Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Learning Classical Planning Transition Functions by Deep Neural Networks

Master's thesis

Michaela Urbanovská

Master programme: Open Informatics
Field of study: Artificial Intelligence
Supervisor: Ing. Antonín Komenda, Ph.D.

Prague, January 2020

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Urbanovská Michaela** Personal ID number: **434664**

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

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Branch of study: **Artificial Intelligence**

II. Master's thesis details

Master's thesis title in English:

Learning Classical Planning Transition Functions by Deep Neural Networks

Master's thesis title in Czech:

Učení přechodových funkcí v klasickém plánování pomocí hlubokých neuronových sítí

Guidelines:

The student will implement a Deep Neural Network (DNN) which will work as a transition function in the execution of a Classical Planning (CP) algorithm. Image of current state and all reachable states will be used as training data for the network. DNN will be trained and tested with data from possible domains such as mazes, Sokoban puzzle and/or multiagent version of mazes. Another implemented DNN will work as a heuristic function. The DNN will learn from costs of already known plans.
1) Study cases of Deep Learning applications to CP, suitable type of a DNN and CP algorithm and Multi-agent planning
2) techniques (incl. privacy preserving planning).
3) Implement the input generator for a chosen DNN and planning algorithm.
4) Implement the DNN which will work as the transition function in the planner.
5) Implement DNN which will work as the heuristic function in the planner.
6) Test implementation of the DNN in the planner on several problem instances of the target planning domains.
7) Evaluate experimentally efficiency of the proposed solution and compare it to existing techniques used for transition functions in classical planners.

Bibliography / sources:

Ian J. Goodfellow, Yoshua Bengio, Aaron C. Courville: Deep Learning. Adaptive computation and machine learning, MIT Press 2016, ISBN 978-0-262-03561-3, pp. 1-775
Masataro Asai, Alex Fukunaga: Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. AAAI 2018: 6094-6101
Michal Stolba, Jan Tozicka, Antonín Komenda: Quantifying Privacy Leakage in Multi-Agent Planning. ACM Trans. Internet Techn. 18(3): 28:1-28:21 (2018)
Christopher M. Bishop, Nasser M. Nasrabadi: Pattern Recognition and Machine Learning. J. Electronic Imaging 16(4): 049901 (2007)

Name and workplace of master's thesis supervisor:

Ing. Antonín Komenda, Ph.D., Department of Computer Science, FEE

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

Date of master's thesis assignment: **28.01.2019** Deadline for master's thesis submission: **07.01.2020**

Assignment valid until: **20.09.2020**

Ing. Antonín Komenda, Ph.D. Supervisor's signature	Head of department's signature	prof. Ing. Pavel Ripka, CSc. Dean's signature

III. Assignment receipt

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

. _____
Date of assignment receipt

_____
Student's signature

Declaration

I hereby declare I have written this thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, January 2020

.....
Michaela Urbanovská

Abstract

Abstract

There are currently two strong research directions in the area of artificial intelligence (AI), namely, machine learning and symbolic AI. Recently, there have been several attempts to bridge these two research directions. Namely, we focus on the connections between deep learning and classical planning. Deep learning provides strong techniques for extracting information from large datasets, while classical planning provides algorithms for general problem solving. Complementarity of these two techniques is therefore natural, and combining these two approaches is a step towards autonomously learning intelligent machines. In this work, we implement a technique combining classical planning and deep learning. Namely, we replace key parts of planning algorithms by solutions devised based on the deep learning paradigm. We provide experimental evaluation to compare the implemented techniques with methods used in classical planning.

Keywords: Classical planning, deep learning, transition function, heuristic function.

Abstrakt

V současné době existují dva silné směry v oblasti umělé inteligence, a to strojové učení a symbolická umělá inteligence. V nedávné době se objevilo několik pokusů o propojení těchto dvou vědeckých směrů. Konkrétně spojením hlubokého učení a klasického plánování. Hluboké učení poskytuje silné nástroje pro získávání znalostí z velkého množství dat, zatímco klasické plánování poskytuje algoritmy pro obecné řešení problémů. Tyto směry se tedy přirozeně doplňují a jejich kombinace je krokem k vytvoření autonomní umělé inteligence. V této práci jsme vytvořili techniku kombinování klasického plánování a hlubokého učení. Docílili jsme toho nahrazením klíčových částí plánovacích algoritmů řešeními, která jsou založena na paradigmatu hlubokého učení. Dále jsme ukázali experimentální evaluaci, která porovnává implementované metody s postupy, které se běžně používají v klasickém plánování.

Klíčová slova: Klasické plánování, hluboké učení, přechodová funkce, heuristická funkce.

Acknowledgements

First, I would like to thank my supervisor Ing. Antonín Komenda, Ph.D. for the support and help during my work on this thesis. Next, I would like to thank colleagues from the faculty, namely Ing. Jan Bím, Ph.D. and doc. Ing. Tomáš Pevný, Ph.D., who provided very helpful discussion and feedback. And last but not least, I would like to thank my family for their never ending support and love.

List of Tables

List of Figures

Contents

Chapter 1

Introduction

The main focus of this thesis is to analyze the possibilities and limitation of using deep learning in combination with classical planning. Instead of replacing the planning process as a whole and trying to make the network learn a search algorithm, we decided to focus on partial replacement of two components involved in many standard planning algorithms, namely, the transition and heuristic functions.

Classical planning provides great methods for general problem solving. Unfortunately, these methods can struggle in large unstructured domains. On the other hand, deep learning methods have been demonstrated to work well on many domains without a clear structure. Therefore, combining both of these methods may lead us to a possible improvement. There have been several works showing, that combination of classical planning and machine learning can be beneficial and can bring better results.

For example, Nasataro Asai in [1] and [2] connected deep learning and classical planning by creating LatPlan, which is a system, that takes in initial and goal state of a problem instance and returns a visualised plan execution. The image on the input is transformed and processed in order to generate a standardized problem representation, which can be then solved by classical planning methods.

A different example is [3], which uses machine learning techniques to create heuristic function to improve the search algorithm. And also one of the newer papers [4], which is learning search policies for the search.

1.1 Specification of Goals

In planning, a transition function takes a current state and a possible action, which can be taken by an agent, and returns a successor state. The first goal of this thesis is implementing a replacement of this function by using techniques from deep learning. Namely, we train a neural network, which will take a visual representation of the current

state on the input and outputs all possible successor states. By doing so, we do not have to create a symbolic representation of the state in order to generate the successor states.

In order to decrease the time and resources needed to solve a problem, we often use heuristic functions, which help us prune the state space and reach defined goal faster. Heuristic function gives us the estimate of a distance from the current state to a goal state. In order to compute it, we usually require a simplified version of the problem, which is constructed via relaxation [5]. The second goal of this thesis is implementing a neural network to replace a heuristic function. By taking a visual representation of the current state on the input, giving the heuristic value for the given state on the output. One advantage of this approach is, that we do not have to create the symbolic representation and we do not have to simplify the problem in order to speed up the heuristic computation.

Having the transition network and the heuristic network, we aim to use both of them in an implemented planner. By doing so, we can then compare them with regularly used transition and heuristic functions and compare their performance. That way, we will be able to tell, if an improvement can be achieved by combining the two approaches. There are several metrics for comparing quality of the search and for heuristics, but since the neural networks tend to be much slower than already implemented heuristic functions, we will not focus on time-based comparisons, mostly because the state-of-the-art planners are highly optimized and well designed software projects, which are specifically implemented to compete against other planners. Everything we used in this research has been implemented in Julia, since it is a language used for different applications including neural networks, in particular we used Flux library. In order to make the compatibility of the networks and the planners possible, everything else was implemented from scratch.

In planning, there are many different problem domains, which can be used in order to evaluate the implemented algorithms. Since we want to use the visual representation of the inputs, we selected domains, which are easily visualized and have grid-like structure. This representation is also easy to process with any neural network and provides us with a well readable input and output as well. The problem domains we use in this work are single-agent maze, multi-goal maze, multi-agent maze and Sokoban puzzle.

Chapter 2

Background

In this chapter, we focus on description of all base aspects important for our work. Starting with the very beginning, which is the classical planning we are trying to improve. We need to define the functions we aim to replace and give an overview on its important parts. Next, we look into history and description of neural networks, describing fundamentals and most important parts, which are currently the state of the art.

2.1 Classical Planning

As we already mentioned, we aim to improve classical planning and in order to do that, we need to properly explain the basics of it. First, we will look at STRIPS planning task [6], which is a definition providing the symbolic representation of any problem instance. Later in this thesis, we use deep learning to avoid this exact representation.

Definition 2.1.1 (STRIPS Planning Task) *STRIPS planning task* Π *is a tuple*

$$\Pi = \langle F, O, s_i, s_g, c \rangle$$

where $F = \{f_1, f_2, ..., f_n\}$ *is a set of facts,* $O = \{o_1, o_2, ..., o_m\}$ *is a set of operators,* s_i *is the initial state, which consists of facts, which hold in the initial state,* s_g *is a goal state condition, which contains facts, which must hold in every goal state and* c *is a cost function* $c(o) : o \to R$ *which gives each operator a cost.*

Every operator $o \in O$ *is a triple* $o = \langle pre(o), add(o), del(o) \rangle$ *where* $pre(o) \subseteq F$ *is a set of preconditions, which are facts, that have to hold in a state in order to apply the operator* o *in the state,* $add(o) \subseteq F$ *is a set of facts, which are added to the state after applying the operator* o *in* s *and* $del(o) \subseteq F$ *are delete effects, which are facts that are no longer true after using the operator* o*.*

We say, that operator o is applicable in state $s \subseteq F$, if $pre(o) \subseteq s$. After applying the operator o in state s, we get state s', which is defined as $s' = (s \setminus del(o)) \cup add(o)$.

Now, that we defined a planning task, let's define the state transition system.

Definition 2.1.2 (State-Transition System) *State transition system is a tuple $\Sigma = \langle S, A, \gamma, c \rangle$, where*

- *S is a finite set of states*
- *A is a finite set of actions*
- *$\gamma : S \times A \to S$ is a state-transition function. $\gamma(s, a)$ is defined iff a is applicable in s, with $\gamma(s, a)$ being the predicted outcome.*
- *$cost : A \to [0, \infty)$ is a cost function assigning a value to each action. The cost value can have various meanings, for example time, price or anything we want to optimize.*

To complete the definition, we must add so called classical planning assumptions as stated in [5] together with this definition, which is a set of restrictive assumptions. We are dealing with static environment, which changes only as a result of taking an action. There is no time defined and everything is happening in discrete sequence of states and actions. And lastly, we are in deterministic world, so there is not uncertainty in taking possible actions, we always know the outcome.

Any problem Π defined as STRIPS by Definition 2.1.1 can be translated to a state-transition system Σ and solved by the means of search algorithms. To do so, we create set of actions A from Σ by taking set of all operators O from Π. To create the set of states S, we have to create them by using possible combinations of facts from F. The cost function remains and the state-transition function γ is defined based on actions in A.

A solution to such defined planning problem Π is a plan [5] represented by a path in the related state-transition system Σ. In case we are trying to minimize the cost of the plan, we are looking for an optimal plan [5]:

Definition 2.1.3 (Plan) *A plan is a finite sequence of actions $\pi = \langle a_1, a_2, ..., a_n \rangle$. We define length of the plan $|\pi| = n$ and cost of the plan as $cost(\pi) = \sum_{i=1}^{n} cost(a_i)$.*

Definition 2.1.4 (Optimal Plan) *Having a planning problem Π and its solution π, we say that π is an optimal solution, if no subsequence of π is a solution for Π and if there is no solution π', such that $|\pi'| \leq |\pi|$.*

We say, that π is cost-optimal solution, if $cost(\pi) = min\{cost(\pi') \mid \pi'$ is a solution for $P\}$.

In order to find a solution to a planning problem, we need to use a problem-solving algorithm. For classical planning problems, such algorithms are typically based on state-space search [5].

Definition 2.1.5 (State-Space Search) *State-space search algorithm performs search over a graph $G = (N, E)$, where N is a set of nodes and E is a set of edges.*

Having a planning problem $\Pi = \langle F, O, s_i, s_g, c\rangle$ and its induced transition system $\Sigma = \langle S, A, \gamma, c\rangle$, N corresponds to S and E corresponds to A.

The search starts in s_i, expanding each found state with γ, until a goal state is reached. In that case, plan π can be returned as a sequence of actions applied at each expansion of the search to reach the goal state.

We can solve any given problem in planning by looking through the whole state space for a solution. That can be very slow and inefficient, since the state spaces can be very big, and some algorithms require storing all the visited states in memory. Therefore, we often use heuristic function in order to improve performance of the search.

