructions, which must be performed in given order, with proper timing (each instruction type has a set execution time, as specified by the model the solver used when computing the problem instance). All agents must execute their instructions within the assigned instruction time ranges $[t_{istart}, t_{i_{end}})$, where $t_{i_{start}}$ is the start time of instruction i (instruction execution cannot start before this time)

and $t_{i_{end}}$ is the end time of instruction i. Instruction execution must be completed before $t_{i_{end}}$.

Used instructions do not require the agents to communicate, but agent synchronization is necessary. This can be achieved by central control of the agents, with central clock, where each agent is sent commands for current timestep. For instructions consisting of multiple commands, each command can be sent in one timestep, provided the instruction duration is long enough to allow such approach. Due to deterministic agent commands, feedback from world state or agents is not necessary.

Another approach is synchronization of precise enough inner agent clocks before construction start. Malmo agents running on the same computer may use the system time for synchronization. The disadvantage of this approach is the added complexity of multiple independent agents, which must be synchronized.

For construction projects with longer duration, without direct access of each agent to central clock and with less precise agent clocks, synchronization signal is necessary to synchronize the agents during construction mission execution. This approach adds similar complexity to the approach with single synchronization event at the start of the mission, as the synchronization can be done through Malmo chat commands, repeatedly.

Since the construction model with action times is expected to be primarily limited by construction duration (as the computation requirements quickly rise for the base model in [4]), and the early testing of Minecraft with Malmo on hardware available for this thesis has indicated that maximum number of available Malmo agents is circa 25, central control of the agents is selected, as it is sufficient, while having the smallest implementation complexity.

Agent controller is required to run with agent actions of different durations. While the simplest design would match every timestep with one command (and in industrial use case such approach would be the most desirable), for testing purposes and to allow visualization of time saved by proposed MILP model, approach with configurable action times is taken. In this approach, information of action type times is passed along with agent instructions between problem instance solver and construction visualizer (see diagram 2.1). Individual instructions are then mapped to Malmo commands which are equally divided into $t \in \mathbb{Z}_+$ command queues in such way, that first $n \in \mathbb{Z} : 0 \leq n < t$ queues contain k commands and the remaining $t-n$ queues contain $k-1$ commands. The number of queues t is equal to instruction action type duration in timesteps (as passed from problem instance solver).

Core functionality of agent controller, where instructions are divided into command queues and commands from the queues are one-by-one sent to the agents, is described by the flowchart 2.2. Before the construction starts, instruction list for each agent is loaded into agent instructions object. This object, upon request, provides one command queue (with commands to be done in one timestep). Since actions can take more than one timestep, agent

instructions object also holds ‘TODO queue’, which holds not-yet-requested command queues of action currently being executed. If the agent instructions object is requested to provide command queue, while ‘TODO queue’ is not empty, top item from ‘TODO queue’ is removed and provided. Otherwise, next instruction is divided into command queues.

After getting requested command queue for all agents, all first items from all command queues are repeatedly removed and sent to their respective agents. There is a waiting time after commands have been sent to all agents (all agents with available commands to be precise), to allow agents to execute their assigned commands. After all queues are executed, a waiting time is present to keep timestep duration constant. Main loop ends, after all instructions have been divided into commands and sent to their respective agents. After main loop ends, the construction mission is ended.

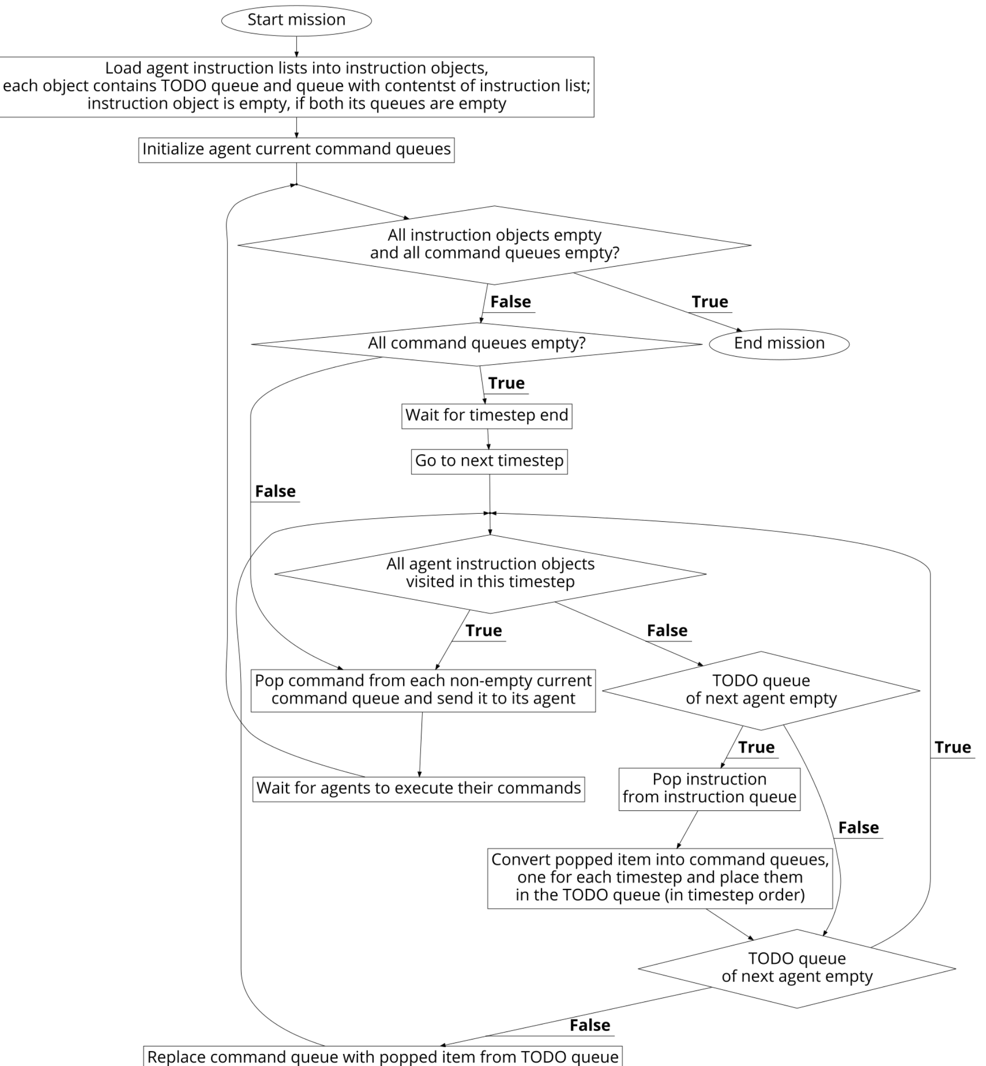


Figure 2.2: Flowchart of suggested central agent controller

Chapter 3

Realisation

3.1 Implementation

Implementation consists of two main applications – `visualizer_main` and `solver_main`. In relation to the analysis block diagram 2.1, `solver_main` serves the role of the problem instance solver and `visualizer_main` serves as the construction visualizer – both are marked on the implementation block diagram 3.1, which shows the finalized overall schema of the application.

Gurobi is selected as the external solver, because it is one of the fastest MILP solvers [12] and because it has been used in [4]. This means, that its use allows for more direct comparison of the time-fraction model with the base model.

Minecraft has been selected as part of the visualization. The main reason for this decision is the core design of the application being sandbox game, which allows blocks-world construction by multiple players. High popularity of Minecraft was second important factor, that led to its choice, as it provides unusual way of promotion of multi-agent construction.

Malmo has been selected, because it provides multiple control and observation schemes for multi-agent systems with API for multiple programming languages. Its position as the only officially supported modification of the game and inclusion of free version of the Minecraft game along the project were also important decision factors, that led to its choice. Mentioned free version of Minecraft (for academic purposes) is Minecraft Java 1.11 and, while being an older version, provides sufficient block construction capabilities to be used in the project.

Python programming language is used for implementation of `solver_main` and `visualizer_main`. Both Malmo and Gurobi provide their API for multiple programming languages (both support C++, C#, Java and Python [21] [12]). Python is selected, because its `pip` utility allows easy installation of both Malmo and Gurobi packages. Python version 3.6 is selected as the only version currently supported by both modules.

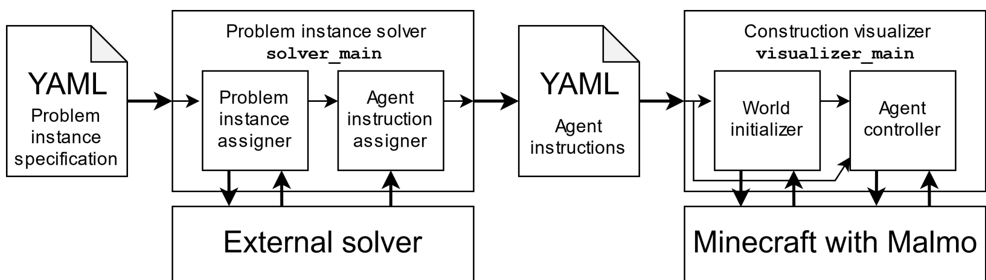


Figure 3.1: Implementation block diagram

Communication between the two applications is done via a YAML file (human and computer readable text-based file format for easy visualization of agent action lists). Problem instance is also described using a YAML file format, which provides a simple way to create the specification.

Architecture consisting of two independent applications is selected because of long computation times of the base model in [4] which are expected to also be present in the proposed model. The application separates computationally intensive and time consuming part from visualization. Saving the intermediate output allows repeated visualization of once-computed problem instance solution. It should be noted, that Minecraft agents with Malmo API are themselves hardware intensive, each requiring a fair amount of operation memory (at least 2 GB to be exact).

3.2 Problem Instance Solver

Implementation of the problem instance solver (described by flowchart 3.2) allows to solve problem instances where makespan is already known (and normally secondary optimization criterion – sum-of-costs – becomes the primary objective) and problem instances with unknown makespan, where minimum makespan is the primary objective (sum-of-costs being the secondary one). Which problem instance is solved by the solver depends on presence (or lack of) `makespan` attribute in the YAML problem instance specification (set `makespan` value naturally means solving instance with known makespan). If the makespan is not set, minimum possible makespan is used (3 to be exact), searched for solution and then increased, until the solution is found. It may be beneficial to estimate the lower bound of minimum construction time (makespan) more precisely – though solving of instances with lower makespan is substantially faster, so the lower bound would have to be very close to the minimum makespan for significant computational time saves to be made.

The application `solver_main` can be run using `run_solver_main.sh` bash script. Command `./run_solver_main.sh --help` can be used to list all command line options. More detailed explanation of these options is available in `readme.txt` file on the enclosed CD. To run the `solver_main` application, use `./run_solver_main.sh -i<PROBLEM_INSTANCE> -o<AGENT_INSTRUCTIONS>`.

