Design and Evaluation of Graph Neural Network-Based Heuristics in Sokoban Search

Bachelor's Thesis

Nikita Filenko
0485074

19. März 2026

Supervisor: Prof. Dr. Benjamin Blankertz
Prof. Dr. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Abstract

Sokoban is a well-known puzzle game that poses a challenging planning problem due to its large state space and the presence of irreversible actions. Optimal Sokoban solvers typically rely on heuristic search algorithms such as A*, where search efficiency critically depends on the quality of the heuristic function. Classical heuristics are hand-crafted and domain-specific, which limits their flexibility and scalability across heterogeneous level distributions.

This thesis investigates whether a Graph Neural Network can be trained as an effective heuristic for Sokoban to reduce search effort in A* search. A push-based state space is considered, and an A* solver is implemented with deadlock detection and a classical baseline heuristic based on Hungarian assignment between boxes and goals. Supervised training targets are obtained by generating labeled datasets from solved instances, where intermediate states are extracted and labeled with the remaining number of pushes to reach a goal state. An edge-aware Graph Isomorphism Network is trained and integrated into the solver as a drop-in heuristic for efficient inference.

Multiple model configurations are trained and compared in different hyperparameter settings, and the resulting learned heuristics are evaluated against the Hungarian baseline. In addition, transferability between level subsets is analyzed by evaluating models across multiple data "blocks," and an automatic clustering procedure is proposed that groups packs based on cross-evaluation performance to identify subsets with similar generalization behavior.

Kurzfassung

Sokoban ist ein bekanntes Puzzlespiel, das aufgrund seines großen Zustandsraums und irreversibler Aktionen ein anspruchsvolles Planungsproblem darstellt. Optimale Sokoban-Löser stützen sich typischerweise auf heuristische Suchalgorithmen wie A*, wobei die Sucheffizienz entscheidend von der Qualität der Heuristikfunktion abhängt. Klassische Heuristiken sind handgefertigt und domänenspezifisch, was ihre Flexibilität und Skalierbarkeit über heterogene Levelverteilungen hinweg einschränkt.

Diese Arbeit untersucht, ob ein Graph Neural Network als effektive Heuristik für Sokoban trainiert werden kann, um den Suchaufwand in der A*-Suche zu reduzieren. Betrachtet wird ein push-basierter Zustandsraum, und es wird ein A*-Solver mit Deadlock-Erkennung sowie einer klassischen Basisheuristik implementiert, die auf der Hungarian-Assignment-Zuordnung zwischen Kisten und Zielen beruht. Für das überwachte Training werden Zielwerte erzeugt, indem gelabelte Datensätze aus gelösten Instanzen generiert werden: Dabei werden Zwischenzustände extrahiert und mit der verbleibenden Anzahl an Pushes bis zum Erreichen eines Zielzustands annotiert. Ein kantenbewusstes Graph Isomorphism Network wird trainiert und als austauschbare Heuristik in den Solver integriert, um effiziente Inferenz zu ermöglichen.

Mehrere Modellkonfigurationen werden unter unterschiedlichen Hyperparameter-Einstellungen trainiert und verglichen, und die resultierenden gelernten Heuristiken werden gegen die Hungarian-Baseline evaluiert. Zusätzlich wird die Übertragbarkeit zwischen Level-Teilgruppen analysiert, indem Modelle über mehrere Daten-„Blöcke“ hinweg bewertet werden, und es wird ein automatisches Clustering-Verfahren vorgeschlagen, das Packs anhand der Cross-Evaluation Performance gruppiert, um Teilmengen mit ähnlichem Generalisierungsverhalten zu identifizieren.

Contents

1 Introduction

1.1 Sokoban and Its Origins

Sokoban[1] is a transport puzzle in which a player moves through a grid-based warehouse and must push all boxes onto designated storage locations. The rules are simple: boxes can only be pushed, only one box can be pushed at a time, and a minor error can make a level unsolvable. This combination of simple mechanics and strict constraints results in puzzles that require careful planning.

Since its release, Sokoban has inspired a large number of ports, clones, and community-created levels[2]. In addition to its cultural impact, Sokoban has become a widely used benchmark in artificial intelligence and planning research, because it demonstrates significant combinatorial complexity, while having a compact state representation[3, 4]. In particular, the presence of irreversible actions and dead-end configurations (deadlocks) makes it challenging for heuristic search methods and interesting for studying how different heuristic functions affect solver performance[4].

1.2 Sokoban as a Search and Planning Problem

Sokoban can be formulated as a deterministic planning problem. A *state* is defined by the layout of the level (static features such as walls and goal tiles) together with the current positions of the player and all boxes. An *action* moves the player to a neighboring free tile, and if the player steps into a box, the box is pushed by one tile in the same direction (only if the target tile is free). A state is a *goal state* if all boxes are placed on goal tiles.

In this thesis, Sokoban is solved in a push-based formulation, where the *move* is defined over *box pushes* instead of individual player moves. In this view, a transition corresponds to pushing a box once, while player movement between pushes is treated as cost-free navigation within the reachable area. This choice is made because pushes represent the irreversible decisions that largely determine whether a level remains solvable.

The problem is difficult for a number of reasons. First, the state space is very large, even for small boards, because there are many possible box configurations. Second, the moves in Sokoban are irreversible, because boxes pushed into corners or against walls may be trapped from reaching any goal position. The detection of deadlocks is complex, and a solver that ignores them may waste most of its time searching states that will never lead to a solution. Third, the branching factor is large because there are often several boxes that can be pushed in several different directions, and each push opens up new paths for the player.

Because of this complexity, solvers commonly use heuristic search, in particular, A*. A* expands states in order of increasing $f(s) = g(s) + h(s)$, where $g(s)$ is the cost from the start state to s (e.g., the number of pushes so far) and $h(s)$ is a heuristic estimate of the remaining cost to reach a goal[5]. The heuristic function is critical: if $h(s)$ is informative, the search focuses on promising states and expands fewer nodes. If $h(s)$ is weak, the search behaves closer to uninformed search and quickly becomes slow.

Classical Sokoban heuristics are typically hand-crafted approximations, for example by matching boxes to goals using simple distance measures, often combined with deadlock detection rules[6, 7]. These heuristics provide a strong baseline and can be very effective on many instances, but they also have important limitations. First, they require manual design and expert knowledge about the domain: improving a heuristic often means adding new rules, additional pruning logic, or more complex problem-specific components[6]. Second, stronger hand-crafted heuristics often come with increased engineering and maintenance effort, because they need careful tuning, extensive testing to avoid false deadlock detections, and may depend on maze-specific structure[8]. These limitations motivated the learning-based approach, where heuristics are learned directly from data.

1.3 Graph Neural Networks for Sokoban

Graph Neural Networks (hereafter abbreviated as GNNs) are neural networks designed to work with data that can be written as a graph[9]. The main idea is simple: each node has a feature vector, and information is exchanged along edges. By repeating this *message passing* step several times, each node can collect information from its neighbors (and, after multiple steps, from a larger part of the graph). The representations of the nodes can then be combined to form a representation of the entire graph, which is then used to predict a value.

There are several properties of GNNs that are useful to Sokoban. First, GNNs can naturally operate on graphs with varying numbers of nodes and edges, since the same local update functions are shared across graph elements[10]. This is significant because different levels have different board sizes and different numbers of reachable tiles. Second, the board has a clear structure: tiles are connected by grid neighbors, and many important constraints are local (e.g. walls, corners, narrow corridors). Message passing fits this well because it propagates local information through the neighborhood structure. Third, GNNs are naturally invariant to the ordering of nodes: the same level represented with a different node ordering should lead to the same prediction, which is important for stable learning and transfer[10].

Compared to models such as CNNs, a GNN can use the level structure directly instead of relying on a fixed grid[11]. This makes it easier to limit the computation to the tiles that actually belong to the level, rather than having to compute over large areas of empty space around the board. The graph representation is also easily scalable to different sizes and shapes of levels, whereas image-based inputs may require padding or resizing, which adds unnecessary structure and noise[11].

Sokoban can therefore be encoded as a graph in a natural way. Walkable tiles can be represented as

nodes, edges can connect tiles that are adjacent on the grid, and node features can describe whether a tile is a wall border, a goal, contains a box, or contains the player. In this representation, the geometry of the level and the current configuration are both available to the model, while still keeping a consistent format across different levels.

In this thesis, a GNN is trained to predict a heuristic value $h(s)$ for A* search. The target value is the remaining cost-to-go from a state s, measured in the number of box pushes needed to reach a goal state. After training, the GNN is integrated into the A* solver as a drop-in heuristic function, so that search performance can be evaluated and compared to classical heuristics.

1.4 Thesis Scope, Research Questions, and Structure

1.4.1 Goal of the Thesis

The goal of this thesis is to design and evaluate a learning-based heuristic for Sokoban that can be used within an A* solver. A Graph Neural Network is trained to predict a heuristic value $h(s)$ for a given Sokoban state s, with the aim of reducing search effort compared to a classical baseline heuristic, while remaining usable across different level sets.

1.4.2 Research Questions

This thesis is guided by the following research questions:

- Can a GNN-based heuristic reduce A* search effort compared to a classical Hungarian matching baseline on Sokoban benchmarks?
- How do different model configurations and hyperparameters affect the quality of the learned heuristic and the resulting search performance?
- How well does a learned heuristic transfer across different subsets of levels, and can transferable subsets be identified automatically?

1.4.3 Scope and Contributions

The scope of this thesis is limited to improving the practical search performance under fixed budgets. The focus lies on the design of state representations, supervised learning of cost-to-go targets, and the integration of the learned model as a heuristic function in A*. Topics such as learning full policies or proving optimality guarantees for the learned heuristic beyond the baseline-combined setting are beyond the scope.

The main contributions are:

- A complete pipeline for generating labeled Sokoban states, training a GNN to predict cost-to-go, and integrating the model into an A* solver as a drop-in heuristic.

- A graph-based state encoding for Sokoban, including node and edge features derived from level geometry and dynamic game objects.
- An experimental comparison of multiple GNN configurations and hyperparameter settings against a classical Hungarian matching baseline, using a consistent evaluation protocol.
- An analysis of transferability across level subsets (data blocks) and a clustering approach based on cross-evaluation results to identify groups with similar generalization behavior.

1.4.4 Structure of the Thesis

Chapter 2 describes the implemented solver, the classical baseline heuristics, the dataset generation process, and the GNN model architecture. Chapter 3 presents the experimental setup and evaluation protocol, followed by results on heuristic accuracy and search performance, including hyperparameter comparisons and transferability analyses. Chapter 4 discusses the findings, limitations, and implications of learned heuristics in Sokoban search. Chapter 5 concludes the thesis and outlines directions for future work.

2 Methods

2.1 Problem Formulation and Notation

This chapter describes the methods used to build an A* solver for Sokoban and to train a learned heuristic that can guide the search. As noted above, a push-based formulation is used throughout this work. Each transition corresponds to a single valid *push*, i.e., moving one box by one tile. The path cost is therefore measured in *pushes*.

This formulation also simplifies the search logic. Between two pushes, the player can often move freely through already open corridors without changing the positions of boxes. For search, these pure movement steps are less informative than pushes, because only pushes change the box configuration and can create irreversible mistakes. Therefore, a push transition $s \rightarrow s'$ is defined as follows: first, all tiles reachable by the player without pushing any box are computed; then, each box is checked for possible pushes that can be executed from the reachable area. Algorithm 1 summarizes the push-based successor generation used in this work.

Algorithm 1 Push-based successor generation

```
function GeneratePushSuccessors(s)
    R ← player-reachable tiles in s without pushing any box        ▷ computed by BFS
    succs ← [ ]
    for all boxes b in state s do
        for all directions d ∈ {up, down, left, right} do
            behind ← tile adjacent to b opposite to d
            front ← tile adjacent to b in direction d
            if behind is outside board or front is outside board then
                continue
            end if
            if behind ∈ R and front is free floor then
                s′ ← s
                move box from b to front in s′
                set player position in s′ to b
                append s′ to succs
            end if
        end for
    end for
    return succs
end function
```

The set of reachable tiles is computed with a breadth-first search (BFS) on the grid, treating walls and boxes as obstacles. A push is considered valid if the player can stand on the tile *behind* a box (relative to the push direction) and the target tile *in front* of the box is free. If the push is applied, the box moves by one tile in the chosen direction, and the player moves into the box's previous position. The static layout (walls and goal tiles) remains unchanged.

A central difficulty in Sokoban is the existence of *deadlocks*[4]. A deadlock is an unsolvable state, meaning that no sequence of valid actions can reach a goal from that state. Deadlocks are hard to detect in general, but simple and reliable rules can still remove a large fraction of hopeless states in practice. In this work, deadlock pruning is used both to speed up A* search and to improve the efficiency of dataset generation for supervised learning.

Deadlock rules are not required to detect every unsolvable state. Missing a deadlock only reduces pruning power and may slow down the search. However, false positives are critical: if a solvable state is classified as deadlocked, the solver may prune the only path to a solution. For this reason, the deadlock checks used in this thesis are designed to be conservative.

The following deadlock rules are applied: *corner deadlocks* (a box not on a goal stuck in a corner), *corridor-line deadlocks* (a box trapped in a corridor segment that contains no goal tile), 2×2 *deadlocks* (a solid 2×2 block of walls/boxes without any box on a goal). States detected as deadlocked are treated as unsolvable and are pruned. Figure 2.1 shows examples of such deadlocks.

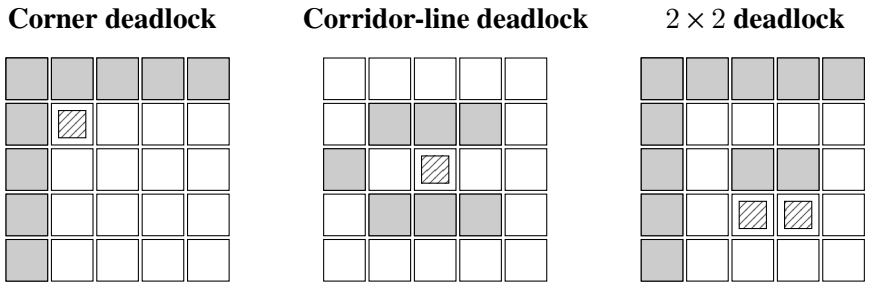


Figure 2.1: Examples of simple deadlock patterns used for pruning.

2.2 A* Solver and Search Infrastructure

A* search[5] is used as the solving algorithm in this thesis. The solver maintains an *open list* of discovered but not yet expanded states, implemented as a priority queue ordered by the evaluation value $f(s) = g(s) + h(s)$. In each iteration, the state with the smallest $f(s)$ is extracted and expanded. For solution reconstruction, a parent pointer is stored whenever a state is reached with an improved path cost.

To avoid expensive priority updates in the queue, a standard "stale entry" check is used. The queue stores pairs (g, s) together with the priority f. When a state is popped, the stored g value is compared against the currently best known $g(s)$. If the entry is outdated (i.e., a better path to s has been found since it was inserted), it is skipped. This keeps the implementation simple while still ensuring that

only the best-known paths are expanded.

For large-scale evaluation and for dataset generation, two resource limits are supported. A *time limit* stops the search after a fixed wall-clock time per instance. In addition, a *node limit* bounds the number of expanded states. If either limit is reached before a goal state is found, the instance is treated as unsolved under the given budget.

Repeated states are a major source of overhead in Sokoban search. To reduce redundant expansions, an optional transposition mechanism is used. Each state is mapped to an integer key using *Zobrist hashing*[12]. In this implementation, the hash depends only on the dynamic parts of the state (box positions and the player position), while static tiles such as walls and goals are not included because they are fixed for a given level. The transposition table stores the best known cost $g(s)$ per hash key. When a successor s' is generated with cost g', it is pruned if the table already contains the same key with a cost $\leq g'$. Otherwise, the stored value is updated.

Algorithm 2 summarizes the A* procedure used in this thesis for push-based Sokoban search. The algorithm operates on push-successor states (Section 2.1), uses the evaluation function $f(s) = g(s) + h(s)$, and applies a stale-entry check to avoid expensive priority updates in the open list. It omits the separate transposition-table bookkeeping for readability.

