Leveraging Learned Graph-Based Heuristics for Efficiently Solving the Combinatorics of Assembly

Verwendung von gelernten graphenbasierten Heuristiken zum effizienten Lösen von kombinatorisch komplexen Assemblierungsproblemen

Bachelor thesis by Svenja Menzenbach

Date of submission: March 7, 2022

1. Review: Niklas Funk
2. Review: Jan Peters

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Svenja Menzenbach, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 7. März 2022

S. Menzenbach

Abstract

In the construction industry, conventional practices still take a large part, but they suffer from problems like the current lack of skilled workers, dangerous working conditions, resource scarcity, and their environmental impact. Automated processes could overcome those problems, but they still lack efficiency and reliability. We look at a difficult combinatorial optimization task where a target shape should be filled with a set of available parts and propose an approach that represents those architectural assembly problems as MILPs. Then, those are solved by sophisticated solvers with an optimality guarantee which is our main motivation using MILPs. However, they are NP-hard and thus get very inefficient for large problem instances. We show that the target shape and parts also influence the problem time, but their impact is insignificant unless the complexity results in a problem with only one solution.

To speed up the solving process, we did first attempts to use GNNs with multi-head attention (MHA). MILPs can be represented as graphs why GNNs are a common choice for this purpose. The network predicts the solution of the MILP that should serve as a warm start for the solver. The network manages to avoid placing parts outside the target shape, but it still has problems respecting the constraints which is most likely due to a weak feature set that does not make the nodes distinguishable enough. This problem could be solved by a sequential approach which is planned for future work.

Zusammenfassung

Hier können Sie Ihre deutsche Zusammenfassung schreiben.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
ACN	amount constraint node
B&B	branch-and-bound
CO	combinatorial optimization
DVN	decision variable node
FC	fully-connected
GCN	grid constraint node
GNN	graph neural network
leaky ReLU	leaky rectified linear unit

LP	linear program
MHA	multi-head attention
MILP	mixed integer linear program
MIP	mixed integer program

List of Symbols

Notation	Description
$\boldsymbol{\theta}$	vector of parameters from a probability distribution

List of Operators

Notation	Description	Operator
abs	the absolute value	$\mathrm{abs}\left(\bullet\right)$
$\log$	the logarithm	$\log\left(\bullet\right)$
sig	the sigmoid activation function	$\mathrm{sig}\left(\bullet\right)$

1 Introduction

In the construction industry, the ratio of automated processes in comparison to conventional practices remains low and is the main bottleneck for increasing its efficiency. However, efficiency improvements are crucial, especially considering the current lack of skilled labor [1], dangerous working conditions [2], resource scarcity, and the environmental impacts [3]. Autonomous assembly can achieve the increase in efficiency by the promising properties to have better scalability, adaptability, and customizability as well as enabling new design possibilities [4].

The applied intelligent robots should take over the decisions over plans, actions and executions [5] which introduces a variety of different research topics. This work will focus on planning the construction's composition. More precisely, the optimal way to fill a target shape with a set of available parts should be found. This is a difficult combinatorial optimization (CO) task as there are many ways to assemble those parts. So on the one hand, the solver has to find a feasible solution holding constraints coming from physical boundaries (e.g., collision/intersection, availability). And on the other hand, the solution should fit the target shape as well as possible.

Despite the task's complexity, it offers great potential to reduce the sector's environmental impact as on the one hand, it would allow using more irregularly shaped objects (i.e., waste elements from a previous building) for construction. On the other hand, it offers the potential to effectively integrate modular building blocks that could also be re-used after a building's lifetime.

Apart from our example of resource assignment in construction scenarios, many other relevant real-world problems are also of combinatorial complexity. Examples include finding the shortest paths in the fields of transportation and telecommunication, the most efficient packaging solutions in logistics, and many more. So, CO can be used in a wide range of applications of different fields. However, because all of the aforementioned problems are typically very hard to solve, the field of combinatorial optimization is still a

very active and interdisciplinary area of research [6]. So some insights discovered for CO in autonomous architectural assembly might also be useful for other applications.

One of the major bottlenecks of combinatorial optimization problems is their discrete nature which in the worst case requires the algorithms to extensively search all the possible solutions to return the optimal one, making them very slow, especially on bigger problem instances. In this work, the combinatorial optimization problem is formulated as a mixed integer program (MIP) which is a common way to represent CO problems. In general MIPs are solved using branch-and-bound (B&B), an algorithm that recursively decomposes the problems into sub-problems based on heuristics [7]. As the solution speed of the solvers heavily depends on the heuristic's quality, lately there has been an increasing interest in learning the heuristic functions [8]. Another way of circumventing the large runtimes of MIP solvers is to directly approximate the solver's solution using graph neural networks (GNNs). As we ultimately also want to solve big problem instances for resource assignments on construction sites, in this thesis, we will also investigate this path of learning efficient heuristics.

2 Motivation

A common approach to solve problems of combinatorics is using MILP [9, 10].

MILP is a model-based approach, i.e., the considered problem is formulated as a constrained optimization problem. As the name suggests, MILP therefore has integer and linear constraints. The integer constraint makes an MILP a combinatorial optimization. They can be solved to global optimality [11] which makes them appealing. In the application of assembly, it is important to find an optimal solution. If a non-optimal solution is found, parts could reach out of the target shape, which for multiple reasons might not be desired. It could be the lack of space, design reasons, or it might even interfere with the structural calculation of the construction. That is also the case with a non-optimal solution where parts of the desired shape are not filled. But note, that even though the optimal solution for a problem instance is found, it might be that the given parts cannot fill the shape perfectly.

The discrete nature of those combinatorial optimizations, implemented as integer constraints in an MILP, makes it a very complex task. So an MILP is NP-hard which can lead to long computation times. But, the solving can be accelerated if a good heuristic is used to lead the solver to a good and feasible solution.

In this work, such heuristics will be learned by using a graph neural network (GNN) with multi-head attention (MHA). However, GNNs are not only used to learn heuristics that support a classical solver. They are also applied as solvers themselves [5, 12]. Since networks, on the one hand, do not come with optimality guarantees, but on the other hand, are typically fast to evaluate, we will use them as a heuristic herein.

3 Foundations

In this chapter, the foundations of MILPs and GNNs are presented. Both methods are the basis of our approach in which we want to solve resource allocation problems fast, and reliably, but also with keeping optimality guarantees.

3.1 Mixed Integer Linear Program

A mixed integer linear program is an optimization problem. The goal is to minimize a linear objective function subject to linear constraints where all (or some) of the variables should be integer-valued. A generic formulation for an MILP problem is

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \\ & x_i \in \mathbb{Z}, \quad i \in \mathcal{I}, \end{aligned} \tag{3.1}$$

where $\mathbf{x} \in \mathbb{R}^n$ are the variables, $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{b} \in \mathbb{R}^m, \mathbf{c} \in \mathbb{R}^n, \mathbf{l} \in (\mathbb{R} \cup \{-\infty\})^n$, and $\mathbf{u} \in (\mathbb{R} \cup \{\infty\})^n$ are the given data, and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the index set of integer variables [8].

Although a linear problem is P-hard, the integer constraints make an MILP NP-hard with exponential complexity [13].

3.2 Linear Problem Relaxation

In order to find a lower bound for our MILP, the integer constraints are removed, which leads to a linear program (LP). A linear program is convex and can be solved efficiently. The optimal solution of the relaxed problem is guaranteed to be a lower bound because removing constraints can only expand the set of feasible solutions. If we got lucky and the optimal solution of the relaxed problem satisfies the integer constraints, even though they were not imposed, this solution is also the optimal solution of the original MILP. If not, which is more common, the branch-and-bound algorithm can be used to further restrict the decision boundaries to find an optimal solution [8, 14].

3.3 Branch-and-Bound

To solve MILPs it is common to use a branch-and-bound approach. In this procedure, a search tree is recursively built by partially assigning integers at each node. The information gathered at each step is used to eventually converge to an optimal or nearly optimal solution. At every step, we have to choose one of the leaf nodes from which to branch. At this node an LP relaxation is solved, where the variables already fixed at that node, are set to the assigned value. The solution results in a lower bound on the objective of the MILP for any further nodes branching from this node. Thus, this branch can easily be pruned if the value is larger than the bound of a known feasible assignment as a better solution cannot be found in this sub-tree. But if the node is expanded instead, a variable to branch from needs to be found out of the set of unfixed variables. After selecting a variable, a branching step is taken. At each branching step, the problem is recursively decomposed into two smaller sub-problems that are easier to solve. After solving the LP relaxation for the selected variable, one of the child nodes constraints the domain of this variable to be less or equal than the floor of the LP relaxation value. The other node constraints this variable to be greater or equal to the LP relaxation value (i.e., if the solution for the variable of the LP relaxation is $x = 4.2$, one child has the constraint $x \leq 4.0$ and the other one has the constraint $x \geq 5.0$). In case of a binary problem where the solution is only allowed to have the integer values 0 and 1, one of the child nodes will assign 0 to the variable and the other one will assign 1 to the selected variable. In the bounding step, it is determined whether a branch can be safely pruned [8, 14].

3.4 Graph Neural Network

Graph representations are used in many areas to describe relationships between data. GNNs are extended neural network models which take different types of graphs, e.g., acyclic, cyclic, directed, and undirected, as input [15]. The network should encode significant graph structures by learning it in an end-to-end fashion [6]. GNNs can either be *graph-focused* or *node-focused*. A *graph-focused* GNN applies a classifier or regressor on a whole graph, whereas a *node-focused* network depends on the properties of the nodes and classifies those [15].

A graph G is defined by a set of nodes $\mathcal{N}$ and edges $\mathcal{E}$ ($G = (\mathcal{N}, \mathcal{E})$) which can be annotated with features, representing the properties of the node or edge [15]. A node's neighborhood is defined by all nodes adjacent to that node. The connection between those is often represented by an adjacency matrix $\mathbf{A} \in \{0,1\}^{N \times N}$, where N is the number of nodes. For the adjacency matrix applies

$$\mathbf{A}(i,j) = \begin{cases} 1, & \text{if node } n_i \text{ is connected to node } n_j \\ 0, & \text{otherwise,} \end{cases}$$

which in case of an undirected graph is a symmetric matrix ($\mathbf{A}(i,j) = \mathbf{A}(j,i)$) [16].

Given the node features and connectivity between the nodes, the goal of a GNN is to learn a state embedding $\mathbf{h}_n \in \mathbb{R}^d$ with dimension d encoding the neighborhood information for each node. Feeding forward this embedding through the network produces an output $\mathbf{o}_n$, such as a prediction for a label classification [17].