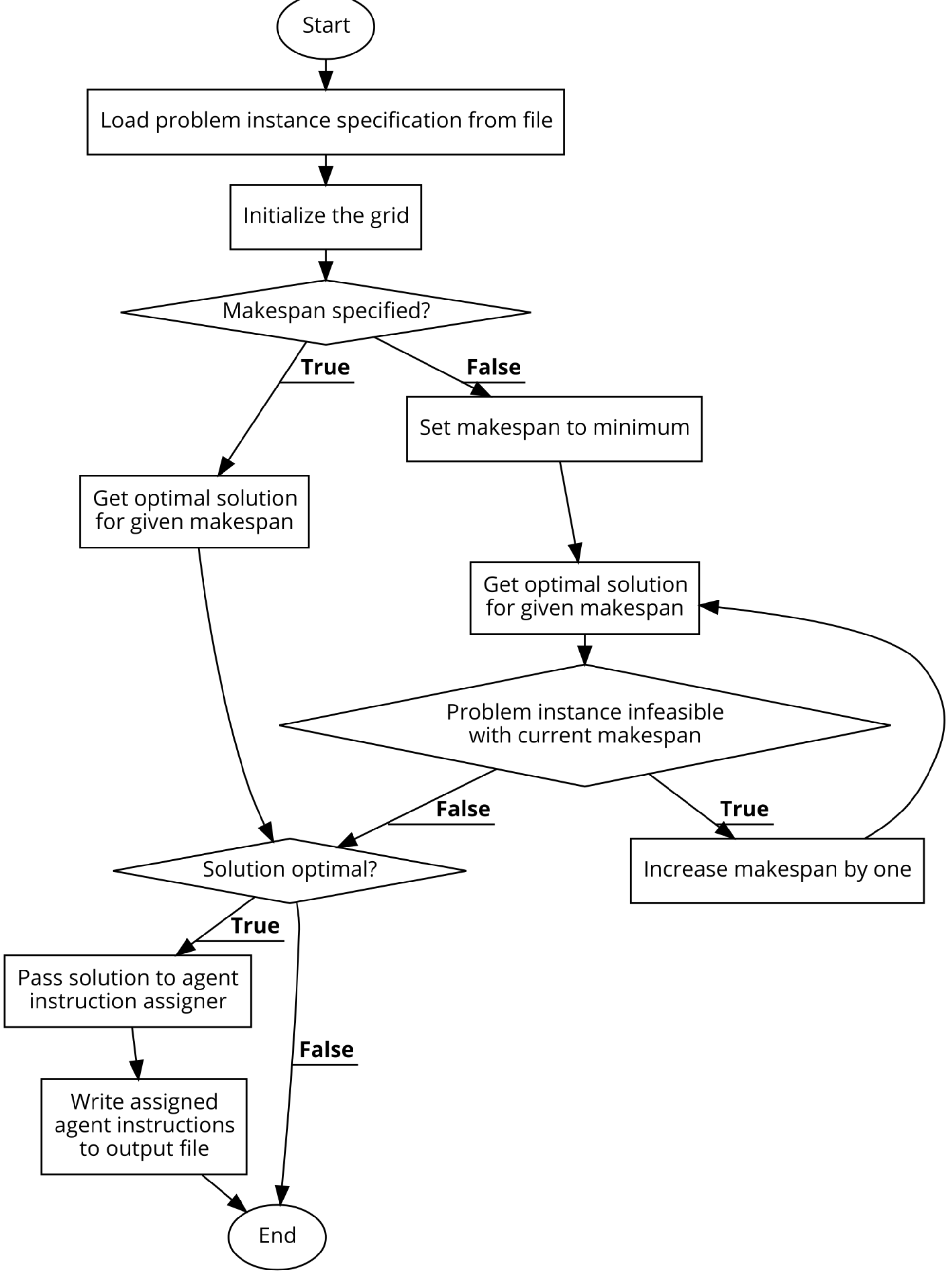


Figure 3.2: Problem instance solver flowchart

3.2.1 Problem Instance Assigner

Flowchart 3.2 of problem instance solver provides more detailed inner view of problem instance assigner, while keeping the agent instruction assigner part very brief. This is done so interaction of both parts is visible, while not cluttering the flow chart with relatively complicated inner structure of agent instruction assigner object, which is described separately.

Problem instance assigner consists of several parts. As part of loading problem instance specification, action times are converted from fractions to integers by multiplying them with least common multiple of their denominators. Converted integer action times are then scaled down by division with their greatest common divisor. Block 'Initialize the grid' prepares grid-related sets $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, $\mathcal{C}$, $\mathcal{P}$, $\mathcal{B}$, as described for the proposed model (range of values in the X, Y and Z axis, all positions within the grid, projections of these positions into first two dimensions and border positions respectively). Functions `N` for getting neighbor positions set ($\mathcal{N}_{(x,y)}$) for position (x, y) and `z_final` for getting required final structure height at position (x, y), are also created, in accordance with their definitions in the time-fraction model. These sets and functions are not related to makespan and so are created first to avoid repeated creation.

After initializing the grid, program execution continues through one of the execution branches, based on the presence of `makespan` attribute. If makespan is specified, the problem instance is assigned to the external solver once. If it is not specified, a problem instance is created for every value of makespan between its lower bound and minimum value. Every instance is passed to the external solver. Assigning the problem instances with set makespan (either known or increasing) to an external solver and calling its optimization method is done inside blocks 'Get optimal solution for given makespan'. Both of these blocks are calls of a single function, `get_optimal_solution_for_makespan`, which returns the solved instance model.

Each variable h_i and r_i from the time-fraction model is made to hold the tuple with information about height change / agent action it represents ($(t, x, y, z_{start}, z_{end})$ and $(t_{start}, t_{end}, x_{start}, y_{start}, z_{start}, c, a, x_{end}, y_{end}, z_{end})$ respectively). When the optimal solution is found, variables belonging to the solution are filtered out and tuples assigned to agent actions are given to the agent instruction assigner.

What the flowchart does not show, is that the solver can also be interrupted by the user, returning interrupted state, which is not the optimal solution and so passes no information to agent instruction assigner. Gurobi solver also allows to set maximum run-time, which is used when running experiments. During search for minimum makespan, sum of solver run-time is used to create total maximum run-time (maximum total solver run-time for one problem instance without set `makespan`).

agent position (equal to action start position)	new action end	action distinguisher	carry	relative instruction
(S, S, S)	$(x', y', z') \in \mathcal{C}$	M	0	enter <x'> <y'>
(S, S, S)	$(x', y', z') \in \mathcal{C}$	M	1	enter <x'> <y'> <minecraft block name>
$(x, y, z) \in \mathcal{B}$	$(\mathrm{E}, \mathrm{E}, \mathrm{E})$	M	0, 1	leave
$(x, y, z) \in \mathcal{C}$	$(x', y', z') \in \mathcal{C}$ $\wedge (x, y, z) = (x', y', z')$	M	0, 1	wait 1
$(x, y, z) \in \mathcal{C}$	$(x', y', z') \in \mathcal{C}$ $\wedge (x', y') \in \mathcal{N}_{(x,y)}$ $\wedge (z = z' \vee z - 1 = z')$	M	0, 1	move <action direction>
$(x, y, z) \in \mathcal{C}$	$(x', y', z') \in \mathcal{C}$ $\wedge (x', y') \in \mathcal{N}_{(x,y)}$ $\wedge z + 1 = z'$	M	0, 1	jump_move <action direction>
$(x, y, z) \in \mathcal{C}$	$(x', y', z') \in \mathcal{C}$ $\wedge (x', y') \in \mathcal{N}_{(x,y)}$ $\wedge z = z'$	D	1	place_block <action direction>
$(x, y, z) \in \mathcal{C}$	$(x', y', z') \in \mathcal{C}$ $\wedge (x', y') \in \mathcal{N}_{(x,y)}$ $\wedge z = z'$	P	0	break_block <action direction>

Table 3.1: Projection of fraction-time model actions to relative instructions

3.2.2 Agent Instruction Assigner

Agent instruction assigner has two main purposes – assign actions from solver solution to agents and to map those fraction-time model actions to relative instructions, easier to translate into Malmo commands. Process of assignation of model actions to agents is depicted in flowchart 3.3. Actions are assigned to agents based on their last assigned action end time and position. When an action is assigned to an agent, it is converted from fraction-time MILP model tuple to relative instruction, used as an intermediate format for the agent instructions YAML file. Conversion is done according to table 3.1. If no option matches, an error is thrown (an option that should not occur, if the external solver fulfils all the constraints of the model).

The assignation of actions to agents, as described in flowchart 3.3, works with two groups of agents – finished agents and unfinished agents. Agents are considered 'finished', when their last assigned action was to leave the grid via 'leave' action type. All other agents are considered 'unfinished'. Actions are assigned to agents in chronological order, so only the last assigned action end position and end time must be tracked for every agent. Actions of 'enter' type are managed separately, because they work with finished agents (if no available finished agents exist, new agent is created). The remaining action types require the agent to be unfinished, with last action end timestep and position matching the start timestep and position of the new action. Action type 'leave' moves agents into 'finished' set to provide them for further 'enter' type actions.