Algorithm 2 A* search over push-based Sokoban states

function ASTARSEARCH(s_0)
  $open \leftarrow$ empty priority queue ordered by $f = g + h$
  $g[key(s_0)] \leftarrow 0$ ▷ state costs indexed by Zobrist hash
  push $(0, s_0)$ into $open$ with priority $h(s_0)$
  while $open$ is not empty **do**
    $(g_q, s) \leftarrow$ pop from $open$ with smallest priority
    if $g_q \neq g[key(s)]$ **then**
      continue ▷ a better path has already been found
    end if
    if s is a goal state **then**
      return reconstructed path
    end if
    for all $s' \in$ GENERATEPUSHSUCCESSORS(s) **do**
      $g' \leftarrow g[key(s)] + 1$
      if s' is not a deadlock **and** $(key(s') \notin g$ **or** $g' < g[key(s')])$ **then**
        $g[key(s')] \leftarrow g'$
        $h' \leftarrow h(s')$
        push (g', s') into $open$ with priority $g' + h'$
      end if
    end for
  end while
  return failure
end function

2.3 Generating Supervised Labels

This section describes how supervised training data is generated in this thesis. The main idea is to solve levels with a classical heuristic search setup and to extract intermediate states along the solution path. Each extracted state is labeled with the remaining number of pushes required to reach a solved configuration. These labels are later used to train a learned heuristic.

2.3.1 Hungarian Matching Baseline

The simple baseline heuristic used in this thesis is based on matching boxes to goal tiles. For a given state s, let $B = \{b_1, \ldots, b_n\}$ be the current box positions and $T = \{t_1, \ldots, t_n\}$ be the goal positions.[1] A cost matrix $C \in \mathbb{R}^{n \times n}$ is constructed using Manhattan distances on the grid:

$$C_{ij} = \|b_i - t_j\|_1.$$

The heuristic value is then defined as the minimum total assignment cost

$$h_{\text{hung}}(s) = \min_{\pi \in S_n} \sum_{i=1}^{n} C_{i,\pi(i)},$$

where π is a permutation of goals. This minimum is computed using the Hungarian algorithm[13].

This heuristic is a good baseline for push-based search because it provides a lower-bound estimate of the amount of work left: each push can move a box no more than one grid cell, so the total Manhattan distance to the assigned goals cannot be decreased faster than one cell per push in the best case. $h_{\text{hung}}(s)$ is also cheap enough to compute often during search.

At the same time, this heuristic has clear limitations. Walls, narrow passages, and interactions between boxes are ignored. As a result, the estimate can be overly optimistic, especially in levels where boxes must take long detours or where pushing one box blocks another.

2.3.2 Label Definition

For supervised learning, the target value is defined as the remaining pushes derived from the solver. For a state s, the label is

$$y(s) = \text{number of pushes needed to reach a goal state from } s.$$

Here, labels are obtained from solved trajectories. If a level is solved with a push sequence that induces a state path $(s_0, s_1, \ldots, s_T)$, where each transition corresponds to one push, then the remaining cost-to-go label for an intermediate state s_t is

$$y(s_t) = T - t.$$

This creates a set of supervised training pairs $(s_t, y(s_t))$ that can be used to train a model to predict a heuristic value $h(s)$.

[1] In standard Sokoban instances, the number of boxes equals the number of goals.

Field	Description
`width`, `height`	Board dimensions (grid width and height)
`walls`	Bitmask of wall tiles
`goals`	Bitmask of goal tiles
`boxes`	Bitmask of box tiles
`player`	Player position as a single tile index
`board_mask`	Bitmask of tiles that belong to the level area
`y`	Label: remaining number of pushes ($y(s)$)
`level_id`	Identifier of the source level

Table 2.1: Fields stored per example in the JSONL datasets.

2.3.3 Data Collection Procedure

On-policy labels. A dataset is generated by solving levels with A* using the Hungarian baseline wrapped with deadlock pruning under fixed time and node limits. To compute optimal solutions more efficiently during label generation, an external Sokoban solver is also used. In this thesis, the open-source solver *Festival* [14, 15] is used as a backend to solve levels and to obtain the remaining push distances needed for the labels. If a level is solved, the reconstructed solution path $(s_0, \ldots, s_T)$ is used to emit labeled records $(s_t, T - t)$. Levels that are not solved within the budget are skipped.

Off-policy labels. Training only on states that lie on solution trajectories can be limiting: such states represent a very narrow part of the search space and mostly reflect "good" decisions. In search, however, a heuristic is queried on many states that are not on the optimal path, including detours and partially explored alternatives. If these states are missing from the training distribution, the learned heuristic may be poorly calibrated in exactly the situations where guidance is needed most.

To cover a broader set of states, an off-policy collection procedure is used. For each level, a bounded best-first search is run for a limited number of expansions and a short time budget. From the expanded states and from the remaining open frontier, a small number of states is sampled, subject to a minimum depth constraint. Each sampled state is then labeled by solving the level *from that state* with Hungarian heuristics or the *Festival* solver [14, 15], and the remaining number of pushes in that solution is used as $y(s)$. This produces additional training examples that reflect the distribution of states encountered during search, not only states on successful solution paths.

Storage format. Collected samples are stored in JSON Lines (JSONL) format, where each line contains one labeled example. To keep files compact and easy to load, the Sokoban state is stored using integer bitmasks together with the board dimensions. Table 2.1 summarizes the fields.

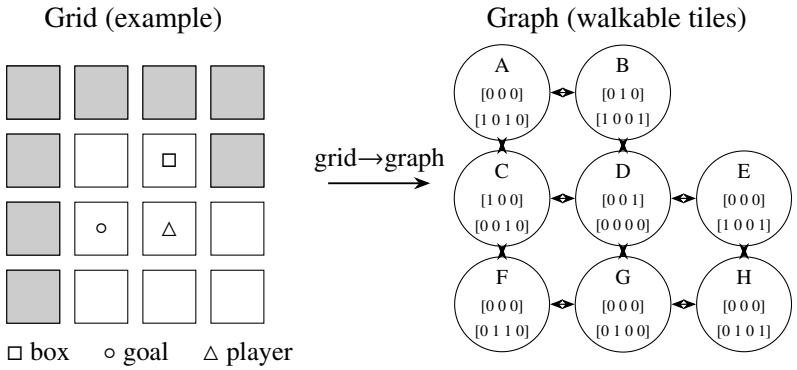


Figure 2.2: Example conversion from a Sokoban grid to a graph.

2.4 Graph Representation of Sokoban States

To apply a Graph Neural Network, each Sokoban state is converted into a graph with node and edge features. The representation is designed to be simple and consistent across different level shapes and sizes. Figure 2.2 illustrates the conversion from a grid-based level to a graph representation.

Nodes and edges. For a given state s, a graph $G(s) = (V, E)$ is built from the level grid. Each node corresponds to one *walkable tile*, i.e., a grid cell that belongs to the level area and is not a wall. Wall cells are excluded from the node set. Edges connect 4-neighborhood tiles: for every pair of adjacent walkable tiles (up, down, left, right), a directed edge is added. Using directed edges allows edge attributes to encode direction explicitly.

Node features. Each node carries a 7-dimensional feature vector:

$$x(v) = [\texttt{is\_goal}, \texttt{has\_box}, \texttt{is\_player}, \texttt{wall\_up}, \texttt{wall\_down}, \texttt{wall\_left}, \texttt{wall\_right}].$$

The first three features describe the current configuration: whether the tile is a goal, contains a box, or contains the player. The remaining four features describe local geometry: for each direction, the feature is set to 1 if the neighbor cell in that direction is a wall or lies outside the grid boundary. This gives the model direct access to local obstacles such as corners and narrow passages.

Edge attributes. Each directed edge has a 4-dimensional one-hot attribute that encodes the direction of the move:

$$e(u, v) \in \{[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]\}$$

for up, down, left, and right. This makes it easier for the model to distinguish different directions during message passing.

Static structure vs. dynamic features. A useful property of this representation is that the graph structure depends only on the level layout. For a fixed level, the set of nodes and the connectivity of edges remain constant. Only a small subset of features changes between states: `has_box` and `is_player` are dynamic, while walls and goals stay fixed. This separation is convenient in practice, because it enables reusing the same graph structure for many states of the same level and updating only the dynamic parts.

2.5 GNN Heuristic Model

This section describes the learned heuristic model used in this thesis. The model takes a Sokoban state s in the graph form introduced in Section 2.4 and predicts a single scalar $\hat{y}(s)$. This value is used as a heuristic estimate $h(s)$ inside A* search.

Figures 2.3 and 2.4 illustrate the learned heuristic pipeline at two levels of detail. Figure 2.3 shows the end-to-end process from a Sokoban state to a scalar prediction and figure 2.4 provides a closer view of a single GINEConv layer.

Inputs and shapes. Each Sokoban state is converted into a graph $G = (V, E)$. Every node $v \in V$ corresponds to one walkable tile. The node feature matrix is denoted by $X \in \mathbb{R}^{|V| \times d_{\text{in}}}$. In this implementation, $d_{\text{in}} = 7$ and the features are

$$X_v = [\texttt{is\_goal}, \texttt{has\_box}, \texttt{is\_player}, \texttt{wall\_up}, \texttt{wall\_down}, \texttt{wall\_left}, \texttt{wall\_right}],$$

Edges encode the 4-neighborhood on the grid. Each directed edge $(u \rightarrow v)$ carries a 4D one-hot direction vector

$$e_{u \rightarrow v} \in \{0, 1\}^4 \quad \text{for } \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}.$$

These edge attributes are used by the edge-aware GINE convolution layers in the model[16].

Hidden size (`hidden`). All internal node embeddings are stored as vectors of size $d_{\text{hid}} = \texttt{hidden}$. At the start of the network, the node features X are mapped into this hidden space by the first graph layer. A larger `hidden` increases model capacity, but also increases runtime and memory. In this architecture, the dominant cost often comes from linear layers of the form `Linear(hidden` $\rightarrow$ `hidden)` inside the per-layer MLP. Since a linear layer uses a weight matrix of shape (`hidden`, `hidden`), the costs scale quadratically with `hidden` for a fixed number of nodes [17].

Edge-aware GIN layers (`layers`). The core of the model is a stack of `layers` $= L$ edge-aware GIN layers[16] (PyTorch Geometric `GINEConv`). Each layer updates node embeddings using the graph structure and the edge direction attributes.
Concretely, at every layer, a small MLP is used inside the graph convolution[18]. That MLP has the form

$$\texttt{Linear} \rightarrow \texttt{ReLU} \rightarrow \texttt{Dropout} \rightarrow \texttt{Linear} \rightarrow \texttt{ReLU} \rightarrow \texttt{Dropout} \rightarrow \texttt{Linear},$$

where ReLU is the standard rectified linear activation[19] and Dropout randomly drops activations during training[20]. After each convolution, another ReLU is applied.

Each additional GNN layer increases the "information range" on the graph: with more layers, a node representation can depend on a larger neighborhood[18]. From a cost perspective, increasing `layers` increases inference time approximately linearly in this implementation, since each additional message-passing layer performs another pass over the graph edges together with local feature transformations [21].

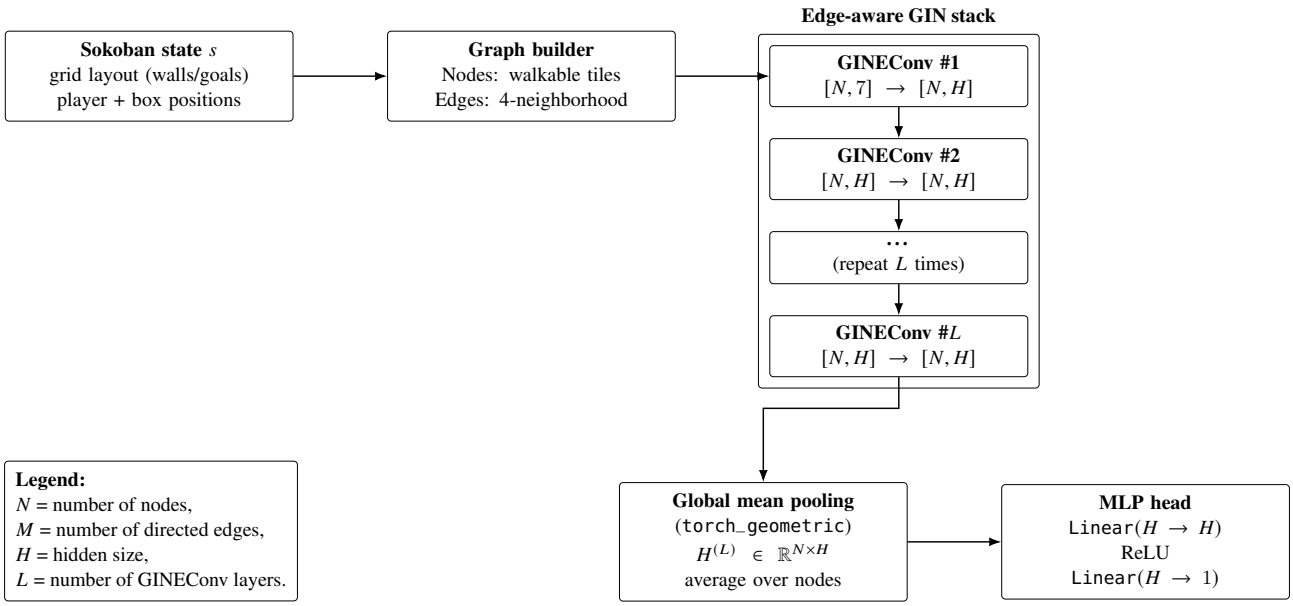


Figure 2.3: End-to-end pipeline of the learned heuristic.

Dropout. Dropout is only active during training and is used inside the MLPs of each graph layer. A higher dropout can help reduce overfitting, but may slow down convergence and can slightly reduce final accuracy if set too high[20]. At inference time, Dropout is disabled, so it does not affect heuristic evaluation speed.

Graph pooling (global mean). After the final graph layer, node embeddings are aggregated into a single graph embedding

$$h_G \in \mathbb{R}^{d_{\text{hid}}}$$

using global mean pooling[22]. This operation averages embeddings across all nodes of the graph, producing a fixed-size vector independent of the board size.

Regression head (output). The pooled vector h_G is fed into a small MLP head:

$$\texttt{Linear}(d_{\text{hid}}, d_{\text{hid}}) \rightarrow \texttt{ReLU} \rightarrow \texttt{Linear}(d_{\text{hid}}, 1),$$

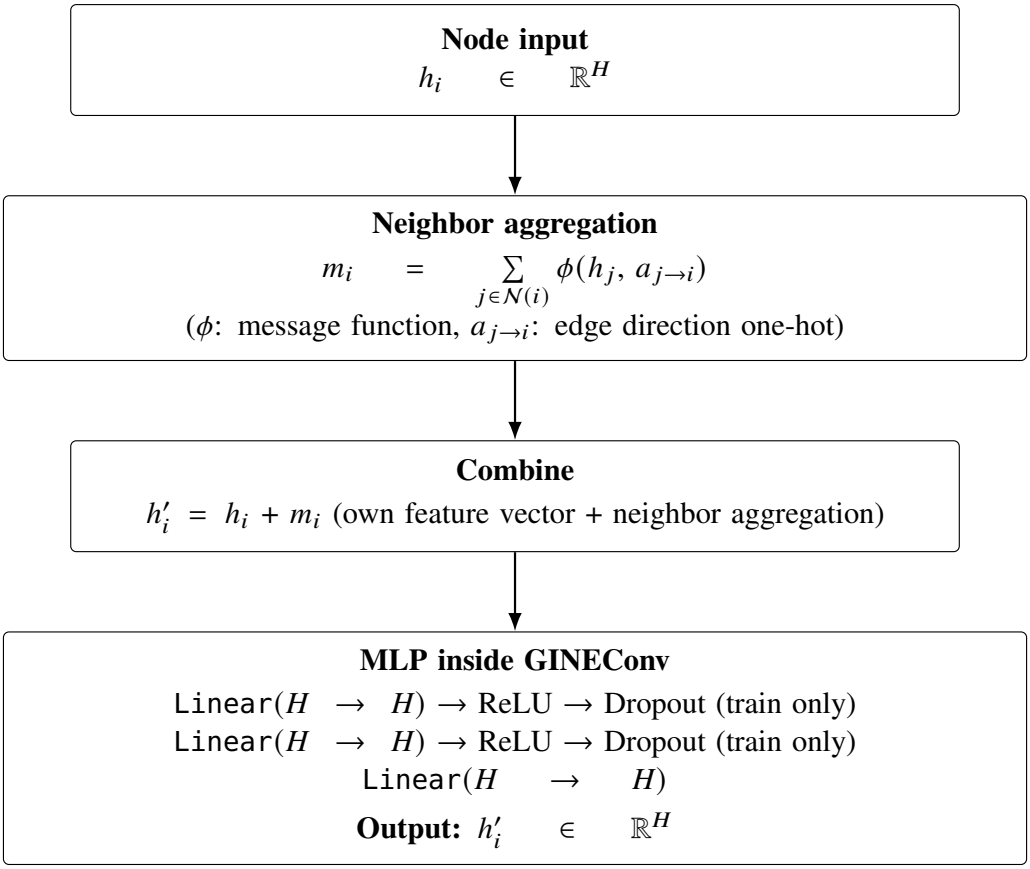
resulting in a single scalar prediction $\hat{y}(s)$. This scalar is used as the learned heuristic value $h_{\text{gnn}}(s)$.