In the original GNN model by Scarselli et al. [15] the state embedding $\mathbf{h}_n$ and output embedding $\mathbf{o}_n$ are defined by

$$\begin{aligned} \mathbf{h}_n &= f_{\mathbf{w}}(\mathbf{x}_n, \mathbf{x}_{\text{co}[n]}, \mathbf{h}_{\text{ne}[n]}, \mathbf{x}_{\text{ne}[n]}) \\ \mathbf{o}_n &= g_{\mathbf{w}}(\mathbf{h}_n, \mathbf{x}_n), \end{aligned} \tag{3.2}$$

where $\mathbf{x}$ is the input feature and $\mathbf{h}$ the hidden state. co$[n]$ denotes the set of edges connected to node n and ne$[n]$ is the set of neighboring nodes of node n connected by those edges. So $\mathbf{x}_n, \mathbf{x}_{\text{co}[n]}, \mathbf{h}_{\text{ne}[n]}, \mathbf{x}_{\text{ne}[n]}$ are the features of node n, the features of the edges connected to node n, the hidden state of the neighbors of node n and the features of the neighbors of node n respectively. $f_{\mathbf{w}}$ is a parametric function called *local transition function* and expresses how dependent node n is on its neighborhood ne$[n]$. Function $g_{\mathbf{w}}$

is the *local output function* describing how the output is generated. $\mathbf{w}$ are the respective parameters which are estimated in the learning process. Those functions can also be applied globally on all states and features at once with the *global transition function* $F_{\mathbf{w}}$ and the *global output function* $G_{\mathbf{w}}$. For that the states and features have to be stacked together to matrices $\mathbf{H}$, $\mathbf{O}$, $\mathbf{X}$, and $\mathbf{X}_N$ representing all states, all outputs, all features, and all node features respectively. That leads to the matrix form

$$\begin{aligned} \mathbf{H} &= F_{\mathbf{w}}(\mathbf{H}, \mathbf{X}) \\ \mathbf{O} &= G_{\mathbf{w}}(\mathbf{H}, \mathbf{X}_N). \end{aligned} \tag{3.3}$$

According to Banach's fixed point theorem [18] the value of $\mathbf{H}$ is the fixed point of Equation (3.3) and is a unique solution with the assumption that $F_{\mathbf{w}}$ is a contraction map according to the state, i.e., there exists a μ, $0 \leq \mu < 1$, such that $||F_{\mathbf{w}}(\mathbf{H}, \mathbf{X}) - F_{\mathbf{w}}(\mathbf{K}, \mathbf{X})|| \leq \mu||\mathbf{H} - \mathbf{K}||$ for any $\mathbf{H}, \mathbf{K}$, where $||\cdot||$ is a vectorial norm. But, Banach's fixed point theorem also suggests the classic iterative scheme

$$\mathbf{H}^{t+1} = F_{\mathbf{w}}(\mathbf{H^t}, \mathbf{X}) \tag{3.4}$$

to compute the state (which is the fixed point), where $\mathbf{H^t}$ is the t^{th} iteration of $\mathbf{H}$. It is the Jacobi iterative method for solving non-linear equations [19]. For any initial value $\mathbf{H^0}$, the dynamic system of Equation (3.4) converges exponentially fast to the solution of Equation (3.3). The state embedding and output embedding of Equation (3.2) can be computed iteratively with

$$\begin{aligned} \mathbf{h}_n^{t+1} &= f_{\mathbf{w}}(\mathbf{x}_n, \mathbf{x}_{\text{co}[n]}, \mathbf{h}_{\text{ne}[n]}^t, \mathbf{x}_{\text{ne}[n]}) \\ \mathbf{o}_n^{t+1} &= g_{\mathbf{w}}(\mathbf{h}_n^{t+1}, \mathbf{x}_n), \quad n \in N. \end{aligned}$$

To compute the state $\mathbf{h}_n^{t+1}$ the current state $\mathbf{h}^t$ and the information from the neighboring nodes are taken into account, i.e., after one iteration the node state not only has information about itself but also its neighbors. After the second iteration, it carries information also about their neighbors and so on such that $\mathbf{h}_n^{t+1}$ has the information of nodes that are t edges away. This is also referred to as *message passing*.

We try to predict the node's label which in our case is whether this node is part or not part of the optimal solution, i.e., we use the GNN for node classification, where the goal is to predict a node's label without knowing the ground truth. In a supervised fashion the model is trained by optimizing a loss function, for instance using the L2 loss function

$$L = \sum_{i=1}^{n} (y_{\text{true}} - y_{\text{pred}})^2.$$

3.4.1 Limitations

Liu et al. [17] state some limitations the original GNN has, and later publications of GNNs try to overcome.

- At first, the iterative update of the hidden state to get the fixed point is computationally inefficient.
- Secondly, each iteration uses the same parameters, while most popular neural networks use different parameters for each layer.
- Third, some informative features on edges might not be properly modeled through the original GNN, e.g., edges carrying a different type of representation should be handled differently according to the type.
- At last, if T is large, the values in the fixed point will get more similar and thus the nodes are less distinguishable.

Other versions of GNNs are for instance graph convolutional networks [20, 21], graph recurrent networks [22, 23], graph attention networks [24, 25] or graph residual networks [26].

3.4.2 MHA for Message Passing

In the following MHA is explained as proposed by [5] and [27]. The message passing is realized by L stacked identical layers which are constructed of sub-layers. A key operation of attention is the dot-product, hence it is also known as "Scaled Dot-Product Attention" [27]. At first the three values key k, query q and value v are computed using a separate weight matrix respectively ($W_{k,o}$, $W_{q,o}$, $W_{v,o}$, where o is the output index of one of the M attention layers)

$$
\begin{aligned}
k_{i,o} &= W_{k,o} n_i^{(l-1)}, \\
q_{i,o} &= W_{q,o} n_i^{(l-1)}, \\
v_{i,o} &= W_{v,o} n_i^{(l-1)},
\end{aligned}
$$

where l is the current round of message passing. In [27] the attention is computed with

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where Q, K and V are q, k and v packed together respectively to jointly compute the attention values for all heads simultaneously and d_k is the dimension of the queries and keys which acts as scaling factor here. Thus, the attention-based message passing effectively computes a weighted update to every node's features, where the weight is given by the scalar product between query and key. In comparison, in [5] the attention is only computed between those nodes that have a connection among each other, which enables to capture partially connected graphs. That is accomplished by defining the compatibility score $c_{i,j}$ for the i^{th}-to-j^{th} node connection as

$$c_{i,j,o} = \begin{cases} \frac{1}{d} q_{i,o}^\top k_{j,o}, & \text{if } \mathbf{A}(i,j) = 1, \\ -\infty, & \text{otherwise}, \end{cases}$$

with normalizing constant d. From that the attention weights are computed using a softmax

$$a_{i,j,o} = \frac{e^{c_{i,j,o}}}{\sum_{j'} e^{c_{i,j',o}}}.$$

The output message of an attention head is calculated with

$$m_{i,o} = \sum_j a_{i,j,o} v_{j,o}.$$

The result of the MHA module is eventually determined by a weighted sum over the outputs of the multi-attention head

$$\text{MHA}(\mathcal{N}^{(l-1)}, i) = \sum_{o=1}^{M} W_{m,o} m_{i,o},$$

where the weights $W_{m,o}$ determine the influence of a single attention head. Those MHA modules are stacked together L times, i.e., one module per round of message passing. So applying

$$\mathbf{h}_i^{(l)} = h(g(h(\text{MHA}(\mathcal{N}^{(l-1)}, i))))$$

L times yields the final embedding vector for each node n_i, with skip connection layer $h(f(x)) = x + f(x)$, the current round of message passing l, and the node embeddings from the previous round $\mathcal{N}^{(l-1)}$. Note that g is the same function as in Equation (4.4), but different weights are assumed at each appearance.

3.4.3 GNN and MILP

Graphs for solving MILPs do often have a bipartite structure [6, 8, 28]. That means a graph represents the MILP with nodes for the decision variables and nodes for the constraints.

Nair et al. [8] state that there are two important properties that should hold true for GNNs that capture the bipartite graph representation of an MILP:

1. The output of the network is invariant to permutations of the input (variables and constraints).
2. The network can be applied for different problem sizes (variable number of decision variables and constraints) using the same parameters.

4 Method

This chapter will introduce our approach to tackle the described combinatorial assembly problem and will also explain the problem in more detail.

4.1 Problem Description

This thesis discusses an assembly problem that is formulated as an MILP. The MILP is then solved using B&B.

The assembly problem can be described as follows: There is a target shape in a 3-dimensional space that should be filled with parts of different shapes. The key property of an MILP is that the variables are integer-valued. Thus the solution of the MILP is discrete. The real world, however, is continuous, so the world has to be discretized. To do that a grid is placed over the world, with cubic grid cells annotated by an identifying index. Every index can be mapped to coordinates in the continuous world in which the point is the center of the grid cell. The other way round, each point in the considered problem area can be mapped to an index in the grid. To simplify the discretization we only consider parts and target shapes that are already discretized, i.e., every part is constructed of cubic blocks of the same size, and the target shape can be perfectly filled with those blocks. Also, each grid cell of the grid equates to the size of those cubic blocks. Figure 4.1 shows an exemplary problem in a 2-dimensional space. Here the target shape is surrounded by the red line and next to the grid the available parts can be seen. The grid also fills space around the target shape to allow pieces to reach out of the grid, even though that is not the desired behavior. It is possible, however, that the parts cannot fill the target shape perfectly. Then the optimal solution might include parts that reach out of the target shape.

Figure 4.2 shows the 3D grid and the assignment of the indices. The grid has the size of $N_x \times N_y \times N_z$, with $N_x,\ N_y,\ N_z \in \mathbb{Z}$.

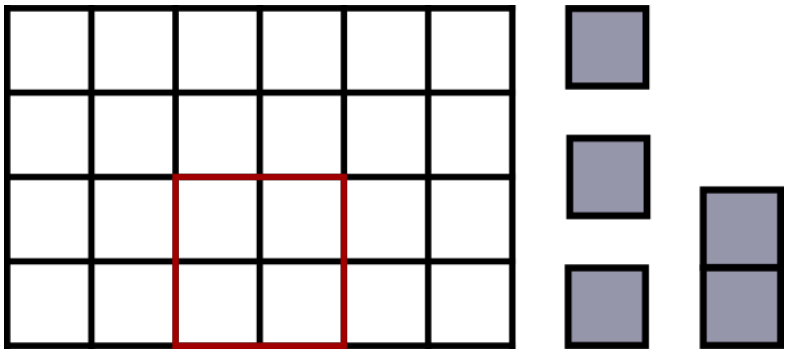


Figure 4.1: Grid with target shape (surrounded by red line) and available parts.

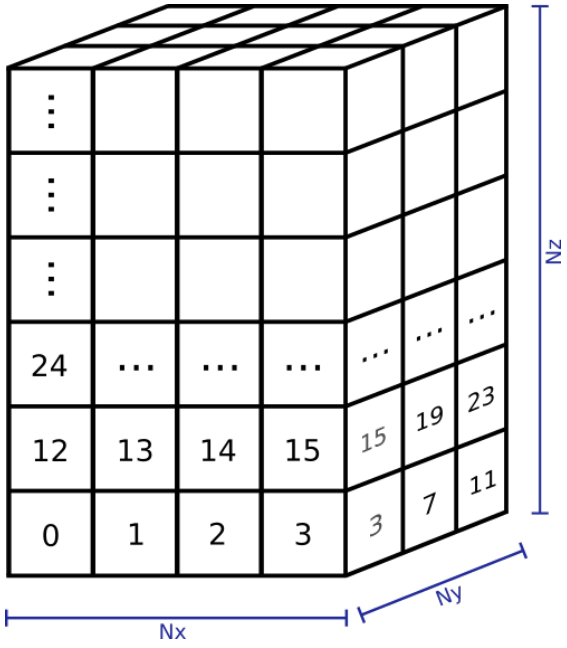


Figure 4.2: 3D Grid with indices.

The discretized problem can be formulated as an MILP. The MIP solver should optimize the problem by the means of maximizing the objective function. The binary decision variables, the solver should determine, describe if a part should be placed at a certain grid cell. The solution also has to fulfill the constraints of the problem formulation. Physically it is not possible that a grid cell is occupied by more than one block, and only parts which are available can be used. Because the MILP is of combinatorial nature, due to the integer constraints, it has an exponential complexity [13]. NP-hard problems are not very scalable because the computation time can get too high. To reduce the time a graph neural network is trained which takes the graph representation of the MILP as input and predicts the decision variables. With help of the prediction, the solving process should be accelerated and thus make the MILP also applicable for large problem instances, as well.

A difficulty, which is faced in the context of architectural assembly, is that the count of decision variables can rise to a very high number because e.g., buildings, especially skyscrapers, are assembled of many parts and thus the grid size can be very high. Which in addition to the high sparsity of the solution makes it more challenging to solve the problem.

4.2 Problem Representation

The continuous world is transformed into a discrete world by representing the position of an object by an index of a grid. In a 3D representation, each grid cell is a cube with a certain side length or a square in a 2D representation. The state of the grid cell is given by vector

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N_g} \end{bmatrix} \in \{0,1\}^{N_g},$$

where N_g is the number of grid cells. $x_i \in \{0,1\}$ with index $i = 0, \ldots, (N_g - 1)$ is 1 if the cell is occupied and 0 otherwise.