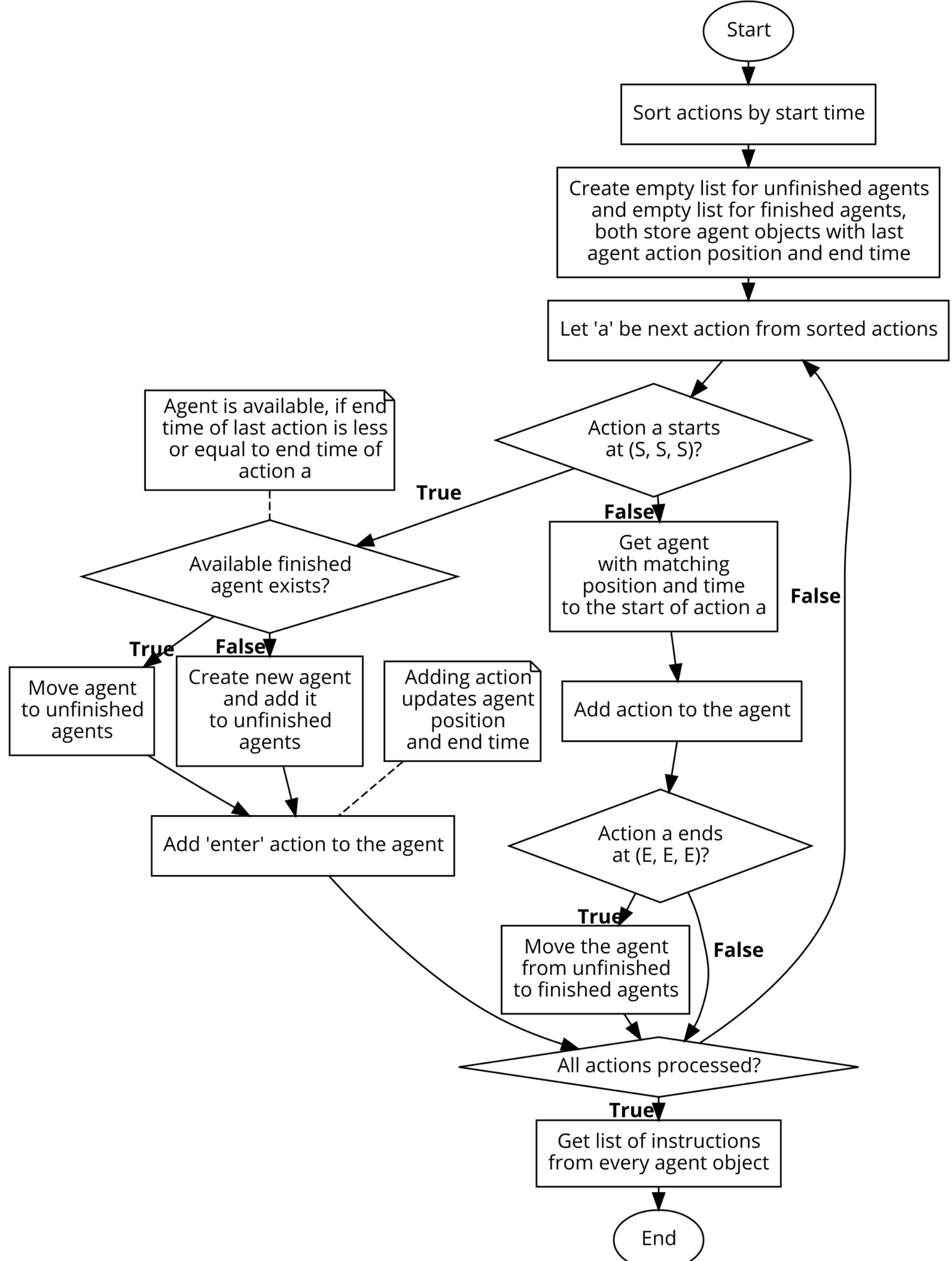


Figure 3.3: Agent instruction assigner flowchart

3.3 Construction Visualizer

Construction visualizer is created to visualize agent instructions from the intermediate YAML file. The application consists of two parts – world initializer and agent controller. Both are connected to a Minecraft world via Malmo API.

3.3.1 World Initializer

World initializer creates a Malmo mission and assigns it to an appropriate number of agents. Mission is described using a XML file, generated internally by the world initializer according to the `Mission.xsd` specification, available at [19].

As part of the Malmo mission, the world initializer creates a superflat world with grass top layer, using flat world generator. To provide maximum visibility for the user, virtual world time is fixed to noon and weather to clear. Random spawning of mobile entities (such as animals and entities hostile to the agent-player) is disabled to keep construction conditions deterministic. Building area perimeter (under border cells) is replaced by red-white markings to allow easier distinguishing of block positions (width of the red marker is 1 meter – the width of a block; white area between markers is also 1 meter wide).

A list of required agents is created in the mission XML. After the mission description is loaded by Malmo, each of the listed agents must be assigned to a Minecraft instance, otherwise the mission does not start. First in the list (and first assigned to a Minecraft instance) is the observer agent. Main purpose of this agent is to provide movable viewport for the construction (controlled by classical WASD key control scheme, `[SPACE]` up, `[SHIFT]` down). Change of viewport angle is done via mouse movement, but must be first enabled by pressing `[ENTER]` after mission starts, to avoid accidental change of viewport angle. The remaining agents in the list are the agents named in the YAML intermediate file, each being assigned a starting position outside the building area. The starting position is marked by a gold block under the feet of the agent. The agent starting position is stored, so the agent returning from the building area using 'leave' action can be teleported directly to its assigned starting area.

Before loading the mission XML, a Minecraft instance is created for every agent on the agent list (including the observer). After each of them is started, they are connected to Python Malmo as clients. Mission is then started and every Python Malmo client is guided through safe mission start (being added to the assigned mission role from the list of required agents). If any of the agents does not respond to repeated attempts at a safe mission start, mission is abandoned and the probable cause is displayed (function for safe agent start is taken from Malmo python samples for its excellent error handling of the Malmo mission start – source cited in the code). After all agents are added to the mission, the mission starts and the agent controller assumes control.

3.3.2 Agent Controller

Implementation of the agent controller is done according to the flowchart 2.2. The main premise is conversion of relative agent instructions to Malmo command queues, each queue for one timestep. One command queue is then gradually sent to each agent for execution, until all commands have been sent.

There are two available conversions of relative instructions to Malmo commands. Version with shorter Malmo command lists allows quicker mission execution, but does not visualize blocks held by the agents. Version with longer command lists provides additional held block visualization (held blocks show above the heads of agents). Command line option --show-held-blocks of the visualizer_main application is used to allow selection between quicker execution and slower execution with shown held blocks. Different speed of execution is caused by additional Malmo commands required for visualization of held blocks. The exact changes to the conversion from relative instructions to Malmo commands are shown in table 3.2. The table shows unification of Malmo control scheme with the use of chat commands. Description of all the command line options for visualizer_main is available in readme.txt. Alternatively, the user can call ./run_visualizer_main.sh --help to get the full list of command line options. Bash script run_visualizer_main.sh provides a simple way for running the visualizer_main application (call ./run_visualizer_main.sh -i<AGENT_INSTRUCTIONS_FILE>).

Malmo chat commands provide simple interface to the Minecraft console (chat <text for Minecraft console>, where the text behind chat keyword is sent without changes to the Minecraft console). Minecraft console text can either be message to be sent to all agents, or a '/' initiated Minecraft console command [20]. Tilde notation is used to write coordinates relative to the agent (for instance, ~1 ~ ~ means position 1 block in the direction of x-axis from the subject of the command, which is generally the agent who sent it). It should be noted, that the fraction-time model uses the coordinate system axis names from the base model, which does not match Minecraft axis name distribution. In Minecraft, y- and z- axis are swapped. Y-axis is the vertical axis. Though due to relative agent instructions, this change has little effect on the implementation – the only affected location is the naming scheme of the enter command, which takes <x'> <y'> as arguments during conversion from MILP agent actions to relative instructions (in 3.1) and <x> <z> as arguments for conversion from relative agent instructions to Malmo commands. Unless explicitly referring to the coordinates in Minecraft, the coordinates in this thesis use fraction-time model axis names.

Direction based rotation in the context of the table means agent rotation in degrees, used according to the <action direction> (action direction N means 180°, S means 0°, W means 90°, E means −90°). Agent waiting position is the position above agent-assigned gold block. Direction based relative position means position one-block in the direction of <action direction>. Ground y is the position of the ground (grass layer) in the Minecraft y-axis.

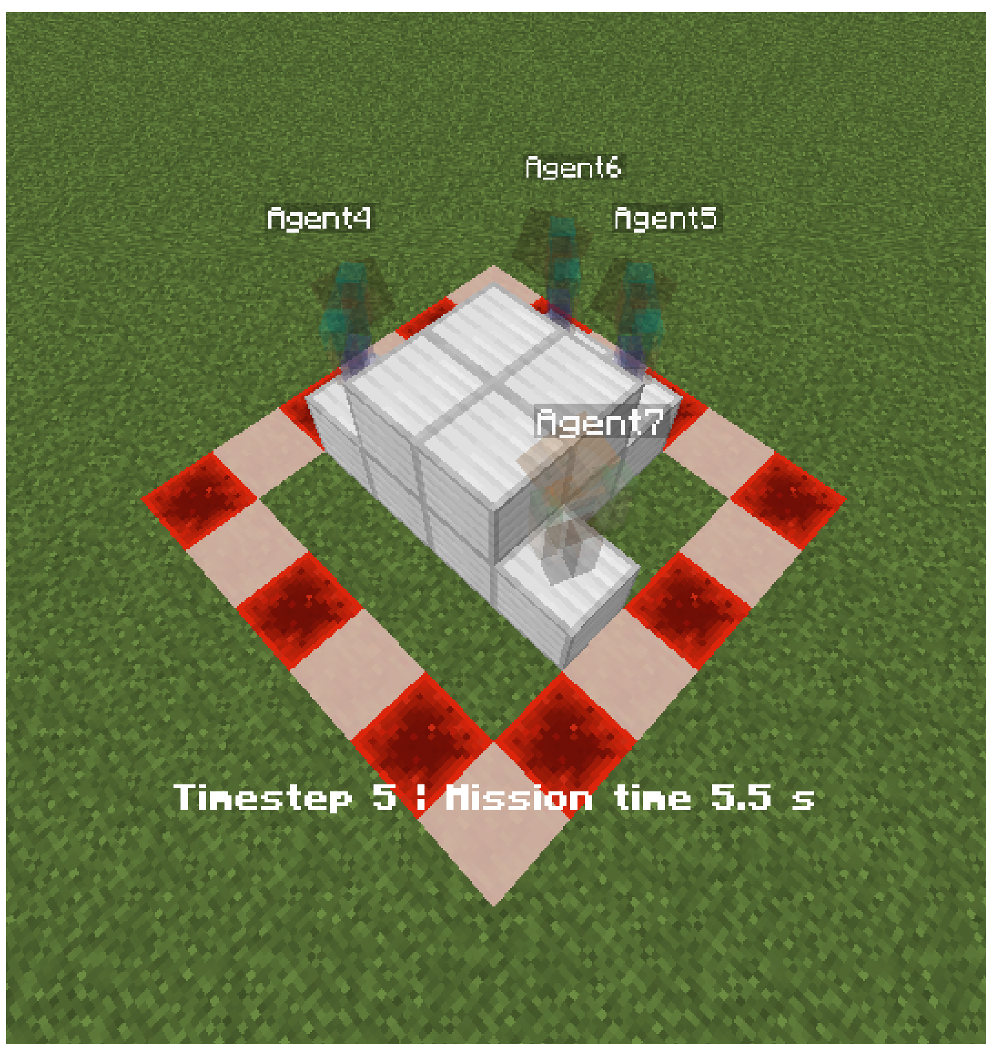


Figure 3.4: Fifth timestep of mission to build a small cube

An example of agent controller directing agents is shown on image 3.4, where agents are about to move down from scaffolding blocks, remove them and leave. The whole construction process is shown in appendix of this thesis, where is also a problem instance defining file and generated agent instructions file. After all relative instructions are converted and all the Malmo commands are sent to their assigned agents, main agent controller loop ends and observer agent calls the end of the mission (using Malmo mission quit command).