Inference cost and the main speed drivers. In this implementation, heuristic evaluation time depends mainly on the number of layers L, the hidden size d_{hid} and the graph size, i.e., number of floor nodes $|V|$ and edges $|E|$. Because the graph contains only walkable tiles (not the full padded grid), the model cost grows with the actual playable area rather than the bounding box size.

Loss function. A Huber loss[23] is used:

$$\mathcal{L}(\hat{y}, y) = \text{Huber}(\hat{y}, y; \delta),$$

with $\delta = 1.0$ in the implementation. Compared to plain squared error, Huber loss is less sensitive to large outliers (very large label values), while still behaving like MSE for small errors.



Here, $a_{j \to i}$ is the directional edge feature for the edge from neighbor j to node i. The symbol ϕ denotes the message function used during neighbor aggregation, i.e., it maps the neighbor embedding h_j together with the edge attribute $a_{j \to i}$ to a contribution vector that is summed into m_i.

Figure 2.4: Detailed simplified view of one GINEConv layer.

2.6 Integrating the Learned Heuristic into A*

After training, the GNN model is used as a drop-in heuristic function in A*. For a state s, the network predicts a scalar value $\hat{y}(s)$, which is interpreted as an estimate of the remaining number of pushes. The learned heuristic is therefore defined as

$$h_{\text{gnn}}(s) = \hat{y}(s),$$

and it is queried whenever A* needs to compute $f(s) = g(s) + h(s)$ for ordering the open list[5].

Deadlock handling. Before evaluating the neural model, the heuristic wrapper optionally applies deadlock checks. If a state is detected as deadlocked, the heuristic returns an effectively infinite value and the state is pruned by the search. This keeps the GNN from wasting computation on states that are known to be unsolvable and makes the behavior consistent with the classical search setup.

2.6.1 Caching and efficient graph reuse

A naive integration would rebuild a full graph object for every state, including node lists, edges, and all node features. This would be unnecessarily expensive, because for a fixed level most of the graph structure does not change between states. In this thesis, the learned heuristic is implemented as a wrapper that reuses static components and updates only the parts that actually change.

Caching by Zobrist hash. The heuristic value is cached per state using Zobrist hashing[12]. For each queried state, a hash key is computed and used to look up the heuristic value in a dictionary cache. If the state has been evaluated before, the cached value is returned immediately without any neural inference. This is important in A*, where the same configuration can be generated multiple times via different push orders.

Static graph structure per level. For each level, a static graph structure is built once and reused across all states of that level. The cached structure includes the mapping from grid cell indices to node indices, the edge list for the 4-neighborhood connectivity, edge attributes (`edge_attr`) encoding the direction as a one-hot vector, and static node features derived from the level layout (goals and local wall-adjacency indicators).

A lightweight *board signature* consisting of width, height, and the board mask is used to detect when the level changes. If the signature changes, the static graph cache is rebuilt.

Fast update of dynamic features. Only two node features change between states: whether a tile currently contains a box and whether it contains the player. Instead of allocating a new feature matrix for each state, a pre-allocated feature buffer is reused. For every call, the dynamic channels are cleared and then filled by iterating over the current box bitmask and the player position. This reduces per-call overhead to a few tensor writes, while all static channels (goals and wall adjacency) stay unchanged.

2.6.2 Attempted admissibility via baseline mixing

The learned model is not guaranteed to be admissible: for some states it may overestimate the true remaining number of pushes. Since admissibility is an important property for A* when optimality guarantees are desired, an additional safety mechanism was explored in this thesis to enforce a conservative heuristic value.

The idea is to combine the learned estimate with the classical Hungarian baseline by taking the minimum:

$$h_{\text{mix}}(s) = \min\big(h_{\text{gnn}}(s),\, h_{\text{hung}}(s)\big).$$

This construction is admissibility-preserving under the assumption that h_{hung} is admissible. By definition, admissibility means that $h_{\text{hung}}(s) \le h^*(s)$ for every state s, where $h^*(s)$ is the true optimal remaining cost to a goal state [5]. Therefore,

$$h_{\text{mix}}(s) = \min\big(h_{\text{gnn}}(s),\, h_{\text{hung}}(s)\big) \le h_{\text{hung}}(s) \le h^*(s),$$

so h_{mix} is also admissible.

This mode is used to evaluate whether an admissible fallback can provide a robust integration path for learned heuristics in Sokoban search.

2.7 Transferability and Block/Packs Clustering

Sokoban level sets are highly heterogeneous: two collections of levels can differ strongly in board topology, corridor structure, and box–goal layout. In practice, this means that a learned heuristic may perform well on one subset of levels but generalize poorly to another. For this reason, it is useful to study *transferability*, i.e., how well a model trained on one block of levels can be applied to a different block.

In this thesis, transferability is defined as follows: a model is trained on a training block A and evaluated on a different block B. Beyond evaluation, clustering can also be used as a data organization method. If blocks with similar structure or similar transfer behavior can be grouped, training and evaluation can be made more systematic: models can be trained on representative groups, and data collection can be balanced across qualitatively different level types. Two approaches are explored in this work: grouping blocks by static structural features of levels and an iterative split/merge scheme based on model transferability across blocks.

2.7.1 Grouping by Static Level Features

The first grouping approach clusters levels using *static* features that can be computed directly from the initial level state, without training a model. This is useful because Sokoban datasets are often heterogeneous: different level subsets may have very different geometry (open rooms vs. long corridors), different amounts of free space, and different box–goal layouts. If all levels are treated as one pool, it becomes harder to reason about generalization and transfer. Static grouping is therefore used as a simple way to organize large level collections into more homogeneous blocks before running more expensive model-based analyses.

Static features range from very simple properties, such as the board size, to more structural descriptors of the walkable-floor graph, and finally to distance-based proxies that relate boxes to goals. A complete list of all features used in the implementation is given in Table 2.2.

Clustering procedure. The grouping pipeline reads level identifiers from input files, computes a static feature vector for each level, and clusters the resulting vectors using KMeans[24]. A practical issue is that KMeans requires the number of clusters K to be chosen in advance[24]. If K is too small, qualitatively different level types are merged into the same group; if K is too large, many tiny clusters are created and the grouping becomes unstable.

To select K automatically, several candidate values $K \in [k_{\min}, k_{\max}]$ are evaluated using the silhouette score[25]. For a level represented by a feature vector x, let $a(x)$ be the average distance from x to other points in the same cluster, and let $b(x)$ be the smallest average distance from x to any other

Feature	Explanation
width	Grid width of the level.
height	Grid height of the level.
board_cells	Number of tiles that belong to the level area.
floor_ratio	Fraction of walkable tiles relative to the full grid area.
wall_ratio	Fraction of wall tiles inside the level area.
n_boxes	Number of boxes.
free_per_box	Free walkable tiles per box.
mean_floor_degree	Mean degree of nodes in the walkable-floor graph.
dead_end_ratio	Fraction of floor tiles with degree ≤ 1 (dead ends).
tunnel_ratio	Fraction of floor tiles that look like straight corridors (degree 2 with neighbors aligned horizontally or vertically).
articulation_ratio	Fraction of floor tiles that are articulation points of the floor graph. An articulation point is a tile whose removal would disconnect walkable space.
corner_ratio	Fraction of floor tiles that are corners, i.e., tiles adjacent to two walls/outside in an L-shape.
box_goal_min_mean	For each box, the Manhattan distance to its nearest goal is computed. The feature is the mean of these nearest distances.
box_goal_min_max	Maximum over boxes of the nearest-goal Manhattan distance.
box_goal_lb_approx_per_box	A cheap lower-bound proxy: sum of nearest-goal distances over all boxes, divided by number of boxes.
reachable_ratio_from_player	Fraction of floor tiles reachable by the player in the initial state without pushing boxes.
initial_pushes	Number of valid push actions available from the initial state.

Table 2.2: Static features used for grouping levels in the implementation.

cluster. The silhouette value for x is defined as

$$\mathrm{sil}(x) = \frac{b(x) - a(x)}{\max(a(x), b(x))},$$

which lies in the range $[-1, 1]$. Values close to 1 indicate that x is much closer to its own cluster than to neighboring clusters, while values near 0 indicate overlapping clusters. The silhouette score for a given K is the average of $\mathrm{sil}(x)$ over all sampled levels, and the value of K that maximizes this score is chosen.

After KMeans clustering, a post-processing step merges clusters smaller than a chosen minimum size into the nearest larger cluster in feature space. This prevents creating blocks that are too small to be useful for training or evaluation.

2.7.2 Transferability-Driven Grouping

The second approach groups levels based on *model behavior* rather than on hand-chosen static features. The motivation is simple: even if two level sets look similar by basic geometry, a learned heuristic may still generalize differently across them. For this reason, grouping is based on cross-evaluation:

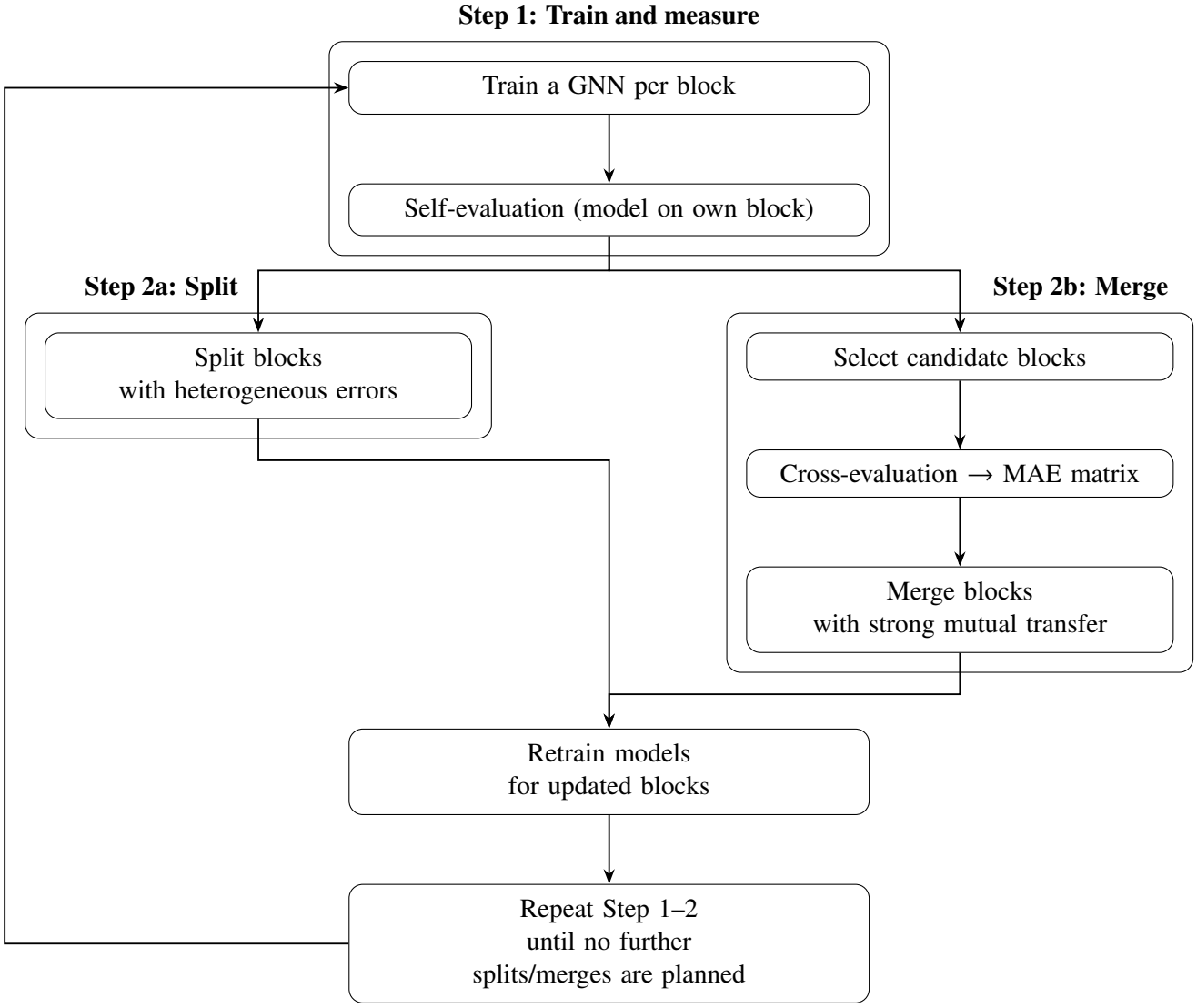


Figure 2.5: Automated pipeline for organizing heterogeneous level blocks based on transferability.

models are trained on one block and then evaluated on other blocks.

The process follows an iterative idea: *(1) train a model per block, (2) measure how it transfers, (3) split blocks that are internally heterogeneous, and (4) merge blocks that transfer well into each other*. This yields blocks that are closer to being "behaviorally consistent". Figure 2.5 summarizes the automated split/merge pipeline used to organize heterogeneous level blocks based on transferability.

Evaluating transferability. The main metric used for cross-evaluation is the mean absolute error (MAE) between predicted and true remaining pushes. MAE is easy to interpret because it has the same units as the label (y measured in pushes). In addition to MAE, the evaluation output can include MSE/RMSE and R^2 for completeness, but MAE is the primary signal used for decisions.

A single evaluation run produces overall metrics for the pair (A, B) and per-level aggregated metrics,

which are used for splitting (see below).

Cross-evaluation matrix. By evaluating many pairs (A, B), a cross-evaluation matrix is built where each entry contains the MAE of a model trained on A when applied to B. In large datasets, evaluating all pairs would be expensive, so the process is restricted to a set of candidate pairs selected automatically. Instead of evaluating all (A, B) combinations, each block is first summarized by a small vector of cheap statistics computed from a limited sample of labeled records. These statistics include, for example, typical board sizes, typical numbers of boxes/goals, and basic statistics of the labels (mean and variance of remaining pushes). Blocks that are close in this coarse summary space are treated as likely candidates for good transfer and are selected as cross-evaluation pairs. This step reduces compute substantially while keeping the selection automatic.

Splitting heterogeneous blocks. A block can still contain levels of very different difficulty or structure. To detect this, a model trained on a block is first evaluated on the *same* block , and MAE is computed *per level*. Levels are then sorted by their per-level MAE. If the sorted list shows a strong gap in error, this indicates that the block contains at least two sub-populations (e.g., levels that are consistently easy for the model and levels that are consistently hard). The block is then split at the strongest statistically meaningful gap. This procedure is applied recursively to obtain multiple segments and enforces a minimum segment size to avoid tiny splits. Thresholds for what counts as a "strong" gap are calibrated automatically from the overall distribution of gaps observed across blocks.

Merging blocks based on transferability. After cross-evaluation results are available, blocks that transfer well into each other are merged. Transferring from A to B is considered good if the MAE on B is not much worse than the MAE that the same model achieves on its own training distribution. For each pair (A, B), a symmetric merge score is computed by combining the two directions of transfer and taking the worse normalized ratio. Lower scores indicate better mutual transfer.
To turn these pairwise scores into groups, blocks are represented as nodes in an undirected graph. Edges are added only for strong pairs (keeping only the best-scoring fraction globally), then the edge scores are converted into affinities and a simple community detection method is used to find groups. The result is a set of merged block groups that reflect mutual transferability rather than only static similarity.

3 Results

3.1 Experimental Protocol

This chapter reports the empirical results of using a learned GNN heuristic in Sokoban search. Unless stated otherwise, all comparisons follow the same evaluation protocol and differ only in the heuristic function, the dataset split, the trained model, and the resource budget used for the run.

3.1.1 Compute environment

All results are obtained with the same A* implementation and the same push-based successor generation described in Chapter 2. Each search run is executed on a single level and terminates either when a goal state is found or when the run budget is exhausted. In the second case the level is considered unsolved. The budgets are expressed in two independent limits: a wall-clock `time_limit` and a `node_limit` (maximum number of expanded states). The concrete values of `time_limit` and `node_limit` differ across experiments and are therefore reported explicitly in the corresponding subsections.

To collect the results in a reasonable amount of time, all calculations were performed on Hydra, a cluster used by the different research groups at Technical University of Berlin. Evaluation is CPU-only (no GPU acceleration was used) to reflect the intended deployment scenario of a search heuristic that is queried frequently during A* expansion.