4.2.1 Getting Occupancy $\mathbf{x}$

To get the occupancy in the grid a matrix $\mathbf{M} \in \{0,1\}^{N_g \times N_d}$, which contains the occupancy of every part at every possible position, is created. N_d is dependent on the number of grid cells N_g and the number of all rotations of all parts: $N_d = N_g N_r$, where $N_r = \sum_{p=1}^{N_p} N_{r,p}$ with $N_{r,p}$ being the number of unique rotations of part p. Here again, 1 means that a cell is occupied and 0 that it is not. The number of unique parts N_p does not consider permutations (i.e., different rotations) of the same part. Each column $\mathbf{g}_{p,i} \in \{0,1\}^{N_g}$ represents a part $p = 1, \ldots, N_p$ placed at a certain index i and stores which indices of the grid are occupied by that part, i.e., the occupancy of part p at index i would be in column $(p-1)N_g + i$. At the edges of the grid, ~~it can happen that~~ the part would reach out of the area where the grid is defined. It is not possible to assign indices to blocks that are outside of the grid because the indices are only part of the discretized world, which again is defined by the grid. So, we decided that it is not possible to place parts that would reach out of the grid. Hence, if a part would reach out of the grid, all values in that column are 0, because a part that is not placed does not occupy any grid cells.

For a part, which is a cube in the size of a grid cell (in the following referred to as unit block), and denoted by $\mathbf{g}_1$, that would be

$$\begin{aligned}
\mathbf{g}_{1,0}^\top &= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \\
\mathbf{g}_{1,1}^\top &= \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \end{bmatrix} \\
&\vdots \\
\mathbf{g}_{1,N_g}^\top &= \begin{bmatrix} 0 & 0 & 0 & \cdots & 1 \end{bmatrix}.
\end{aligned}$$

For a part out of two unit blocks (2-block part denoted by $\mathbf{g}_2$) in a vertical rotation, placed in an exemplary 4×4 grid, the columns have the form

$$\begin{aligned}
\mathbf{g}_{2,0}^\top &= \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \end{bmatrix} \\
\mathbf{g}_{2,1}^\top &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & \cdots & 0 \end{bmatrix} \\
&\vdots \\
\mathbf{g}_{2,N_g}^\top &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}.
\end{aligned}$$

For a problem with those two parts, the matrix is composed to

$$\mathbf{M} = \begin{bmatrix} \mathbf{g}_{1,0} & \mathbf{g}_{1,1} & \cdots & \mathbf{g}_{1,N_g} & \mathbf{g}_{2,0} & \mathbf{g}_{2,1} & \cdots & \mathbf{g}_{2,N_g} \end{bmatrix}. \tag{4.1}$$

For each column, there is a decision variable that determines if the part is placed at this index. The decision variables are stacked together to a vector

$$\mathbf{d}^\top = \begin{bmatrix} d_{\mathbf{g}_{1,0}} & d_{\mathbf{g}_{1,1}} & \cdots & d_{\mathbf{g}_{1,N_g}} & d_{\mathbf{g}_{2,0}} & d_{\mathbf{g}_{2,1}} & \cdots & d_{\mathbf{g}_{2,N_g}} \end{bmatrix} \in \{0,1\}^{N_d}.$$

The occupancy $\mathbf{x}$ is then determined by

$$\mathbf{x} = \mathbf{M}\mathbf{d}.$$

4.2.2 Cost Function

The goal is to fill the target shape. The target shape is represented by vector

$$\mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} \in \{a,b\}^{N_g}, \tag{4.2}$$

where $c_i \in \{a,b\}$. a is the reward if a cell belonging to the target shape is occupied and b is the negative reward (penalty) when a block is placed at a cell where it should not be placed. As a default $a = 1$ and $b = -1$ are set. So if there should not be a penalty for placing a part outside of the target shape, b should be set to 0.

The cost function is then determined by

$$L = \mathbf{c} \cdot \mathbf{x} = \mathbf{c}^\top \mathbf{x}.$$

4.2.3 Constraints

Besides the integer constraints, two other constraints have to be incorporated:

1. One grid cell cannot be occupied by more than one part. For the occupancy that means
$$x_i \leq 1, \qquad \forall x_i \in \mathbf{x}.$$

2. There are only a limited number of parts available. So for the decision variables of one part $\mathbf{g}_p$ applies
$$\sum_{i=0}^{N_g-1} d_{\mathbf{g_{p,i}}} \leq m_p, \qquad \forall p = 1, \ldots, N_p,$$
where m_p is the available amount of part $\mathbf{g}_p$. This constraint applies to all N_p parts.

So, in total, there are $N_g + N_p$ constraints.

not sure tho

4.2.4 MILP

The given objective function and constraints of the discussed problem lead to the MILP representation

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^\top \mathbf{x} = \mathbf{c}^\top \mathbf{M} \mathbf{d} \\ \text{subject to} \quad & \mathbf{x} \leq \mathbf{1} \\ & \sum_{i=0}^{N_g-1} d_{\mathbf{g_{p,i}}} \leq m_p, \qquad \forall p = 1, \ldots, N_p \\ & d_j \in \{0, 1\}, \quad j = 0, 1, \ldots, N_d - 1. \end{aligned} \tag{4.3}$$

Note that the decision variables are binary so in comparison to Equation (3.1) the decision variables $\mathbf{d}$ in Equation (4.3) are not constraint by boundaries $\mathbf{l}$ and $\mathbf{u}$, but instead take the values 0 or 1.

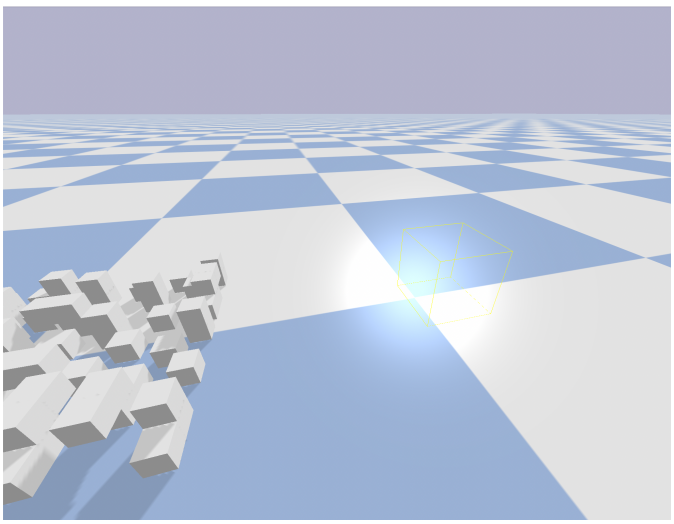


Figure 4.3: PyBullet environment with parts on the left and yellow wireframe of target shape on the right.

4.3 Framework

4.3.1 Pybullet Environment

The environment used is an adaption from Funk et al.'s environment used in [5]. The environment is built in the physics simulator PyBullet [29].

Here, all parts are created, which are assembled by cubes with a side length of $0.05\,\mathrm{m}$, and the target shape is specified by giving the coordinates of the corner points of the shape.

4.3.2 Visualization

PyBullet also serves as a visualization framework.

The target shape is shown by a yellow wireframe as shown in Figure 4.3, and all available parts are placed into the world.

4.3.3 Solver

To solve the MILP, the commercial solver Gurobi [14] is used. To set the objective and the constraints the indices a part occupies, the available amount of a part and the indices of the target shape are needed. Remember that every rotation of the part has to be considered as its own part. With that given, the matrix $\mathbf{M}$ as in Equation (4.1) is created as well as the vector $\mathbf{c}$ (cf. Equation (4.2)), which stores the reward at the target shape indices or the penalty at the other indices. As $\mathbf{M}$ can quickly grow to a large matrix when adding new parts or enlarging the grid, $\mathbf{M}$ is stored as a sparse matrix.

After constructing $\mathbf{M}$ and $\mathbf{c}$ the solver can be configured. For that, the Gurobi Python library is used. In comparison to other solvers like PySCIPOpt [30], Gurobi allows creating a vector (or matrix) to represent the optimization variables, instead of creating each variable separately in a for-loop. So the decision variables as a vector of size N_d are added and the type is set to binary. Then the objective which should be maximized is set and the constraints of the MILP are added. Here, Gurobi also enables to use matrices and vectors.

After configuring the solver, the problem can be solved. As we have binary variables, that means the solver assigns 0 or 1 to each decision variable.

4.3.4 Interface

To translate between the environment in PyBullet and the Gurobi solver an interface is implemented. The interface takes the continuous values of the blocks and the corner points of the target shape from the PyBullet environment and transforms them into the discrete space so that the solver can solve it. The solution is then transformed back to the continuous space to be visualized in the simulation. The interface has functions that return the index of given coordinates and vice versa.

To determine the index i from the coordinates $x,\ y,\ z$, the coordinates are rounded to the next closest grid cell center point and the center of the overall grid is set to the origin (0, 0, 0). Together with $N_x,\ N_y,\ N_z$ and the side length of a grid cell in the continuous space, discrete coordinates (x_d, y_d, z_d) can be determined in a coordinate system, where the most lowest-left-front grid cell center is the origin. The index is then calculated with

$$i = x_d + y_d N_x + z_d N_x N_y.$$

The center of the lowest xy-plane of the grid is placed on the center (regarding $x,\ y$ coordinates) of the target shape in the continuous space. The bottom side of the grid is

aligned with the bottom of the target shape. That way the grid is generated around the target shape and covers it completely.

The size of the grid depends on the target shape. But it also lets room at the sides and at the top for parts that reach out of the target shape. The bottom is the floor, so there is no additional space under the target shape. In the best case, the parts will not be placed outside the target shape but it is possible that the target shape cannot be perfectly filled with the available parts. Then it might be possible that the optimum involves parts that reach out.

Target Shape Indices

The observation from the environment provides the corner points of the target shape. However, the shape of the volume which is spanned by these points is not distinct. In order to make the target shape clearer, more points lying in the target shape are added. This step, however, needs to be hardcoded for every shape. It is a functioning workaround for now, but it is a point that should be tackled again in the future. With enough points, a concave hull takes the form of the desired target shape. To create the bounding polygons and find inlaying grid cells, the python library alphashape [31] is used. To find the inlaying grid cells each index is mapped into its coordinates. Then a function is called, which determines if this point lies in the alpha shape. The indices for which this holds true are collected in a list.

Part Indices

We want to get the indices a block would occupy if it is placed at the lowest-left-front possible position in the grid. That is useful because in that way equal parts can be recognized and it simplifies creating matrix $\mathbf{M}$. The indices obtained here just have to be added to the index at which the part is placed to get the indices the part occupies there. The process to obtain the indices is depicted in Figure 4.4. Each block in the environment holds its coordinates and information about the connected blocks. To obtain the indices of a part, the part is first rotated to the desired orientation. In Figure 4.4 the original part is blue and the rotated part is green. The rotation center is the center of the first block of a part (pink point). Then the part is moved to the center of the grid. This is necessary because the part needs to be completely in the grid to translate the coordinates into an index. After the indices are determined, the block is moved such that the lowest-left-front

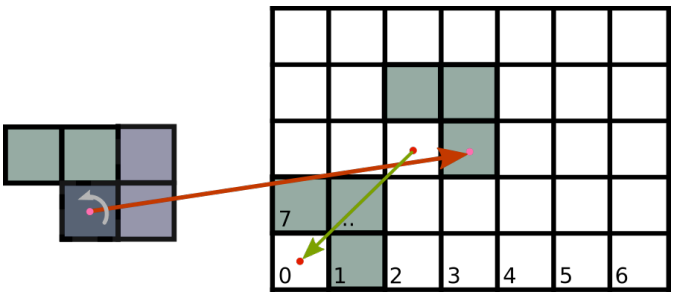


Figure 4.4: Visualization of the algorithm to get the relative indices of a part. First, the part is rotated to the desired orientation. The rotation is around the center of the first block assembling the part (blue part rotated around pink point). The green part is the rotated part which is then moved to the center of the grid, again w.r.t. the center of the first block assembling the part. Then the index of the lowest-left-front grid cell of the bounding box of the part is determined (red point). This index is subtracted from all indices of the part. So it is moved to the lowest-left-front possible position in the grid.