relative instruction	Malmo command list
enter <x> <z> <minecraft block name>	chat /setblock <AWP** x> <ground y - 1> <AWP** z> <minecraft block name> chat /tp @p <x> ~ <z> 0 60 if show_held_blocks, chat /clone <AWP** x> <ground y - 1> <AWP** z> <AWP** x> <ground y - 1> <AWP** z> ~ ~2 ~ replace
enter <x> <z>	chat /setblock <AWP** x> <ground y - 1> <AWP** z> air chat /tp @p <x> ~ <z> 0 60 if show_held_blocks, chat /clone <AWP** x> <ground y - 1> <AWP** z> <AWP** x> <ground y - 1> <AWP** z> ~ ~2 ~ replace
leave	if show_held_blocks, chat /setblock ~ ~2 ~ air chat /tp @p <AWP** x> ~ <AWP** z> 0 60
move <action direction>	if show_held_blocks and agent holding block, chat /setblock ~ ~2 ~ air chat /tp @p <DBRP*** x> ~ <DBRP*** z> <DBR*> 60 if show_held_blocks and agent holding block, chat /clone <AWP** x> <ground y - 1> <AWP** z> <AWP** x> <ground y - 1> <AWP** z> ~ ~2 ~ replace
jump_move <action direction>	if show_held_blocks and agent holding block, chat /setblock ~ ~2 ~ air chat /tp @p <DBRP*** x> ~1 <DBRP*** z> <DBR*> 60 if show_held_blocks and agent holding block, chat /clone <AWP** x> <ground y - 1> <AWP** z> <AWP** x> <ground y - 1> <AWP** z> ~ ~2 ~ replace
place_block <action direction>	chat /tp @p ~ ~ ~ <DBR*> 60 if show_held_blocks, chat /setblock ~ ~2 ~ air chat /clone <AWP** x> <ground y - 1> <AWP** z> <AWP** x> <ground y - 1> <AWP** z> <DBRP*** x> ~ <DBRP*** z> replace move
break_block <action direction>	chat /tp @p ~ ~ ~ <DBR*> 60 chat /clone <DBRP*** x> ~ <DBRP*** z> <DBRP*** x> ~ <DBRP*** z> <AWP** x> <ground y - 1> <AWP** z> replace move if show_held_blocks, chat /clone <AWP** x> <ground y - 1> <AWP** z> <AWP** x> <ground y - 1> <AWP** z> ~ ~2 ~ replace
wait <timestep count>	Empty list; if <timestep count> > 1, replace with wait <timestep count - 1>

* direction based rotation
** agent waiting position
*** direction based relative position

Table 3.2: Projection from relative instructions to Malmo commands

3.4 Implementation Challenges

Implementation of the designed solution is often met with challenges and setbacks. Some of the more unexpected problems and their chosen solutions are shortly described in this section.

3.4.1 Minecraft Local Server Agent Limit

Malmo environment uses Minecraft local server for multi-agent missions, which limits the connected agents to 8. Minecraft without Malmo offers version for public servers, where the number of allowed connections is configurable, but choosing this option would remove the advantages for which Malmo was chosen (namely Malmo API and being able to use multiple Minecraft instances without requiring a paid account). Ultimately, much better solution was found – the limit of local server connections is stored in a not-easily accessible inner variable, accessible inside the source code of Malmo. Small modification of the Malmo source code is therefore done as simple means to remove the software limit of mission agents. This modification is marked in the attached code, located in the directory `application/src/MalmoModifiedFiles`. Modifications are done at lines 961–963 in `ClientStateMachine.java` and at lines 75–80 and 189–202 in `MalmoMod.java`.

3.4.2 Malmo Discrete Movement Commands Non-determinism

During early testing of construction visualizer (with originally selected discrete movement commands), small errors, such as misplaced or missing blocks were observed within the built structure. After brief investigation, skipped execution of a discrete movement command has been determined to be the cause. On average, construction of structure of problem instance 1 from the experiments section contained one error (occurring at different parts of the structure, when visualization was run multiple times from the same intermediate YAML file, with 21 agents). To avoid the complications of non-deterministic action execution, switch to equivalent Malmo chat commands was made, which proved to be deterministic.

3.5 Experiments

3.5.1 Action Duration Experiment Description

To prove that the fraction-time model allows reduction of the construction duration, three structures are selected to be built using three sets of action type durations – 1-timestep actions, 1-2-timestep actions and 1-2-3-timestep actions. 1-timestep-action-type-duration-set assigns 1-timestep duration to

every action type. 1-2-timestep- and 1-2-3-timestep-action-type-duration-sets assign durations to action types according to the number of Malmo commands they are mapped to (without held blocks shown and with held blocks shown respectively – according to the two projection tables 3.1 and 3.2). Exact action timestep durations for 1-2-timestep- and 1-2-3-timestep-action-type-duration-sets are written in table 3.3. 1-timestep actions are chosen as a benchmark, as they reduce the fraction-time model to a state similar to the base model in [4] – the difference being the inclusion of 'enter' action type into the objective function for sum-of-costs (secondary optimization criterion) and constraints for vertex 1.18 and edge collisions 1.19 replaced by single constraint 2.9 preventing both by not allowing concurrent actions to share vertices (and assigning both start- and end-position as action vertices for the duration of the action). As such, the 1-timestep actions set used in fraction-time model is used to simulate constant-time actions model (a variation on the base model from [4]), which serves for the three experiment instances as a comparison model. 1-2-timestep and 1-2-3-timestep actions are measured against it. For the comparison, the constant-time model is assumed to use timesteps with the duration of maximum number of Malmo commands any action can map to (labeled t_{max}). This assumption is chosen, because the implementation of construction visualizer uses it to perform actions when action durations of the problem instance for the model are too short for executing the full Malmo command list. Makespan and sum-of-costs in 3.4 are multiplied by t_{max} (shown to the right of '/'), because both of these are time-related and timesteps for the constant-time model would require t_{max} times longer time to execute.

The three problem instances 3.5 selected for the test are modified problem instances from the paper of the base model, selected for their low computational requirements (compared to the rest) in anticipation of higher computational requirements of the generalized model. Their modification consists of removal of surrounding area without blocks around the built structure. Each instance is run 10 times for every configuration (i. e. every set of the three sets of action type durations).

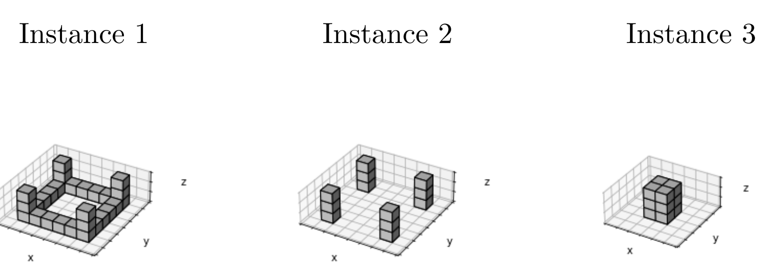


Figure 3.5: Three problem instances used for experiments

3.5.2 Action Duration Experiment Setup

The experiment is performed with Gurobi 9.1.1, using up to 32 threads on Intel Skylake processor with 16 physical cores with multi-threading feature (show as 32 cores within the system), running at 2993 MHz with 132025000 kB of RAM. For each experiment instance, 3 problem instance YAML files are created (for 1-timestep, 1-2-timestep and 1-2-3-timestep action type sets), 9 input files in total. Every input file is run 10 times, optimizing both makespan and sum-of-costs. The maximum solver run-time is set to 24 hours (for each of the 10 repeats independently). Maximum number of agents set to 20. Action type durations are displayed in table 3.3. Maximum action durations are used by the constant-time model to fit all the action types. Due to this fact, 1-2-timestep action set uses 2-timestep actions in the constant model and 1-2-3-timestep action set uses 3-timestep actions. This means, that constant-time model with 2-timestep actions has double the makespan and sum-of-costs of constant-time model with 1-timestep actions. Constant-time model with 3-timestep actions has three times the makespan and sum-of-costs of constant-time model with 1-timestep actions. Comparison of fraction-time model with 1-2-timestep actions and constant-time model with 2-timestep actions, as well as comparison of fraction-time model with 1-2-3-timestep actions and constant-time model with 3-timestep actions is shown in table 3.4. Let b_t be the amount of time spent executing just Malmo commands within the constant-time model, relative to total instruction list execution time (including the synchronization waiting times) and call it 'busy time'. 'Busy time' then provides lower bound for time required for executing Malmo commands. In an ideal situation, where every action type (but 'wait') is equally likely to occur, 'wait' actions do not occur and infinite plan is used, can value of b_t be calculated by the equation 3.1.

$$b_t = \frac{T_{enter} + T_{leave} + T_{move_block} + T_{move_empty} + T_{deliver} + T_{pick_up}}{6 \max \left(T_{enter} + T_{leave} + T_{move_block} + T_{move_empty} + T_{deliver} + T_{pick_up}\right)} \tag{3.1}$$

3.5.3 Action Duration Experiment Results

The results are shown in table 3.4. Since the problem instance is solved to optimality, both of its optimization criteria (makespan and sum-of-costs) are the same for every out of 10 runs. It is clearly visible, that fraction-time model provides solutions with smaller makespan, then would provide an equivalent constant-time action model. When averaged over the three instances, the fraction time model provides approximately 1.3 times ($\doteq \frac{1}{0.771}$ – see 'average' section of table 3.4) better solution than the simulated constant-time model for 1-2-timestep actions. Similarly, for 1-2-3-timestep actions, the fraction-time model provides approximately 1.2 times ($\doteq \frac{1}{0.839}$ – see 'average' section of table

Action type name	1-timestep actions	1-2-timestep actions	1-2-3-timestep actions
enter	1	2	3
leave	1	1	2
move_block	1	1	3
move_empty	1	1	1
pick_up	1	2	3
deliver	1	2	3
max	1	2	3
‘busy time’ fraction	1	0.75	0.83

Table 3.3: Action type duration for 1-timestep, 1-2-timestep and 1-2-3-timestep action sets

Instance	Actions type set	Run-time Mean [s]	Run-time Sampling variance [s^2]	Makespan / 1-timestep action equivalent	Sum-of-costs / 1-timestep action equivalent	Robots
1	1-timestep	5.474	8,116E-02	11 / 11	132 / 132	20
	1-2-timestep	24.092	6.324E-01	17 / 22	200 / 264	20
	1-2-3-timestep	93.131	1.919E+02	29 / 33	340 / 396	20
2	1-timestep	15.176	1.985E+00	12 / 12	152 / 152	20
	1-2-timestep	252.616	9.339E+03	19 / 24	232 / 304	20
	1-2-3-timestep	475.056	2.176E+03	32 / 36	396 / 456	20
3	1-timestep	450.664	1.205E+04	14 / 14	142 / 142	16
	1-2-timestep	786.215	8.507E+03	21 / 28	213 / 284	17
	1-2-3-timestep	11891.322	4.075E+06	37 / 42	352 / 426	17
average				1	1	
				0.771	0.757	
				0.839	0.851	

Table 3.4: Experimental results

3.4) better solution than the constant-time model. ‘Better’ solution in this context means 1.3 (respectively 1.2) times lower makespan and sum-of-costs value, when compared to constant-time model with 2-timestep (respectively 3-timestep) action durations. The decrease of makespan and sum-of-costs for the fraction-time model can be seen in figure 3.6. The main cause of the difference is probably the amount of ‘busy time’ within the constant-time model (see equation 3.1), as it almost completely matches the measured average values in table 3.4. While the equation 3.1 preconditions are not met, so the approximated b_t value is not the actual lower bound, the equation still seems to predict the added efficiency of fraction-time model, which may be beneficial for determining, if the reduced makespan is worth the considerably higher computational time.