Heuristic function is a $h(s) : s \to R$, which maps any state $s \in S$ to a real value [5]. Heuristic function gives us an estimate of path length from the current state s to a goal state. A function, which always maps $h(s)$ to the length of shortest possible path is called perfect or optimal heuristic and it is denoted as h^* in [5].

Heuristic function can be any function in general, however, there are some properties of heuristic functions, which can further assure certain properties of search algorithms.

Definition 2.1.6 (Admissible Heuristic) *Heuristic function h is admissible, if $\forall s \in S : 0 \leq h(s) \leq h^*(s)$.*

Definition 2.1.7 (Consistent Heuristic) *Heuristic function h is consistent (monotone), if $h(s) \leq cost(\gamma(s, a)) + h(\gamma(s, a))$*

Definition 2.1.8 (Goal-aware Heuristic) *Heuristic function h is goal-aware, if $h^*(s) = 0$ then $h(s) = 0$.*

Those are three important properties we might come across while describing heuristic functions, which we used in our experiments.

Computing of heuristic estimate can be a very complicated problem, in fact, it may be as complicated as solving the problem itself. To decrease the complexity of this task, we use different techniques to simplify the problem and solve its simpler version. One of these techniques is relaxation [5], which is a widely used approach. By relaxing a problem, we delete some of its constraints making it much easier to solve. For example, in the maze

domain, we could delete the walls and find length of the path to the goal by computing the Euclidean distance.

Now that we have every tool to solve our problems, and even the heuristic function to find the solution faster, we have to look at the type of solution we want. There are multiple types of planning and the type we use depends on the solution we want to get.

One branch in planning focuses on finding cost-optimal solutions, hence called optimal planning. An essential tool for finding cost-optimal solutions is a shortest path search algorithm. The most famous representative of such algorithms is A* algorithm [7], which gradually expands the state space and relies on heuristic evaluation of considered search nodes. However, in order to obtain optimality guarantees, the heuristic evaluation function is required to be admissible, by Definition 2.1.6 and/or consistent, by Definition 2.1.7. Since neural networks represent a black-box approximation scheme, we cannot expect to ensure these properties. Therefore, we use satisficing planning for our experiments.

2.2 Satisficing Planning

Satisficing planning does not require finding an optimal solution to a problem but only one possible solution. There are many algorithms, which can be used, practically any search algorithm, which is complete. Completeness does assure finding a solution to a problem if the solution exists [7]. Because we have a heuristic function to our disposal, we are using only informed search algorithms [7], which are using additional information in order to search the state space. In our case, that additional information are values from a heuristic function. Namely, we are implementing two version of the best-first search algorithm.

2.2.1 Best-first Search

Best-first search is a search algorithm introduced by J. Pearl [8]. The algorithm relies on a heuristic evaluation function, which takes a state on the input and returns a value. Since it is not closely specified, it can depend on basically anything, the state specification, goal specification, information gained on the way to the current node, etc.

For experiments, we will be using two variants of best-first search. The first one is going to be implemented as a greedy best-first search, which uses strictly computed heuristic values. The second one will be using values from the heuristic function together with information gained during the search.

Greedy Best-first Search

Greedy best first search [5] is one of the most commonly used algorithms in satisficing planning. It uses an open list to store states from the state-space, which are going to be expanded, and a closed list, which is used to store already expanded states. To expand a state and obtain its possible successors, we use the state-transition function γ. For each successor, we compute its heuristic value and put it into the open list.

Its greedy property comes from the way it picks the nodes from the open list. Nodes are picked only according to their heuristic value, node with the smallest heuristic value, meaning it is closest to the goal, will be expanded first. We are using the closed list to prevent any cycles in the search and to ensure completeness of the algorithm. It is a suboptimal algorithm, unless we have strong restrictions in the heuristic function. However, that is not a problem in case we are looking for an arbitrary solution.

We can see all the key parts of the algorithm in Listing **??**. States in the open list are ordered based on computed value of heuristics, so we use priority queue in the implementation. Once a state is expanded, it is added to the closed list, which is implemented by a set.

```
function GBFS(s_init)
    closed_list = Set()
    open_list = PriorityQueue()
    open_list.add(s_init, compute_heuristic(s_init))

    while length(open_list) > 0
        s = open_list.pop()
        if s == goal
            return s
            break
        end

        if s in closed_list
            continue
        end
        closed_list.add(s)

        for next_s in s.get_successors()
            h = compute_heuristic(next_s)
            if h != Inf
```

```
                open_list.add(next_s, h)
            end
        end
    end
    return false
end
\label{listing:GBFS}
```

Listing 2.1: GBFS Pseudocode

2.2.2 History-considering Best-first Search

The second algorithm we will be using for our experiments is a best-first search, which also considers the history of the search up to the current state. It still uses open list for storing nodes to be expanded and a closed list for nodes, which were already visited. Implementation details are the same as in the previously mentioned greedy best-first search, however, the difference is in the function used to evaluate the nodes. This time, we will not be using only the heuristic function, meaning the neural network, but we will be taking shortest length to the current node as a $g(n)$ value and we will sum it up with the heuristic value.

```
function HCBFS(s_init)
    closed_list = Set()
    open_list = PriorityQueue()
    g_values = Dict()

    h_init = compute_heuristic(s_init)
    g_init = 0
    g_values[s_init] = g
    open_list.add(s_init, h_init + g_init)

    while length(open_list) > 0
        s = open_list.pop()
        if s == goal
            return s
            break
        end
```

```
            if s in closed_list
                continue
            end
            closed_list.add(s)

            # Expand the selected state
            for next_s in s.get_successors()
                if next_s in closed_list
                    continue
                end

                h = compute_heuristic(next_s)
                g = g_values[s] + step_cost

                if not(next_s in g_values) or g < g_values[next_s]
                    g_values[next_s] = g
                    open_list.add(next_s, h + g)
                end
            end
        end
        return false
    end
```

Listing 2.2: HCBFS Pseudocode

Note that the algorithm is similar to A* [7]. However, to emphasize that our algorithm does not provide any guarantees (due to the absence of admissible and consistent heuristics), we call this search algorithm History-considering Best-first Search.

2.2.3 Multi-heuristic Search

We are trying to compare the neural networks with state of the art in satisficing planning. One of the approaches which has proven to work well in classical planning is the multi-heuristic search [9]. Multiple approaches were proposed in the paper, on how to use more than one heuristic estimated in the search. The most simple and straightforward possibility is using sum or maximum of the multiple values. However, in [9] it was shown that this is not the most successful approach.

The first approach described in [9], which achieves good results is tie-breaking, where we have a set of heuristic functions $h = \{h_1, h_2, ..., h_n\}$. Instead of using only one function

to order states in the open list, we use h_1 for the initial ordering, h_2 for the tie-breaking when there are multiple states with the same heuristic value. Then h_3 in case the values are still the same and so on. One complication in this approach is determining the order of used heuristics. If we use wrong ordering of the heuristics, it can cause worse performance.

Another option is alternation, which uses all the heuristic values and expands a state based on a different heuristic value every iteration. We always select a state, which minimizes selected heuristic h_i and also which is the oldest. This approach has proven to be the most successful one in [9].

2.2.4 Used Heuristic Functions

To evaluate results of a neural network replacing the heuristic function, we have to compare it to already existing baseline approaches. In satisficing planning, one of the most successful planners is [10], which uses H^{FF} heuristic [11] in combination with landmarks. In experiments, we use H^{FF} as well, with the addition of LM-cut heuristic [12], which also uses landmarks. We also included zero heuristic and Euclidean distance as examples of very simple heuristics, which will hopefully be outperformed.

H^{FF} - Fast Forward Heuristic

The name of the heuristic comes from the name of the planner, where it has been first used [11]. It is a heuristic function, which performs delete-relaxation on the problem, meaning, that all delete effects of operators in Definition 2.1.1 are simply ignored. With modified operators, we get relaxed planning task, which is important for further computations. Let's look at pseudocode of the algorithm from [5].

HFF(Σ, s, g)
  $\hat{s}_0 = s$; $A_0 = \varnothing$
  for $k = 1$ by 1 until a subset of $\hat{s}_k$ r-satisfies g do *(i)*
    $A_k \leftarrow$ {all actions that are r-applicable in $\hat{s}_{k-1}$}
    $\hat{s}_k \leftarrow \gamma^+(\hat{s}_{k-1}, A_k)$
    if $\hat{s}_k = \hat{s}_{k-1}$ then // (Σ, s, g) has no solution *(ii)*
      return ∞
  $\hat{g}_k \leftarrow g$
  for $i = k$ down to 1 do *(iii)*
    arbitrarily choose a minimal set of actions
      $\hat{a}_i \subseteq A_i$ such that $\gamma^+(\hat{s}_i, \hat{a}_i)$ satisfies $\hat{g}_i$
    $\hat{g}_{i-1} \leftarrow (\hat{g}_i - \mathrm{eff}(\hat{a}_i)) \cup \mathrm{pre}(\hat{a}_i)$
  $\hat{\pi} \leftarrow \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$ *(iv)*
  return $\sum$ {cost(a) | a is an action in $\hat{\pi}$}

Figure 2.1: H^{FF} algorithm for Fast Forward heuristic computation taken from [5]

We can see that the algorithm gets a planning task Σ, current state s and a goal condition g on the input. We first construct a graph, which starts by taking the state s and applying all actions applicable in s to generate successor states. This continues until

we find a state that satisfies the goal condition. Now that we found the goal in the relaxed graph, we can start the second part of the algorithm, which is backtracking. We start from the goal node and select minimal set of actions, which satisfies the goal. This way we backtrack through the whole relaxed graph, until we arrive in the initial state again. Length of the discovered path is the returned heuristic value for state s.

This heuristic is not admissible as stated in [5], since choosing actions in the backtracking part of the algorithm is not done in an optimal fashion. Therefore, we cannot actually say, that the path found in the relaxed transition graph is optimal. However, for satisficing planning H^{FF} heuristic is widely used.

LM-cut

To introduce the LM-cut heuristic [12], we first need to explain landmarks, which are the key part of the heuristic computation. Landmarks are logical formulas, that need to be true at some point in the search. The already mentioned LAMA planner [10] uses landmark based heuristic together with the H^{FF} heuristic, which has proven to be a great combination according to the IPS 2008 competition results, as it won the satisficing planning track.

There are multiple heuristic functions, which are using landmarks in various ways. In this thesis, we use operator landmarks. Operator landmark L is a subset of operator set O such that in every plan, there is at least one operator $o \in L$. With these landmarks, we compute the LM-cut heuristic. To do so, we need to use h^{max} heuristic.

For a set of literals $s = \{s_1, s_2, ..., s_n\}$ max-cost is defined as a largest max-cost of each s_i individually. Such value is computed by function Δ^{max}, which is defined as stated in [5]:

$$\Delta^{max}(s, g) = max_{g_i \in g} \Delta^{max}(s, g_i)$$

$$\Delta^{max}(s, g_i) = \begin{cases} 0, & \text{if } g_i \in s \\ min\{\Delta^{max}(s, o) \mid o \in O \text{ and } g_i \in add(o)\}, & otherwise; \end{cases}$$

$$\Delta^{max}(s, a) = cost(a) + \Delta^{max}(s, g)$$

Max-cost heuristic can be computed as

$$h^{max} = \Delta^{max}(s, g)$$

where s is a state and g is a goal condition.

Now that we are familiar with the h^{max} heuristic, we can explain LM-cut computation itself. To compute LM-cut of a state s, we first compute its h^{max} heuristic, if it is equal to 0 or infinity, we return the value. In case it is not, the cost of solution to the

relaxed planning task is non-zero. We initialize value of the heuristic to zero and start the computation. We construct an action landmark L, which consists of operators with non-zero cost, from which at least one $o \in L$ must be in the plan. We take $o \in L$ which has the minimal cost from all operators in L and we add its cost to the heuristic value. We also subtract this cost from costs of all operators in L. This process is repeated until value of h^{max} remains greater than zero. Once it is equal to zero, we return the computed heuristic value.

This heuristic function is computationally challenging mostly because deciding, if a set of operators is disjunctive landmark, is P-SPACE complete problem [5]. There are many different ways of discovering landmarks. In this implementation, we took similar approach to [12] and implemented landmark generating function using a justification graph and s-t cut. We construct the graph by connecting each add effect of operator to it is most costly precondition. We then perform s-t cut on this graph, which returns edges from the graph - operators. We look for edges, which connect two sets of nodes, the ones that are accessible from initial state with cost equal to zero, and the rest of the nodes. These operators then form the landmark L.

2.3 Neural Networks

Since human brain can learn a large amount tasks without being specifically programmed to do so, it has been a great inspiration for artificial intelligence. Namely, the field of machine learning. Sometimes, it is not possible to program a machine to perfectly perform a task. The task can be only shown by example, or it is too complicated to be encoded by human. There are many cases like that, for which machine learning can be beneficial.