Each used cluster node is equipped with:

- **CPU:** Intel® Xeon® Platinum 8480CL
  - 2 sockets, 56 cores per socket, 2 threads per core, 112 physical cores (224 logical CPUs)
- **Memory:** $\approx 2{,}113{,}466{,}104$ kB RAM
- Ubuntu 22.04.1 LTS
- Linux kernel 5.15.0-101-generic

The node-level hardware capacity exceeds the resources allocated to a single evaluation job. For all runtime comparisons, each job was executed with:

- `cpus-per-task` = 1
- `mem` = 16 GB

This distinction is important because runtime depends on the effective per-job allocation rather than the full node capacity.

3.1.2 Datasets

Two level sources are used throughout this thesis: a large synthetic benchmark (Boxoban)[26] and a heterogeneous community collection aggregated by LetsLogic[2]. The goal of using both is to evaluate learned heuristics under two complementary settings: a very large, uniform distribution that enables large-scale supervised learning, and a diverse real-world distribution that stresses transferability across heterogeneous level styles.

Boxoban. Boxoban is a large collection of procedurally generated Sokoban levels released by DeepMind[26]. 503,000 levels were fetched with a fixed board size of 10×10 and exactly 4 boxes per level. The main advantage of Boxoban for this work is its scale: the large number of distinct instances enables training on a broad variety of configurations while keeping the underlying format consistent. This reduces the risk that a learned heuristic overfits to a small set of handcrafted levels and provides a strong basis for systematic hyperparameter comparisons under controlled conditions.

LetsLogic (community level packs). To evaluate generalization across diverse level distributions, additional levels were collected from LetsLogic[2], a community site that aggregates existing Sokoban level packs. In total, 53,799 levels from 697 different packs were extracted. This collection is substantially more heterogeneous than Boxoban: levels differ widely in size, number of boxes, and structural style. In the raw form, some levels are extremely large and are not solvable within a reasonable computational budget. In particular, the observed ranges were:

width: $\min = 3$, $\max = 200$; height: $\min = 3$, $\max = 200$; #boxes: $\min = 1$, $\max = 19{,}208$.

To obtain a practically solvable benchmark subset, levels were filtered to a maximum board size of 50×50 and at most 50 boxes, yielding 44,166 levels. From this subset, only those levels were retained that the external *Festival* solver could solve within a time limit of 1 hour per level. This final community levels dataset consists of 41,140 solvable levels and is used for the transferability and clustering experiments reported later in this chapter.

3.2 Speed vs. Mixed Heuristic Modes

This section compares the two inference modes implemented for the learned heuristic: `speed`, which directly uses the GNN prediction $h_{\text{gnn}}(s)$, and `mixed`, which uses the mixed heuristic $h_{\text{mix}}(s) = \min(h_{\text{gnn}}(s), h_{\text{hung}}(s))$ from Section 2.6.2. The purpose of this comparison is to decide which mode is used for the remaining experiments in this chapter.

Evaluation setup. The comparison is performed on two domains:

- **Boxoban:** 10,000 test levels from the Boxoban split.
- **Static cluster:** the test split of one of the static-feature clusters (described in Section 2.7.1), consisting of 525 levels.

Both domains are evaluated under the same search budget:

$$\texttt{time\_limit} = 180\,\text{s}, \qquad \texttt{node\_limit} = 2{,}000{,}000,$$

Search performance. On the Boxoban subset, `speed` solved all 10,000 instances and expanded 2,490,848 nodes in total, with a total runtime of 2,535.60 s. In contrast, `mixed` expanded 42,694,389 nodes and required 30,426.46 s total runtime, while solving 9,997 instances.

The same effect is visible on the static cluster block. Here, `speed` achieved a success rate of 92.76% (487/525) with 16,883,460 expanded nodes and 10,967.20 s runtime, while `mixed` achieved 90.29% (474/525) with 19,847,598 expanded nodes and 15,170.94 s runtime.

Table 3.1 summarizes the observed search outcomes. Figure 3.1 visualizes the same comparison in terms of total runtime.

Domain	Mode	Solved	Solved rate	Total nodes	Total time (s)
Boxoban	`speed`	10,000/10,000	1.0000	2,490,848	2,535.60
Boxoban	`mixed`	9,997/10,000	0.9997	42,694,389	30,426.46
`cluster`	`speed`	487/525	0.9276	16,883,460	10,967.20
`cluster`	`mixed`	474/525	0.9029	19,847,598	15,170.94

Table 3.1: Comparison of `speed` and `mixed` under the same A* budget.

How often does `mixed` fall back to Hungarian? To quantify how frequently the minimum operation selects the Hungarian baseline, both heuristics were evaluated on the states expanded during search and compared whenever both values were available. On Boxoban, in `speed` runs, Hungarian was smaller than the GNN estimate in 94.18% of the comparable states (5,108,642 out of 5,424,437). In `mixed` runs, this rate increased to 98.88% (60,171,890 out of 60,855,358).

Domain	Mode	Hungarian < GNN	GNN < Hungarian	GNN = Hungarian
Boxoban	`speed`	5,108,642	304,464	11,331
Boxoban	`mixed`	60,171,890	664,886	18,582
`cluster`	`speed`	17,760,390	546,416	378
`cluster`	`mixed`	26,311,303	411,317	522

Table 3.2: Frequency of comparisons between $h_{\text{hung}}(s)$ and $h_{\text{gnn}}(s)$ on states expanded during search.

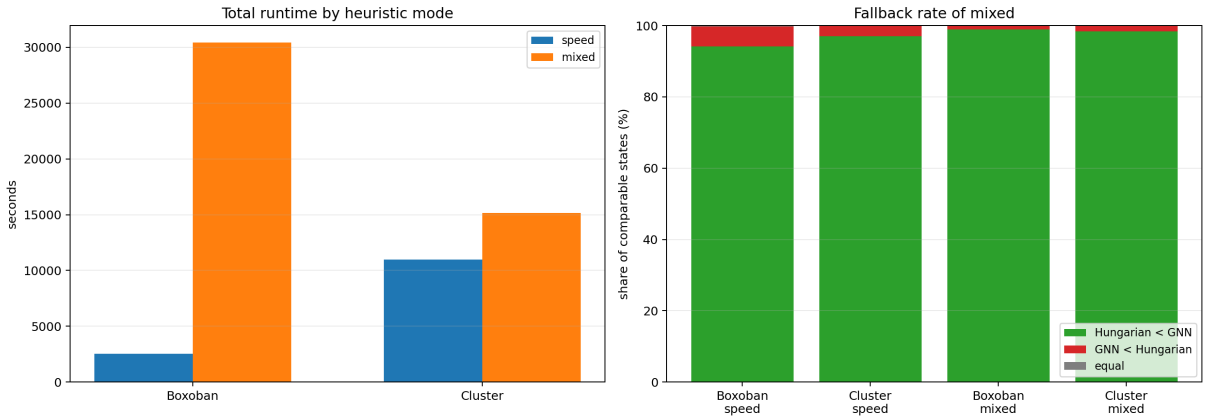


Figure 3.1: Search cost comparison and composition of heuristic comparisons on expanded states.

For the static cluster, the same pattern holds: Hungarian was smaller than the GNN estimate in 97.01% of comparable states in `speed` runs (17,760,390 out of 18,307,184), and in 98.46% of comparable states in `mixed` runs (26,311,303 out of 26,723,142).

Table 3.2 summarizes the results. Figure 3.1 additionally reports the composition of heuristic comparisons on expanded states, illustrating how frequently the minimum operation selects the Hungarian value.

Summary and choice of evaluation mode. Across both domains, `mixed` selects the Hungarian baseline on the vast majority of states (94–99% in the measured runs), which makes its behavior close to Hungarian while still incurring the overhead of evaluating the GNN. In addition, because taking the minimum often decreases the heuristic value compared to `speed`, the search tends to expand more nodes.

Based on these results, all remaining experiments in this thesis use the `speed` mode. This choice does *not* enforce heuristic admissibility: the learned heuristic is treated as an informative guidance signal rather than a guaranteed lower bound. Consequently, A* with `speed` mode is not expected to be push-optimal in general, and the evaluation in the following sections does not only report search effort (nodes and runtime), but also measures solution quality by tracking the returned solution length.

3.3 Hyperparameter Study

This section compares several GNN configurations on Boxoban and LetsLogic levels. The compared models differ primarily in hidden size (`hidden`) and number of GINEConv layers (`layers`); other training details are kept fixed. Besides prediction accuracy, the main criterion for a heuristic model is its impact on A* search, i.e., solved rate and search cost (expanded nodes and runtime).

3.3.1 Boxoban

Evaluation setup. All configurations are evaluated on a fixed set of 1,000 Boxoban levels. For each configuration, solved rate, total expanded nodes, total runtime, average time per expanded node, and node overhead with respect to the Hungarian baseline were reported.

Search results. Table 3.3 lists all tested (`hidden`, `layers`) pairs and summarizes the A* outcomes for all tested configurations. The configuration identifier follows the naming `<hidden>_<layers>`.

Config	Solved	Total nodes	Total time (s)	Time/node (ms)	Overhead vs. Hun
32_2	1000/1000	1,239,442	1,626.44	1.3122	2.15
32_3	1000/1000	1,021,184	1,633.00	1.5991	2.62
32_4	1000/1000	1,012,967	1,848.45	1.8248	2.99
32_5	1000/1000	1,074,561	2,237.74	2.0825	3.41
32_6	1000/1000	1,494,753	3,244.86	2.1708	3.56
64_2	1000/1000	1,118,915	1,656.21	1.4802	2.43
64_3	1000/1000	770,957	1,453.54	1.8854	3.09
64_4	1000/1000	595,429	1,376.59	2.3119	3.79
64_5	1000/1000	505,195	1,361.19	2.6944	4.42
64_6	1000/1000	618,269	1,787.49	2.8911	4.74
128_4	1000/1000	427,939	1,333.74	3.1166	5.11
128_5	1000/1000	356,944	1,396.40	3.9121	6.41
128_6	1000/1000	330,122	1,513.41	4.5844	7.51
256_4	1000/1000	348,142	2,451.29	7.0411	11.54
256_5	1000/1000	318,618	3,685.36	11.5667	18.95
256_6	1000/1000	275,454	3,186.22	11.5671	18.96
512_4	1000/1000	296,916	13,719.61	46.2075	75.72
512_5	999/1000	272,710	16,110.39	59.0752	96.81
512_6	999/1000	268,987	18,926.71	70.3629	115.31

Table 3.3: Boxoban hyperparameter sweep: A* results for all tested configurations (1,000 levels, `speed` mode).

Across most settings up to `hidden`=256, all 1,000 instances are solved. For `hidden`=512, the solved rate slightly drops to 999/1000 for 5 and 6 layers, while runtime increases substantially.

A clear trade-off is observed between search effort (expanded nodes) and heuristic evaluation cost. Increasing model capacity generally reduces the total number of expanded nodes. For example, moving from `32_2` to `128_4` reduces total expansions from 1,239,442 to 427,939. However, this comes with a higher time per node (1.31 ms vs. 3.12 ms). Beyond `hidden`=256, per-node cost grows rapidly (7–12 ms), and for `hidden`=512 it becomes dominant (46–70 ms), leading to very large overall runtimes despite fewer expansions. Figure 3.2 visualizes the search efforts of each model and compares them with the Hungarian baseline.

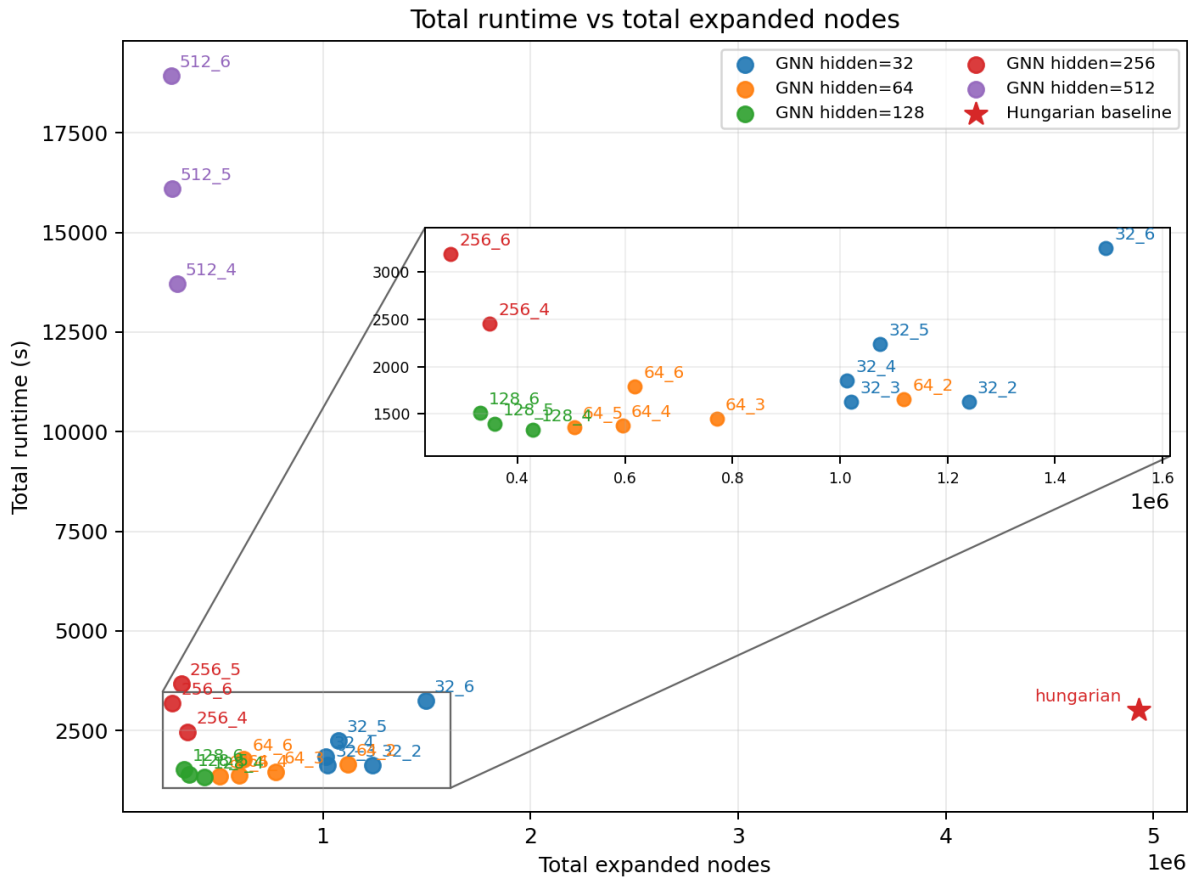


Figure 3.2: Trade-off between search effort and runtime for different model sizes.

Among all tested configurations, `128_4` achieves the lowest total runtime (1,333.74 s) while still solving all instances. Although deeper or wider models can further reduce node expansions (e.g., `256_6` expands 275,454 nodes), their higher inference cost results in longer total runtimes. Based on this sweep, the configuration `hidden`=128 and `layers`=4 is selected as the default model for all subsequent experiments in this thesis.

3.3.2 Community Levels

Evaluation setup. To complement the Boxoban sweep, the smaller set of configurations was evaluated on a heterogeneous community levels subset using a single global model setting. Each level was evaluated with a runtime budget of 180 s. In this setting, the time limit is the dominant constraint: many unsuccessful runs terminate due to the time budget before reaching a node-based bound.

Search results. Table 3.4 reports the outcomes for four representative configurations. As model capacity increases, the number of expanded nodes decreases strongly (e.g., from 29.33M total expansions for `128_4` down to 2.90M for `512_6`). However, these larger models do not solve more instances

Config	Solved	Solved rate	Total nodes	Total time (s)	Time/node (ms)
128_4	560/1000	0.560	29,329,123	82,815.50	2.8237
256_5	536/1000	0.536	6,679,902	86,648.37	12.9715
256_6	530/1000	0.530	6,059,840	88,476.06	14.6004
512_6	512/1000	0.512	2,900,705	97,199.37	33.5089

Table 3.4: Community levels hyperparameter comparison.

within the fixed time budget. Instead, solved rate slightly decreases (from 56.0% for `128_4` to 51.2% for `512_6`), while the average time per expanded node increases substantially (from 2.82 ms to 33.51 ms).

A notable effect is that "fewer expanded nodes" does not necessarily indicate a more efficient search in this time-limited regime: when inference becomes expensive, the solver expands fewer nodes simply because it spends more time per node and reaches the wall-clock budget earlier. This is reflected by the unsolved runs, whose median runtime is essentially at the limit for all configurations.