Computational time (‘run-time’ in the table 3.4) is increasing as expected (significant increase of computation time with increase of makespan has already been noted in [4]). The quick increase of computation time can also be seen in figure 3.7, which uses standard box-plots to show the increases in Gurobi runtime. Significant spread of possible computation times for single instance is caused by heuristic optimization methods of Gurobi Solver. The main reason for measuring it is to approximate how significant solution computational time variations can be. When looking back at table 3.4, the

most notable sampling variation is in instance 2, 2-timestep action types, with value of standard deviation approximation of 96.64 s, which is quite significant, considering the mean value is 252.616 s. This observation confirms, that, for future studies of this model, at least 10 measurements must be taken for every data-point connected to solving time.

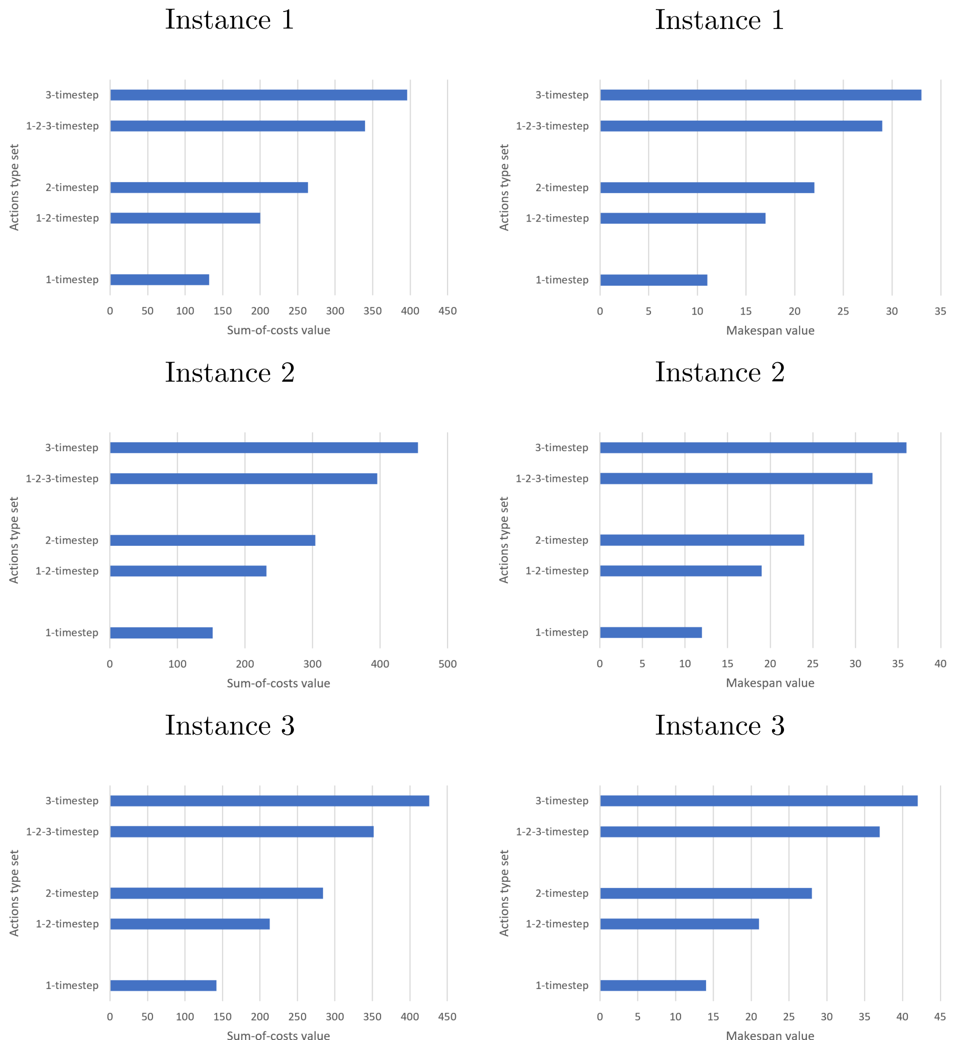


Figure 3.6: Makespan and sum-of-costs for instances 1, 2 and 3

3.5.3.1 Influence of Action Type Durations on Action Type Count

It is expected, that action types with shorter durations will be more prevalent, relative to constant-time model, than action types with longer durations. This expectation (and its associated difference in action type prevalence) is tested

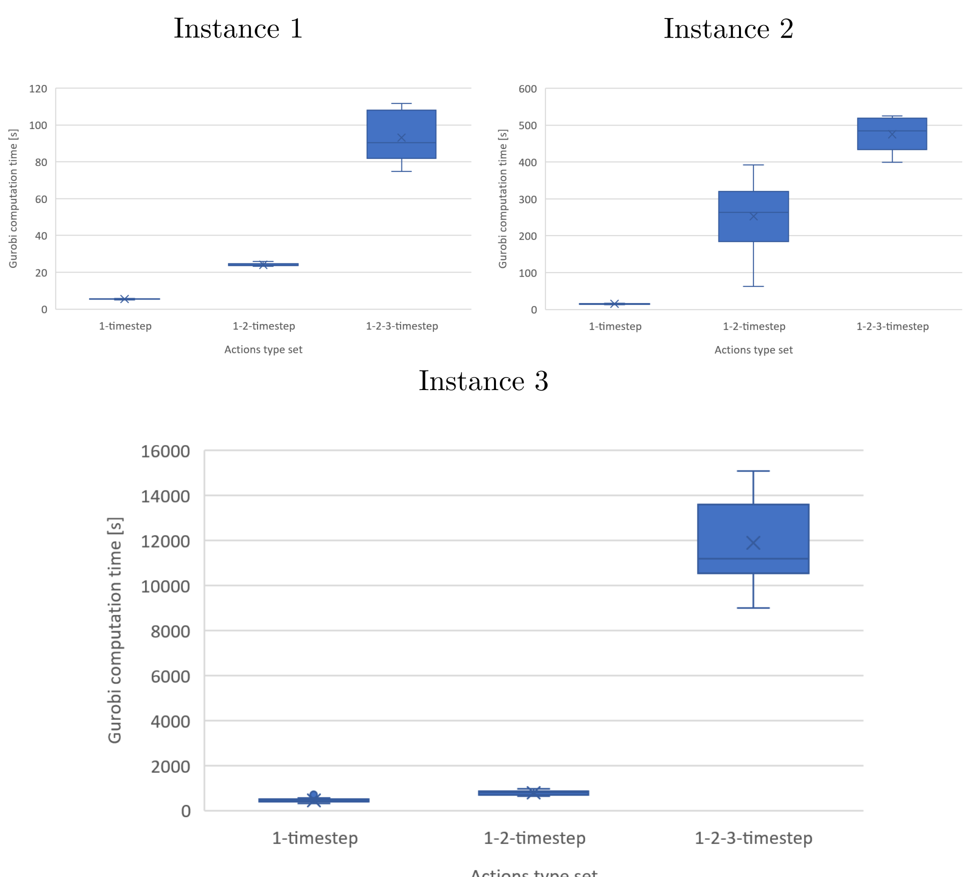


Figure 3.7: Gurobi computational time for instances 1, 2 and 3

using additional data from the first experiment (outputs of `solver_main`). Based on the data, visualized in graphs 3.8, this is not necessarily the case as instances 1 and 2 show the same number of relative instructions per type. The very likely cause of this phenomenon is the close proximity of structures of both instances to the border cells, as there is little to optimize on sub-plan `enter`, `place_block`, `leave` (the spires in the corners also proved to be too simple to optimize). In contrast, relative instruction counts for instance 3 show the expected change in the instruction type counts. One exception is the count of `enter` instruction, which increased for 1-2-action times, while having the duration of 2 timesteps (maximum duration for this action set). This is likely caused by shorter-duration instructions requiring more robots to enter the building area. Graph 3.9 shows total execution time in timesteps of relative instructions. Each horizontal double column consists of upper column, showing execution time for constant-time actions, and lower column, showing execution time for fraction-time actions. Graphs for instances 1 and 2 show expected lowered execution times due to exact action-duration mapping of the

fraction time model. Graph for instance 3 shows additional time-saves being made by performing more shorter duration actions and less longer duration actions.

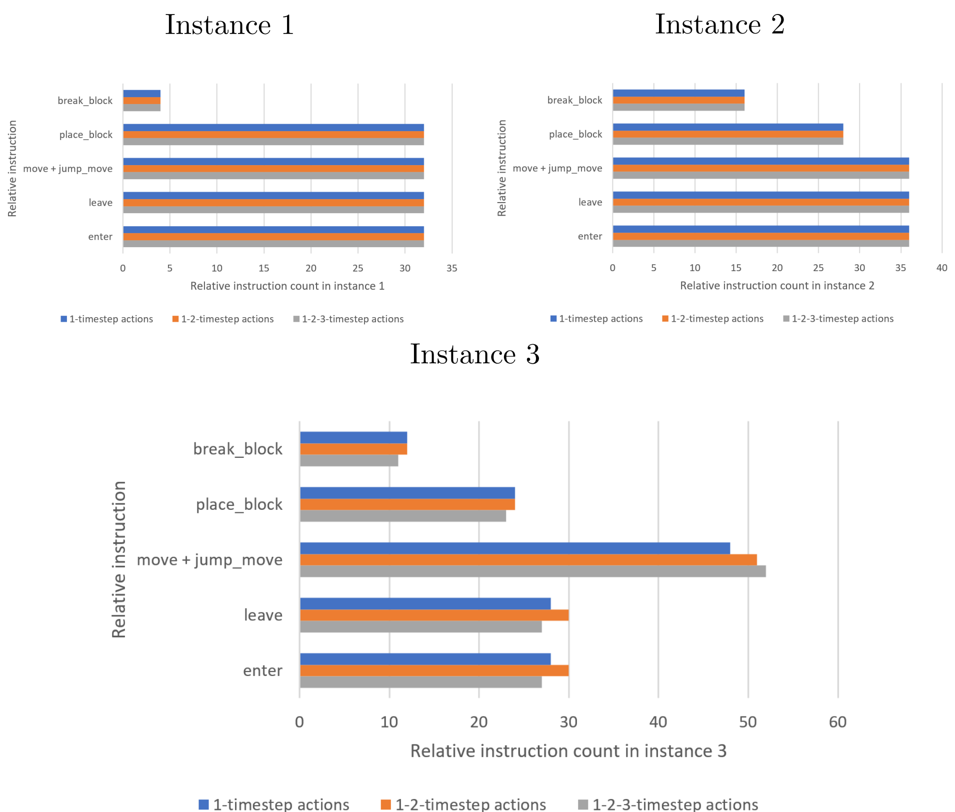


Figure 3.8: Relative instruction counts for the three instances

3.5.4 Robot Count Experiment Description and Setup

The second experiment studies the behaviour of the fraction-time model, when limited number of agents is available. For this experiment, instance 3 (see 3.10) of the last experiment is chosen, as the maximum number of usable agents has already been found (16 for 1-timestep actions and 17 for 1-2-timestep- and 1-2-3-timestep-action-type-sets). The experiment uses only 1-timestep actions, because the makespan and sum-of-costs of 1-2-timestep- and 1-2-3-timestep-action-type-sets can be estimated from it using equation 3.1 (as has been established in the previous experiment). The maximum number of robots will be decreasing from 17 to 1 (by 1), until instance with 1 agent limit is computed, or an instance takes longer than 8 hours for the Gurobi solver to compute. The experimental setup used the same hardware, as the previous experiment.