For a machine to learn, we need a structure, that is capable of updating itself. Inspiration in creating such structure often came from biological examples. One of these structures is a neural network.

As said in [13], the term 'neural network' originated in attempts of finding mathematical representations of information processing in biological systems. The first artificial neuron was proposed by McCulloch and Pitts in [14] and later on, other researchers built on top of this original idea to introduce significant improvements in the field, such as multilayer perceptron [15].

From a mathematical perspective, a universal approximation theorem states, that neural networks can learn any continuous function [16]. This theorem has been explored by many others. In [17] there was a focus mostly on the architecture, which is considered to be the feature, which gives the neural networks the ability of being a universal approximator.

And in [18] it was shown that it holds for any non-linear activation function.

Therefore, we can say that with a mild assumption about the activation function, we can approximate any function by a neural network.

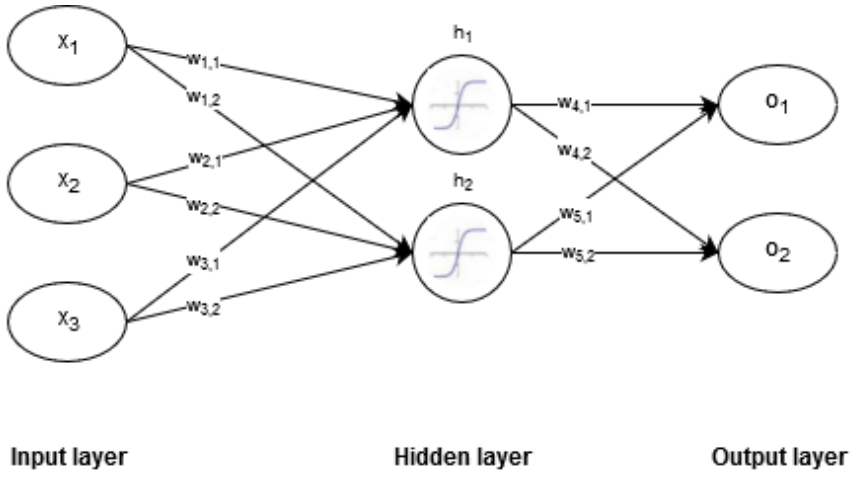

Figure 2.2: Example of a simple neural network

To introduce a very basic structure of neural network, let's look at this simple example in Figure 2.2. Leftmost layer is the input layer, which has three inputs $\{x_1, x_2, x_3\}$. Next we have hidden layer, which consists of two neurons $\{h_1, h_2\}$. Both neurons in the hidden layer have sigmoid as activation function. Last are two outputs located in the output layer. Next, there are edges from inputs to hidden layer and from hidden layer to outputs, with weight assigned to them.

To compute the input for any neuron, we have to compute a weighted sum of outputs from the previous layer. In case of h_1, its input is

$$h_1^{in} = x_1 w_{1,1} + x_2 w_{2,1} + x_3 w_{3,1}$$

Because h_1 has a sigmoid activation function, output of this node is

$$h_1^{out} = \frac{1}{1 + e^{-h_1^{in}}}$$

To finish the computation, the output o_1 is computed as

$$o_1 = h_1^{out} w_{4,1} + h_2^{out} w_{5,1}$$

Now that we have output of the neural network, we can look at how correct it is compared to the expected output. To do so, we need a loss function, which returns a value based on computed outputs and expected outputs.

In order to decrease the loss and to output values closer to the expected output, we

have to start training the network. To train the network, we have to update weights in the network, which can be done by the backpropagation algorithm.

2.3.1 Backpropagation

We have a loss function, which gives us error for a set of computed and expected outputs. Backpropagation takes this error value and computes its partial derivatives with respect to every parameter in the network. Value of the derivative for each weight denotes its credit. Credit value says how much does a weight contribute to the output of the network. Computation of backpropagation runs in the exact opposite direction than computation of the outputs of the network. The efficiency of the algorithm lays in the chain rule, which is applied through the whole network in the backwards direction and allows to reuse already computed derivatives instead of computing the derivatives for each layer separately [19].

It was one of the first ways of showing, that neural networks are actually capable of learning non-trivial features. Until then, hand-crafting features was often taken approach, which was limiting because of time and computational power required to include more fields of problems. Now, it is commonly used together with a gradient descent type of algorithm, to complete the whole training process of the neural network.

2.3.2 Stochastic Gradient Descent

Our main goal is still optimization, meaning that we want to converge to a minimal possible error regarding the results of our network. To do the right adjustments to the weights of our network, we will use gradient descent algorithm. As described in [7], we pick a point in the weight space by initializing all the weights in our network. By computing the gradient by backpropagation algorithm, we will move to a neighboring point, which is downhill and repeat until we converge to a minimum. Very important part of this process is the learning rate which determines how large is the step taken. It can either be a constant or it can change overtime.

However, using gradient descent method over the whole data set may be very costly, because data set for neural networks tend to be large. One solution to solve this issue is the stochastic gradient descent, which is probably one of the most used optimization algorithms when it comes to deep learning. The SGD works over mini-batches and not the whole data.

Algorithm 2.3.1 (Stochastic Gradient Descent) *Stochastic gradient descent update in time step k requires learning rate ϵ_k and initial parameters Θ on the input. The function f denotes computation done by neural network.*

Until a stopping criterion is met, repeat:

1. *Sample a mini-batch of data samples* $\{x^1, x^2, ..., x^n\}$ *and their corresponding labels* $\{y^1, y^2, ..., y^n\}$

2. *Compute the gradient*

$$\hat{g} \leftarrow +\frac{1}{n}\nabla_\Theta \sum_i loss(f(x^i; \Theta), y^i)$$

3. *Update the parameters*

$$\Theta \leftarrow \Theta - \epsilon\hat{g}$$

That is the very basic description of stochastic gradient descent, which shows how the network learns. Updating parameters to approach the minimal error is the main goal.

2.3.3 Convolutional Neural Networks

During creating architectures of our networks, convolutional networks played a great role. All of our data domains have image-like structure and we are trying to extract information from them based just on the visual representation. A usual approach when working with such data is deploying convolutional neural networks.

As the name suggests, the operation, which is performed when using these networks is convolution. Convolution function has two arguments, the input array I and the kernel K. Since we are dealing with multidimensional arrays, we will denote them as tensors. In this case, input I has four dimensions, width, height, number of channels and number of samples.

The kernel K is a tensor as well, with multiple trainable parameters. When creating it, we need to specify its width and height, as well as the number of input channels and number of output channels. The number of input channels has to equal to the number of channels in the input tensor in order to process it.

Besides the size or the kernel, we have other hyperparameters, which influence the convolution computation and modify the size of the output. The first one is padding, most common padding schemes are valid and same. The valid padding performs convolution only on valid pixels, which causes the output to be smaller than the input. The same padding pads the input with zeros, so that the convolution produces output of the same size as the input. It is also possible to define padding of a custom size.

Another one is stride, which defines the movement of kernel across the input data. Stride equal to one means, that we move the kernel by one pixel at a time, which causes heavy overlapping during the computation. We can also define the stride so that there is no overlapping at all. The output dimension is decreasing with increasing the stride.

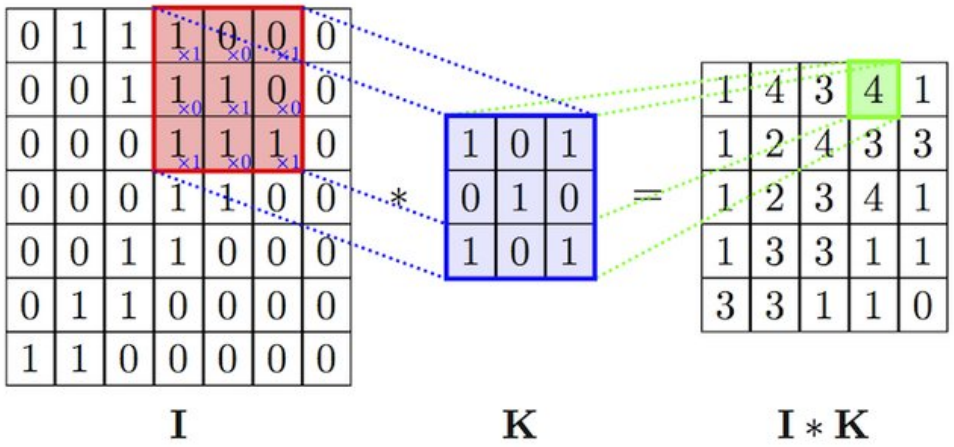

Figure 2.3: Example of convolution taken from [20]

We can look at the kernel as multiple independent kernels, one for each channel. We can then compute the convolution as

$$(I * K)_{i,j,k} = \sum_{m,n,o} I_{i \cdot S+m, j \cdot S+n, o} K_{m,n,o,k}$$

where S is the given stride for the convolution computation as stated in [15].

2.3.4 Attention

Attention networks have proven to be a great success as showed in [21], by introducing the Transformer architecture. In general, attention allows the network to focus only on subsets of inputs.

In this work, we use attention masks created by convolutional layers and softmax. We use multiple masks, not only one. Standard procedure in such case is concatenating results of the multiplications of input and all the masks, in channel dimension. Also, each mask is created by a separate convolutional network, which is also trained with the rest of the architecture.

The created mask of values may contain only zeros and ones, in that case it is a hard attention. In our case, we use soft attention, which uses values between zero and one.

Having such mask, we can then multiply the input features. That results in obtaining modified input, with some of the features "emphasized" by the attention mask.

2.3.5 Recurrent Neural Networks

Another explored direction, which we will be discussing, are recurrent neural networks. These networks are used mostly for sequential data. This type of neural network has

been a great success in many fields, one example is natural language processing, sequence classification or sequence prediction. Recurrent neural networks are a great tool for processing sequences, however there can be a problem with long-term dependencies, as the sequences can be arbitrarily long.

The main building block of recurrent neural networks are the recurrent cells, there are multiple types of them, but we will first look at the very basic one.

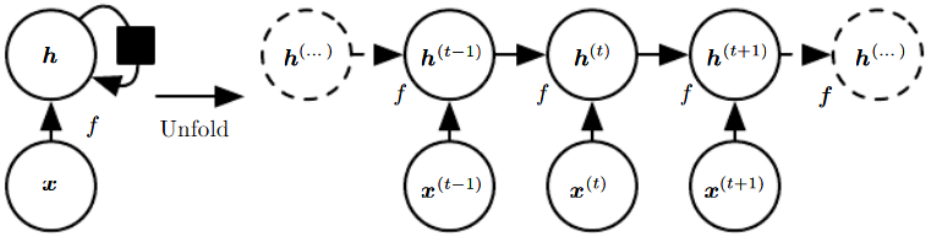

Figure 2.4: Recurrent network example with no outputs [15]

On the input of a recurrent cell, there is not only an input tensor, but also the previous state, meaning the last generated output. After unfolding the cell for n-steps, we can see, how the computation works. We can also have a network, which generates an output at each single time step, or a network, which does use recurrent cells and returns only one output for the whole input sequence.

One of the well-known recurrent cells is the LSTM cell, which was introduced in [22]]. LSTM stands for long short-term memory, which introduced the possibility of forgetting information stored in memory cell. LSTM consists of input gate, output gate and a forget gate. It has been a great discovery, since storing information about more than a couple iterations in the past has been a great problem in the field of recurrent neural networks. It has worked very well in the area of handwriting recognition [23], speech recognition [24], and many others.

In this work in particular, we focused mostly on the reasoning network presented in [25], which is also a form of recurrent neural network. Instead of just a simple feeding the previous output to the input, there is a cell containing memory and control modules, which supports reasoning over the input. Unlike LSTM or the regular recurrent cell, the MAC cell introduced in [25] had to be implemented from scratch in order to train it, since it is not a standardized recurrent cell, which appears in machine learning libraries.