Overall, `128_4` achieves both the highest solved rate and the lowest total runtime among the compared settings on community levels. For consistency with the Boxoban sweep, and because it provides the best runtime–accuracy trade-off across both domains, the configuration `128_4` is used as the default model in all subsequent experiments.

3.4 Boxoban: GNN vs. Hungarian Baseline

This section presents results of the best performing model on Boxoban.

Dataset and label generation. 503,000 Boxoban levels were fetched, all with a fixed board size of 10×10 and four boxes. All Boxoban levels were solved with the external *Festival* solver[14, 15] using a time budget of one hour per level, resulting in complete coverage of the dataset. From the solved trajectories, a total of 31,129,541 labeled states were extracted.

For supervised training and evaluation, a subset of levels was selected and split into train/validation/test sets. Table 3.5 summarizes the resulting splits and label counts.

Split	#levels	#labeled states
Train	453,000	27,631,822
Validation	40,000	2,451,007
Test	10,000	–
Total	503000	31,129,541

Table 3.5: Boxoban dataset splits and total number of extracted labeled states.

Model configuration and training outcome. The evaluated model uses the default architecture selected in the hyperparameter study (Section 3.3). Training was run for 200 epochs. The best

Parameter	Value
Architecture	edge-aware GIN (GINEConv) + global mean pooling
Hidden size (`hidden`)	128
Number of layers (`layers`)	4
Dropout	0.1
Loss	Huber ($\delta = 1.0$)
Training epochs	200 (best checkpoint at epoch 143)
Best validation loss	1.3848

Table 3.6: Configuration of the Boxoban GNN heuristic model used in the evaluation.

checkpoint (used in all Boxoban evaluations in this section) achieved a validation loss of

$$\texttt{val\_loss} = 1.3848$$

at epoch 143. The final resumed training run reached epoch 200 with `val_loss`=1.3987. Table 3.6 summarizes the characteristics of the model.

Search performance on 10,000 Boxoban test levels. The GNN heuristic is compared against the Hungarian baseline using the same A* implementation and the same evaluation budget. Both heuristics solve all 10,000 test instances. However, the search effort differs substantially.

Over the full test set, the Hungarian baseline expands 49,004,858 nodes in total, whereas the GNN heuristic expands 2,490,848 nodes. This corresponds to a reduction factor of $19.67\times$ in expanded nodes. Runtime is also reduced: Hungarian requires 10,810.12 s total runtime, while the GNN heuristic requires 2,973.79 s, corresponding to a $3.64\times$ reduction in total wall-clock time.

Table 3.7 reports both aggregate totals and robust per-level statistics (median and interquartile range).

Heuristic	Solved	Total nodes	Median nodes [IQR]	Median time [IQR] (s)
Hungarian	10,000/10,000	49,004,858	1,264 [3,520]	0.2698 [0.7364]
GNN (`speed`)	10,000/10,000	2,490,848	57 [174]	0.1064 [0.2032]

Table 3.7: Boxoban test results (10,000 levels): Hungarian baseline vs. GNN heuristic in `speed` mode.

To complement the aggregate results in Table 3.7, Figure 3.3 shows a per-level comparison of search cost. Each point corresponds to one level, and the diagonal indicates equal cost for both heuristics. Most points lie below the diagonal in the node plot, indicating fewer expansions with the GNN on the majority of instances.

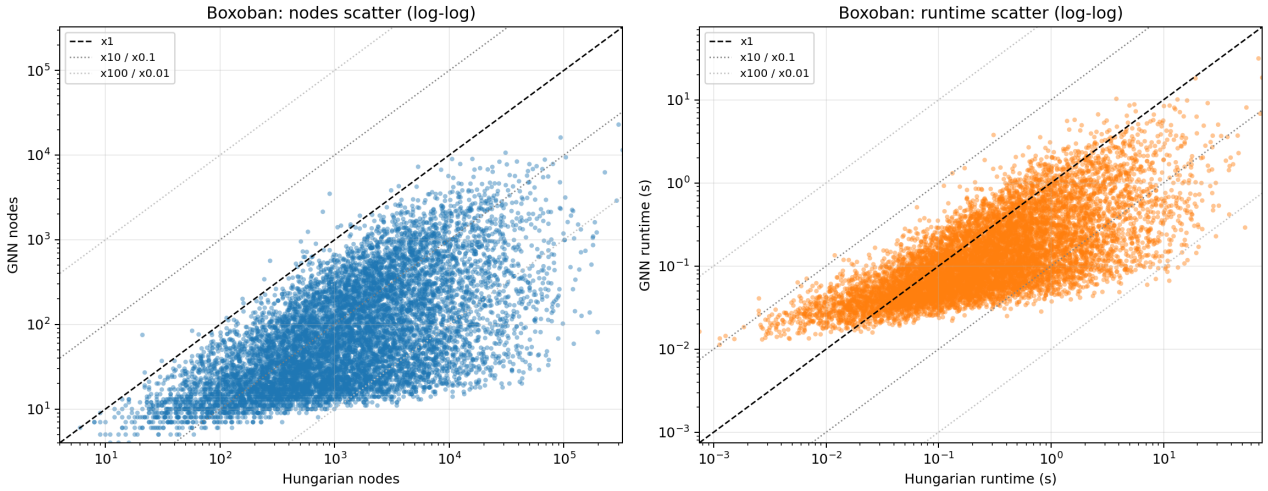


Figure 3.3: Per-level comparison on Boxoban test levels (log–log). Left: expanded nodes (GNN vs. Hungarian). Right: runtime (GNN vs. Hungarian). The dashed diagonal indicates parity; dotted lines show factors of 10 and 100.

To visualize runtime behavior across easy and hard instances, Figure 3.4 plots runtimes sorted independently per method. This view highlights that the GNN-guided search remains faster across most of the test set, including in the long-runtime tail.

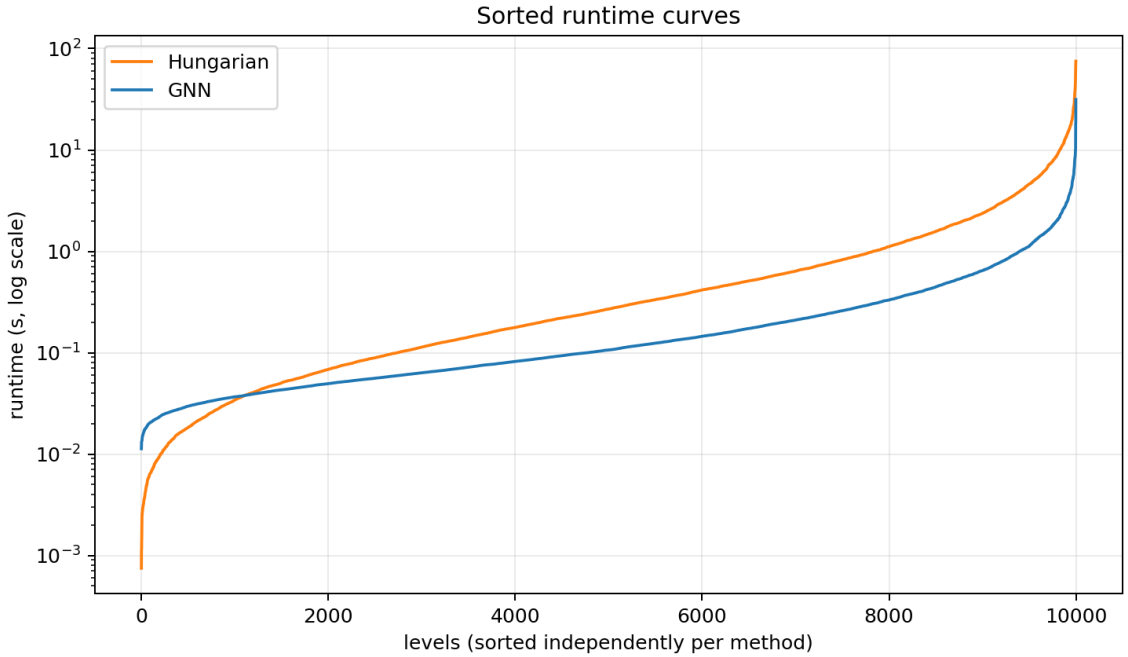


Figure 3.4: Sorted per-level runtime curves on the Boxoban test set (log scale). Levels are sorted independently per method, so the x-axis ranks instances by runtime for each curve rather than matching the same level across curves.

Solution length deviation. Comparing the solution length found by the GNN-guided search to the Hungarian-guided solution length on the same levels yields the following.

On 93.6% of instances, both methods return solutions of identical length. On the remaining 6.4% of instances, the GNN-guided search returns a longer solution, with a mean increase of 0.156 pushes across the full test set. The 95th percentile of the length increase is 2 pushes, and the maximum observed increase is 10 pushes.

Table 3.8 reports the distribution of the absolute solution-length deviation between solution lengths.

Absolute difference	0	1	2	3	4	5	6	7	8	9	10
Number of levels	9360	0	523	0	98	0	17	0	1	0	1

Table 3.8: Distribution of the absolute difference in solution length between the GNN-guided and Hungarian-guided searches on the Boxoban test set (10,000 levels).

Breakdown by difficulty. To provide a more detailed picture, results are grouped by the push-optimal solution length (Hungarian). Table 3.9 shows a consistent reduction in expanded nodes across buckets. For longer solutions, Hungarian expands very large search trees, while the GNN heuristic keeps expansions substantially smaller. At the same time, the rate of longer-than-optimal GNN solutions remains low across buckets.

Opt. length	**#levels**	**Hun**		**GNN**	
		Median nodes	**Median time (s)**	**Median nodes**	**Median time (s)**
≤ 10	1,326	128	0.0352	13	0.0398
11–20	6,791	1,162	0.2493	54	0.1013
21–30	1,739	7,288	1.5070	275	0.3844
31–40	137	28,338	5.8229	774	1.0434
41–60	7	14,602	2.3833	2,375	2.2893

Table 3.9: Per-bucket median search cost on Boxoban levels, grouped by optimal solution length.

3.5 Community levels: Global model vs. Hungarian

This section evaluates how a single global GNN heuristic generalizes to a heterogeneous collection of community Sokoban levels aggregated from LetsLogic[2].

3.5.1 Setup

Dataset and splits. The community levels dataset used in this thesis consists of 41,140 levels after the filtering and solvability criteria described in Section 3.1. The dataset is split into train/validation/test by level instances, using the following split sizes: 32,912 levels for training, and 4,114 levels each for validation and test (Table 3.10).

Split	#levels	#labeled states
Train	32,912	12,641,863
Validation	4,114	1,469,824
Test	4,114	–
Total	41,140	–

Table 3.10: Community levels dataset splits used for the global-model evaluation.

Model and heuristic mode. The evaluation in this section uses a single global model trained on the community levels training split. The architecture follows the default configuration selected in Section 3.3. Table 3.11 summarizes the characteristics of the model.

Parameter	Value
Architecture	edge-aware GIN (GINEConv) + global mean pooling
Hidden size (`hidden`)	128
Number of layers (`layers`)	4
Dropout	0.1
Loss	Huber ($\delta = 1.0$)
Training epochs	200 (best checkpoint at epoch 33)
Best validation loss	63.2405

Table 3.11: Configuration of the community levels global GNN heuristic model used in the evaluation.

Search budget and reference heuristic. All comparisons use the same A* implementation and the same push-based state space as in Chapter 2. Each level is solved with a wall-clock budget of `time_limit`=180 s and a node-expansion limit of `node_limit`=2,000,000.

The Hungarian heuristic serves as the classical baseline. When solution length differences are reported later in this section, they are computed on the subset of levels solved by both methods under the same budget.

3.5.2 Main results

In this benchmark, the time limit dominates: all unsuccessful runs terminate due to the wall-clock budget, and the node limit is never reached.

The GNN heuristic solves 1,475 out of 4,114 levels (35.85%), while Hungarian solves 1,243 out of 4,114 levels (30.21%). Because a large fraction of instances times out for both methods, the overall runtime distribution is heavily censored at 180 s (median runtime is at the time limit for both heuristics). Table 3.12 summarizes the overall outcomes.

Heuristic	Solved	Total nodes	Median nodes [IQR]	Median time [IQR] (s)
Hungarian	1,243/4,114	1,560,666,146	251,219 [554,410]	180.0001 [127.8637]
GNN (`speed`)	1,475/4,114	225,100,663	31,179.5 [73,204]	180.0012 [161.9477]

Table 3.12: Community levels test results.

Solved-instance cost. To isolate search cost on instances that are actually solved within the budget, Table 3.13 reports median node expansions and runtime restricted to solved runs only. On this subset, the GNN heuristic expands substantially fewer nodes (median 3,687 vs. 24,479), while runtimes are of the same order of magnitude (both with a wide IQR due to varying difficulty). Moreover, on the 1,196 instances solved by both methods, the GNN expands fewer nodes in 93.9% of cases.

Heuristic	Solved	Median nodes [IQR]	Median time [IQR] (s)	Total nodes
Hungarian	1,243	24,479 [177,883]	2.8710 [26.4981]	200,293,072
GNN (`speed`)	1,475	3,687 [26,112]	4.1502 [26.3776]	36,265,674

Table 3.13: Community levels test results restricted to solved runs only.

To complement the aggregate results, Figure 3.5 shows a per-level comparison of node expansions and runtime (log–log). In the node plot, most points lie below the diagonal, indicating that the GNN expands fewer states on the majority of comparable instances. Figure 3.6 visualizes the runtime distribution by plotting runtimes sorted independently per method. The flat tail at the top reflects the 180 s time limit for unsolved instances.

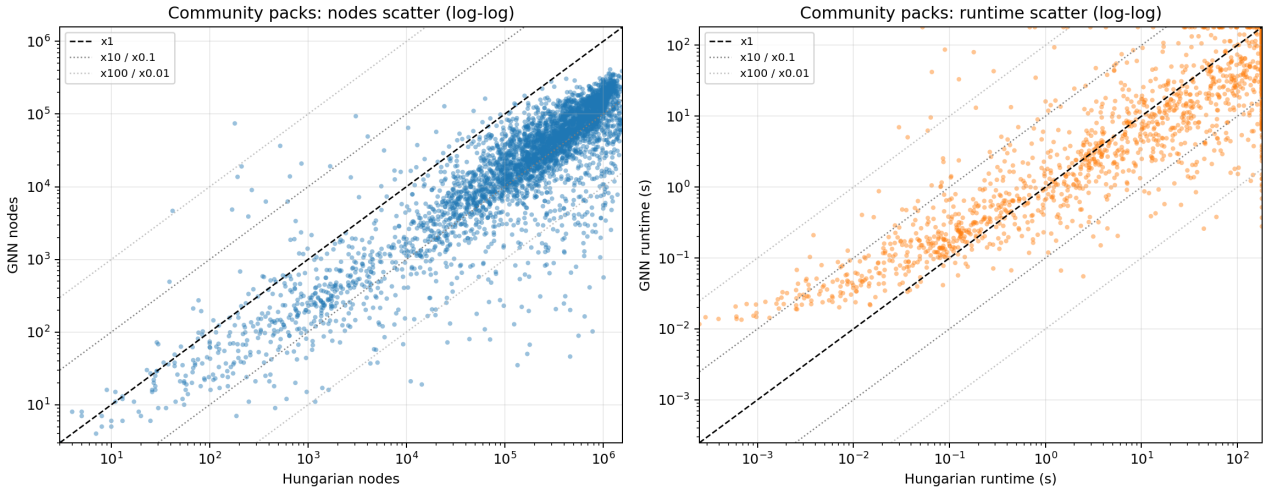


Figure 3.5: Per-level comparison on LetsLogic test levels (log–log). Left: expanded nodes (GNN vs. Hungarian). Right: runtime (GNN vs. Hungarian). The dashed diagonal indicates parity, dotted lines show factors of 10 and 100.