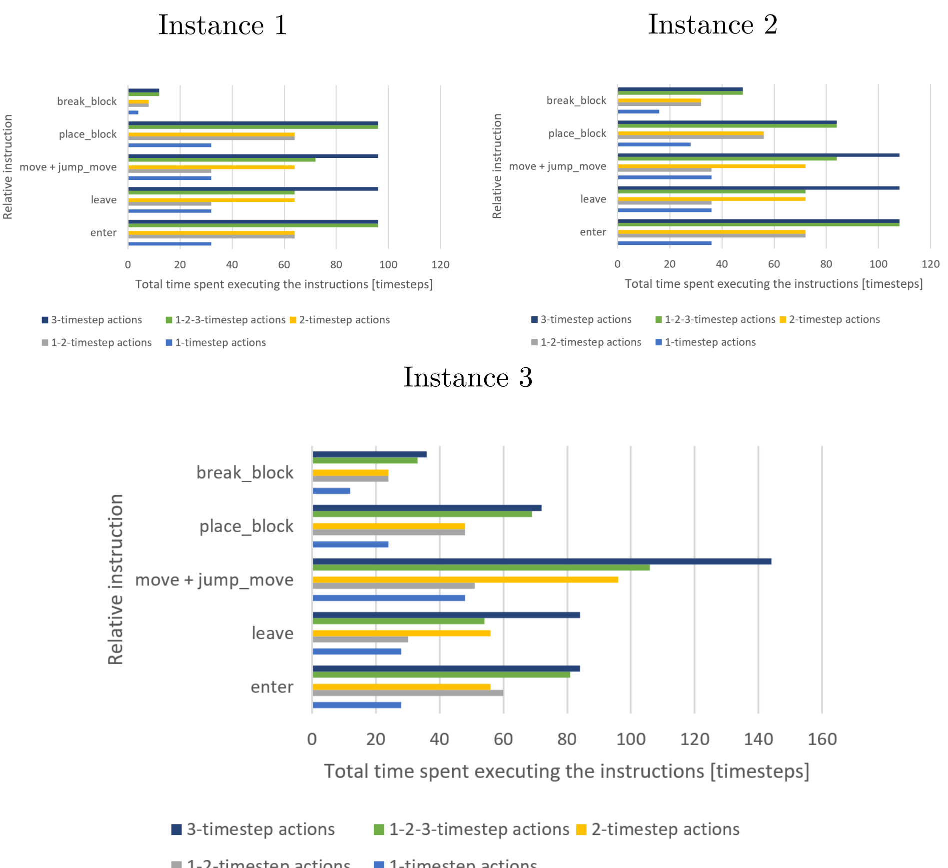


Figure 3.9: Cumulative relative instruction execution time for the three instances

3.5.5 Robot Count Experiment Results

The data from the experiment are shown in figures 3.11, 3.12 and 3.13. The figures stop at 5 robots, because the experiment with 4 robots exceeded computational time limit. Figure 3.11 shows the expected increase in makespan and decrease in sum-of-cost values (lower numbers of agents are more easily assigned to tasks, that are not dependent on each other, rising the efficiency of individual agents, thus lowering the sum-of-costs at the cost of higher makespan). It should be noted, that relatively high number of agents can be removed at first, without significantly increasing the makespan (4 agents from 16 usable can be removed at the cost of increasing the makespan by 1 timestep). This is in line with an assumption, that the increase in makespan should be approximately exponential (would be exponential, if the problem instance could be divided into the-agent-number of equal sub-problems, where agents could work entirely independently).

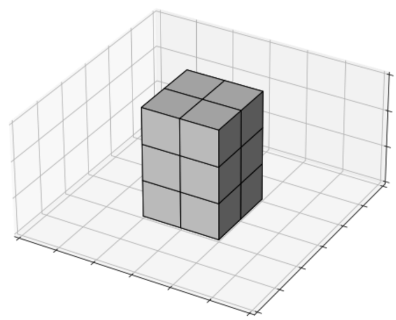


Figure 3.10: Instance 3, used again for experiment with robot count

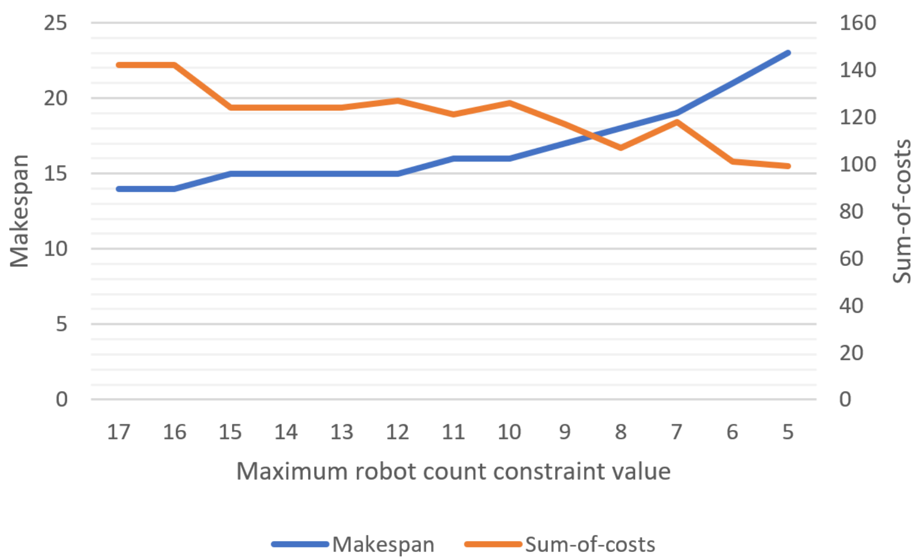


Figure 3.11: Makespan and sum-of-costs for decreasing maximum agent count

Figure 3.12 shows the overall computational complexity to be rising with the decreasing number of agents, while also being subject to local decreases. This trend is caused by two factors with opposite effects – increasing the makespan increases the computational complexity and reducing the number of robots decreases the computational complexity (because of the stricter constraint). The side effects can be quite clearly seen during the initial slow increase of makespan, when going from 16 to 15 robots and from 12 to 11 robots (where makespan increases from 14 to 15 and from 15 to 16 respectively). In both cases, the increase of makespan ends local decrease in computational complexity.

Figure 3.13 shows already mentioned relative increases in agent time spent on the grid. It is not an objective of the fraction-time model to use all

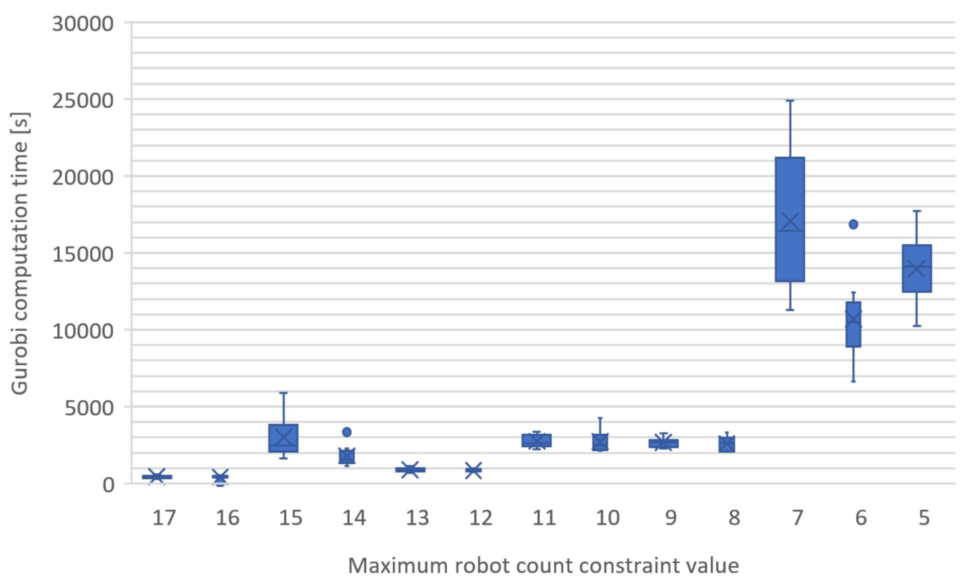


Figure 3.12: Computational runtime for decreasing maximum agent count

the agents for the entire duration of the mission. When the agent is no longer needed on the grid, the sum-of-costs secondary criterion ensures that the agent leaves the grid immediately.

The graph shown in figure 3.13 is meant to show that not all the agents are required to be on the grid for the whole duration of the mission and the fraction of time, when they are needed, decreases with increasing number of the agents. This is caused by the critical path. Some actions (building of access ramps for instance) must be performed in an exact order and provide minimum possible makespan for given number of agents and minimum makespan of the whole structure, if maximum usable number of agents for given structure is reached (though this number has been – for the current model so far – only experimentally gained). A less visible example can also be seen in the problem instance in the appendix of this thesis – agents 0, 1, 2 and 3 must wait outside the grid for agents 4, 5, 6 and 7 to finish performing their critical path actions. In this case, there are four, equally long, critical paths. To reduce the makespan further, all four would have to be shortened.

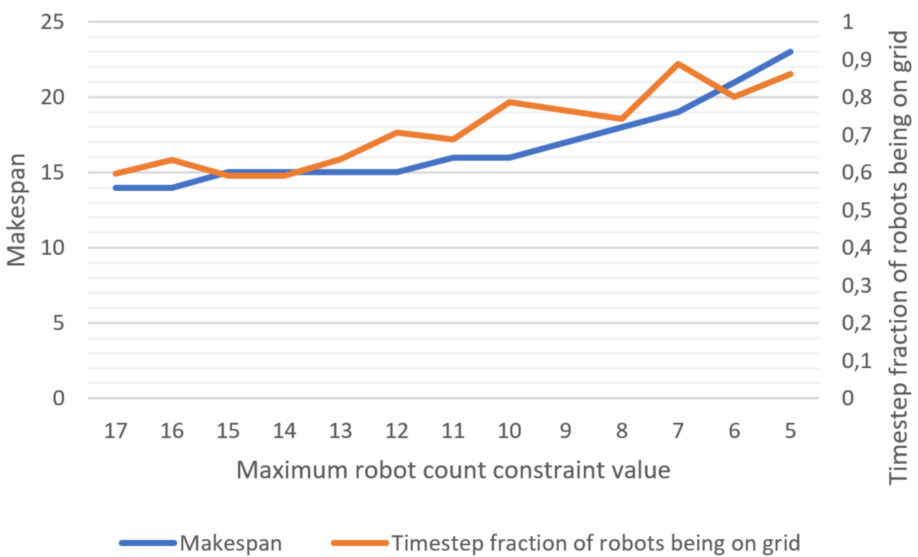


Figure 3.13: Makespan and relative time spent on the grid for decreasing maximum agent count

Conclusion

This bachelor thesis covers the creation of two-part multi-agent construction system application. The first part is a problem solver, which uses proposed fraction-time MILP model to convert user-entered structure description to an optimal construction plan. The plan consists of instructions lists assigned to agents and its execution time is proven to be optimal (given set execution times for each robot action type). Second application serves as a visualizer, using Malmo API to run agent instruction lists within the Minecraft world. Since the solver and visualizer are separated into two independent applications, any construction plan needs to be computed only once and then can be visualized as many times as necessary, without the need to run the solver.