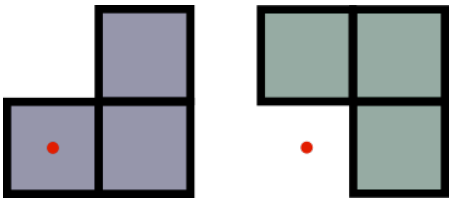


Figure 4.5: Lowest-left-front point (red) of a part.

point (red point) is at index 0. Note that index 0 does not necessarily need to be occupied here (cf. Figure 4.5). The block is moved to the lowest-left-front corner to get the lowest possible index numbers. Then the indices of the blocks belonging to the part are collected in a list (for the part in Figure 4.4 that would be [1, 7, 8]). This is done for all $N_{r,p}$ rotations of a part p.

Visualizing the Solution

Given the solution for the decision variables, it is known where to put which part. Here we use the function again which returns the coordinates for a given index.

4.4 Graph Neural Network

To accelerate the solving time for large problem instances this section proposes a GNN with multi-head attention (MHA) which should predict the solution of the MILP which is then given to the solver as a warm start.

4.4.1 Graph

MILP instances can get well transferred into a graph representation $G = (\mathcal{N}, \mathcal{E})$ with nodes $\mathcal{N}$ and edges $\mathcal{E}$. In many other works, as in [8] and [28], the MIP graph representation has a bipartite structure. The reason for this is that their optimization problem contains two different kinds of entities, i.e., decision variables that are subject to optimization and constraint variables. These variables are transferred into two different kinds of nodes $\mathcal{N} = \{\mathcal{N}_v, \mathcal{N}_c\}$ in the graph, with $\mathcal{N}_v$ representing the decision variables and nodes in $\mathcal{N}_c$ representing the constraints. To illustrate, which decision variables are influenced by which constraints, an edge is added between the corresponding nodes. This results in a bipartite graph.

In our case, however, the graph is tripartite as depicted in Figure 4.6, because we introduce two different kinds of constraint nodes $\mathcal{N}_c = \{\mathcal{N}_{c,\text{grid}}, \mathcal{N}_{c,\text{amount}}\}$. That gives additional structure to the problem, which should also facilitate the learning. So besides the decision variable nodes (red), there are two kinds of constraint nodes: 1) constraint nodes in $\mathcal{N}_{c,\text{grid}}$ that represent the occupancy at each grid cell (green), and 2) constraint nodes in $\mathcal{N}_{c,\text{amount}}$ that represent the available amount of a part (blue). Figure 4.6 shows an exemplary 2×2 grid with a unit block and a 2-block part in vertical and horizontal orientation.

Each node in $\mathcal{N}_{c,\text{amount}}$ represents one part. Here a part includes all of its possible rotations. So each decision variable node has exactly one connection to a node of $\mathcal{N}_{c,\text{amount}}$. Each node in $\mathcal{N}_{c,\text{grid}}$ represents a grid cell. So one decision variable node has a connection to a node in $\mathcal{N}_{c,\text{grid}}$, for each grid cell the part occupies. There is no connection if the part would reach out of the grid. Each decision variable node is also connected to all decision variable nodes which belong to the same part. Those connections are not drawn in Figure 4.6 because that would overload the Figure. Moreover, all nodes have a self-connection.

This split of the constraint nodes $\mathcal{N}_c$ can later be exploited to add additional features for instance to the grid constraint nodes (such as grid position), which are not meaningful for the amount constraint nodes.

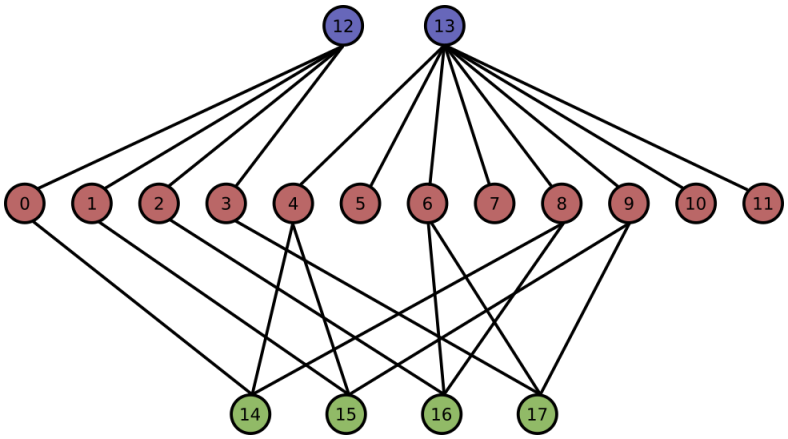


Figure 4.6: Tripartite graph structure for a 2×2 grid with a unit block and a 2-block part in vertical and horizontal orientation. The red nodes depict the decision variable nodes, blue the amount constraint nodes and green the grid constraint nodes. Each decision variable node is also connected to all decision variable nodes which belong to the same part, but those connections are not drawn here.

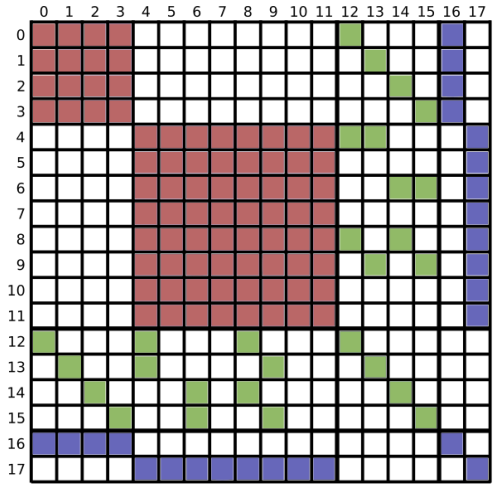


Figure 4.7: Adjacency matrix of the graph in Figure 4.6. All colored entries (i, j) show a connection between node n_i and n_j.

An adjacency matrix $\mathbf{A} \in \{0,1\}^{N_n \times N_n}$ embodies the connectivity between those nodes where N_n is the number of nodes. If an edge connects two nodes n_i and n_j, the entry at $\mathbf{A}(i,j)$ equals 1, and 0 otherwise. As the graph is non-directed, the adjacency matrix is symmetric ($\mathbf{A}(i,j) = \mathbf{A}(j,i)$).

4.4.2 Features

Nodes in $\mathcal{N}_v$, $\mathcal{N}_{c,\text{grid}}$ and $\mathcal{N}_{c,\text{amount}}$ have a different set of features (cf. Table 4.1).
In the following, the features considered for the respective nodes are presented. However, their value for the network has to be evaluated (cf. Section 6.2.1).

All three feature node types share the feature *kind*. That feature determines if the observed node is in $\mathcal{N}_v$, $\mathcal{N}_{c,\text{grid}}$ and $\mathcal{N}_{c,\text{amount}}$, with the values 1, -1 and 0 respectively. This feature should help the network to distinguish the kind of nodes.

The feature *# primitive elements* equals the number of parts without considering the possible orientations of a part and should belong to the features of nodes in $\mathcal{N}_v$.

Possibly helpful is the *part id* which indicates which part the decision node affects. It could help to enforce the connections between the decision variables belonging to the same part which might help to respect the amount constraint.

The features *# edges reward* and *# edges penalty* represent the number of connections to nodes in $\mathcal{N}_{c,\text{grid}}$, dependent on the objective value of the grid cell correspondent to the node, i.e., if the node represents a grid cell in the target shape or not. So, *# edges reward* counts the number of edges to nodes with a positive objective value and *# edges penalty* to a negative objective value. This is useful, as it helps the network to distinguish if a part is placed in, partially in, or out of the target shape.

The *decision variable index* enumerates each decision variable, so that each decision variable node has a unique index, and can be distinguished. That feature might be helpful because some decision variable node features could have the same value for different nodes e.g., *kind*, *# edges reward* and *# edges penalty*, which could lead to the same prediction for those, even though that is not desired.

Then, there is the index of the grid cell (*grid idx*) which could be considered for nodes in $\mathcal{N}_{c,\text{grid}}$ and $\mathcal{N}_v$. For the decision variable nodes, it could help the network to recognize that parts placed at the same index are less likely to be placed at the same time. But, on the other hand, that can happen and then might also confuse the network.

For the grid constraint nodes that feature helps to distinguish them from each other, because they could also be very similar, as described for the decision variable nodes before.

We will also consider the *coordinates* (x, y, z) of a grid cell, one time for the decision variable nodes and also for the grid constraint nodes. Those features also are a unique identifier for each grid cell and have a direct translation to the grid index, but they also carry the information on how close the grid cells are, which might be helpful.

The constraint node feature *objective value* can take the values 1 and -1. The nodes in $\mathcal{N}_{c,\text{grid}}$ implement two functions. They not only set constraints on the occupancy in the grid, but they also hold information of coefficients in the objective function, which is represented in this feature. So the value is 1 if the grid cell corresponding to this node belongs to the target shape and -1 otherwise. That is important to distinguish which grid cells should be occupied or not.

The features *lower bound* of constraint and *upper bound* of constraint prescribe the boundaries for the constraints, i.e., the quantity of a part or by how many parts one grid cell can be occupied (which in our case will be 1 for all cells). The upper bounds are key elements of the MILP and thus part of the minimum feature set. The lower bounds, however, are excluded from that, because first, there are no explicit lower bounds in our constraints of the MILP (cf. Equation (4.3)). And second, there is a natural lower bound of 0 by construction of the problem which would also be the value of the feature and thus might make it obsolete. Because for grid constraint nodes, the upper bound is also equal, this feature will not help the network much. But we keep the constraint in case other upper bounds will be set later.

For amount constraint nodes we also consider the feature *# edges decision variables*, which is the number of edges to decision variable nodes going from one amount constraint node.

At last, we also look at the *# rotations* per part. This could help because the network might extract the information that a higher amount of rotations also means that there are more decision variables, but the number of parts that can be placed is equal, i.e., that all rotations still represent one part.

Table 4.1 gives an overview of the features and to which kinds of node it belongs.

All features are normalized so that they take values in $[-1, 1]$. Normalization is important because unnormalized data can slow down the training and can be the reason to be stuck in a local optimum. Normalization to the range between -1 and 1 is useful if a sigmoid function is used, which is the case for our network. Input values in that range fall in the

feature	decision variable	grid constraint	amount constraint
kind	x	x	x
# primitive elements	x		
part id	x		
# edges reward	x		
# edges penalty	x		
decision variable index	x		
grid idx	x	x	
coordinates	x	x	
objective value		x	
lower bound		x	x
upper bound		x	x
# edges decision variables			x
# rotations			x

Table 4.1: (Considered) features for different kind of nodes.

region where the output of the sigmoid function is most sensitive to changes in the input values. So, scaling speeds up the learning and is therefore recommended [32].

4.4.3 Architecture

Now the architecture of the used neural network is introduced. As input, the network should take the graph representation of the problem (see Section 4.4.1) with the features presented in Section 4.4.2. The output of the network should return the prediction of every decision node, more precisely the value of every decision node which should take 1 if the node is inside the solution subset, and 0 otherwise. Thus we are dealing with a binary classification problem for every node and have to perform multidimensional regression within $[0, 1]$. The class (0 or 1) can then be determined by applying a threshold at 0.5 on the predicted value.

For this purpose, a GNN is used which is based on the one presented by Funk et al. [5]. They use a GNN which combines graphs and multi-head attention (MHA) [24, 27]. Because the feature size of the three types of nodes is different, the architecture of the network is designed in such a way that it can take three inputs. The one input layer receives the decision variable nodes, and the other two input layers, the two kinds of constraint nodes. With the three input layers, the feature *kind* is unnecessary, because each of these input layers only gets the feature vectors of the respective node and thus does not have to distinguish them. The three inputs are passed to the first layers which embed the nodes to create a richer and more high-dimensional representation. The initial embedding layers are built of a fully-connected (FC) layer with a subsequent leaky rectified linear unit (leaky ReLU) activation function

$$\mathbf{h}_i^{(1)} = g(\mathbf{h}_i^{(0)}) = \mathrm{leakyReLU}(\mathrm{FC}(\mathbf{h}_i^{(0)})), \tag{4.4}$$

where $\mathbf{h}_i^{(0)} = \tilde{\mathbf{x}}_i$, the input feature vector of node n_i and $\mathbf{h}_i^{(1)}$ is the initial embedding. Because after embedding, each node's feature dimension is equal, the output of those layers is concatenated and then passed to L layers of message passing. The purpose of the message passing is to compute updated features for each node, which not only depends on its own previous embedding but is also influenced by its neighboring nodes. Therefore, this mechanism is combining the local feature information while taking the graph's topology into account. The message passing is realized by MHA as described in Subsection 3.4.2.