The MAC cell has two main units, the control unit and the memory unit. In the control unit, the main reasoning operation is performed in each time step and the internal state of this unit is updated accordingly. The memory unit consists of the read and write modules. In the read module, information is extracted from the knowledge base, which is the maze in our case, and a retrieved information is then passed into the write module. The write

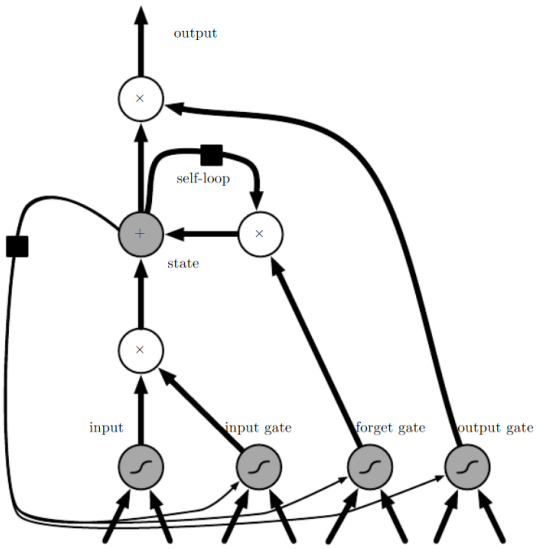

Figure 2.5: Block diagram of LSTM cell [15]

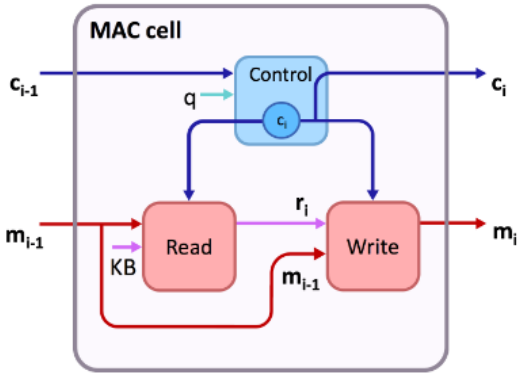

Figure 2.6: MAC cell [25]

module takes the retrieved information and current memory state and produces a new memory state.

This MAC cell core has been kept in our implementation as well, since its structure does not depend on the definition of the task as much. However, in the paper the solved task is much different from ours, so we had to modify the input and output modules in order to fulfill our requirements.

Chapter 3

Proposed Solutions

Now that we discussed both classical planning and neural networks, we can talk about the main focus of this work and that is combining the two fields. The proposed combination technique consists of creating two networks, each one to substitute a different part of the state-space search algorithm we defined earlier in Section 2.1.5.

First network is used to replace the state-transition function γ in order to generate possible successor states in the state-transition system as defined in Definition 2.1.2. The second network is used to replace a heuristic function $h(s)$, which returns a number, that represents an estimate of distance from state s to a goal state.

By replacing these parts of state-space search, we avoid the need of creating symbolic representation of the states. That would be usually necessary in order to obtain new successor states or to compute a heuristic value. By doing so, we narrow the gap between model-based classical planning and model-free machine learning [26].

One key challenge of implementing such approach is obtaining a good quality data sets in order to train the networks. The other challenge is creating the architectures themselves. We first focus on generation of all required data sets and then proceed with creating the neural network architectures.

3.1 Used Problem Domains

Since our main goal is using visual representation of problems, we use convolutional networks described in Section 2.3.3. Such networks are well suited to process data organized in grids. To this end, we also consider domains, which can be represented by a grid. This allows convenient visualization of the problems. The grid representation w.l.o.g. also provided, that the neural networks can process higher-dimensional "visual" representation of more complex planning problems.

Each described data set consists of train and test set of samples, which are disjunctive.

Both train and test set contain data samples and their corresponding labels. Batches for the training of the network were randomly sampled from the training data sets.

There is one data set for each selected problem domains. In classical planning, there are many problem domains, which are used as benchmark problems when comparing planning algorithms or heuristics. In this section, we describe all selected problem domains and the process of generation of their corresponding data sets.

3.1.1 Maze Domain

To start off with a simple data domain, which is easy to solve for humans as well, we use the single-agent maze domain. We have four cell types, agent, goal, free cell and a wall. There is one agent and one goal located in the maze, movement possible in 4-neighborhood and all free cells are accessible by the agent. As for every domain, we implemented a data generator from scratch to be able to get enough data for the networks to train. The maze generator is based on Prim's algorithm [27], which can be used in a randomized form to generate mazes, which fulfill our demands. The generator is written in a way, so it can generate any size of the maze and it places the goal and the agent randomly into the created maze. Examples of randomly generated mazes are in Figure 3.1.

Figure 3.1: Examples of generated mazes
Yellow denotes the agent position, orange denotes the goal position, purple denotes the walls and black denotes the corridors.

```
function generate_maze_prim(maze_size)
    grid = ones(maze_size)
    wall_list = []

    cell = (rand(x),rand(y))
    grid[cell] = 0
    neighbors = get_neighbors(cell)
    wall_list.add(neighbors)
```

```
        while wall_list not empty
            wall = rand(wall_list)
            grid[wall] = 0
            neighbors = get_neighbors(wall)
            wall_list.add(neighbors)
            wall_list.remove(wall)
        end
    end
    \label{code:rand_prim}
```

Listing 3.1: Randomized Prim's Algorithm Pseudocode

This algorithm in Listing **??** satisfies the requirements we have on our single-agent maze instances and creates enough random data samples. Of course, there are different types of mazes as well, not all mazes have the same restrictions. There are special cases just like a spiral, which can be generated by the Prim's algorithm, but there are many others, for example mazes with wide hallways or empty mazes, which cannot be generated with our algorithm.

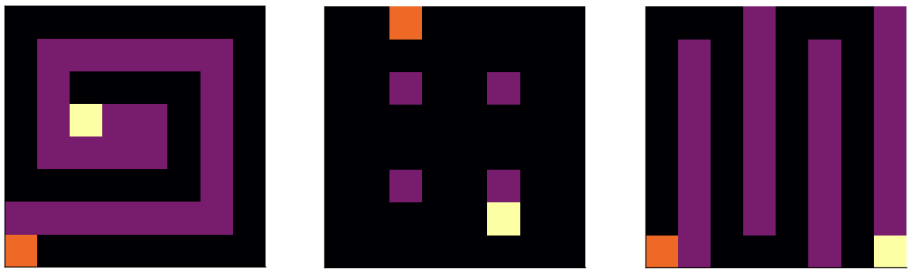

Figure 3.2: Examples of handwritten mazes
Yellow denotes the agent position, orange denotes the goal position, purple denotes the walls and black denotes the corridors.

Just to see, how well our networks work on different data, we generated some samples manually. We wanted to use these unique cases, to see the results, since they are not in the training data set. We do not expect the results to be perfect, since the structure of some of these samples is very different from the generated data. However, it is interesting to see, if the trained knowledge can be transferred to a slightly different domain.

3.1.2 Multi-goal and Multi-agent Maze Domains

In order to test the neural network architectures properly, it is good to have multiple domains we can train on. Even though it is probable, that we will not be able to reuse the networks between the domains, we still want to increase complexity of our data domains to track the results.

Figure 3.3: Examples of generated multi-goal mazes
Yellow denotes the agent position, orange denotes the goal position, purple denotes the walls and black denotes the corridors.

Increasing the complexity of the single-agent maze can be done by simply adding more goals to the map. This creates more complicated task, because there are multiple possible goals on the map. However, it still maintains the number of agents and the rules of movement from the simple mazes.

Next step, in increasing the complexity, is using multiple agents with multiple goals. This increases the difficulty of the task quite a lot, because now, we have multiple moving entities in the maze. Introducing this problem domain also comes with a few unique scenarios we have to deal with. Agent can still move in their 4-neighborhood, but they cannot jump over each other or switch places. It is also important to say, that goals in the maze are not assigned.

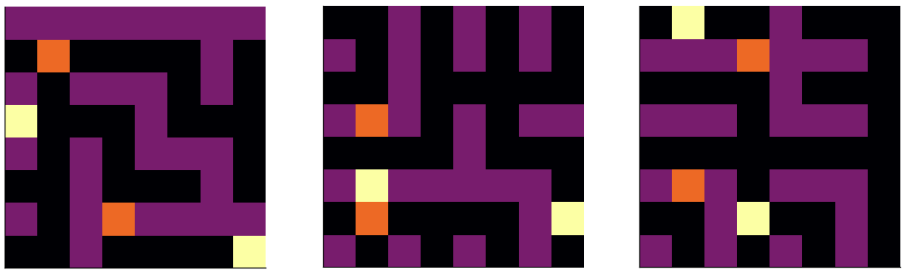

Figure 3.4: Examples of generated multi-agent mazes
Yellow denotes the agent position, orange denotes the goal position, purple denotes the walls and black denotes the corridors.

To generate data for both of these domains, we used the same randomized Prim's Algorithm as mentioned before. After generating the mazes in similar fashion, we proceed with placing goals and agents into the maze depending on the domain. The positions of both are chosen at random.

In case of multi-agent domain, we have to deal with one new problem. The generated problem instance don't have to be solvable. In case of regular mazes and multi-goal mazes, all the instances are always solvable, but that is not the case with this domain. Therefore, the complexity of the data generation increases, since we want to use solvable problem

instances.

3.1.3 Sokoban Domain

To add even more complexity, we decided to add a problem domain, which is often used as a classical planning benchmark. It is a well-known P-SPACE complete puzzle game, Sokoban. It was first introduced in Japan in 1981 by Hiroyuki Imabayashi and later released as a video game. Sokoban can be easily represented as a grid, which suits well our set of problem domains. The rules of movement are very similar to movement in mazes, the only difference is the box. Boxes can be pushed, cannot be pulled and we cannot push two or more boxes at the same time. The goals of the game is to move boxes around the map until each goal has a box on it.

Compared to all the other domains we introduced earlier, Sokoban maps are very different. Not only the number of entities is higher, but we also tend to have a lot more free space in the grid. Another complication is, that we have only one agent, but all the boxes count as moving entities, which changes the transition function for this problem.

Another great problem we came across with Sokoban domain was the data. Generating Sokoban levels, and planning problems in general, is a difficult problem. Its difficulty even increases, when we want solvable levels, which is also an important property for us. It is not beneficial for us to train the networks on levels, which cannot be solved.

We already mentioned, that we need lots of data in deep learning to obtain the best possible results. Getting the data can be a difficult task, especially with a problem domain like this one, where generating the levels is by itself a very interesting and complicated problem.

Solvable Sokoban Level Generation

The first step of creating the Sokoban level generator was actually trying to find already implemented one. There are multiple implementations of Sokoban levels online and there are even ones, which focus on quality of the levels. For our cause, we needed quantity rather than quality, so we implemented generator proposed in [28].

We first start with an empty map of requested size, into which we start placing patterns. The Sokoban map generator uses seventeen 3x3 patterns, which are randomly flipped and rotated and then placed onto an empty map next to each other. In each step, we pick one random template and place it in the first free top left space. After using up all the free space on the map, goals, boxes and the agent are placed. Filled map is then processed by a solver, which determines if it is solvable or not. This step in particular is a complication, since solving Sokoban is P-SPACE complete [29]. To slightly improve

performance on this step, there is a limit on how many states can the solver expand. Even with such restriction, the algorithm is very slow.

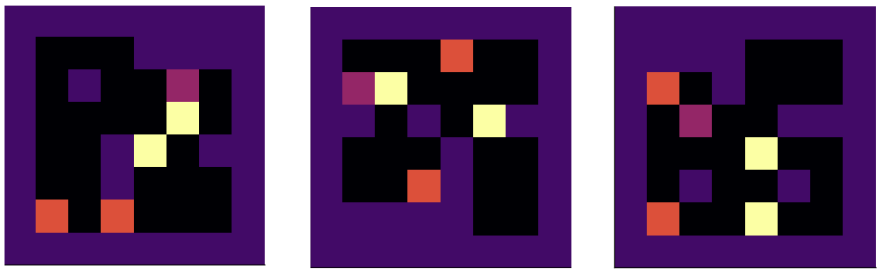

Figure 3.5: Examples of generated solvable Sokoban levels
In case of this maze, dark violet denotes the walls and black are the floor tiles. The boxes are yellow, goals are orange and the agent is dark pink.

There are also multiple conditions, which should help discarding the unsolvable maps quicker. One of them is a check for the continuous floor, since we do not want any inaccessible islands of free space in the map. Next is looking for "trapped" floor tiles, meaning tiles that are surrounded from at least 3 sides by the walls and the last one is checking it there is not too much free space in the map, since according to [28] levels with large empty spaces tend to be too easy to solve and are not as interesting as ones with small free spaces. Since we were generating quite small maps, we decided to put 2 boxes in the map to not make the map too overcrowded. However, the generator is written in a way that arbitrary many boxes and goals can be placed into the map. The last step is solving the level and returning it in case it can be solved.

Some of the maps are not too interesting for a human player, since the agent and boxes are placed randomly. It does happen that there are boxes right next to the goal or that the maps are too easy, however we need those data samples to train the networks properly, so it can see as many scenarios as possible. Also, it does not mean that a map, which is easy to solve for a human will be easy to solve for a planner.

3.1.4 Sokoban Data for the Expansion Network

Now that we know, how complicated is generating Sokoban levels, we need to consider data for the expansion network. Expansion network needs a large data set, especially in the case of Sokoban, because all possible movement combination including boxes are not present in the generated maps.

To have a better chance at training the expansion network for Sokoban domain, we had to create a separate data set containing all possible steps. We can make any possible step on 5x5 grid by placing the agent to the middle, therefore all the samples in this data set are 5x5. After placing the agent, we create all possible combinations of walls,

free spaces and boxes in the possible agent movement directions. That leaves us with a "cross" of values with agent in the middle. From this set of samples, we delete the ones with no moves possible. The rest of the samples is copied several times, until we obtain a sufficient number of samples. Then, we go through the whole data set and randomize all the cells, which were not assigned in each sample. Even though there are same step scenarios in the data, due to this randomization, they are not likely to be exactly the same.