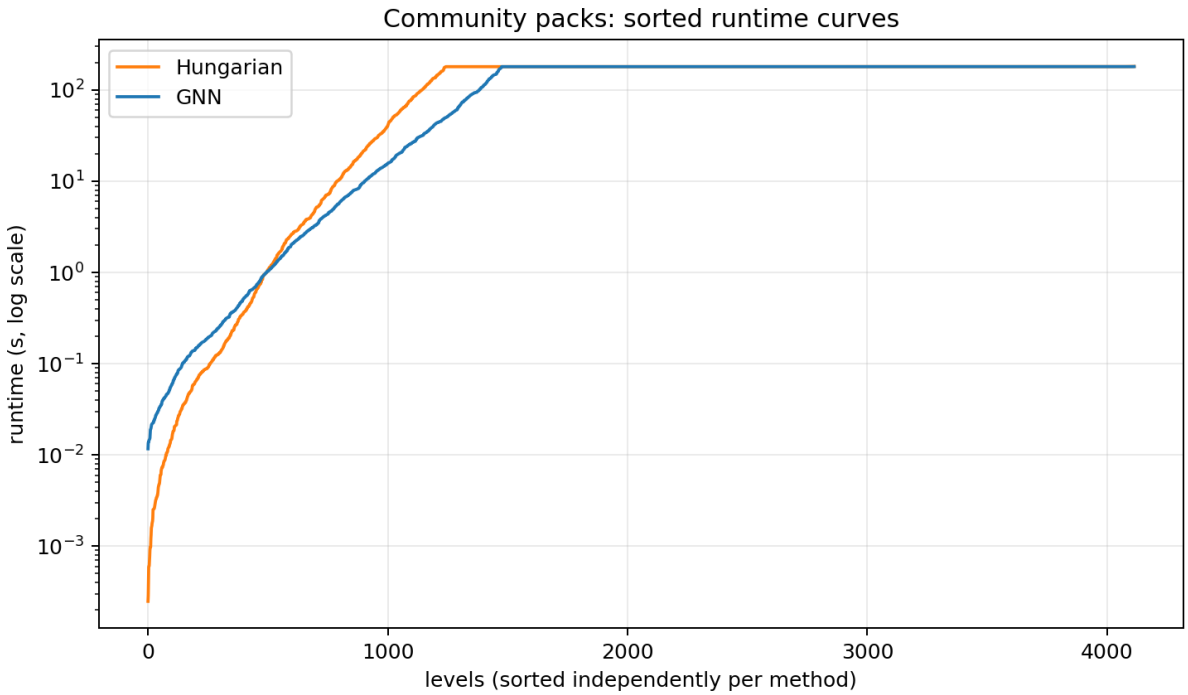


Figure 3.6: Sorted per-level runtime curves on LetsLogic test levels (log scale). Levels are sorted independently per method.

3.5.3 Solution length deviation

Because the GNN heuristic is used in `speed` mode, solutions are not guaranteed to be push-optimal. Solution-length deviations are therefore evaluated on the intersection of instances solved by both methods within the same budget (1,196 levels).

On 62.63% of the comparable instances, both methods return solutions of identical length. On the remaining 37.37%, the GNN-guided search returns a longer solution. Across the full comparable set, the mean increase is 1.36 pushes (mean relative increase 6.42%), the 95th percentile is 6 pushes, and the maximum observed increase is 18 pushes. Table 3.14 reports the distribution of the absolute solution-length deviation.

Absolute difference	0	2	4	6	8	10	12	14	16	18
Number of levels	749	252	107	47	20	8	7	4	1	1

Table 3.14: Distribution of the absolute difference in solution length between the GNN-guided and Hungarian-guided searches on LetsLogic test levels, restricted to instances solved by both methods.

3.6 Static-feature clusters

This section evaluates static-feature clustering as a way to organize heterogeneous community-created levels into more homogeneous blocks. The goal is to test whether training one model per cluster improves within-cluster performance and to analyze transfer across clusters.

3.6.1 Cluster summary

Clustering procedure. Levels were grouped by applying KMeans[24] to standardized static feature vectors (Section 2.7.1). The number of clusters was selected automatically via silhouette[25] score on a random sample, and very small clusters were merged into the nearest larger cluster in feature space.

In the community levels setting, silhouette selection chose $K = 13$ clusters before post-processing, and after merging small clusters the final partition contains $K = 12$ clusters. Cluster sizes range from 315 to 6,826 levels, with a median cluster size of 3,632 levels.

To make the static-feature clusters easier to interpret, Table 3.15 reports the cluster size, typical board size, number of boxes, and a short list of *distinctive feature traits* for each cluster.

The traits in the last column are computed automatically by comparing each cluster's median feature values to the global feature statistics: for a small set of highly separating features, a cluster is tagged as `high` or `low` depending on whether its median is substantially above or below the global median.[1]

3.6.2 Within-cluster evaluation

This subsection evaluates whether training one model per static-feature cluster improves performance on the corresponding cluster test split. For each group, three heuristics under the same budget (`time_limit`=300 s, `node_limit`=2,000,000) are compared: the Hungarian baseline, a single global model (*General GNN*), and a cluster-specific model trained on the same group (*Clustered GNN*).

Macro summary. Across the full static-cluster test set (4,114 levels), the overall solved rates are close: Hungarian solves 37.55%, the General GNN solves 37.38%, and the Clustered GNN solves 38.07% (weighted by the number of test levels per group). A macro-average over groups (unweighted) yields similar values (30.56%, 30.76%, 30.77%), reflecting that some small but difficult clusters have very low solved rates for all methods.

Solved rate per group. Figure 3.7 reports the solved rate for each group. Cluster-specific training can improve the solved rate for some clusters, while for others it is similar or slightly worse than the General GNN. Several clusters are extremely hard under the fixed 300 s budget (notably group_008 and group_010), where all methods solve only a negligible fraction.

[1] The meaning of the individual features is defined in Table 2.2.

Cluster	#levels	#labels	Median size	Med. boxes	Distinctive traits
00	6,826	2,192,185	10×9	8	low height, low width, low initial_pushes
01	5,248	619,935	9×8	4	low height, low width, low board_cells
02	5,104	1,938,950	15×14	17	high board_cells, low free_per_box
03	4,247	2,373,801	18×15	14	high box_goal_min_mean, high approx_per_box, high reachable_ratio
04	4,007	782,558	8×8	6	low width, low height, low board_cells, low box_goal_min_max
05	3,878	2,060,849	12×11	7	low initial_pushes, low width, low board_cells, high free_per_box
06	3,386	4,905,424	11×11	17	low box_goal_min_max, low box_goal_min_mean, low box_goal_lb_approx_per_box, low reachable_ratio_from_player
07	2,711	1,494,782	11×10	7	low box_goal_min_max, low board_cells, low width, high free_per_box
08	2,402	1,867,299	18×15.5	34	high height, high width, high board_cells, low free_per_box
09	1,553	599,541	18×16	14	high height, high width, high board_cells, high initial_pushes
10	1,463	948,421	19×17	30	high initial_pushes, high height, high board_cells, high box_goal_min_max
11	315	485,534	33×24	16	high box_goal_min_mean, high box_goal_lb_approx_per_box, high box_goal_min_max, high board_cells

Table 3.15: Static-feature cluster summary. The last column lists automatically generated high/low tags for a subset of highly separating features (definitions in Table 2.2).

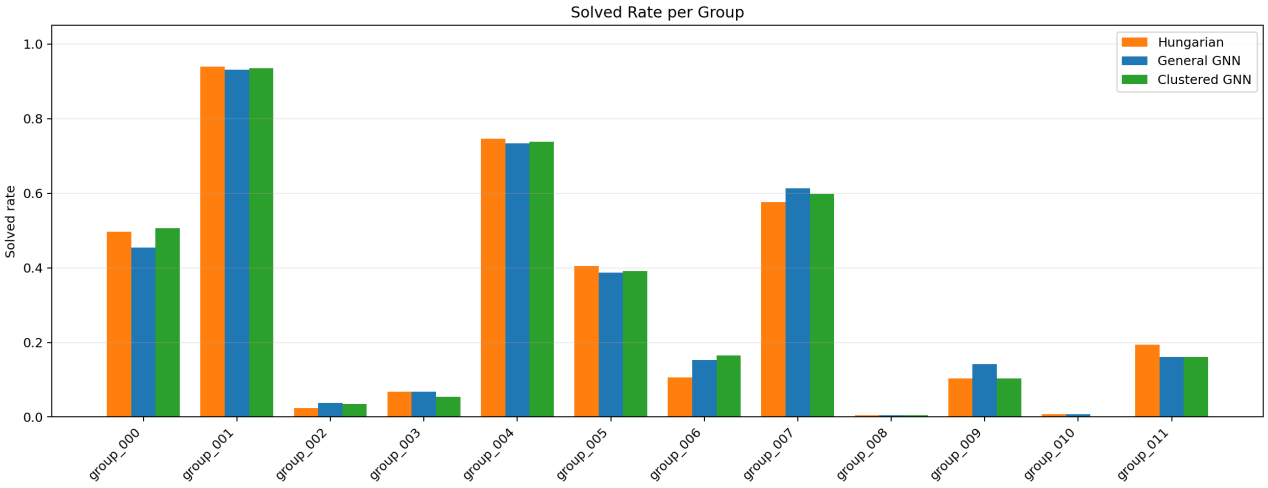


Figure 3.7: Solved rate per static-feature group on the community levels test split: Hungarian baseline vs. General GNN vs. Clustered GNN.

Search cost on comparable solved instances. Figure 3.8 reports runtime and expanded nodes restricted to levels that are solved by *all three* methods within the budget (1,378 out of 4,114 levels in total). On these comparable instances, both GNN variants reduce expansions substantially compared to Hungarian in most groups.

Aggregating over groups and weighting by the number of comparable instances per group, Hungarian expands on average 74,665 nodes, while the General GNN expands 33,032 nodes and the Clustered GNN expands 31,380 nodes. The corresponding weighted mean of within-group median runtimes is 16.48 s (Hungarian), 25.44 s (General GNN), and 22.88 s (Clustered GNN).

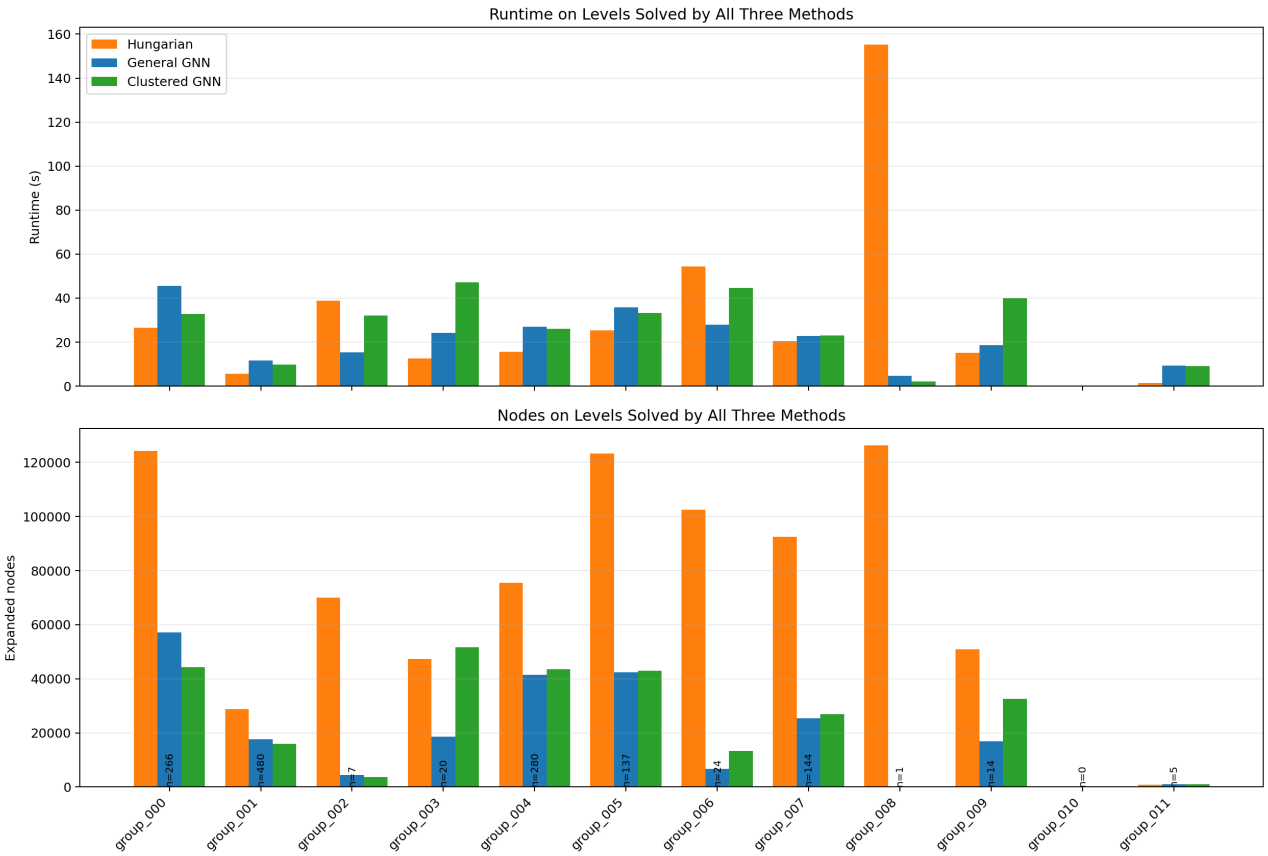


Figure 3.8: Runtime (top) and expanded nodes (bottom) on community test levels solved by all three methods, shown per static-feature group.

Delta solved rate and training data volume. Table 3.16 reports, for each group, the solved-rate difference between Clustered GNN and General GNN, together with the group size in the full dataset (proxy for the amount of training data available for the cluster-specific model). Positive values indicate that the cluster-specific model solves more instances than the General GNN on that group.

Group	Solved rate (General)	Δ solved (Clustered – General)
00	0.4539	+0.0527
01	0.9314	+0.0038
02	0.0373	-0.0020
03	0.0682	-0.0141
04	0.7332	+0.0050
05	0.3866	+0.0052
06	0.1534	+0.0118
07	0.6125	-0.0148
08	0.0042	0.0000
09	0.1419	-0.0387
10	0.0068	-0.0068
11	0.1613	0.0000

Table 3.16: Solved rate of the General GNN and the difference in solved rate between the Clustered GNN and the General GNN per group.

3.6.3 Cross-evaluation matrix

Within-cluster evaluation shows that cluster-specific training can improve performance on several groups. To understand whether this improvement comes from learning *cluster-specific* structure, a cross-evaluation is performed: a model trained on one source cluster is evaluated on the test split of another target cluster.

Figure 3.9 summarizes the cross-evaluation between cluster-specific models. The heatmap shows solved rate on each target cluster (rows) for each source-model cluster (columns). The diagonal corresponds to the matched setting (train and test on the same cluster), and is highlighted. Overall, solved rates vary strongly by target cluster, which is consistent with the heterogeneous difficulty observed earlier (Figure 3.7).

The diagonal is not uniformly optimal: for some targets the best-performing model comes from a different source cluster. The right panel of Figure 3.9 makes this more explicit by ranking, for each target row, where the matched model falls among all source models. For example, for several clusters the matched model is ranked first, whereas for others the matched model is noticeably lower-ranked, indicating that those target distributions are better covered by a model trained on a different static cluster.

Another visible pattern is that a small number of source models generalize comparatively well across multiple targets. This aligns with a trend that clusters that were trained on more labels (for example `group_006`) tend to transfer better. Quantitatively, the share of targets where a source cluster achieves the best solved rate increases with the number of training labels, and the average rank of a source model across targets tends to improve as label count grows.

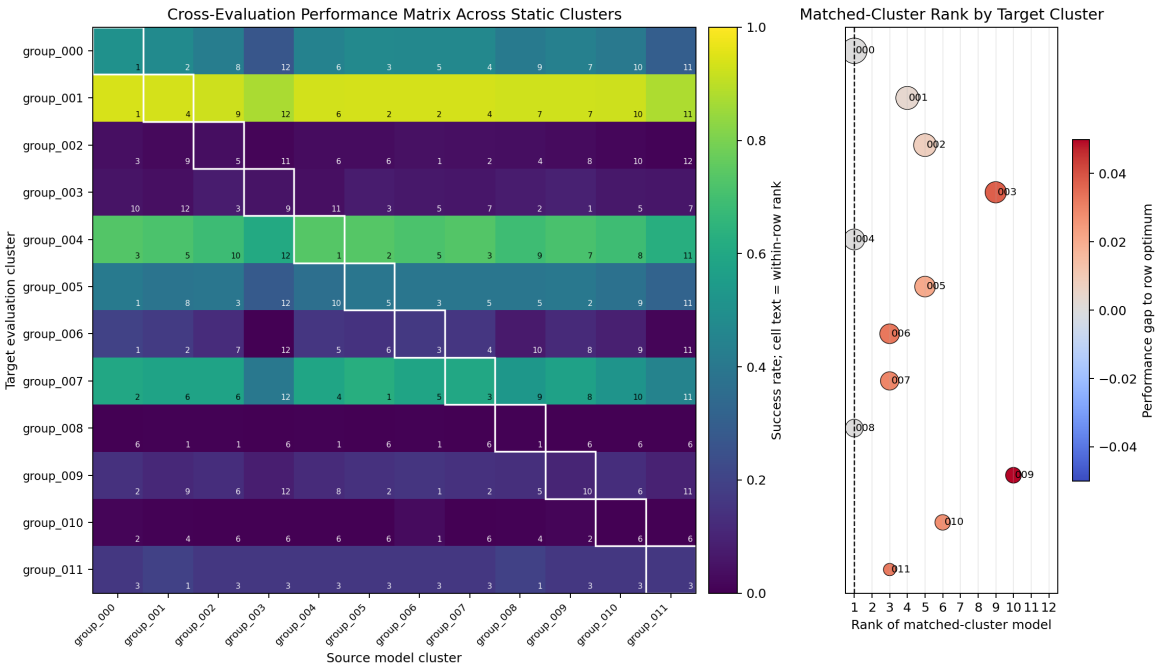


Figure 3.9: Cross-evaluation across static clusters. Left: solved-rate matrix for evaluating a model trained on source cluster (columns) on target cluster test splits (rows). Diagonal cells correspond to matched cluster models and are highlighted. Right: for each target cluster, the rank of the matched model within its row (1 = best in that row), with color indicating the solved-rate gap between the row-best model and the matched model.