The output layer of the network uses the sigmoid activation function

$$\text{sig}(t) = \frac{1}{1 + \exp^{-t}}$$

because it generates an output in the range of 0 and 1, which is our desired range.

The networked is trained in a supervised fashion. We use the Adam optimizer [33] which is an algorithm for first-order gradient-based optimization of stochastic objective functions, and weighted binary cross-entropy loss [34]

$$L = -\frac{1}{N_d} \sum_{j=0}^{N_d-1} w_j [y_{\text{true},j} \cdot \log(y_{\text{pred},j}) + (1 - y_{\text{true},j}) \cdot \log(1 - y_{\text{pred},j})]. \tag{4.5}$$

We use the weights because the there are a lot more decision variables which are 0 than 1. So, without the weights, the network would probably just predict everything close to 0, because it is cheaper not to place any part instead of placing a part at a wrong index. To overcome that problem, we use weights. The factor for the accumulation is defined as follows

$$w_j = \begin{cases} \frac{\#\ (y_{\text{true},j} = 0)}{\#\ (y_{\text{true},j} = 1)}, & \text{if } y_{\text{true},j} = 1, \\ 1, & \text{otherwise.} \end{cases}$$

4.5 Use GNN Model for MIP Solver

After the model is trained it can be applied as a heuristic for the Gurobi MIP solver [14] we use. The goal in incorporating the prediction of the model is that the solution is already quite good so that more branches can be pruned which leads to a shorter time to find the solution. Gurobi offers several possibilities to incorporate a heuristic from whose the best option for our approach needs to be evaluated. The functions or attributes we will consider are `Model.cbSetSolution()`, `VarHintVal`, `VarHintPri`, and `Start`.

which?

4.5.1 Model.cbSetSolution()

`Model.cbSetSolution()` enables to import a solution of the decision variables that is then used as heuristic solution via a callback. It is also possible to only set some of

the variables. After setting the solution `Model.cbUseSolution()` can be called to immediately use the values to try computing a feasible solution from the specified values.

When using `Model.cbSetSolution()` we use the binary prediction after applying the threshold as solution.

4.5.2 VarHintVal

The user can give hints to the solver if it is known that a variable is likely to take a specific value. Those hints affect the heuristic the Gurobi solver uses to find feasible solutions, and the branching decisions of the solver while exploring the search tree. So the hints act as guidance for the solver in which they influence the entire solution process. Here it is also possible to just partially give hints to the variables.

Here we also use the binary prediction after applying the threshold as hints.

4.5.3 VarHintPri

This variable assigns a priority to each variable hint which provides information about the confidence of a variable. The hint priorities are relative. So higher confidence in one hint simply needs a higher hint priority than the other one.

We use the prediction of the GNN model to compute the priority with

$$\mathrm{abs}((1 - y_{\text{true}}) - y_{\text{pred}}).$$

4.5.4 Start

This variable attribute represents the current start vector of the MIP. From this vector, the solver will try to build an initial solution, if available. Those variables can again also just partially be assigned. If the start vector is infeasible, the solver will not find a new incumbent solution. This can also happen if the heuristic of Gurobi finds an equal valued solution as produced by the start vector.

5 Discussion

In this chapter, we compare MILPs with GNNs and discuss alternative ways how a GNN can be used to accelerate the solving of an MILP.

5.1 MILP vs. GNN

A combinatorial optimization problem can be solved through classical solvers or through a neural network. To solve the problem with classical solvers, on the one hand, the problem is often represented as an MIP and then solved with an MIP-solver. To solve the problem with a neural network, on the other hand, the problem is often represented as a graph. Both problem representations discretize the world to solve the combinatorial problem.

In comparison to neural networks, MIPs can guarantee to find the optimal solution of a problem instance. There are applications where it is crucial to find the optimal solution because sub-optimal solutions are not realizable. In the context of an autonomous architectural assembly, a sub-optimal solution can lead to problems with the structural calculations or to holes in a wall. Clearing those faults can be very costly hence sub-optimal solutions should be avoided. This is an advantage of MILPs over GNNs and the reason why in this thesis the problem is represented as a MILP.

GNNs, however, have the potential to be faster than classical solvers which are especially important for real-time applications. The GNN can also give a sequential solution that has other benefits, e.g., the network returns which part should be placed at which position in the next time step. After each step, the assembled parts should be stable because you cannot construct something which collides during the assembling. Also, it should be feasible for the robot to place a part at a certain position without colliding with something. Those points could be considered when providing a sequential solution as it is done in Funk et al.'s work [5]. So the sequential order of the parts which are placed is valuable

information when the robot places the part at the construction because otherwise resources (parts, robot) could be destroyed.

The proposed approach does not have any sequential information because the final solution is provided by a MILP. But it is taken advantage of the computational efficiency of an GNN. A non-sequential solution of the GNN is given to the MILP which then can prune unfeasible solutions faster. So the GNN only accelerates the solving here.

So in conclusion, both, using an MIP or a GNN to solve the optimization problems has their benefits. The MIP can guarantee optimality whereas the GNN can potentially provide a sequential solution.

5.2 Alternative Ways to Use the GNN

There are different approaches to how the solving process of an MIP can be accelerated using GNNs. Here I will present Nair et al.'s [8] approach who use two GNNs two execute *Neural Diving* and *Neural Branching* in a MIP solver.

Neural Diving is a trained deep neural network that assigns the decision variables partially, after that smaller 'sub-MIPs' defined by the unassigned variables are given to standard MIP solver like Gurobi [14] or PySCIPOpt [30] which can then give a guaranteed optimal assignment of the smaller set, which, however, might not be the optimum of the whole problem. Those sub-MIPs can even be solved parallel, if the resources are available, which again increases the efficiency. Neural Branching is a learned deep neural network to assist the B&B algorithm. The network should make variable selection decisions such that not many iterations are necessary and the objective value gap is bound more efficiently with a small tree.

Their approach quickly finds a good solution which is a benefit of using it. But it is not guaranteed that an optimal solution is found because the problem is portioned into smaller sub-MIPs and not solved altogether. The solver indeed finds the optimal solution for each sub-problem but those combined are not necessarily the optimal solution of the original problem. However, their approach still finds good solutions, and only sometimes fails to find an optimal or near-optimal solution.

6 Experiments

In order to evaluate the MIP solver, we let it solve different problem instances with variations in target shape, size, and shape of the parts.
Then we will evaluate our GNN model with different node features and problem instances to find out which features are meaningful and how the result is for training on different problem instances.
The device used for all experiments is a MacBook Pro (2021) having an Apple M1 Pro with 10-core CPU, 16-core GPU, 16-core Neural Engine, and 16GB unified memory. It is to mention, that Rosetta 2 is necessary which interprets Intel programs and can also influence the running time.

6.1 Computation Time MILP

Computation time is one of the key aspects for people deciding over which solver to use. Especially in real-time applications, the required time is crucial. Generating a building plan in architectural assembly is not a real-time application per se, nevertheless, fast results are still important and desired. Short computation times also enable testing several scenarios. In the context of architectural assembly, such scenarios can cover different target shapes and different sets of parts [14]. In this experiment, we will look at the time for every step in the solving process. Those steps are

1. add variables,
2. set objective,
3. add constraint (grid),
4. add constraint (amount),
5. and solving.

6.1.1 Different Problem Sizes

At first, the solving time concerning the problem size is evaluated. This experiment shows how scalable a problem is, which should be solved with the MIP solver. In this experiment, the target shape is a cube with increasing edge length a, i.e., it fills a^3 grid cells. Because the target shape increases, the total number of grid cells also increases. The number of blocks from which the parts are composed is rounded to a number that guarantees that the shape can be filled perfectly. In this experiment, the grid size N_g goes from 384 to 12672 and object class 26 is used (cf. Table 6.1).

6.1.2 Different Target Shapes

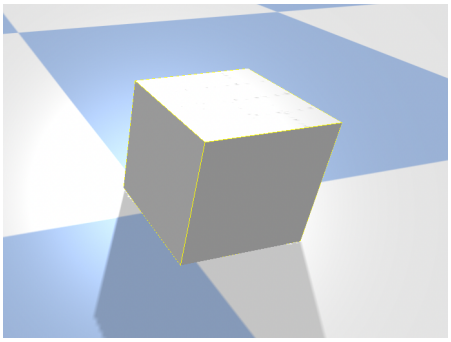
Secondly, we look at the solving time with respect to the target shape. Here four different shapes will be evaluated: a cube, a circle, a plateau with two towers, and an arbitrary constructed. The shapes are shown in Figure 4.1, but note that the experiments use different scales of the shapes. Here the grid size is set to a constant number and the number of grid cells that have to be filled are almost the same, so that truly the shape and not the problem size is evaluated. The grid size is $N_x = N_y = 14$, $N_z = 5$ which results in $N_g = 980$. The number of grid cells in the target shape is 160, except for the circle it is 156. Here we also use object class 26 (cf. Table 6.1).

6.1.3 Different Sets of Parts

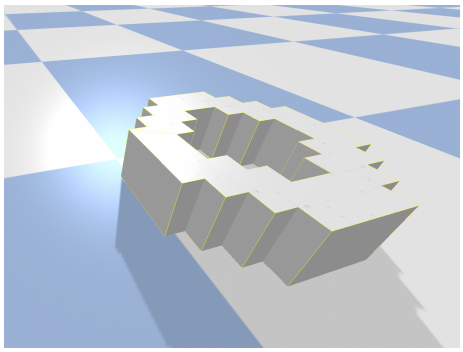
Another point that can variate the problem instances is the set of parts that are used to fit the shape. Here we evaluate how the complexity and the number of the parts used, influences the solving time. Here again the target shape is a $4 \times 4 \times 4$ cube which leads to $N_g = 384$. In this experiment the part sets in Table 6.1 are used. Each part p has $N_{r,p}$ unique orientations. For some sets, we only allowed the rotation around the z-axis so they have fewer unique rotations.

6.1.4 Abraxis Cube

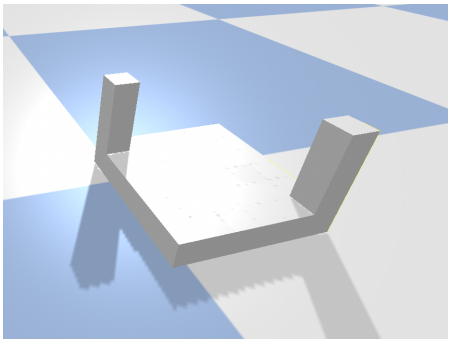
The Abraxis cube is a 3D puzzle with 13 unique parts (see Figure 6.2). The goal is to fill a $4 \times 4 \times 4$ cube with those parts. The Abraxis cube is interesting because there is only



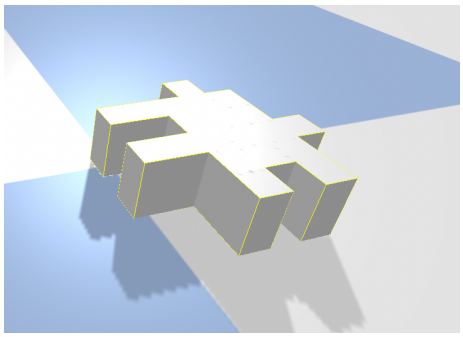
(a) Cube.



(b) Circle.



(c) Plateau with towers.



(d) Arbitrary Shape.

Figure 6.1: Target Shapes.

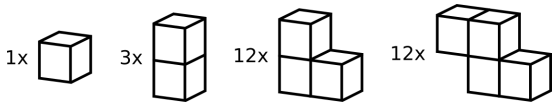
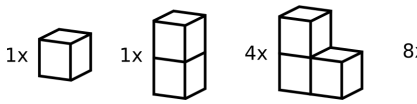
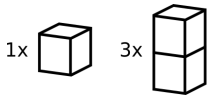
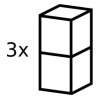
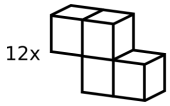
object class	parts in object class
26	
26 (z-rot)	
32	
33	
34	
34 (z-rot)	
35	

Table 6.1: Parts which are included in the respective object class. The count is, how many unique orientations of a part in 90 degrees steps exist ($N_{r,p}$).