3.1.5 Sokoban Data for the Heuristic Network

To train the heuristic network, we need to have a data set consisting of solvable Sokoban maps. As we mentioned before, generating solvable maps is an issue. This problem is common in the deep learning field and can be solved by multiple methods. The one we decided to use is data augmentation.

Data augmentation uses various modifications of the already existing data in order to generate new data samples without the need of generating them again. In our case, we augmented the data by randomly rotating and flipping it, which resulted in different maps. One advantage of this approach is, that even though the maps looks different, length of the optimal solution is still the same. That means, that we do not need to generate the labels for the augmented data samples again, which saves us a great amount of time.

3.1.6 Label Generation

We already established that labels are important part of the data set for neural networks. In order to compute loss and to train the network, we need the correct labels. In this case, we are looking at two different data sets, where each one needs a different set of labels. For the expansion network we give a maze in a certain state on the input and the output show how the maze looks after taking all the possible steps. On the other hand, for the heuristic network we require a number on the output, which tells us the heuristic value for the input maze.

In order to generate the label set for expansion network, we just need a function similar to the transition function used in search algorithms. In fact, almost exactly the function we are trying to replace. Such function just returns possible next states and for each of them, we add one input maze to the data set. If there are multiple steps possible in a current state of a maze, the function returns all the possible next states. In such case, we copy the input maze into the data, so that every label has an input. That can lead to increasing number of samples in the data set, however in the simple maze and multi-goal

mazes, it cannot grow more than 4 times and even that is highly impossible, because that would mean, that the agent is always on a crossroad, which is not really possible. In Sokoban, it is a very similar scenario, because the only entity creating steps is the agent. The only extra thing we have to take into account is possibly pushing boxes and being sure that the generated steps are valid.

In the case of multi-agent mazes, the data set can grow a lot more, because we are dealing with combinations of movements, which in the worst case means that there can be 16 samples for one scenario. That would be the case if there were multiple agents on crossroads. Again, it does not really happen, because all the samples are not just two agents in separate crossroads, but the data set does grow when generating the labels.

Labels for the heuristic network are a little more complicated to create, since they actually demand solving the problems. We are going to be solving our problem instances in the means of satisficing planning during the experiments and evaluation, however, we want the labels to be the values of h^*, so we need to use an optimal planning algorithm, which does return cost of the shortest possible plan to the goal state. For that, we used A* algorithm.

3.2 Planner Implementation

Since our main goal is replacement of parts of the state-space search algorithm in a planner, we must implement the planner regardless. Not only because we want to use neural networks instead of transition function and heuristic function, but also because planners like these always use the symbolical representation we are trying to avoid by using neural networks. Therefore, implementing our own planners, which are using different representation, is necessary.

Because we are acting upon the visual representation, it is necessary to create a domain-dependent planner for each of our domains. Each of these planners uses the same planning algorithms and the same heuristics.

Input of the planner is

- initial state
  - 2D array with the initial state of the problem
- selection of transition function
  - true for expansion network
  - false for regular transition function
- type of heuristic

  - "none" - always returns 0
  - "euclidean" - Euclidean distance
  - "hff" - fast forward heuristic
  - "lmcut" - LM-cut heuristic
  - "neural" - heuristic network

- array with heuristic network
  - empty when not using the "neural" heuristic
  - non-empty when using the "neural" heuristic

The expansion function in the algorithm can be replaced by a neural network by loading the proper model and one-hot encoding the maze, which is supposed to be expanded. Another thing, that has to be done is extracting the goals from the maze, since goal position is not an important information in the task of making a valid step. Adding another entity to the input is unnecessary, so we process a maze without goals. That is performed by encoding the maze, creating a free space on the goal positions, and passing it to the input of the neural network. The expansion network then outputs a maze, which contains multiple agent placements, from which we then extract all possible states. It is also possible to turn on a manual check of the returned states, in order to see if the network makes any mistakes.

For the heuristic function, we implemented the already mentioned H^{FF} and LM-cut heuristics as a very successful state of the art heuristic functions. In case of using the neural network as a heuristic function, there is a parameter in the function, which returns values of heuristic of a state, which can be set to "neural". In that case, an already loaded network gets a one-hot encoded maze as its input and returns a heuristic value. This value is then returned and treated as a regular heuristic estimate. In addition to these heuristic functions, we also use the zero heuristic and Euclidean distance.

In order to evaluate, how well do the networks perform, we need to compare the performance of them with regular transition function and regular heuristics. The planner returns length of the solution, number of states, which were expanded in the search and array with the whole path, which is needed for further evaluations and comparisons.

3.3 Expansion Network

In the state-space search (Definition 2.1.5), transition function takes in a current state of the search and returns all its successors. It does so, by applying all actions applicable in

that state. One great advantage of this function, is that it has the symbolic representation of the actions. There could be a case, where symbolic definition of the actions is too difficult to create. Or a case where the action is not defined at all and the only way to obtain information about it, is by observation.

That is the problem we are trying to solve with the expansion network. By observing pairs of states, without knowledge about actions that connect them, we want to learn possible actions for the domain. Even for the most simple maze domain, size of the data set is important in order to train the network. The task does not only lay in the location of free spaces around the agent, but we need to make sure, that the maze structure remains the same and no rules are broken when performing the learned actions.

We could say, that we are working with very small maze images, therefore it is convenient to use convolutional neural networks for this task. A key hyperparameter, when using convolutional networks, is size of the used kernel. In this case, we want to focus on the 4-neighborhood of the agent, which means that if we took a 3x3 window, we could evaluate all the possible steps. Therefore, we use a 3x3 kernel in the convolutional network. Another parameter is padding. The padding we use in this network is equal to one, since we want to preserve the size of the input through the whole network. The maze structure is very important to us, so changing the data size during the process does not seem to be very reasonable or beneficial. Convolutional networks tend to be chained, so we created the first architecture by chaining three of these kernels.

A great improvement for this network is usage of residual connections. Residual connection is an architecture modification, which is often used in deep learning and achieved great results for example in ResNet image classification network [30]. In ResNet, residual connections were added to a deep architecture, which had problems with learning identity function. After adding the residual connections, complexity of the network decreased and there was an improvement in the results.

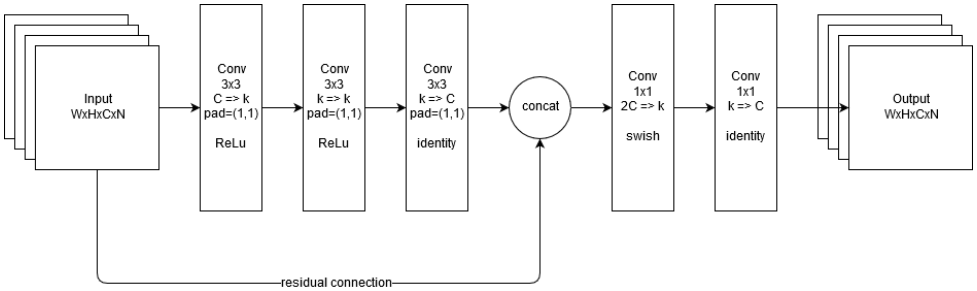

Figure 3.6: Expansion network architecture

Expansion network has one residual connection, which connects the input data with output of the three chained convolutional layers. After concatenating these two parts of data, we process them through 1x1 convolution to adjust the number of channels to match

the input. In order to obtain even better results, we added normalization in the form of dropout between the first three convolutional layers. We can see the whole architecture in Figure 3.6.

This architecture can be parametrized by several values. We can determine the variable k, which in Figure 3.6 denotes number of channels in the hidden convolutional networks. We can also determine size of kernel for all the convolutional layers.

3.3.1 Loss Function

Using a proper loss function is a key to training any network. In this case, we train the network to learn distribution of values on a grid, so cross-entropy is a fitting loss function. Since we are using one-hot encoded tensors, we are using its version intended for this representation.

```
function logitcrossentropy(x, y; weight = 1)
    return -sum(y .* logsoftmax(x) .* weight) *
        1 // size(y, 2)
end
```

Listing 3.2: Logistic entropy loss from [31]

Unfortunately, the one-hot encoding has its limitations, because we cannot express such thing as an agent standing on the goal. To do so, we would have to add another dimension, increase the complexity of the problem, or use fraction numbers. The loss function is not suitable for usage of fractions and another added dimension is unnecessary addition. If we go the other way and replace the goal cells with empty cells, we keep the task the same. The goal is treated the same way as the empty cells and to make a valid move, it is not important to know the location of the goal.

3.3.2 Expanding to Multi-mazes

One great advantage of creating the no-goal representation is, that multi-goal mazes act exactly the same as regular mazes in the eyes of the expansion network. Therefore, the same network, which we implemented for the regular maze domain, can be reused for the multi-goal maze domain.

Sadly, that is not the case with the multi-agent mazes. In this domain, we have several special cases, which cannot appear in the other maze domains. Movement rules for all the agents are still the same, but now, there are collisions possible. Without placing such collisions in the training data set, there is no way for the network to learn how to evaluate them.

Because we are not dealing with anything else, than increase in the number of agents, there could be a possible reusability in this domain as well. A maze with single agent could be denoted as a special case of multi-agent maze. That leads us to a possibility of using only one expansion network for all our maze domains.

3.3.3 Expanding to Sokoban

Unfortunately, reusing the same architecture for Sokoban domain is not fully possible. We can keep the structure the same, but with added entity in the maze, the number of channels on the input is higher. That is why we need to train another expansion network just for the Sokoban domain.

Again, we replace the goal cells with empty cells, since it does not change the characteristics of our task. We have very similar rules of movement in all our domains, but Sokoban the most complicated one. We have to learn not only, how to move the agent, but also how and when we can move the boxes.

Until now, we used only 3x3 convolutional kernels, as we can see in Figure 3.6, because it is enough to see all valid moves the agent can make. When adding the boxes, we have a lot more complicated situation. The 3x3 view is no longer enough, because we have to think about boxes and their movement. For pushing a box, it necessary to see not only the box, but also one step further, to know if we can push it. That is why we increased the kernel size and trained networks with 5x5 and 7x7 kernels. Except from this change, the structure of the network stays the same as displayed in Figure 3.6.

3.4 Heuristic Network

Even in planning, heuristic computation is a complicated task. We are trying to compute it based on a non-simplified visual problem representation, which is certainly not an easy task. We built on top of the existing expansion network in the early works on the heuristic network. Very soon, we figured that there needs to be a more complex network architecture created, in order to return a usable heuristic function.

Since the input data representation is still based on the visual representation, convolutional networks are still a good direction to explore. Since our task is very complex, we wanted to get closer to the classical planning approaches for computing heuristics. That is why we decided to use attention, to emulate simplification of the problem.

The other direction, we tried to explore were recurrent networks. We are still training the network on relatively small data samples. That gave us an idea of internal simulation of the search, which could be done by the network. Search is a sequential process, so

recurrent networks are a good candidate to approach this idea. Namely, we are talking about the reasoning network using MAC cells.

Before we create any suitable architecture, we have to reevaluate used loss function and the structure of data.

3.4.1 Loss Function and Generating Batches

We started designing the architecture with the same data set as we had in the case of expansion network, just with different labels, which were telling us the optimal heuristic for each data sample. At the very beginning we used mean-square error as a loss function, to train the network to return as close values to the optimal heuristic values as possible. After training several simple architecture using this dataset and loss function, it was obvious from the results, that there was a need of using a different approach.

The first great change is in the dataset. If we look at the dataset from human perspective, it looks very random. Each maze is different and has a number assigned to it. It would take us a very long time to figure out the relation between the maze and the number. Even after that, it would still be hard to come up with numbers to mazes we never saw.

On the other hand, if we got ten images of the same maze with different agent positions, it would be much easier to come up with a relation between the values assigned to the mazes. That is how we approached the data as well.

For each maze instance, which was located in the original data set, we randomly picked multiple positions, from all possible agent placements, and added those randomly picked samples with their computed labels into the data set. To train the network, we created the batches by taking a selected number of positions from each maze instance, so there were always multiple different agent placements in the maze at one batch.

With batches created in such manner, we then could compute the loss function for each part of the batch with the same maze structure, summing the partial losses together and returning the sum as a loss for the whole batch. Since MSE was not working very well, we created our own loss function, which was more suitable for our data and the results we are trying to achieve.

In satisficing planning, success of the planner depends highly on the heuristic function. If the functions sorts our open list well, we will expand less states and find a solution faster. In order to do so, it is not that helpful to have values close to h^* if they are not ordered well. What we need is a function, which will achieve the best ordering of the states possible. A great example of such function is a monotonic function.