3.7 Split–merge clustering

This section reports the same analyses as in Section 3.6, but for the iterative split–merge clustering pipeline introduced in Section 2.7. Only the key properties of the resulting partition are summarized here; evaluation and cross-evaluation follow the same protocol as before.

3.7.1 Cluster summary

The split–merge pipeline was iterated until the per-group MAE no longer improved and the number of clusters stopped decreasing. In total, 6 pipeline rounds were required, resulting in $K = 15$ clusters.

Table 3.17 summarizes the resulting groups. Compared to static-feature clustering (Section 3.6.1), the split–merge partition is more imbalanced: two large clusters (`group_000` and `group_001`) contain the majority of levels and labels, while several small clusters contain only a few hundred levels and yield very small test splits.

Across all groups, the split contains 41,140 total levels, 4,128 test levels, and 21,531,047 labeled training states.

Cluster	All levels	Test levels	Solved test levels	Train labels	Med. size	Med. boxes
00	12,412	1,242	166	7,856,283	13×14	13
01	12,906	1,292	597	5,624,502	10×11	8
02	1,361	137	1	1,671,782	17×20	23
03	4,700	470	77	2,761,421	11×13	12
04	1,162	117	3	666,539	13×15	13
05	348	36	1	222,292	13×15	18
06	317	33	1	220,569	13×15	22
07	67	8	0	42,890	15×17	16
08	4,883	489	439	859,981	8×9	4
09	363	37	0	262,567	16×18	23
10	281	29	26	67,739	9×9	6
11	146	16	0	110,625	13×15	16
12	728	74	73	45,898	7×8	3
13	322	33	2	238,174	16×19	22
14	1,144	115	7	879,785	15×17	22
TOTAL	41,140	4,128	1,393	21,531,047	–	–

Table 3.17: Split–merge cluster summary: group sizes, test sizes, number of test instances solved by all three methods (used for comparable-cost plots), and training label counts.

3.7.2 Within-cluster evaluation

As in Section 3.6.2, three heuristics under the same budget (`time_limit`=300 s, `node_limit`=2,000,000) are compared on the split–merge groups: Hungarian, the General GNN, and a group-specific Clustered GNN trained on the corresponding split–merge cluster.

Across the full split–merge test split (4,128 levels), the Clustered GNN achieves the highest overall solved rate: Hungarian solves 37.06% (1530/4128), the General GNN solves 37.14% (1533/4128), and the Clustered GNN solves 39.03% (1611/4128). Figure 3.10 shows solved rates per split–merge group. The clustered models improve solved rate in several large groups, while some very small groups show unstable behavior due to the small number of test instances (e.g., `group_007`, `group_011`).

Search cost on solved instances. Figure 3.11 reports runtime and expanded nodes restricted to levels that are solved by *all three* methods (group counts in Table 3.17). On these comparable instances, both GNN variants reduce node expansions compared to Hungarian in most large groups. Weighting by the number of comparable instances per group, Hungarian expands on average 16,729 nodes, while the General GNN expands 6,508 nodes and the Clustered GNN expands 5,351 nodes.

In runtime, Hungarian remains faster on this comparable subset (3.43 s weighted mean) than the learned heuristics (7.00 s for General GNN, 5.74 s for Clustered GNN), reflecting the additional per-node inference cost of the GNN heuristic.

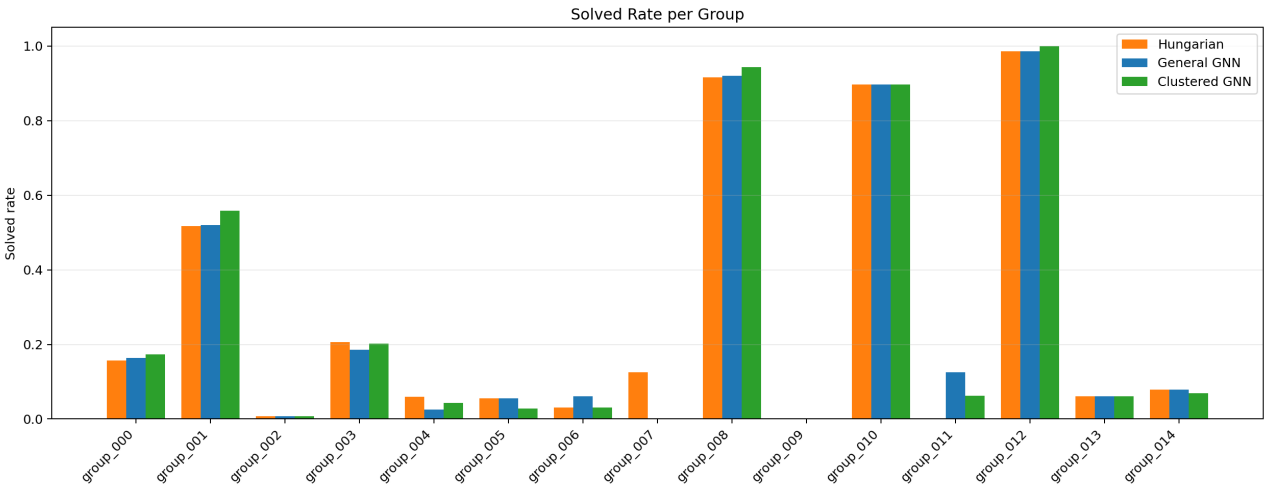


Figure 3.10: Solved rate per split–merge group on the community test split: Hungarian baseline vs. General GNN vs. Clustered GNN.

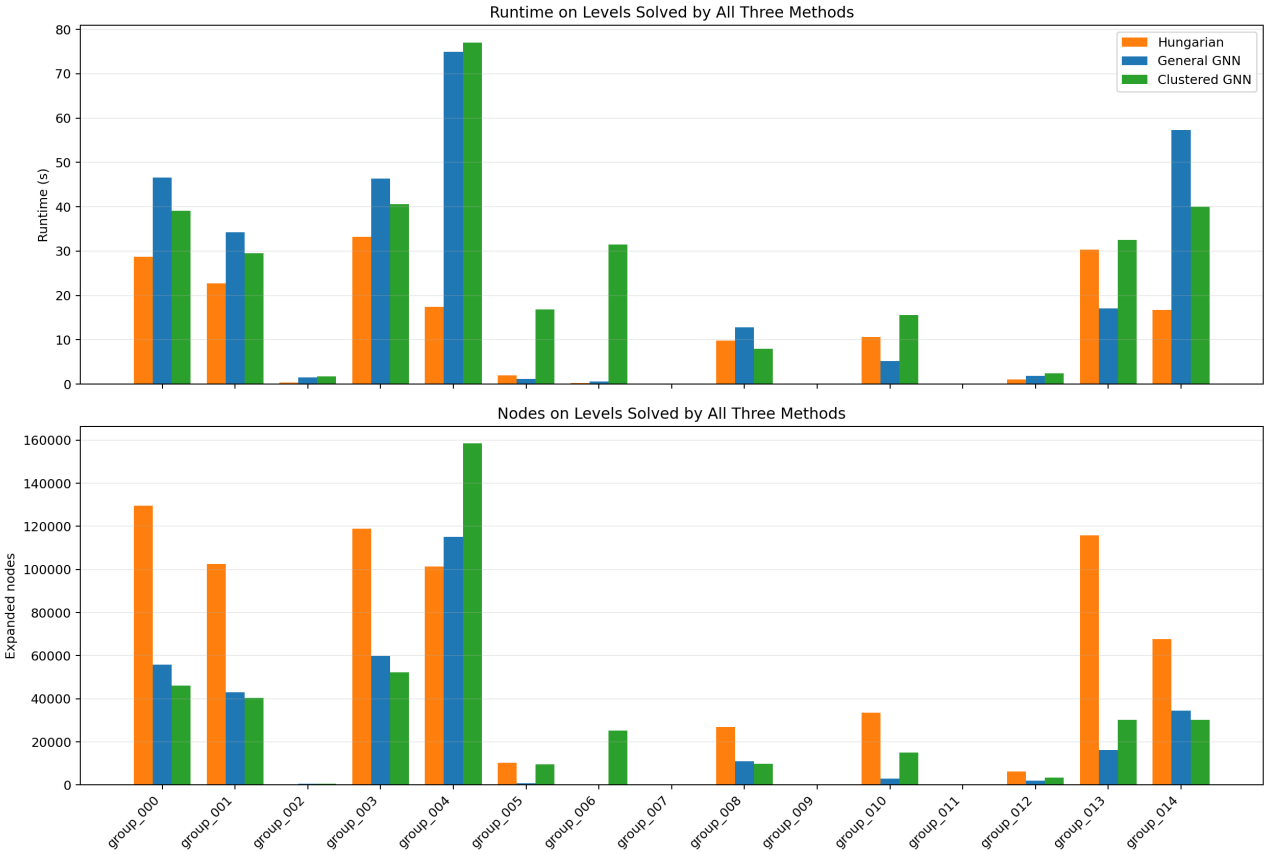


Figure 3.11: Runtime (top) and expanded nodes (bottom) on community test levels solved by all three methods, shown per split–merge group.

Delta solved rate and training data volume. Table 3.18 reports, for each group, the solved-rate difference between Clustered GNN and General GNN, together with the number of labeled training states available for that cluster. Positive values indicate that the cluster-specific model solves more instances than the General GNN on that group.

Group	Levels	Solved (General)	Solved (Clustered)	Δ solved	Train labels
00	1,242	0.1643	0.1731	+0.0089	7,856,283
01	1,292	0.5201	0.5581	+0.0379	5,624,502
02	137	0.0073	0.0073	+0.0000	1,671,782
03	470	0.1851	0.2021	+0.0170	2,761,421
04	117	0.0256	0.0427	+0.0171	666,539
05	36	0.0556	0.0278	-0.0278	222,292
06	33	0.0606	0.0303	-0.0303	220,569
07	8	0.0000	0.0000	+0.0000	42,890
08	489	0.9151	0.9407	+0.0256	859,981
09	37	0.0000	0.0000	+0.0000	262,567
10	29	0.8966	0.8966	+0.0000	67,739
11	16	0.1250	0.0625	-0.0625	110,625
12	74	0.9865	1.0000	+0.0135	45,898
13	33	0.0606	0.0606	+0.0000	238,174
14	115	0.0783	0.0696	-0.0087	879,785

Table 3.18: The difference in solved rate between the Clustered GNN and the General GNN.

A mild trend is visible that clusters with larger training label volumes tend to yield more reliable gains (e.g., group_001, group_003), while several small clusters show noisy changes in solved rate under the fixed budget.

3.7.3 Cross-evaluation matrix

Figure 3.12 reports the cross-evaluation results for the split–merge groups in the same format as Figure 3.9. Compared to static-feature clustering, the split–merge matrix shows a stronger diagonal structure for several target groups: matched models are more frequently among the top-ranked choices within their target row.

At the same time, the matrix is not perfectly block-diagonal. For multiple targets, the best-performing source model comes from a different cluster, which is also reflected by the matched-model ranks in the right panel. The split–merge grouping also separates clusters strongly by difficulty: there are several very easy clusters that are solved at high rates by many source models (bright rows, e.g., group_008 and group_012), while several difficult clusters remain dark regardless of the training source, indicating low transfer across the entire row. Overall, the cross-evaluation confirms that the split–merge grouping yields clusters that are more self-consistent for some targets, while substantial cross-cluster transfer remains in other cases.

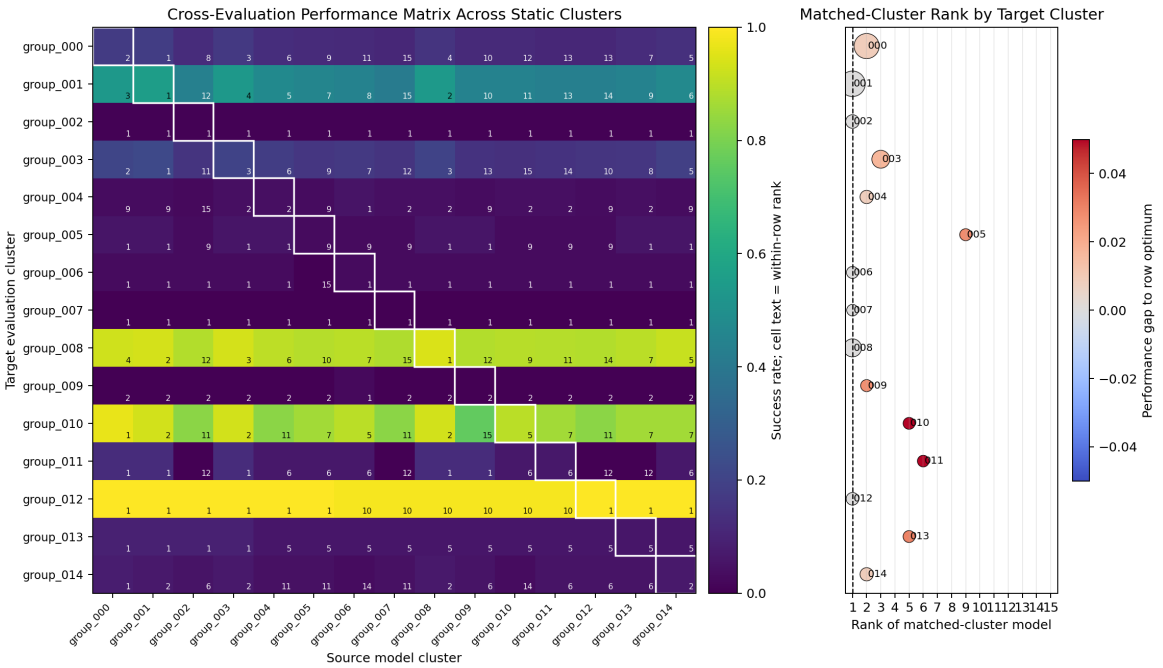


Figure 3.12: Cross-evaluation across split–merge groups. Left: solved-rate matrix for evaluating a model trained on source group (columns) on target group test splits (rows). Diagonal cells correspond to matched group models and are highlighted. Right: for each target group, the rank of the matched model within its row (1 = best in that row), with color indicating the solved-rate gap between the row-best model and the matched model.

3.8 Summary of results

Across all experiments, the learned GNN heuristic consistently reduces search effort in terms of expanded nodes, while remaining more expensive per node in wall-clock time. On Boxoban, this translates into a strong overall speedup: the GNN-guided search is also faster in runtime, while only rarely producing longer solutions than Hungarian heuristic. On the community levels, the same qualitative pattern holds in terms of search effort, but absolute runtime is dominated by the fixed time budget for unsolved instances, which makes solved rate the primary differentiator.

For community levels, the global model provides a competitive baseline but remains close to Hungarian in overall solved rate under a small 5 minutes budget. The strongest improvements come from cluster-specific training: on both clusterings considered, the clustered (own) models achieve the best overall solved rate under the fixed budget. The gains are unevenly distributed across groups: a small number of large clusters accounts for most of the additional solved instances, while several small clusters are too small to yield stable conclusions.

A consistent trade-off is visible when comparing runtime and node expansions on comparable solved instances. Hungarian tends to be faster in wall-clock time, while the GNN variants expand substantially

fewer nodes. This separation is stable across both clustering schemes and reflects that the learned heuristic reduces search effort but incurs additional inference overhead. Comparing learned heuristics directly, cluster-specific models achieve slightly higher solved rates while also reducing expansions and runtime relative to the general GNN.