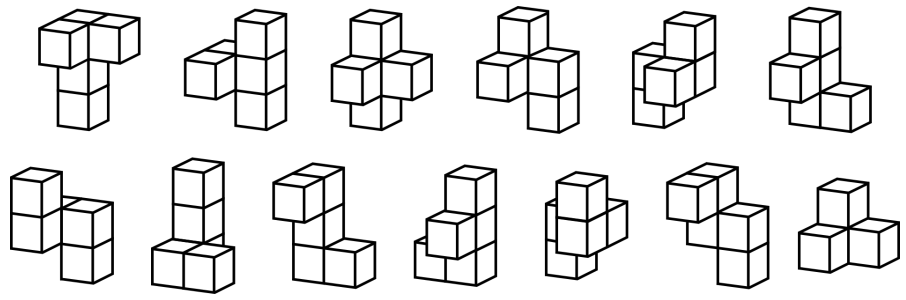


Figure 6.2: Parts of the abraxis cube puzzle.

one way to assemble the parts to get the cube. So with this experiment, a more complex problem is evaluated.

This experiment introduces a new object class, which can be compared to the other, less complex parts. To also look at bigger, more complex problems, we also solve compound $4 \times 4 \times 4$ cubes with the respective multiplication of the part set. The multiplications we look at are two and four.

6.2 GNN

In the following, we will describe the experiments on the GNN. At first, we will evaluate which features are useful, and then we will take the best feature set and let the network train on different problem instances. The first goal, during the development of the model, is to show that our approach works, i.e., it is more like a proof of concept. That means that we let the model learn on just one sample, which is prone to overfitting. To keep the problem simpler only two kinds of parts are used (unit block, 2-block part). All data used to train the model have a cuboid target shape of size $4 \times 4 \times 4$.

6.2.1 Different Features

In Section 4.4.2 we introduce a lot of features that have to be evaluated in terms of their benefit for the prediction. Some features which arise from key properties of the MILP like the lower bounds or the objective value seem very likely to be beneficial. So, the features representing the number of edges to positive or negative valued objective values,

the objective itself, the upper bounds, and the kind of the node build a minimum feature set. Besides those, the other features are added to the feature set one by one, to evaluate their influence. Table 7.4 shows the feature sets which are evaluated at first. After figuring out which features improve the result of the minimum set, we also test other combinations of features concerning those results.

Improvement is measured by the loss and the number of false predictions. Therefore the prediction is rounded to the binary values 0 and 1 with respect to a threshold at 0.5, so that values above 0.5 are 1, and 0 otherwise. We will refer to that as binary prediction (b_pred). The number of false predictions is then calculated by an L1 loss between that and the ground truth

$$\#\text{ false predictions} = \sum_{i=1}^{n_{DV}} |y_{\text{true}} - y_{\text{b\_pred}}|. \tag{6.1}$$

The loss function is a normalized, weighted cross-entropy loss, as defined in Equation (4.5).

The hyperparameters for this experiment are constant with a batch size of 1, 150 epochs and a learning rate of 0.00005. It is possible that 150 epochs are not sufficient to get the best result, but because the number of epochs is constant, the results are still comparable to each other. We choose a batch size of 1 because at first we only want to learn the model on one sample at first to check if that leads to a good result. If the result is not good when overfitting is not considered, then it is unlikely it will be good with higher batch size.

6.2.2 Different Problem Instances

After figuring out which the best combination among those features is, we train the model on different problem instances shown in Table 6.3. Those problem instances use different part sets, so the amount and kind of parts change. We also test the part on a grid that does not have the offset around the target shape, i.e., the parts cannot be placed outside the target shape. That also reduces the problem size a lot.

The metric used to compare the results is again the number of false predictions which is computed as in Equation (6.1) and the loss.

Here, the hyperparameters are also constant, but the experiment will be run on 200 epochs and 1000 epochs. The batch size and learning rate stay 1 and 0.00005 respectively.

feature	0	1	2	3	4	5	6	7	8	9	10	11	12
kind	x	x	x	x	x	x	x	x	x	x	x	x	x
# primitive elements		x											
part id			x										
# edges reward	x	x	x	x	x	x	x	x	x	x	x	x	x
# edges penalty	x	x	x	x	x	x	x	x	x	x	x	x	x
decision variable index				x									
grid idx DVN					x	x							
grid idx GCN						x	x						
coordinates DVN								x	x				
coordinates GCN									x	x			
objective value	x	x	x	x	x	x	x	x	x	x	x	x	x
lower bound GCN											x		
lower bound ACN											x		
upper bound GCN	x	x	x	x	x	x	x	x	x	x	x	x	x
upper bound ACN	x	x	x	x	x	x	x	x	x	x	x	x	x
# edges decision variables												x	
# rotations													x

Table 6.2: Feature sets for the experiment. DVN means decision variable node and GCN means grid constraint node.

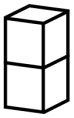
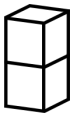
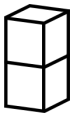
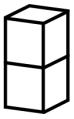
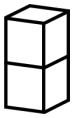
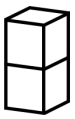
id	object class	w/o offset	N_d	parts in object class
1	33		384	1x 
2	34 (z-rot)		384	1x 
3	34		1152	3x 
4	32 (z-rot)		768	1x  1x 
5	32		1536	1x  3x 
6	34 (z-rot)	x	64	1x 
7	34	x	192	3x 
8	32 (z-rot)	x	128	1x  1x 
9	32	x	256	1x  3x 

Table 6.3: Problem instances given to the GNN.

6.2.3 Different Connectivity

To evaluate how important the connectivity for the prediction of the network is we will also run the network on a fully connected graph and a randomly connected graph.

The target shape is again a $4 \times 4 \times 4$ cube that should be filled with parts of object class 32. And the model is trained in 1000 epochs with batch size 1 and a learning rate of 0.00005. The metrics to compare the results are the number of false predictions and the loss as in the other experiments.

6.2.4 Use Model as Heuristic for the MIP Solver

Finally, the (on 1000 epochs) trained model is applied on problem instance 9 and its result is given to the GNN. After training, the model predicts the solution correctly. That means we will not only give a good solution but even the correct solution to Gurobi's solver.

In Section 4.5 a few approaches to incorporate the GNN's solution are presented. We will test the approaches `Model.cbSetSolution()`, `VarHintVal`, and `VarHintPri` with the whole solution. `Start` seems to be more likely to fail especially if given an infeasible start vector which could happen because the model does not guarantee feasible solutions. Thus we will exclude this approach for now but we might pick it up in future work with partial assignments of the variables.

7 Results

In this chapter, we will discuss the results of the experiments described in chapter 6.

7.1 Computation Time

In the following the results of the experiments regarding the influence of factors like the problem size, target shape, and used parts, on the solving time are presented.

7.1.1 Different Problem Sizes

In Figure 7.1 the time of the different steps in the solving process is plotted over the number of grid cells in the respective problem. It shows that during the solving process, most of the time is used for the solving itself. Adding the variables also seems to take some amount of time in comparison to setting the objective and adding the constraints looking at Figure 7.2, which only shows the first six problem sizes of Figure 7.1. But considering bigger problems everything except the solving time takes an insignificant time in the whole solving process (cf. Figure 7.1).

The graph of the solving time in Figure 7.1 lets assume a polynomial or exponential growth w.r.t. the problem size N_g. Which is consistent with the exponential complexity of a MILP.

Figure 7.3 shows the solving time per grid cell. The blue curve is the solving time divided by the number of all grid cells N_g and the orange curve plots the solving time divided by the number of grid cells that are in the target shape N_t Here it can also be observed that the trend is not linear. The bigger a problem instance is the more time is needed per grid cell.

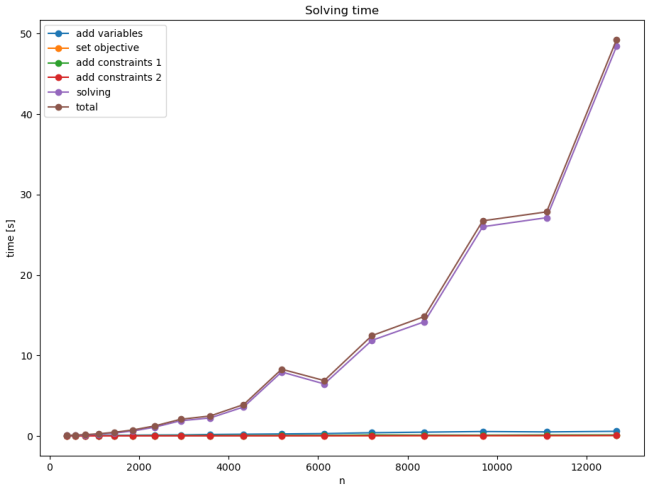


Figure 7.1: Time in seconds of different steps in the solving process over the number of grid cells N_g.

Because the solving time rises a lot with increasing problem size, it is reasonable to accelerate the solving with for instance learning a heuristic with a neural network.

7.1.2 Different Target Shapes

Here different target shapes are evaluated. Figure 7.4 shows that the solving does not change significantly for different shapes. Everything except the solving time is constant which makes sense because the number of grid cells and the number of blocks that assemble the parts are (almost) equal for all shapes. The deviation in time is about $50\,\mathrm{ms}$ and could come from factors like process scheduling, which would mean that the shape is not very important for the solver. To check that, we determined the variance of the times over 100 runs. The result is a variance in the magnitude of 10^{-5}, so the deviation in the solving time is neglectable. That means, that the shape influences the solving time. The two highest solving times are observed for the cuboid and the arbitrary shape which are those with a higher cross-sectional area. But, the result is not clear enough to give certain properties significance. So we come to the conclusion that the shape does not play a substantial role.

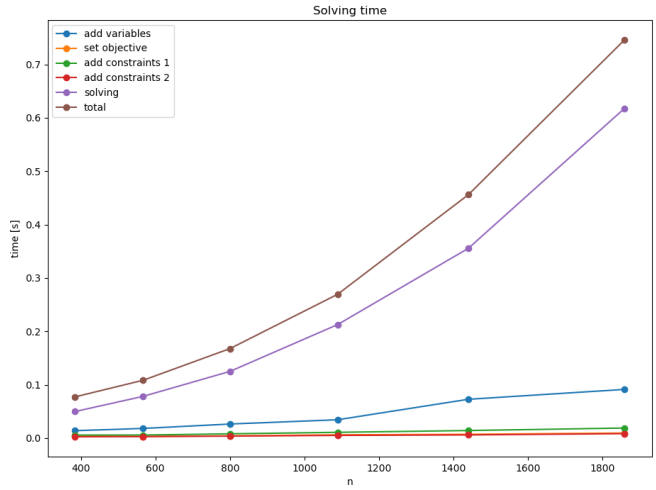


Figure 7.2: Time in seconds of different steps in the solving process over the number of grid cells N_g. Here less N_g to compare the different parts of solving better.

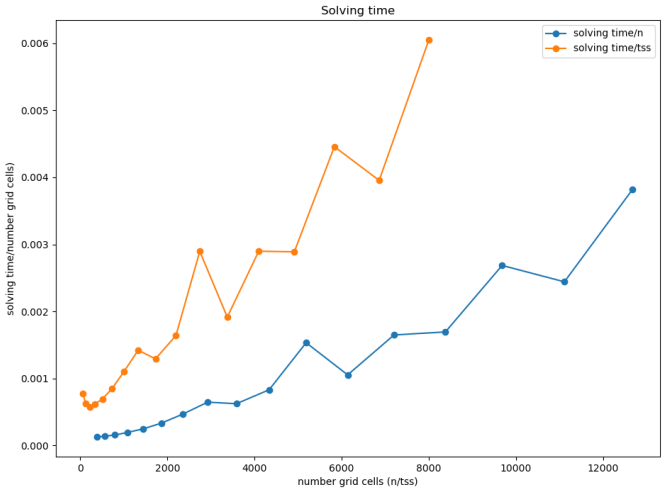


Figure 7.3: Solving time divided by the number of grid cells N_g (blue) and the number of grid cells in the target shape tss (orange).