To achieve this result, study of problem modeling formalisms has been performed, with focus on mixed integer linear programming and constraint programming. Ultimately, MILP has been selected for its significant computational speed advantage over CP, when solving model of multi-agent construction problem similar to the one of this thesis. Gurobi solver has been selected for its high solving speed and because it has been used by the base construction model.

The most important contribution of this thesis is the proposed fraction-time MILP model. The model has been created as a generalization of a previous multi-agent construction model, replacing its one-tick duration actions with n-tick duration action types, where n can be – theoretically – any positive integer. In practice, high values of n cannot be currently used because of large computational requirements of long-makespan construction tasks. Though large values of n are not necessary – the fraction-time model is used to accommodate different durations of actions caused by conversion to Malmo API commands, where $n \in \{1, 2, 3\}$. The model is called 'fraction-time', because any set of simple fractions can be converted to integers by multiplying them with least common multiple of their denominators. This method is incorporated in the application (both the solver and the visualizer) to allow the user to enter fraction action times.

A set of experiments has been created to test the the approach. In the first experiment, three different cube structures have been used to measure the relative construction time reduction when comparing one-tick action times to n-tick action times used by Malmo construction visualizer. Computational time increase for the n-tick action times has been also measured. By using the proposed fraction-time model, construction duration has been reduced by approximately 25 %, when carried blocks are not visualized, otherwise by 16 % (relative to previous best exact optimization model, which uses constant action times). The decrease in makespan can be roughly predicted based on the constant-time model solution and used action times. This can help determine, if benefits of the fraction-time makespan reduction outweigh the cost of longer computation time.

The second experiment observes the relation between the maximum number of agents and the solution properties, such as makespan, sum-of-costs, computation time and robot utilization. Lowering the number of agents led to expected increase in makespan and overall increase in computational complexity. Local, less significant, decreases in computational complexity have been observed as part of a stricter constraint on the number of agents. Furthermore, a critical path has been observed in the generated agent instructions file. Increases in the lengths of critical paths of the experimental problem instance have been observed when decreasing the number of available agents for the instance. In other words, the fraction-time model is shown to handle tens of agents, even in small construction spaces, as long as the makespan is sufficiently small.

In conclusion, all the goals have been fulfilled. The proposed fraction-time model provides significant reduction of building time for Malmo agents. Developed application set uses this model to convert user-entered structure to an optimal building plan, which is then executed in Minecraft. Moreover, the fraction-time model created as part of this thesis, can also be used to map construction problems outside the Minecraft environment – namely the TERMES multi-agent system, which can also benefit from accommodation of different action times.

The main challenge for the model is large makespan (construction time), which is required to build large structures or medium structures with small numbers of agents. Future work might be required to reduce the computation requirements of exact multi-agent construction models and allow for planning of larger, more complex structures.

Bibliography

1. BARROS DOS SANTOS, Sérgio R.; GIVIGI, Sidney; NASCIMENTO, Cairo L.; FERNANDES, Jose M.; BUONOCORE, Luciano; ALMEIDA NETO, Areolino de. Iterative Decentralized Planning for Collective Construction Tasks with Quadrotors. *Journal of Intelligent & Robotic Systems*. 2018, vol. 90, no. 1, pp. 217–234. ISSN 1573-0409. Available from DOI: `10.1007/s10846-017-0659-6`.
2. SABOIA, Maira; THANGAVELU, Vivek; NAPP, Nils. Autonomous multi-material construction with a heterogeneous robot team. *Robotics and Autonomous Systems*. 2019, vol. 121, p. 103239. ISSN 0921-8890. Available from DOI: `https://doi.org/10.1016/j.robot.2019.07.009`.
3. PETERSEN, Kirstin Hagelskjaer; NAGPAL, Radhika; WERFEL, Justin K. Termes: An autonomous robotic system for three-dimensional collective construction. *Robotics: science and systems VII*. 2011.
4. KOENIG, Sven; KUMAR, TK Satish. Exact Approaches to the Multi-Agent Collective Construction Problem. [N.d.].
5. JOHNSON, Matthew; HOFMANN, Katja; HUTTON, Tim; BIGNELL, David. The Malmo Platform for Artificial Intelligence Experimentation. In: *IJCAI*. 2016, pp. 4246–4247.
6. RUSSELL, Stuart. *Artificial intelligence : a modern approach*. Englewood Cliffs, N.J: Prentice Hall, 1995. ISBN 0-13-103805-2.
7. AHUJA, Ravindra K.; MAGNANTI, Thomas L.; ORLIN, James B. *Network Flows: Theory, Algorithms, and Applications*. USA: Prentice-Hall, Inc., 1993. ISBN 013617549X.
8. ROSSI, Francesca; VAN BEEK, Peter; WALSH, Toby (eds.). *Handbook of constraint programming*. 1st ed. Amsterdam ; Boston: Elsevier, 2006. Foundations of artificial intelligence. ISBN 9780444527264. OCLC: ocm70408044.

9. *OR-Tools* [online] [visited on 2021-03-31]. Available from: https://opensource.google/projects/or-tools.
10. JÜNGER, M. *50 years of integer programming 1958-2008 : the early years and state-of-the-art surveys*. Berlin London: Springer, 2010. ISBN 978-3-540-68274-5.
11. *Tutorial: Mixed-Integer Linear Programming* [online]. Gurobi Optimization [visited on 2021-04-15]. Available from: https://www.gurobi.com/resource/tutorial-mixed-integer-linear-programming/.
12. *Gurobi Optimizer - Gurobi* [online]. Gurobi Optimization [visited on 2021-04-01]. Available from: https://www.gurobi.com/products/gurobi-optimizer/.
13. *Tutorial: Mixed-Integer Linear Programming* [online]. Gurobi Optimization [visited on 2021-04-15]. Available from: https://www.gurobi.com/resource/tutorial-mixed-integer-linear-programming/.
14. PERSSON, Markus Alexej; BERGENSTEN, Jens Peder. *Minecraft* [comp. software]. Mojang Studios, 2016. Version 1.11 [visited on 2021-04-06]. Available from: https://www.minecraft.net/.
15. *Official Minecraft Wiki – The Ultimate Resource for Minecraft* [online]. Fandom, 2021 [visited on 2021-03-25]. Available from: https://minecraft.fandom.com/wiki/Minecraft_Wiki.
16. *World type – Official Minecraft Wiki* [online]. Fandom, 2021 [visited on 2021-03-25]. Available from: https://minecraft.fandom.com/wiki/World_type.
17. *Player – Official Minecraft Wiki* [online]. Fandom, 2021 [visited on 2021-03-25]. Available from: https://minecraft.fandom.com/wiki/Player%5C#Gamemodes.
18. LINN, Allison. *Project Malmo, which lets researchers use Minecraft for AI research, makes public debut* [online]. Microsoft, c2021 [visited on 2021-03-21]. Available from: https://blogs.microsoft.com/ai/project-malmo-lets-researchers-use-minecraft-ai-research-makes-public-debut/%5C#sm.00001sywt8ebfmf2ttmmz73uyjgl6.
19. *XML Schema Documentation* [online]. 2017 [visited on 2021-03-21]. Available from: https://microsoft.github.io/malmo/0.30.0/Schemas/Mission.html.
20. *Commands – Official Minecraft Wiki* [online]. Fandom, 2021 [visited on 2021-04-06]. Available from: https://minecraft.fandom.com/wiki/Commands.
21. *Project Malmo: Main Page* [online]. 2017 [visited on 2021-04-22]. Available from: https://microsoft.github.io/malmo/0.30.0/Documentation/index.html.