To train a neural network, which acts as a monotonic function is not a trivial problem and it requires a custom loss function. The structure of our new data allows us to compute

the loss for each maze instance in the data. Meaning, that we take every sample, with the same arrangement of walls and goal and we compute the loss for all these samples. We then sum the values for all the maze instances and return it as a loss. For each maze instance, it does a pairwise comparison of the elements, to determine if their order is correct. The original order is based on the provided labels.

```
function loss(mx,y)
    partial_losses = []
    for every maze instance in batch
        ixs = all indexes of the instance
        data_diffs = mx[ixs] - transpose(mx[ixs])
        labels_diffs = y[ixs] - transpose(y[ixs])
        tmp = -data_diffs * sign(labels_diffs)+1
        l = sum(max(0, tmp)
        partial_losses.add(l)
    end
    return sum(partial_losses)
end
```

Listing 3.3: Pseudocode of loss for the heuristic network

3.4.2 Architecture Design

With a better designed data set and a new loss function, next step is creating the architecture. As mentioned before, our main focus went into recurrent networks and convolutional networks.

Recurrent Network Architecture

Our first idea in the field of recurrent networks was to use LSTM. It has been a great success in so many fields, so we thought it may benefit us as well. Unfortunately, using simple LSTM architecture was not a success and visualizations of the results were showing rather random values across the mazes.

Since the state space of planning problems can be very large, one of the intuitions behind the poor quality of the results was, that maybe the network architecture is not complicated enough. We hoped that the recurrent network could simulate the search, but that is a very complex task. Maybe, a more complex architecture could manage to do so. And on that note, we decided to explore the reasoning neural networks and implement a MAC recurrent network.

MAC recurrent network is quite different from the LSTM recurrent cell. Also, the LSTM is usually implemented in every library, which is used to implement neural networks, but in case of MAC, we had to create the whole implementation from scratch with modification, so the architecture matches the needs of our task.

The task presented in [25] is very different from ours. They have a question and an image on the input, and the output is just one of predefined answers. We have only an image, so we had to modify the input and output modules of the network.

On the input, we used a simple convolutional network, to extract features from the maze image. There is no need to use a pretrained image processing network, mostly because our images are so small, they could not be processed by it anyway. The output module is modified, so it returns a number instead of a vector.

We aimed to create a complex recurrent architecture, which we fulfilled. The outputted results from this network were better than the first ones we obtained by using LSTM. However, there were still many issues with certain data samples.

One great drawback of this architecture is the number of hyperparameters it has. Due to its complexity, it is very complicated to train. Manual changing of the hyperparameters causes the outcomes to change a lot and it does require a lot of time to train. It would require a lot more time and work in order to figure out if it is possible to train this architecture to work in our favor. The reasoning of the network can be arbitrarily long, depending on the complexity of the maze on the input. One way of training it effectively is implementing a self-stopping mechanism, which is an interesting idea for future work.

Attention Network

The key success in this architecture is due to usage of attention, as described in Section 2.3.4. The thought behind this approach also reassembles the relaxation used in planning. If we imagine looking at a maze and identifying interesting parts of it, such as crossroads or long straight paths, we might simplify the problem enough to obtain a distance estimate from the agent to the goal.

Implementation of attention was done by using convolutional layers and using softmax over first two dimensions of the input. Meaning, at the end we received the attention mask. One problem, which comes up when using attention is how many attention masks do we want in order to find enough attention-worthy places in the data. We experimented with different numbers of attentions to see if there is difference in the results.

The input of this network is still of the size $W \times H \times C \times N$, first two dimensions are width and height of the data, the third is number of channels and the last one is number of samples in the processed mini-batch. Size of the mini-batch has been deliberately omitted in the architecture diagrams for simplification.

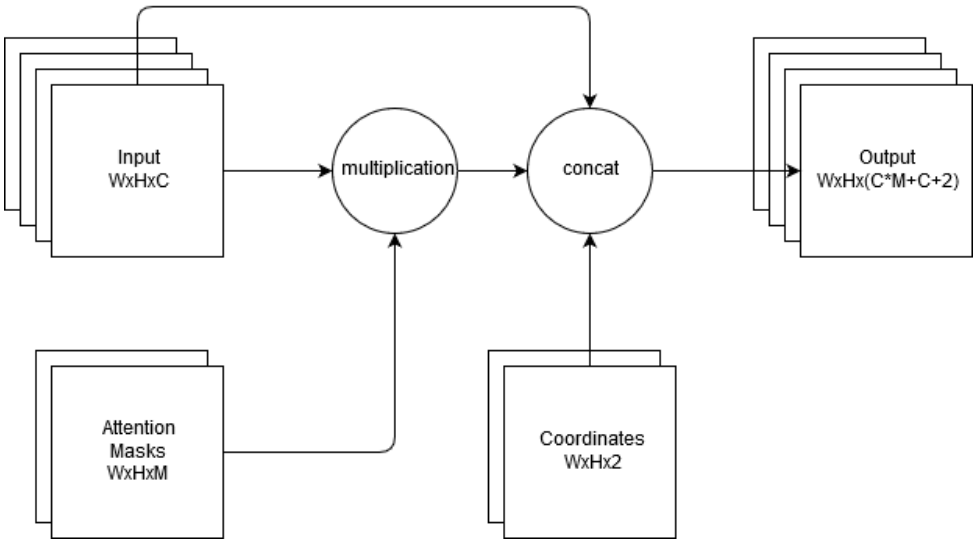

Figure 3.7: Attention block diagram

After creating M attention masks, we multiply the input data by each of the masks, concatenating the results. That results in a data tensor of size $M \times C + C + 2$. We multiply each channel of the data by each attention and concatenating the results with the original data, this multiplication is denoted in Figure 3.7 as "multiplication". After that, we add two last channels, which are x and y coordinates for the mazes. It was shown in the [32] that for learning spacial information it can be highly beneficial to provide coordinates for the data. Therefore, we added coordinates at the end of our data tensor.

This created data tensor is then processed by multiple convolutional layers with same padding, which keeps the width and height of the data the same throughout the whole network. After processing through the convolutional layers, aggregation in form of a sum is performed over the first two dimensions, creating a vector of the same size as the number of channels in the last convolutional layer. Then it is processed through a dense layer, returning one value, which is our final heuristic value for the input. The whole architecture is displayed in Figure 3.8.

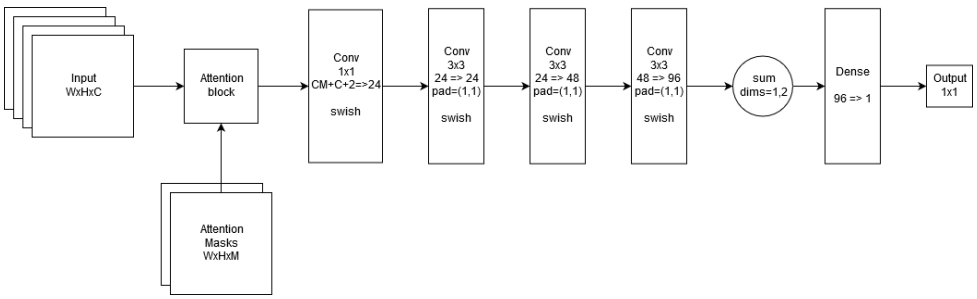

Figure 3.8: Heuristic network architecture using one attention block

That is the top level description of the architecture. We experimented it with multiple small modifications, to see if they influence the results. One modification is the number of

attention layers, which I mentioned earlier. The second modification, which we denoted as an "attention block", states, how many times we repeat creating the attention masks and the large multiplication of these with the input data. One case is using is only once, as described above, at the beginning of the network. The other case is using five of them, one between each two convolutional layers. This case is displayed in Figure 3.9.

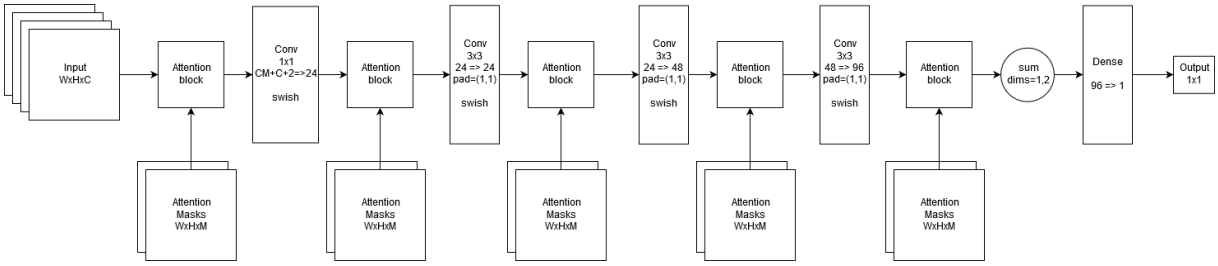

Figure 3.9: Heuristic network architecture using five attention blocks
Every attention blocks has its own set of attention masks.

3.4.3 Expanding to Multi-mazes

With the expansion network, there were not many changes when applying the network to other maze domains, since the movement rules are very similar for all three maze domains.

With the heuristic network, the situation is very different. In the multi-goal maze, we have multiple goals, so every label represents length of the path to the closest goal, because there are multiple goals in the maze. That may present a problem, because analysis of the maze structure has more to focus on, as we are not only looking at one agent position and one goal position, we need to add that deciding factor of which goal is closer to the agent.

The multi-agent maze is even more complicated, because there are not only multiple goals but also multiple agents. We need to consider placement of all agents at the same time. This scenario is also more difficult to imagine for humans as well. Once there is more than one moving entity in the maze, it becomes difficult to imagine how many steps are left until we reach the goal.

3.4.4 Expanding to Sokoban

Similar to the multi-agent maze, we have multiple entities in the maze now, which can move. We do not have to move them all at the same time, so that is an advantage, but the goal is defined differently in Sokoban than in all the other domains. We are looking at a different representation of the goal state. Also the structure of maps in Sokoban is very different to structure of mazes we talked about previously. We do not have only

narrow hallways now, we are dealing with open space, where we have to position the boxes correctly.

Labels of Sokoban map samples represent the optimal number of steps made to solve the map, however that does not depend on distance of the agent from goal, like in the maze domain. We need to also focus on the boxes and their distances from goal. Even for a human, it is much more complicated to make an estimation of number of steps necessary to solve a Sokoban level. That is why, it is an interesting problem.

Chapter 4

Experiments

Experiments in this work can be divided into two parts. The first part is training and evaluating the designed networks and the second one is evaluating these trained networks against classical planning methods.

Each of the established architectures has multiple adjustable hyperparameters. To find the right architecture setup, we trained each network with varying selection of hyperparameters, to see if any of the settings can improve results of the network.

To justify selection of the hyperparameters, it is necessary to have a metric that helps us evaluate and compare all the trained networks in order to pick the best ones. The best selected networks are further used in the planning experiments.

4.1 Network Evaluation

To compare different trained networks and evaluate their results, we implemented several evaluation functions. We focus less on the value of loss function achieved during training, and more on the actual functionality of the networks on real data. These function try to emulate functionality, which is required for the networks to perform in a classical planner.

4.1.1 Expansion Network Evaluation

For the expansion network, we implemented two evaluation functions, which were designed to check the resulting maze structure, since that is something, that the output must preserve.

The expansion network takes a grid representation of a state on the input and generates a maze with possible successors. To this end, it is important to compare the generated maze with actual state generated by the state-transition function.

The first function is used to check, whether the placement of wall cells is the same in the input and output maze. The networks generated a whole new maze base on the

input, therefore it is important to check, whether it is a maze with the same structure.

It also checks, if there are not any walls added to or deleted from the maze at the output. Number at the output of this evaluation function denotes the largest value assigned to a cell, which is not supposed to be a wall. In the evaluation tables, output of this evaluation is denoted as *wall diff*.

The other function evaluated the steps generated by the expansion network. It takes the input and runs it through the expansion network. The output maze generated by the expansion network creates a tensor of the same size as the input. The tensor consists of a one-hot encoded vector for each cell, where each vector position represents one cell type. Then it takes the input mazes and generates all possible successor states with the state-expansion function.

Now with outputs from both state-transition function and the expansion network, we can compare the successors generated from each. We first look at all the successor states from state-transition functions and note all positions, where the agent was transitioned. We look at one-hot vectors for all these cell positions and denote the smallest value, which is at this position of the vector, decoding agent cell type.

Then, we do the exact opposite and look at all the cells, which are not possible to be a valid agent placement. Now we look for the maximal possible value for agent at the one-hot vector for all these cells.

That gives us the minimal correct step and the maximal wrong step created by the expansion network. In the evaluation tables, there values are denoted as *min corr* and *max wr*. In case of Sokoban, there is also a *min corr box* and *max wr box*, which gives these values but for boxes on the map.