Cross-evaluation further shows that matched models often rank near the best choices for their target group, but they are not always optimal. This effect is weaker for the split–merge partition than for static-feature clustering: in the split–merge case, matched models are more frequently among the top-ranked sources for their target rows, indicating a more self-consistent grouping with respect to model performance. At the same time, "hub" models exist that transfer well across multiple targets, so the best source model for a target group can come from a different cluster. A mild trend is that clusters with larger numbers of supervised training states tend to produce source models that rank highly across a broader range of targets.

Finally, the magnitude of the GNN advantage over Hungarian varies with structural properties of the levels. On the transferability-driven grouping, the improvement in solved rate tends to be larger in groups of moderate structural complexity and smaller in very compact or highly box-dense groups. At the instance level, levels solved only by GNNs tend to have larger playable areas and slightly more boxes, lower box density, and larger box–goal distance proxies than levels solved only by Hungarian. These observations provide a data-driven summary of where the learned heuristic most often adds additional solved instances under the fixed budget, and they motivate the discussion in the next chapter.

4 Discussion

This chapter discusses the main conclusions that can be drawn from the experiments. The discussion focuses on the practical role of the learned GNN heuristic in Sokoban search, the trade-off between heuristic quality and search efficiency, and the extent to which the observed behavior generalizes across different level sets and grouping strategies.

4.1 Main findings

The main findings of this thesis can be summarized as follows:

- A GNN-based heuristic can improve practical Sokoban search compared to the Hungarian baseline, although the observed gains depend on the evaluation setting, the search mode, and the level distribution.
- The improvement carries over not only to large collections, but also to the more heterogeneous community levels setting, although in a weaker and less uniform form.
- The admissibility-preserving mixed mode is not competitive in this setup. In the measured runs, the minimum operation selects the Hungarian value on the large majority of expanded states, which makes mixed behave similarly to the baseline while still paying the overhead of evaluating the GNN.
- Larger models do not automatically give better search performance. Increasing model capacity often reduces the number of expanded nodes, but after a certain point the per-node inference cost grows too much. The results therefore suggest that practical search performance depends on balancing heuristic quality against evaluation overhead rather than maximizing model size.
- Generalization across heterogeneous community levels is possible, but it is not uniform. The clustering experiments show that some parts of the benchmark benefit from cluster-specific training, while others are already handled well by a more general model. The split–merge grouping yields clearer gains than static-feature clustering, which suggests that transferability is only partly explained by static level descriptors. At the same time, there is a visible tendency that source groups with more supervised labels transfer more robustly across targets. This indicates that broader training coverage may matter at least as much as cluster matching itself.

4.2 Heuristic quality and evaluation cost

The results suggest that the learned heuristic helps because it ranks states more informatively than the Hungarian baseline. This leads to a much stronger focus of the search and therefore to substantially

fewer node expansions. This effect is visible both on Boxoban and on the community levels benchmark, even though the absolute gains differ across the two settings.

At the same time, search performance does not depend on heuristic quality alone. A heuristic is evaluated at every expanded state, so its practical value also depends on how expensive a single evaluation is. This is especially important for the GNN architecture used in this thesis. As described in Section 2.5, increasing the hidden size raises the cost of the internal linear layers roughly quadratically, while increasing the number of message-passing layers adds cost approximately linearly. A larger model can therefore provide a stronger heuristic signal, but it also makes every node evaluation more expensive.

Across both domains, larger models usually reduce the number of expanded nodes. However, beyond a certain point the runtime no longer improves, because the reduction in search effort is outweighed by the higher inference cost. A lower node count does not necessarily mean a better solver, because expensive models may spend more time per state and hit the wall-clock limit earlier.

Within the tested configurations, the model with hidden size 128 and 4 GINEConv layers provides the best practical operating point. This should not be interpreted as a universally optimal architecture, because the best balance is likely to depend on the level distribution, the search budget, and the hardware setup. Still, the experiments show that model selection should be calibrated instead of choosing the maximum model capacity.

The behavior of the mixed mode supports the same conclusion from a different angle. In theory, taking

$$h_{\min}(s) = \min\big(h_{\text{GNN}}(s), h_{\text{Hungarian}}(s)\big)$$

is attractive because it preserves admissibility. The learned heuristic can contribute whenever it is more conservative, while the classical heuristic remains a safe fallback. However, in practice, this advantage is hardly useful.

The reason is that the Hungarian estimate is smaller on the large majority of expanded states. The Hungarian matching cost is a conservative lower bound based on assigning boxes to goals, whereas the learned heuristic is trained to predict the remaining cost more directly and therefore often outputs larger values. As a result, the minimum operation selects the Hungarian value most of the time. Mixed search then behaves very similarly to the baseline in terms of state ordering, but still has to evaluate the GNN at each state. The outcome is an unfavorable combination: weaker guidance than speed mode, more node expansions, and additional neural overhead.

4.3 Performance across level distributions

The effect of the learned heuristic strongly depends on the level distribution in which it is used. In Boxoban, the GNN leads to a clear improvement over the Hungarian baseline. The reduction in expanded nodes is so large, and this reduction also translates into a substantial runtime gain despite the increased overhead for computing the heuristic. The reason is that Boxoban forms a broad but still

relatively controlled distribution: levels follow a common generation process, have fixed dimensions, and use a fixed number of boxes within each subset. This makes the training data diverse enough to learn useful patterns, but still regular enough for these patterns to transfer reliably within the same benchmark.

The situation is different on the community levels benchmark. Here, the learned heuristic still helps, but the improvement is noticeably smaller and less uniform. The reduction in search effort remains clear, but the runtime gain is much smaller. This is consistent with the fact that the community levels are considerably more heterogeneous in board size, box count, geometry, and overall level style. As a result, the model faces a less coherent training and evaluation distribution, so a heuristic that works well on one part of the benchmark may transfer only partially to another.

The plausible interpretation is that the prediction problem becomes harder once the level distribution is less structured. Even a useful heuristic may only help moderately in runtime, because harder instances consume more of the available wall-clock budget and because model errors are less consistent across instances. The experiments therefore suggest that the gap between Boxoban and community levels is mainly a distribution effect: the same method remains useful in both settings, but it benefits much more from regularity than from heterogeneity.

At the instance level, the results also suggest that the benefit of the learned heuristic depends on structural properties of the level. The GNN tends to add solved instances mainly on moderately large levels with slightly more boxes, lower box density, and larger box–goal distance proxies. By contrast, Hungarian remains particularly strong on compact and dense levels where assignment distances are short and the box–goal structure is comparatively simple. A plausible interpretation is that the learned heuristic is most useful when the search requires recognizing more global geometric and coordination patterns, rather than only estimating a local matching cost. Its advantage therefore appears largest on levels of intermediate structural complexity: not the smallest and most assignment-driven instances, but also not the most overloaded large instances where both methods degrade and the gain over Hungarian becomes smaller.

A similar contrast appears in the quality of the solution. In Boxoban, GNN search usually stays close to the push-optimal solutions, so the practical cost of non-admissibility is small. On community levels, deviations are more visible, which is expected in a more varied benchmark where heuristic errors are harder to control uniformly. However, even there, the loss in solution quality remains moderate compared to the reduction in search effort. This makes the trade-off acceptable: the learned heuristic does not preserve optimality, but it often improves search enough for the resulting solutions to remain practically competitive.

4.4 Clustering and transferability

The clustering experiments were useful mainly as a way to understand structure in the community levels benchmark. Unlike Boxoban, this benchmark is not a single coherent distribution. It contains levels that differ strongly in overall design style. Grouping levels into clusters therefore served two

purposes: it tests whether some subsets benefit from specialization, and it makes it possible to study how well models transfer across different parts of the benchmark.

The results show that specialization can help, but the effect depends on how the groups are constructed. For static-feature clustering, the gains over the General GNN are small and uneven. Some groups benefit slightly from a cluster-specific model, while others show no change or a small decrease in solved rate. This suggests that levels that look similar under hand-crafted features are not necessarily the ones that are best served by the same learned heuristic.

The split–merge grouping leads to a more convincing picture. Here, the clustered models achieve the highest overall solved rate, and the improvements are concentrated in several of the larger groups rather than being spread thinly across the benchmark. This is a more meaningful result than a uniform gain of a few tenths of a percentage point, because it shows that grouping can identify parts of the dataset where specialization has a practical effect under the fixed budget. At the same time, the smallest groups remain hard to interpret, since their test splits are too small for stable conclusions.

The matched models often perform well, especially for the split–merge partition, but they are not uniformly optimal. Some clusters benefit from their own model, while others are better covered by a model trained on a different subset of levels. A useful pattern is that a small number of source groups transfer well across many targets. This is visible in both clustering schemes, but especially clear in the cross-evaluation results where some source models rank near the top across multiple rows. A plausible explanation is training coverage: groups with more supervised labels tend to produce better source models, because they expose the GNN to a wider range of states during training. The data does not support a strict rule that the largest group is always best, but there is a visible tendency that such groups transfer more reliably. This suggests that transferability depends not only on similarity between source and target groups, but also on how much supervised experience the source model has seen.

The results also suggest that the practical advantage of the learned heuristic depends on level structure in a systematic way. On the transferability-driven grouping, the solved-rate gain of the clustered GNN over Hungarian tends to decrease as structural complexity increases, with negative rank correlations for median playable area, number of boxes, and box density. At the same time, the instance-level comparison between levels solved only by the clustered GNN and levels solved only by Hungarian points in a more specific direction: the GNN more often wins on levels with larger playable areas, slightly more boxes, lower box density, and larger box–goal distance proxies, whereas Hungarian is relatively stronger on more compact and denser instances. A plausible interpretation is that the learned heuristic is particularly useful when good search guidance depends on capturing broader spatial structure and longer-range coordination patterns, while the Hungarian baseline remains competitive on tightly constrained levels where a simple conservative lower bound already aligns well with the search difficulty.

Taken together, these results suggest that clustering can be useful, but not simply as a way to create one model per visually similar subset of levels. Its main value is analytical: it reveals that the community levels benchmark contains regions where a general model is already sufficient, regions where specialization helps, and a few source groups that act as broadly transferable training hubs. In

this thesis, the split–merge approach appears more informative than static-feature clustering, which suggests that transferability is only partly explained by fixed structural descriptors. For heterogeneous Sokoban data, a good grouping strategy should therefore capture not only static similarity, but also how well learned heuristics actually transfer between groups.

4.5 Limitations and open questions

The main limitation of the evaluation is the fixed search budget of 180–300 s per level. This budget is already large enough to make full benchmark runs very expensive, but it is still too small to fully separate the methods on the hardest instances. Under this constraint, many community levels remain unsolved by all approaches, so the measured gains appear mainly as a few percentage points in solved rate or as moderate relative improvements. It remains an open question how the comparison would change under much larger budgets, where stronger but more expensive heuristics might have more time to pay off.

A related limitation is that model selection was performed under the same practical budget. The chosen configuration provides the best trade-off in the present setup, but this does not mean that it would remain optimal under different conditions. With more training data, longer training time, or larger search budgets, higher-capacity models could become more competitive. This is especially plausible because larger models already tend to reduce node expansions, even when their runtime is not yet favorable under the current limits.

Another limitation is that the analysis is restricted to the available labeled data and to the tested grouping strategies. Cluster-specific performance depends not only on structural similarity, but also on the amount and coverage of supervised labels in each group. This makes it difficult to cleanly separate the effect of grouping from the effect of training data volume. A natural next step would therefore be to study more explicitly how transferability depends on label count, group difficulty, and the choice of clustering criterion.

Finally, the learned heuristic in speed mode is not admissible. In this thesis, the empirical loss in solution quality remains moderate, but exact push-optimality is no longer guaranteed. This raises a broader open question: whether it is possible to retain more of the practical speed advantage of learned heuristics while recovering stronger guarantees on solution quality.

5 Conclusion

This thesis investigated whether a Graph Neural Network can be used as an effective heuristic for Sokoban search. More specifically, the goal was to evaluate whether a learned GNN heuristic can reduce A* search effort compared to a classical Hungarian matching baseline, how this depends on model configuration, and how well such a heuristic transfers across different subsets of levels.

The results show that a learned heuristic can indeed improve practical search. Across the experiments, the GNN consistently reduced the number of expanded nodes and on the large, coherent benchmark this also translated into a clear runtime improvement. On the more heterogeneous community levels benchmark, the gains were smaller and less uniform, but the learned heuristic still remained competitive and slightly improved solved rate under the fixed budget. At the same time, the experiments showed that stronger models are not automatically better in practice: the best results were obtained when heuristic quality and inference cost were balanced carefully.

A second central outcome of this thesis is that transferability depends on the structure of the level distribution. The clustering experiments suggest that some subsets benefit from specialization, while others are already covered well by a general model. In particular, the split–merge grouping produced clearer gains than static-feature clustering, which indicates that transferability is only partly explained by simple structural descriptors.

Taken together, these findings show that learned heuristics are a promising practical extension of classical Sokoban search. They do not replace search, but they can improve how search is guided, especially when the training and evaluation distribution is sufficiently regular and when model size is chosen with runtime constraints in mind.

Future work could study the same approach under larger search budgets, with larger training sets and stronger models, or with grouping methods that are optimized more directly for transfer behavior. Another important direction is to investigate whether the practical speed advantages of learned heuristics can be combined with stronger guarantees on solution quality.

Overall, this thesis shows that GNN-based heuristics are a viable tool for reducing search effort in Sokoban, and that their effectiveness depends not only on predictive quality, but also on distributional structure, computational cost, and transfer across level families.

Declaration on the Use of AI Tools

The following AI tools were used in a supporting role in the preparation of this thesis:

- ChatGPT (GPT-5.4 Thinking), OpenAI. Accessed via `chat.openai.com`.
- DeepL, DeepL SE. Accessed via `www.deepl.com`.

Scope and nature of use
These tools were used for linguistic revision of the thesis text, including grammar checks, shorter and clearer formulations, and general improvements in wording. In addition, ChatGPT was used for support with Python plotting scripts based on `matplotlib`, for writing comments in code, and for assistance in drafting README files. DeepL was used for translation support and for occasional suggestions on word choice and phrasing.

Responsibility
All substantive academic content of this thesis, including the model design, experiments, data analysis, and interpretation of results, was developed by me. AI-generated suggestions were reviewed, modified where necessary, and adopted solely under my own responsibility.

List of Figures

List of Tables

Bibliography

[1] Sokobano.de. History of sokoban. https://www.sokobano.de/en/history.php, 2024. Accessed: 2026-01-17.

[2] Let's Logic. Let's logic – the online sokoban community. https://www.letslogic.com/, 2026. Website and level collection platform, accessed 2025-10-13.

[3] Joseph Culberson. Sokoban is PSPACE-complete. Technical report, 1997.

[4] Andreas Junghanns and Jonathan Schaeffer. Sokoban: A challenging single-agent search problem. In *IJCAI Workshop on Using Games as an Experimental Testbed for AI Research*, pages 27–36. Morgan Kaufmann Publishers San Francisco, CA, 1997.

[5] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[7] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 1–15. Springer, 1998.

[8] Tristan Cazenave and Nicolas Jouandeau. Towards deadlock free sokoban. In *Board Game Studies XIIIth Colloquium*, 2010.

[9] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[10] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[11] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[12] Albert L Zobrist. A new hashing method with application for game playing. *ICGA Journal*, 13(2):69–73, 1990.

[13] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[14] Yaron Shoham and Jonathan Schaeffer. Fess: A feature based approach to single-agent search. In *2020 IEEE Conference on Games (CoG)*, 2020. Presented at IEEE CoG 2020.

[15] Joris Wit. Festival: A sokoban solver. GitHub repository, 2023. Accessed: 2025-10-13.

[16] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv: Learning*, 2019.

[17] PyTorch Developers. torch.nn.linear — pytorch documentation. https://docs.pytorch.org/docs/stable/generated/torch.nn.Linear.html, 2026. Accessed: 2026-01-17.

[18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[21] PyTorch Geometric Contributors. Creating message passing networks — pytorch geometric documentation. https://pytorch-geometric.readthedocs.io/en/2.6.0/notes/create_gnn.html, 2026. Accessed: 2026-01-17.

[22] PyTorch Geometric Contributors. torch_geometric.nn.pool.global_mean_pool. https://pytorch-geometric.readthedocs.io/en/2.5.2/generated/torch_geometric.nn.pool.global_mean_pool.html, 2026. Accessed: 2026-01-17.

[23] Peter J. Huber. *Robust Estimation of a Location Parameter*, pages 492–518. Springer New York, New York, NY, 1992.

[24] J MacQueen. Multivariate observations. In *Proceedings of the 5th Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. University of California press Oakland, CA, USA, 1967.

[25] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[26] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sebastien Racaniere, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, Timothy Lillicrap, and Victor Valdes. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018.