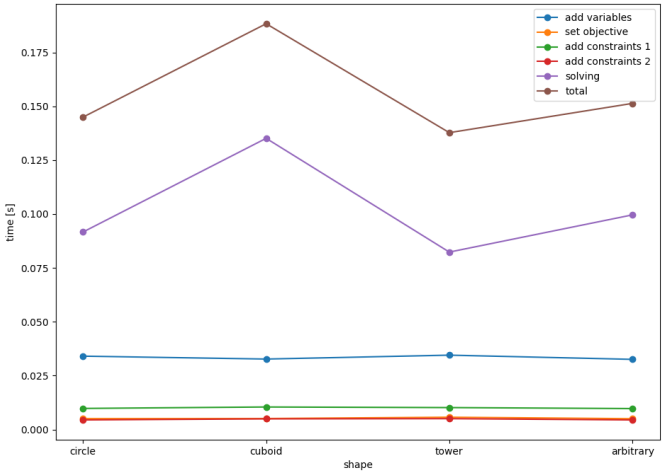


Figure 7.4: Time in seconds of different steps in the solving process over different target shapes.

7.1.3 Different Sets of Parts

In Figure 7.5 the solving time is plotted over the object class which represents the set of available parts. Here again, the deviation is under a second (for all object classes except the Abraxis part set (cf. Section 7.1.4)). But this time the graph shows expected behavior. Object class 33 only consists of the unit block. So the solver only has to place a block at every grid cell in the target shape. This object class happens to be solved the fastest, which makes sense. Object class 26 needs the most time to solve which is also reasonable because it is the set with the most parts and also more complex parts. Since object class 26 with only rotations around the z-axis has fewer parts than object class 26 with all rotations, it needs less time to be solved. The same applies to objects class 34 and its version with only z-rotations. Object classes 32 and 34 almost need the same amount of time which is interesting because object class 34 is a subset of object class 32 so it has fewer parts. Object class 35 has more complex parts again which explains the higher solving time. So the type of parts that are used seems to be important regarding the solving time, i.e., the more complex the used parts are, the more time is needed for solving.

Table 7.1 shows some information over the problem instance and the solver. The more

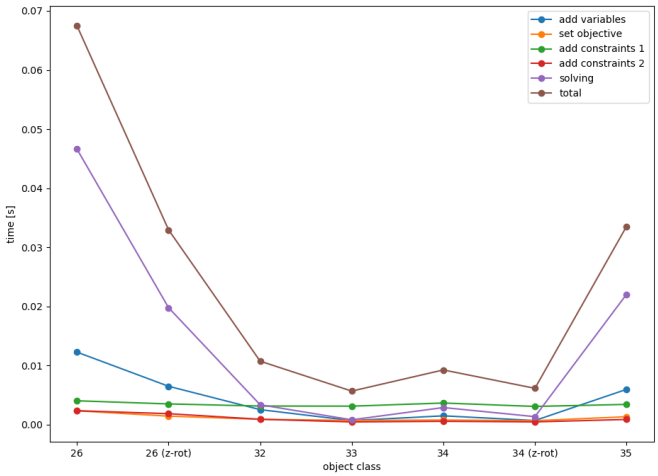


Figure 7.5: Time in seconds of different steps in the solving process over different times of part sets used.

parts a problem instance has, the more simplex iterations are required and thus the time needed increases. That makes sense because the solver has to find the solution for more decision variables, the more parts are considered. Interesting is, a maximum of one node is explored. Those with only one part do not even explore one node. It is possible that here the solution of the relaxed LP already only has binary integer values and thus is already feasible. Then no further nodes need to be explored because the relaxed LP solution is already the optimum.

7.1.4 Abraxis Cube

If we compare Figure 7.6 with Figure 7.5, we can see that the more complex problem needs significant more time for solving. The solving for the single Abraxis cube needs $11.31\,\mathrm{s}$ whereas the solving with object class 26 needs $0.05\,\mathrm{s}$. The cuboid with $8 \times 4 \times 4$ grid cells needs only a little bit more time, but the $8 \times 8 \times 4$ cuboid needs a lot more time again. This is consistent with the observation from the problem size experiment, which shows that the solving time does not grow linear with the problem size.

object class	# parts	explored nodes	simplex iterations	time
26	28	1	807	0.04
26 (z-rot)	14	1	613	0.02
32	4	1	1	0.00
33	1	0	0	0.00
34	3	1	8	0.00
34 (z-rot)	1	0	0	0.00
35	12	1	574	0.02

Table 7.1: Information about a problem instance and its solving for different sets of parts. The number of parts is the number of parts with its rotations.

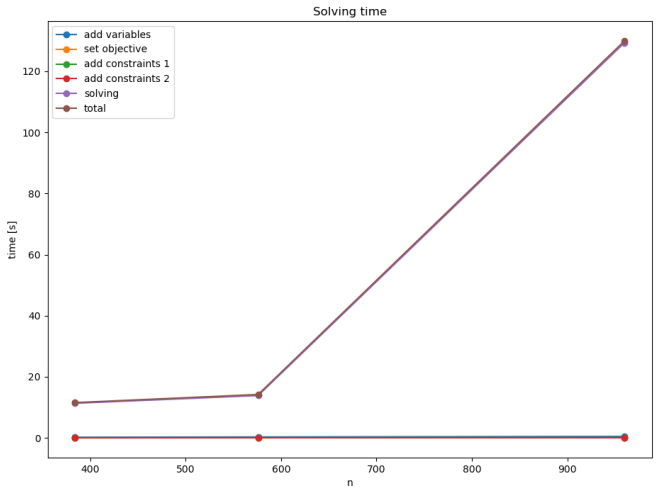


Figure 7.6: Time in seconds of different steps in the solving process over the number of grid cells N_g with combined abraxis cubes to solve.

multiplication	N_r	explored nodes	simplex iterations	time
1	248	94	40874	11.28
2	248	1	41113	13.73
4	248	1	340586	129.33

Table 7.2: Information about a problem instance and its solving for the abraxis cube. The number of parts is the number of parts with its rotations.

Table 7.2 again shows that there is a correlation between the number of simplex iterations and the solving time. Interesting is that the solver explores 94 nodes for the original Abraxis cube but only one for the bigger problem sizes constructed of the Abraxis puzzle.

7.2 GNN

In this section, the results of the experiments for the GNN are discussed.

7.2.1 Different Features

To determine, which features are useful for the GNN, we compare the number of false predictions between the minimum set (set 0) with the other feature set. The results of that are shown in Table 7.3. The loss correlates with that number and even allows a more precise comparison between instances with the same number of false predictions. However, for now, we want to find all features that improve the result in terms of the number of false predictions and thus do not look at the loss in more detail.

It turns out that the features *decision variable index*, *coordinates DVN* and *coordinates GCN* together with *coordinates DVN* improve the result, in which *decision variable index* has the most impact (cf. Table 7.3 left). DVN stands for decision variable node, and GCN stands for grid constraint node. All three are features that make the feature vector of a node more discriminable. So, it is very important that the features of different nodes are not equal, which can easily happen in our scenario. Equal feature vectors result in the same output feature, which will lead to a violation of the constraints of the MILP. Suppose we

have a part with two different rotations, which is assembled out of two unit blocks. Then a lot of decision nodes would be the same (without the identifying features) because they both occupy two grid cells. But in our MILP they cannot occupy the same grid cells because that would violate the constraint. The network, however, does treat them the same and thus places a part at every position where the part is fully in the target shape. This observation is made by looking thoroughly at the predicted decision variables.

The coordinates also give a spatial relation between the grid cells, which also seems to help. Especially if they are given to both the decision variable nodes and the grid constraint nodes.

After figuring out which features improve the result we tried other combinations of features that might be helpful together. If we compare the result of feature set 6, 7, and 8 we can see, that the coordinates given to the grid constraint nodes in addition to the coordinates for the decision variable nodes improved the result in comparison to the coordinates for the decision variable nodes alone. But the coordinates given to the grid constraint nodes alone did not reduce the number of false predictions. Hence, some features are stronger together. That motivated us to combine features that represent an index or the coordinates. We also combine all features together which resulted in a better prediction.

Looking at the right Table in Table 7.3 we can notice that all combinations are better than the minimum feature set 0, which is expectable. Adding another feature to a feature set can also impair the result for feature sets 3 and 13. The best result, however, has the feature set with all features which improved the minimum set together (set 15).

7.2.2 Different Problem Instances

We trained the model on different problem instances with different parts and with or without offset. The results in Table 7.5 show, that the network has no problems predicting the solution right for problem instance 1. That problem instance only uses unit blocks, which explains the result. From analyzing the predictions of the model we figured out, that the network places a part at every position where the part does not reach out of the target shape. In the case of the unit blocks, the result of that equals the solution and does not violate the grid constraint. For the 2-block part, however, that does not hold true. Here the part is also placed at every possible index where the part is still in the target shape, but it should only be placed at every other grid cell, to avoid overlapping. So for all problem sizes with more parts the network makes false predictions.

	# false predictions	loss
0	164	0.2707
1	164	0.2707
2	164	0.2754
3	**104**	**0.1857**
4	164	0.2569
5	164	0.2649
6	164	0.2753
7	**157**	0.2418
8	**151**	0.2227
9	164	0.2699
10	164	0.2764
11	164	0.2764
12	164	0.2740
13	**116**	0.2020
14	**119**	0.2026
15	**68**	**0.1297**
16	**103**	0.1708
17	**107**	0.1788

Table 7.3: Amount of false predictions and loss after 150 epochs. The number corresponds to the number of the feature set in Table 7.4.

feature	13	14	15	16	17
kind	x	x	x	x	x
# primitive elements					
part id					
# edges reward	x	x	x	x	x
# edges penalty	x	x	x	x	x
decision variable index	x		x	x	x
grid idx DVN	x	x		x	x
grid idx GCN					x
coordinates DVN		x	x	x	x
coordinates GCN			x	x	
objective value	x	x	x	x	x
lower bound GCN					
lower bound ACN					
upper bound GCN	x	x	x	x	x
upper bound ACN	x	x	x	x	x
# edges decision variables					
# rotations					

Table 7.4: Feature sets for the experiment, after first evaluation of feature sets in Table 7.4. DVN means decision variable node, GCN means grid constraint node, and ACN amount constraint node.

But if we look at the problem instances without offset, the network can make the right prediction if trained for 1000 epochs. Here the network cannot place a block outside of the target shape because the grid only spans the target shape, and the number of decision variables is smaller. The network did not place a part outside of the target shape before, so we think the smaller problem size is the reason for the good result. But on the other hand, the network might also just learn a pattern. Because of the simple parts, we are using the ground truth solution is very regular which could be exploited by the network. It might also be that the network, in the case with offset, sets the focus too much on the decision if a part is in the target shape or not, and does not utilize the other information much. While the network without offset might exploit the information coming from the constraint nodes, and thus actually follow the constraints.

For problem instance 5 the error is higher after 1000 epochs than after 200 epochs. The learning curve in Figure 7.7 indicates that the learning rate is too high and the optimum was overshot.

When comparing the losses it stands out that a lower number of false predictions can still have a higher loss, than a problem instance with more false predicted variables (cf. problem instances 2 and 5). That makes sense because problem instance 2 has fewer decision variables. So the ratio between false predicted variables and decision variables in total is higher here. So, it is important to consider a different amount of decision variables or a different count of decision variables that are assigned to 1 in the ground truth, when comparing the loss of those problem instances. That is why the losses for the two biggest problem instances 3 and 5 are the lowest, except for the first problem instance. The first problem instance is even smaller which shows that it is a correct prediction with high confidence.