4.1.2 Heuristic Network Evaluation

For the heuristic network, the evaluation of the output is more complicated, since the output is just one value. The heuristic network takes a maze on the input and returns a value, which represents an estimate of the distance from the current state to the goal.

The visualization is a possibility, however in order to obtain it, we have to explore the whole state space and compute the heuristic value for every possible agent placement. This approach can be computationally very expensive for large maze instances and also the readability of the visualization gets more difficult, depending on the difference between neighboring heuristic values.

In the Figure 4.1, we can see examples of such visualization for heuristic network for maze domains. The maze on the left contains heuristic values computed for each cell and normalized. The maze on the right is the maze instance with placed goal. In the Figure 4.2, we can see the same visualization but for the multi-goal maze domain.

We decided to solve this problem by computing a sequence ranking distance. To do so, we take a problem sample, we want to evaluate the network on. We first optimally solve the problem, which leaves us with the optimal path to solution.

The path consists of all the states, which are encountered on the optimal path, so we can give each one of these states to the heuristic network and obtain their neural heuristic value. We then order them according to the computed neural heuristic values and compare the optimal ordering with this one. We do so by computing the normalized Kendall tau distance [33]. Regular Kendall tau distance, in this case, could be computed as a number of inversions in a permutation, since the first sequence is perfectly ordered sequence of numbers. However, since we need to take the length of the path in consideration, we have to normalize the value. The normalized Kendall tau distance tells us what percentage of pair from both orderings, which differ in order.

The normalized Kendall's tau K for a permutation σ is computed as

$$K(\sigma) = \frac{\sum_{(i,j):i>j}[\sigma(i) < \sigma(j)]}{\frac{|\sigma|(|\sigma|-1)}{2}}$$

In the evaluation tables, minimal and maximal Kendall tau distance are denoted as *min kt* and *max kt*.

4.2 Training the Expansion Networks

In this section, we present results obtained from trained expansion networks. We trained one network for the three maze-related domains and one for the Sokoban domain. All metrics used to evaluate success of the architectures are described in Section 4.1.1.

4.2.1 Expansion Network for Maze Domains

We trained the expansion network architecture described in Section 3.3 with 64 output channels in the first convolutional layer for 250 epochs. In Table 4.1 it is denoted as **nogoal-1** and on the regular maze data, it shows great results.

We attempted reusing the same trained network on the multi-agent data as well, but as we can see in Table 4.1, the results were not sufficient at all. We can see, that the *min corr* value is very low. That can cause ignoring possible successor states in the state-space search.

All these low values for correct steps come from possible agent collisions. Collisions of agent are not in the training data for the network, so it did not know how to properly evaluate them.

We solved this issue by combining the original maze data set with the multi-agent data set and creating the training batches from both. The resulting network **nogoal-ma-1** returned much better results, as we can see in Table 4.1.

Model	maze data			ma-maze data		
	wall diff	min corr	max wr	wall diff	min corr	max wr
nogoal-1	9.71483e-6	0.24469	6.39833f-8	0.01587	3.11338f-11	0.00053
nogoal-ma-1	4.23067e-8	0.24124	8.42643f-10	1.85346e-5	0.23101	1.5056f-7

Table 4.1: Expansion network evaluation for maze domains

The **nogoal-ma-1** trained network is used for all three maze domains in the planning experiments in Section 4.4.

4.2.2 Expansion Network for Sokoban

As we mentioned before, Sokoban requires a separate network due to its data structure. Because we are dealing with one more entity in the map, the box, we require the data to have more channels.

We keep the same architecture as before, with minor modifications of the convolutional kernel as mentioned in Section 3.4.4. In this case, we have to train the network with multiple different parameter configurations.

We parametrize the experiments with the following parameters. The first one denotes the number of channels in hidden convolutional layers. Next two denote size of the convolutional kernels and number of used convolutional layers. The last one is number of training epochs. Names of models are constructed from values of these four parameters.

We trained 17 models in total, which are evaluated on both training and test data set. Results are displayed in Table 4.2.

Evaluation shows the real problem in this task. The walls are not a big problem even for the test data samples, unlike the correct steps.

The minimal correct step values are so small, and the maximum wrong values so large, that we probably would not be able to fully rely on any of these networks. Correct placements of boxes and the agent have assigned values close to zero, which makes the steps impossible to detect by the planner. The displacements of both boxes and agent are assigned a lot higher values. This leads to not only generating invalid steps, but also to ignoring existing valid ones.

Based on these results, we can assume, that training a Sokoban expansion network is a much more complicated task than we first anticipated. Regardless, we will use a Sokoban expansion network in the second part of our experiments. The network configuration we will use is **64-5-2-250**, which showed the best results.

Model	Test data set				
	wall diff	min corr	min corr box	max wr	max wr box
128-5-2-250	0.00034	4.02678e-16	9.86054e-7	0.91806	0.38099
128-5-2-500	0.00106	1.1512e-21	3.2776e-7	0.96902	0.60831
128-5-2-750	0.0005	1.46605e-27	3.6783e-10	0.68624	0.57492
128-7-3-250	0.00029	2.12268e-38	1.04296e-10	0.98131	0.60293
128-7-3-500	2.71443e-6	3.8239e-23	4.77875e-7	0.68839	0.33603
128-7-3-750	0.00011	1.24393e-12	1.60283e-5	0.58009	0.54652
256-5-2-250	1.1452e-5	1.09583e-13	1.58407e-7	0.70511	0.41824
256-5-2-500	4.32075e-5	3.78003e-19	6.35464e-9	0.73162	0.55639
256-5-2-750	5.19998e-5	6.83472e-16	3.53193e-10	0.8935	0.58464
256-7-3-250	5.11103e-5	4.62526e-29	1.23327e-8	0.80184	0.43226
256-7-3-500	6.08975e-5	3.2576e-37	7.38266e-16	0.57952	0.60324
64-5-2-250	0.00241	1.33777e-11	4.79369e-6	0.96589	0.37592
64-5-2-500	5.04715e-6	1.52633e-22	4.95629e-7	0.99828	0.43834
64-5-2-750	1.12057e-5	5.23684e-25	2.6235e-7	0.99995	0.88743
64-7-3-250	3.70442e-5	3.19981e-18	3.50131e-8	0.47539	0.02295
64-7-3-500	1.43414e-6	5.33057e-19	9.47323e-7	0.74012	0.20341
64-7-3-750	1.62785e-5	2.37593e-37	5.94413e-8	0.77367	0.20639

Table 4.2: Expansion network evaluation for Sokoban

4.3 Training the Heuristic Networks

Heuristic network is trying to solve a much more complex task than the expansion network. Therefore, we trained multiple configurations for each of the domains. The numbers of trained configurations may differ, since we did not succeed in training all the configurations on all problems. It is usually the case with the more complicated networks, which require a lot of memory and time to train.

The trained networks were then evaluated by methods described in Section 4.1.2 and compared manually according to the results. We sampled 1000 samples from both data sets and used them to evaluate each trained network configuration.

The evaluation on train data means, that samples from the training dataset were solved by a planner and the resulting path orderings were then compared as described in 4.1.2.

4.3.1 Heuristic Network for Maze Domain

For the single-agent maze domain, we first trained prototypes of both convolutional network with attention, mentioned in Section 3.4.2 and the MAC network mentioned in Section 3.4.2.

The MAC network requires a lot of parameter tuning and adjustments, since the architecture is much more complicated and the results we were able to get were not convincing

enough in order to continue in its development. Therefore, we performed experiments with the convolutional attention network and we will leave the MAC architecture for possible future development. One direction of such development is the self-stopping learning mechanism we mentioned in Section 3.4.2.

During training, we tested multiple configurations of networks by adjusting four parameters. First one is padding, which is equal to 0 or 1 depending on whether or not we want to pad the maze with walls on the input. Then we have number of attention layers, which tells us how many attention masks is created for each attention block, which performs operation described in Section 2.3.4. Next one is number of attention blocks. If it is equal to one, the attention process is performed only at the very beginning of the network. If it is equal to 5, there is one attention block between every two convolutional layers in the network. The last parameter is number of epochs, which we consider to be 50 or 100.

In Table 4.3, we have minimal *(min kt)* and *(max kt)* maximal obtained Kendall tau distance for each of the data sets. The number in brackets next to the value is how many samples from 1000 has Kendall tau distance equal to the value.

Model	Train data		Test data	
	min kt	max kt	min kt	max kt
0-10-1-100	0.08696 (2)	1.0 (44)	0.10526 (14)	0.66667 (9)
0-10-1-50	0.0 (57)	1.0 (43)	0.0 (57)	0.66667 (4)
0-10-5-100	0.08889 (1)	1.0 (44)	0.15238 (1)	0.74546 (1)
0-10-5-50	0.0 (21)	1.0 (43)	0.0 (9)	0.66667 (8)
0-5-1-50	0.0 (480)	1.0 (18)	0.0 (480)	0.33333 (3)
0-5-5-50	0.00654 (1)	1.0 (44)	0.00952 (2)	0.66667 (9)
1-10-1-100	**0.0 (929)**	**1.0 (2)**	**0.0 (950)**	**0.04762 (3)**
1-10-1-50	0.08 (1)	1.0 (44)	0.10526 (13)	0.66667 (9)
1-10-5-100	**0.0 (321)**	**0.59048 (1)**	**0.0 (248)**	**0.55556 (2)**
1-10-5-50	0.01282 (1)	1.0 (44)	0.01905 (1)	0.66667 (9)
1-5-1-50	0.08333 (1)	1.0 (44)	0.10526 (13)	0.66667 (9)

Table 4.3: Heuristic network evaluation for maze domain

From evaluation in Table 4.3, there are two values, which stand out. The first one is *min kt* of model **1-10-1-100** on training data. There are 929 samples from 1000, which had Kendall tau distance equal to 0. That means, that all these were ordered correctly by the network and there were no inversions in permutations needed to get the correct order of the path. We can see, that the success was repeated with the test data as well and there are 950 samples with the same property. This configuration also has a very low *max kt* value for test data, which means that all orderings in the test data set were performed near perfect.

The other value is the max kt for training data of **1-10-5-100**. It is the only configu-

ration, which achieved max kt lower than one. That is actually the highest possible value of normalized Kendall tau distance, which denotes that maximum number of inversions is necessary to create the correct permutation.

We decided to take both **1-10-5-100** and **1-10-1-100** configurations and use them separately in the planner. Notice, that it is the same architecture, which differs only in the number of used attention blocks. Both have some standout results, but it is hard to tell, which one could have better performance. Therefore, we will run the planning experiments with both.

To see the resulting attention masks used in the planning experiments, we also have a visualization of them. An example for such visualization for one attention block with ten attention masks is shown in Figure 4.3. The top-left image is the problem instance on the input, next 10 images are attention masks from the first block of the heuristic network.

In the Figure 4.3, we can see, that there are various attention focuses in the masks. Some of the masks focus primarily on the area around the goal. Some of them focus more on the structure of the maze. Some of the masks look similar, however, none of them are the same. Based on that, we can assume that the number of attention masks is not too high.

4.3.2 Heuristic Network for Multi-goal Maze Domain

There are 11 trained heuristic networks for the multi-goal maze domain. We adjusted four parameters of the configuration, just as in the previous cases. First is padding of the data with walls, second two are number of attention layers in each block and number of blocks. The last one is number of epochs.

To evaluate the networks we use normalized Kendall tau values *min kt* and *max kt* described in Section 4.1.2. In Table 4.4, we can see values for all the trained networks. The one, which stands out is the **1-5-1-50** configuration. It has the highest number of samples with the minimal *min kt*, which means that about half of orderings from both train and test data set, were ordered correctly. Therefore, we use the **1-5-1-50** configuration for further planning experiments.

In Figure 4.4, we can see visualization of all the attention layers in the selected model **1-5-1-50**. The original problem instance is the top-left image. The other 5 images are the attention masks. We can tell, that two of the masks focus on the position of the agent. Those almost look like duplicates. Mask in the middle of the bottom row looks exactly like a path through the whole maze. Another one looks at the positions of goals. That shows, that the attention masks can focus on different aspects of the maze.