Appendix A

Acronyms

LP Linear programming

MILP Mixed integer linear programming

CP Constraint programming

Appendix B

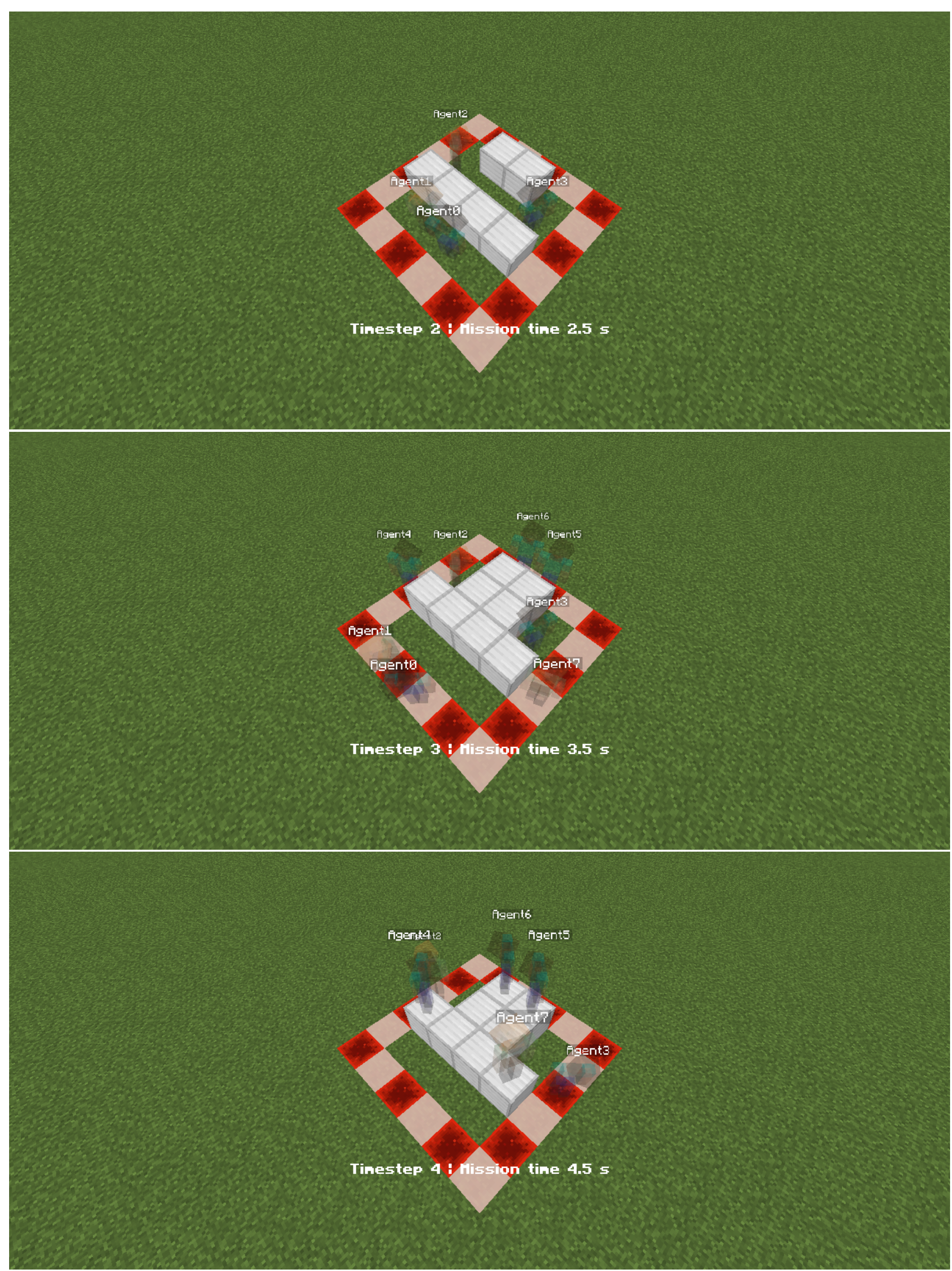
Appendix

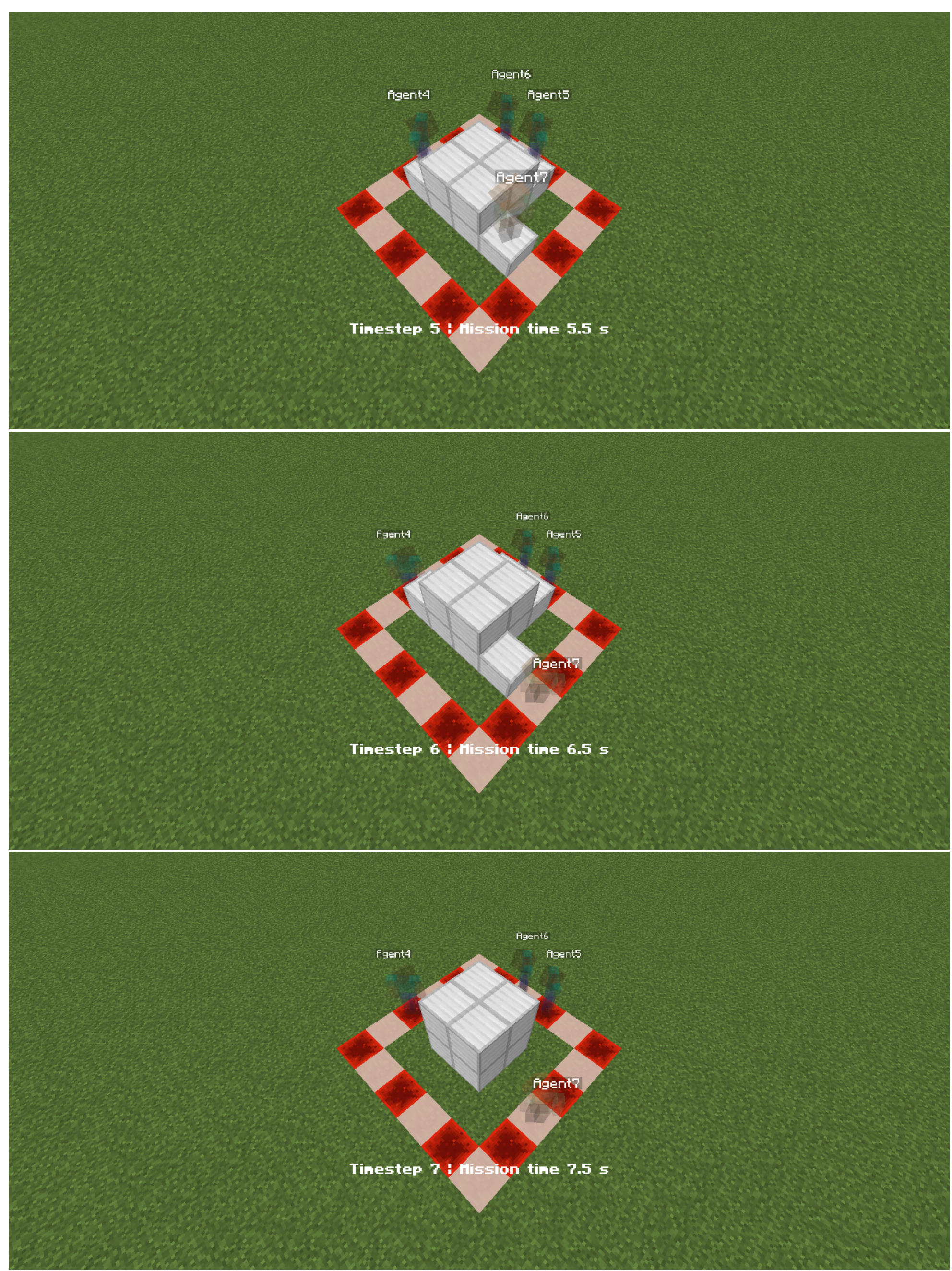
B.1 Example Problem Instance

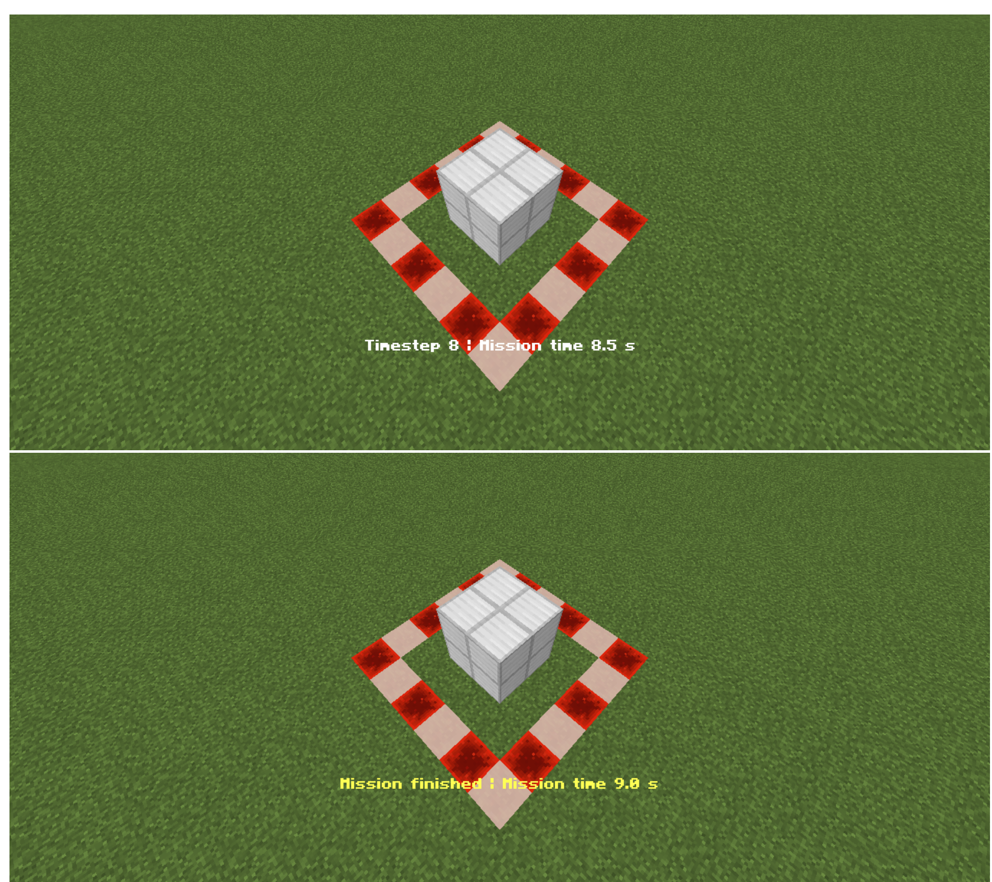
B.1.1 Problem Instance Specification File

```yaml
# base constraints
max_agent_count: 8
building_material: iron_block
# action type times
entry_time: 1
leave_time: 1
move_time_without_block: 1
move_time_with_block: 1
pick_up_time: 1
deliver_time: 1
# height map of the building
building_heights:
- [0, 0, 0, 0]
- [0, 2, 2, 0]
- [0, 2, 2, 0]
- [0, 0, 0, 0]
```

B.1.2 Construction Visualizer Controlled Minecraft Graphical Output


Agent2
Agent1
Agent3
Agent0
Timestep 2 | Mission time 2.5 s
Agent6
Agent4
Agent2
Agent5
Agent3
Agent1
Agent0
Agent7
Timestep 3 | Mission time 3.5 s
Agent6
Agent5
Agent7
Agent3
Timestep 4 | Mission time 4.5 s


Agent6
Agent4
Agent5
Agent7
Timestep 5 | Mission time 5.5 s
Agent6
Agent4
Agent5
Agent7
Timestep 6 | Mission time 6.5 s
Agent6
Agent4
Agent5
Agent7
Timestep 7 | Mission time 7.5 s



B.1.3 Agent Instructions File

```
agents:
- agent_name: Agent0
  instructions:
  - enter 2 0 iron_block
  - move S
  - place_block S
  - move N
  - leave
- agent_name: Agent1
  instructions:
  - enter 3 0 iron_block
  - move S
  - place_block S
  - move N
  - leave
- agent_name: Agent2
```

```
17    instructions:
18    - wait 2
19    - enter 5 3 iron_block
20    - move W
21    - place_block W
22    - move E
23    - leave
24  - agent_name: Agent3
25    instructions:
26    - wait 2
27    - enter 0 3 iron_block
28    - move E
29    - place_block E
30    - move W
31    - leave
32  - agent_name: Agent4
33    instructions:
34    - enter 5 2 iron_block
35    - place_block W
36    - leave
37    - enter 5 2 iron_block
38    - jump_move W
39    - place_block W
40    - move E
41    - break_block W
42    - leave
43  - agent_name: Agent5
44    instructions:
45    - enter 2 5 iron_block
46    - place_block N
47    - leave
48    - enter 2 5 iron_block
49    - jump_move N
50    - place_block N
51    - move S
52    - break_block N
53    - leave
54  - agent_name: Agent6
55    instructions:
56    - enter 3 5 iron_block
57    - place_block N
58    - leave
59    - enter 3 5 iron_block
60    - jump_move N
```

```yaml
    - place_block N
    - move S
    - break_block N
    - leave
- agent_name: Agent7
  instructions:
    - enter 0 2 iron_block
    - place_block E
    - leave
    - enter 0 2 iron_block
    - jump_move E
    - place_block E
    - move W
    - break_block E
    - leave
building_area:
  size_x: 6
  size_z: 6
deliver_time: 1
entry_time: 1
leave_time: 1
mission_ticks: 9
move_time_with_block: 1
move_time_without_block: 1
pick_up_time: 1
show_held_blocks: false
```

Appendix C

Contents of Enclosed CD

```
readme.txt..........................the file with CD contents description
application..............the directory with the application related files
    src .....................the directory with the application source files
    examples...................the directory with the example input files
    run_solver_main.sh ..........run script for solver_main application
    run_visualizer_main.sh..run script for visualizer_main application
    setup.sh.......................installation script for the application
text ..........................................the thesis text directory
    src ...................the directory of LaTeX source codes of the thesis
    thesis.pdf ...........................the thesis text in PDF format
```