7.2.3 Different Connectivity

Table 7.6 shows the result of training the model with graphs having different connectivity. All numbers are the same for the original connection of the graph representing the connectivity between the decision nodes, and for a fully or randomly connected graph. That lets us assume that the connections between the nodes do not influence the result at all because the loss probably would have different values otherwise. This is bad because those connections are important to consider the constraints. That might also be the reason why the predictions in the other experiments mostly do not respect the constraints.
On the other hand, it is also possible that the connections only have an impact if the features are meaningful enough. So, that might indicate that the feature set is not good

id	N_d	# 1s in ground truth	# false predictions		loss	
			200 epochs	1000 epochs	200 epochs	1000 epochs
1	384	64	**0**	**0**	**0.0085**	**5.6794e-5**
2	384	32	16	16	0.1104	0.1058
3	1152	32	16	16	0.0555	0.0540
4	768	42	70	70	0.2424	0.2384
5	1536	44	29	109	0.0549	0.1455
6	64	32	16	**0**	0.7392	0.0003
7	192	32	16	**0**	0.3397	0.0001
8	128	39	89	**0**	1.5250	0.0020
9	256	41	71	**0**	0.7209	0.0014

Table 7.5: Amount of false predictions and loss for 200 epochs and 1000 epochs. The id corresponds to the id of the problem instance in Table 6.3.

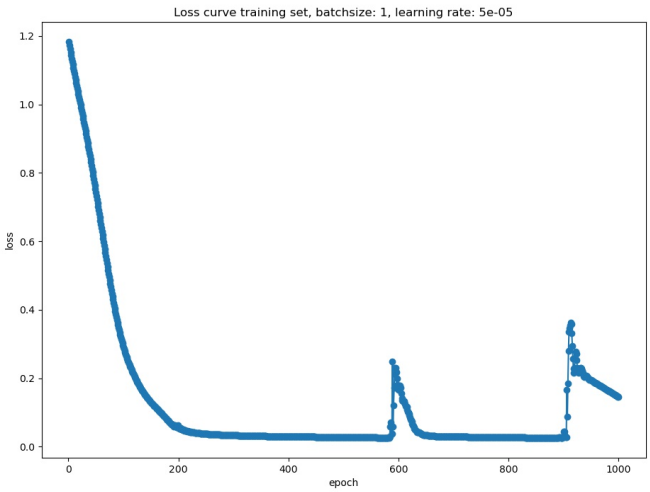


Figure 7.7: Loss curve of training with problem instance 9 for 1000 epochs.

enough, yet, such that the connection between the nodes is not seen as important and does not influence the prediction.

7.2.4 Use Model as heuristic for the MIP Solver

Figure 7.8 shows the solving time for the problem when incorporating the GNNs prediction with the different, presented methods, and the time without the usage of the GNN. All times where the GNN is used are higher than the solving time without the GNN. We think that it takes longer because the problem is so small that the solver quickly finds the solution while the GNN creates an overhead because of additional computations. Hence, it is useful to undertake this experiment again, once a model, which can predict satisfactory predictions on big problem instances, is trained in future work.

Among the methods using the GNN, however, usage of variable hints together with its certainty outperforms the method where the solution is set in the callback. That the variable hints with certainty are better than without makes sense because it is additional information that can be exploited by the solver.

	# false predictions	loss
original connection	**0**	**0.0014**
fully connected	**0**	**0.0014**
randomly connected	**0**	**0.0014**

Table 7.6: Amount of false predictions and loss after 1000 epochs for different connectivity between the graph nodes.

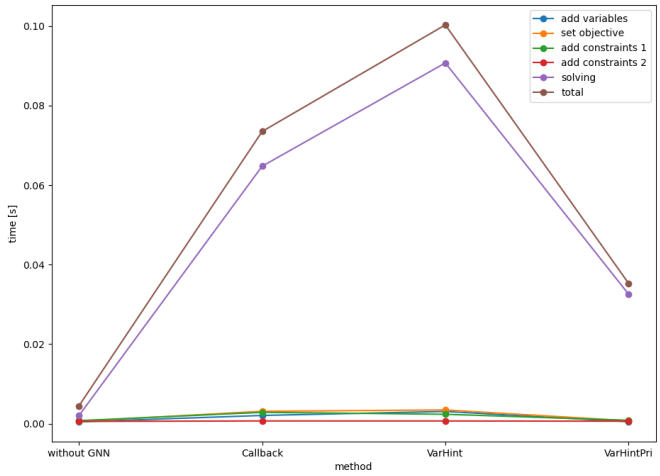


Figure 7.8: Time in seconds of different steps in the solving process over different methods to incorporate the GNN in Gurobi.

8 Conclusion

In this chapter the results of the experiments presented in chapter 6 and 7 are discussed.

8.1 MILP Results

The results of Section 6.1 show that the computation time of a MIP solver is mostly influenced by the problem size. Problem instances in a real assembly problem can get very big. If we look for instance at a skyscraper, thousands of parts of different shapes are needed to construct such a building. But not only the amount of parts, but also the size of the grid will be very big because the overall building is big in comparison to the parts it is built of. The complexity of parts also plays a role in the computation time albeit it might not be as significant as the problem size. But if the parts are that complex that there is only one solution the computation time also rises quite fast, as we can see in the example of the Abraxis cube (cf. Subsection 7.1.4). The shape, however, seems to play a minor role in the computation time.

Even though the solution does not necessarily need to be computed in real-time, it is desirable to have short computation times. Thus it is reasonable to apply a method that accelerates the solving time. In this work, the use of a GNN for this purpose is evaluated.

8.2 GNN Results

The training of the GNN turned out to be more difficult than we expected at the beginning. Even though we train the network just on one sample and do not care about overfitting for now, the network keeps predicting some decision variables wrong. And for the cases the network predicts right, we cannot certainly say that the network does not exploit

patterns in the ground truth instead of actually respecting the constraints, but especially because the connection does not seem to be of any matter for the network, it is more likely that just a pattern is exploited. Also, in most cases the network does not follow the constraints, i.e., some grid cells are occupied by more than one block and more parts are placed than available. To avoid that problem it is likely that a sequential approach would be beneficial because then the network could also take into account which grid cells are already occupied and also how many parts still are available. This information could be included in the feature set and would make the features more distinct, i.e., if a node is chosen it should have a significantly different feature vector that informs the rest, such that in the best case the rest is not chosen if they violate the constraints.
But apart from that, the network already manages to place parts only into the target shape. So we think that the network has the potential to yield good results after implementing sequential decision making. When applying the network on the MIP solver Gurobi, it is still slower than without using the GNN. But we only solved a small problem with the GNN so far, which produced fast solutions anyway. So, we believe that the GNN will improve the solving time for bigger problems. The method to give variable hints together with its confidence is the best method among the tested ones, to incorporate the prediction.

9 Outlook

We presented an MILP environment which represents an assembly problem in a discrete environment and also did first attempts to speed up the solving process using a GNN. Initial results show that it generally could work to accelerate the MIP solver with a GNN but also reveal problems with more complex instances.
So, in future work, the GNN has to be tackled again as it is not producing satisfactory results, yet, however, we could gain valuable insights for our future approaches. It seems, that the network cannot solve all problems at once but rather a sequential decision making approach is necessary. The model appears to have problems with considering the constraints. A sequential approach with additional features providing information about the current state could help the network because those features could represent the constraints and those violations better. Those features could be the number of blocks that occupy a grid cell, and how many parts are still available, after some are already placed, etc. The sequential approach could either again try to give a solution for the MILP which is then given to the MIP solver to guarantee optimality. Or the network could guide the branching and value assignment of the decision variables during the solving process of the MIP solver as in [8]. With additional features representing the current state, we think the network could produce better results.
Once the network produces better results the experiment evaluating the different methods should be repeated. It could also be interesting to give the partial prediction based on its certainty to the solver. Here the variable `Start` which represents the current start vector can also be taken into account again.

Also, our current solution does not have any sequential information over the placement of the parts, which, however, is crucial information if it comes to the actual assembly task. Here the robot needs to know which part has to be placed next and at which position it should be placed. This information is provided in the work of Funk et al. [5], but they cannot guarantee optimality. So we will investigate this and try to also incorporate the sequential information. This also allows checking if a partially assembled construction is

feasible, i.e., if it is stable enough and if the robot can place the part within its working space without collisions.

Bibliography

[1] B. Akomah, L. Ahinaquah, and Z. Mustapha, "Skilled labour shortage in the building construction industry within the central region," 08 2020.

[2] M. Abrey and J. Smallwood, "The effects of unsatisfactory working conditions on productivity in the construction industry," *Procedia Engineering*, vol. 85, pp. 3–9, 2014. Selected papers from Creative Construction Conference 2014.

[3] R. Spence and H. Mulligan, "Sustainable development and the construction industry," *Habitat International*, vol. 19, no. 3, pp. 279–292, 1995.

[4] S. Tibbits, *Autonomous Assembly Designing for a New Era of Collective Construction*. 2018. OCLC: 1187431078.

[5] N. Funk, G. Chalvatzaki, B. Belousov, and J. Peters, "Learn2assemble with structured representations and search for robotic architectural construction," in *5th Annual Conference on Robot Learning*, 2021.

[6] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," *arXiv:2102.09544 [cs, math, stat]*, Apr. 2021. arXiv: 2102.09544.

[7] Y. Shen, Y. Sun, A. Eberhard, and X. Li, "Learning Primal Heuristics for Mixed Integer Programs," *arXiv:2107.00866 [cs, math]*, July 2021. arXiv: 2107.00866.

[8] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals, and Y. Zwols, "Solving Mixed Integer Programs Using Neural Networks," *arXiv:2012.13349 [cs, math]*, July 2021. arXiv: 2012.13349.

[9] L. A. Wolsey, *Integer Programming* -. New York: John Wiley Sons, 1998.

[10] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations*, pp. 85–103, Springer US, 1972.

[11] B. Bah and J. Kurtz, “An integer programming approach to deep neural networks with binary activation functions,” 2020.

[12] Y. Peng, B. Choi, and J. Xu, “Graph embedding for combinatorial optimization: A survey,” *CoRR*, vol. abs/2008.12646, 2020.

[13] M. Elwekeil, M. Alghoniemy, H. Furukawa, and O. Muta, “Lagrangian relaxation approach for low complexity channel assignment in multi-cell wlans,” in *2013 International Conference on Computing, Networking and Communications (ICNC)*, pp. 138–142, 2013.

[14] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2021.

[15] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[16] C. Gallicchio and A. Micheli, “Fast and deep graph neural networks,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 3898–3905, Apr. 2020.

[17] Z. Liu and J. Zhou, “Introduction to graph neural networks,” *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, pp. 1–127, Mar. 2020.

[18] M. Khamsi, “An introduction to metric spaces and fixed point theory,” 01 2001.

[19] M. J. D. Powell, “An efficient method for finding the minimum of a function of several variables without calculating derivatives,” *The Computer Journal*, vol. 7, pp. 155–162, 01 1964.

[20] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, vol. abs/1609.02907, 2016.

[21] R. Li, S. Wang, F. Zhu, and J. Huang, “Adaptive graph convolutional neural networks,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.

[22] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” 2017.

[23] T. Yan, H. Zhang, Z. Li, and Y. Xia, “Stochastic graph recurrent neural network,” 2020.

[24] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[25] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The World Wide Web Conference*, WWW '19, (New York, NY, USA), p. 2022–2032, Association for Computing Machinery, 2019.

[26] J. Zhang and L. Meng, "Gresnet: Graph residual network for reviving deep gnns from suspended animation," *CoRR*, vol. abs/1909.05729, 2019.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[28] J. Oren, C. Ross, M. Lefarov, F. Richter, A. Taitler, Z. Feldman, C. Daniel, and D. Di Castro, "SOLO: Search Online, Learn Offline for Combinatorial Optimization Problems," *arXiv:2104.01646 [cs, math]*, May 2021. arXiv: 2104.01646.

[29] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning." `http://pybullet.org`, 2016–2020.

[30] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "PySCIPOpt: Mathematical programming in python with the SCIP optimization suite," in *Mathematical Software – ICMS 2016*, pp. 301–307, Springer International Publishing, 2016.

[31] K. Bellock, N. Godber, and P. Kahn, "bellockk/alphashape: v1.3.1 release," Apr. 2021.

[32] M. Rafiq, G. Bugmann, and D. Easterbrook, "Neural network design for engineering applications," *Computers Structures*, vol. 79, no. 17, pp. 1541–1552, 2001.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[34] R. Y. Rubinstein and D. P. Kroese, *The Cross-Entropy Method - A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Berlin Heidelberg: Springer Science Business Media, 2013.