Model	Train data		Test data	
	min kt	max kt	min kt	max kt
0-10-1-50	0.10476 (1)	1.0 (92)	0.1 (2)	1.0 (88)
0-10-5-50	0.0 (18)	1.0 (92)	0.0 (29)	1.0 (86)
0-5-1-50	0.10526 (3)	1.0 (92)	0.09524 (1)	1.0 (88)
0-5-5-50	0.0 (28)	1.0 (92)	0.0 (34)	1.0 (87)
1-10-1-50	0.0 (23)	1.0 (119)	0.0 (21)	1.0 (125)
1-10-5-50	0.0 (29)	1.0 (83)	0.0 (20)	1.0 (90)
1-5-1-50	**0.0 (492)**	**1.0 (48)**	**0.0 (497)**	**1.0 (55)**
1-5-5-50	0.0 (445)	1.0 (57)	0.0 (437)	1.0 (73)
0-5-1-100	0.11111 (4)	1.0 (97)	0.1 (1)	1.0 (99)
1-5-1-100	0.0 (30)	1.0 (131)	0.0 (30)	1.0 (135)
1-10-1-100	0.0 (25)	1.0 (119)	0.0 (29)	1.0 (120)

Table 4.4: Heuristic network evaluation for multi-goal maze domain

4.3.3 Heuristic Network for Multi-agent Maze Domain

There are 9 trained networks for the multi-agent maze domain. The training configurations are parametrized by four values. There is a parameter for padding the mazes with walls, parameter for number of attention layers in a block, number of attention blocks and a number of epochs. All results are in Table 4.5. In this case, a standout network configuration is **0-10-1-100**, as is has the highest number of samples with *min kt* value in both train and test data set. We use this network configuration further in the planning experiments.

Model	Train data		Test data	
	min kt	max kt	min kt	max kt
0-10-1-100	**0.0 (28)**	**1.0 (5)**	**0.0 (32)**	**1.0 (4)**
0-10-1-50	0.0 (26)	1.0 (5)	0.0 (18)	1.0 (4)
0-5-1-100	0.0 (4)	1.0 (16)	0.0 (5)	1.0 (6)
0-5-1-50	0.0 (9)	1.0 (13)	0.0 (5)	1.0 (6)
1-10-1-100	0.0 (5)	1.0 (10)	0.0 (3)	1.0 (7)
1-10-1-50	0.0 (1)	1.0 (10)	0.0 (1)	1.0 (7)
1-10-5-50	0.01515 (1)	1.0 (11)	0.0 (1)	1.0 (8)
1-5-1-100	0.0 (1)	1.0 (13)	0.0 (2)	1.0 (12)
1-5-1-50	0.03297 (1)	1.0 (15)	0.0 (1)	1.0 (10)

Table 4.5: Heuristic network evaluation for multi-agent maze domain

In Figure 4.5, we can see visualization of all attention masks applied on an input maze. From the images, we can tell that the network focuses on locations of agents, as well as on locations of goal. Several of the masks seem to focus more on wall placement and the structure of the maze.

4.3.4 Heuristic Network for Sokoban Domain

For the Sokoban domain, there are 5 trained networks. Unlike the maze domains, here we do not use the wall padding parameter, since all the Sokoban maps are padded by default.

The two possible parameters are number of attention blocks and number of attention layers in one attention block. All networks were trained for 50 epochs. Evaluation using Kendall tau distance is showed in Table 4.6. The *min kt* and *max kt* denote minimal and maximal Kendall tau distances achieved on the evaluation data. Number in brackets next to that says, for how many samples was this value computed.

Model	Train data		Test data	
	min kt	max kt	min kt	max kt
10-1-50	0.0 (118)	0.75 (1)	0.0 (116)	0.61905 (1)
10-5-50	**0.0 (266)**	**0.39286 (1)**	**0.0 (226)**	**0.45455 (1)**
20-1-50	0.0 (237)	0.5 (2)	0.0 (210)	0.52381 (1)
5-1-50	0.0 (106)	0.6 (1)	0.0 (103)	0.6 (2)
5-5-50	0.0 (148)	0.51111 (1)	0.0 (142)	0.50909 (1)

Table 4.6: Heuristic network evaluation for Sokoban domain

Every configuration in Table 4.6 achieved *min kt* equal to zero for both train and test data. The *max kt* values did not reach 1.0 at any of the configurations. That is something we saw repeatedly in the maze domains.

To select a network for further experiments, let's look at the counts next to Kendall tau distance values. The configuration **10-5-50** has overall lowest values for both train and test data. Also, the number of samples with the lowest Kendall tau distance is the largest of all the configurations. Therefore, we use this trained network in further experiments.

In Figure 4.6, we can see attention masks from the first attention block applied on an input Sokoban map. We can see, that there is a mask, which looks directly at placement of the goals and another one, which focuses on the boxes. Other ones focus on the whole map and highlight different groups of cells.

4.4 Planning Experiments

In sections 4.2 and 4.3 we selected expansion networks and heuristic network to use in the planning experiments. There are two expansion networks, one for Sokoban and one for the all three maze-related domains. There are two heuristic networks for maze domain, since each of them had different positive properties, we decided to use both. There is one heuristic network for multi-goal maze, one for multi-agent maze and one for Sokoban domain.

We are using domain dependent planners, because of the visual representation of problems, as stated in Section 3.2. For each domain, we implemented a planner with the same algorithms and the same heuristics. There are three state-space search algorithms, greedy best-first search (Section 2.2.1), history considering best-first search (Section 2.2.2) and multi-heuristic search using tie-breaking (Section 2.2.3). In the multi-heuristic search, we can select as many heuristics as desired. The other two algorithms support using only one heuristic at a time.

For heuristic, we can select none, which always returns zero, Euclidean distance, H^{FF} heuristic (Section 2.2.4), LM-cut (Section 2.2.4) and the heuristic neural network. For use in multi-heuristic search, they can be arbitrarily combined.

For each problem domain, we created 50 new problem instances, which are not in any of the training data sets we used earlier. To compare the performance of all the planner configurations, we measured length of the output plans and number of expanded states during the search. There is also coverage denoted as cvg, which says percentage of solved problems from the set of 50.

4.4.1 Planning Experiments for Maze Domain

With selected expansion network and heuristic network, we finished experiments for the maze domain with results shown in Table 4.7. As mentioned in Section 4.3.1, we selected two heuristic networks for this domain due to their results. In the Table 4.7, configuration **1-10-5-100** is denoted as **nn** and **1-10-1-100** is denoted as **nn2**.

We can see, that using the expansion network doesn't change the coverage of any configurations. Therefore, we can assume that is generates the successor states correctly, since it's possible to find solutions to all problems.

In GBFS experiments, LM-cut has the shortest paths together with Euclidean heuristic, and the least expanded states. The next heuristic function after that is the heuristic network **1-10-5-100**. That means, that is outperformed the zero heuristic and h^{FF} heuristic as well.

In HCBFS experiments, all the configurations, except the **1-10-1-100** heuristic network configuration (denoted nn2), achieved the same results for the "Path len" values. For the "Expanded states" values, best is LM-cut and second is the heuristic network. That shows, that it outperforms every other heuristic in the average number of expanded states, except LM-cut. That holds for both expansion network and state-transition function.

In multi-heuristic search experiments, LM-cut achieved the best results with every other heuristic as a tie-breaker. From all configurations with heuristic network as the primary heuristic, **1-10-1-100** has overall best results when used with LM-cut as a tie-breaker.

Search	Heuristic	Default expansion							Expansion network						
		Path len			Expanded states				Path len			Expanded states			
		avg	min	max	avg	min	max	cvg	avg	min	max	avg	min	max	cvg
gbfs	none	11.64	4.0	52.0	27.42	6.0	63.0	1.0	10.88	4.0	38.0	23.48	7.0	38.0	1.0
gbfs	eucl	10.76	4.0	38.0	14.58	4.0	38.0	1.0	10.76	4.0	38.0	14.2	4.0	38.0	1.0
gbfs	hff	11.64	4.0	52.0	27.42	6.0	63.0	1.0	10.88	4.0	38.0	23.48	7.0	38.0	1.0
gbfs	lmcut	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0
gbfs	nn	10.8	4.0	38.0	14.12	4.0	38.0	1.0	10.8	4.0	38.0	14.12	4.0	38.0	1.0
gbfs	nn2	10.92	4.0	38.0	35.16	30.0	62.0	1.0	10.84	4.0	38.0	35.16	30.0	62.0	1.0
hcbfs	none	10.76	4.0	38.0	33.1	12.0	112.0	1.0	10.76	4.0	38.0	32.78	11.0	112.0	1.0
hcbfs	eucl	10.76	4.0	38.0	20.9	5.0	90.0	1.0	10.76	4.0	38.0	20.86	5.0	88.0	1.0
hcbfs	hff	10.76	4.0	38.0	33.1	12.0	112.0	1.0	10.76	4.0	38.0	32.78	11.0	112.0	1.0
hcbfs	lmcut	10.76	4.0	38.0	12.98	5.0	68.0	1.0	10.76	4.0	38.0	12.2	5.0	39.0	1.0
hcbfs	nn	10.76	4.0	38.0	20.38	5.0	84.0	1.0	10.76	4.0	38.0	20.38	5.0	84.0	1.0
hcbfs	nn2	10.84	4.0	38.0	37.66	31.0	109.0	1.0	10.84	4.0	38.0	37.66	31.0	109.0	1.0
mh-gbfs	eucl, hff	10.76	4.0	38.0	14.58	4.0	38.0	1.0	10.76	4.0	38.0	14.2	4.0	38.0	1.0
mh-gbfs	eucl, lmcut	10.76	4.0	38.0	13.42	4.0	38.0	1.0	10.76	4.0	38.0	13.42	4.0	38.0	1.0
mh-gbfs	eucl, nn	10.76	4.0	38.0	13.84	4.0	38.0	1.0	10.76	4.0	38.0	13.84	4.0	38.0	1.0
mh-gbfs	eucl, nn2	10.76	4.0	38.0	14.58	4.0	38.0	1.0	10.76	4.0	38.0	14.2	4.0	38.0	1.0
mh-gbfs	hff, eucl	10.76	4.0	38.0	14.58	4.0	38.0	1.0	10.76	4.0	38.0	14.2	4.0	38.0	1.0
mh-gbfs	hff, lmcut	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0
mh-gbfs	hff, nn	10.8	4.0	38.0	14.12	4.0	38.0	1.0	10.8	4.0	38.0	14.12	4.0	38.0	1.0
mh-gbfs	hff, nn2	11.64	4.0	52.0	27.42	6.0	63.0	1.0	10.88	4.0	38.0	23.48	7.0	38.0	1.0
mh-gbfs	lmcut, eucl	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0
mh-gbfs	lmcut, hff	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0
mh-gbfs	lmcut, nn	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0
mh-gbfs	lmcut, nn2	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0
mh-gbfs	nn, eucl	10.8	4.0	38.0	14.1	4.0	38.0	1.0	10.8	4.0	38.0	14.1	4.0	38.0	1.0
mh-gbfs	nn, hff	10.8	4.0	38.0	14.12	4.0	38.0	1.0	10.8	4.0	38.0	14.12	4.0	38.0	1.0
mh-gbfs	nn, lmcut	10.8	4.0	38.0	14.1	4.0	38.0	1.0	10.8	4.0	38.0	14.1	4.0	38.0	1.0
mh-gbfs	nn2, eucl	10.76	4.0	38.0	14.58	4.0	38.0	1.0	10.76	4.0	38.0	14.2	4.0	38.0	1.0
mh-gbfs	nn2, hff	11.64	4.0	52.0	27.42	6.0	63.0	1.0	10.88	4.0	38.0	23.48	7.0	38.0	1.0
mh-gbfs	nn2, lmcut	10.76	4.0	38.0	10.76	4.0	38.0	1.0	10.76	4.0	38.0	10.76	4.0	38.0	1.0

Table 4.7: Planning experiments for maze domain

4.4.2 Planning Experiments for Multi-goal Maze Domain

Results for the multi-goal maze domain are shown in Table 4.8. With a few minor exceptions, results for regular transition function and the expansion network are very similar. The coverage achieved with both transition functions is equal to 1.0, therefore all problems were solved with every planner configuration.

In case of GBFS, heuristic network achieves best results together with LM-cut and h^{FF} heuristics. In case of HCBFS, it even outperforms them in the number of expanded states, using both expansion network and regular state-transition function.

In the multi-heuristic search, majority of the configurations achieved the same results, which can be classified as the best. Every configuration with heuristic network as the primary heuristic achieved those results as well. That shows us, that heuristic network outperformed Euclidean heuristic at the primary position at the multi-heuristic search. As a tie-breaker, heuristic network performed well with h^{FF}, LM-cut, but not Euclidean

heuristic.

Search	Heuristic	Default expansion							Expansion network						
		Path len			Expanded states				Path len			Expanded states			
		avg	min	max	avg	min	max	cvg	avg	min	max	avg	min	max	cvg
gbfs	none	6.7	1.0	26.0	14.38	1.0	31.0	1.0	6.88	1.0	26.0	14.26	1.0	32.0	1.0
gbfs	eucl	5.92	1.0	26.0	7.12	1.0	26.0	1.0	5.92	1.0	26.0	7.2	1.0	26.0	1.0
gbfs	hff	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
gbfs	lmcut	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
gbfs	nn	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
hcbfs	none	5.88	1.0	26.0	18.18	2.0	78.0	1.0	5.88	1.0	26.0	18.34	2.0	77.0	1.0
hcbfs	eucl	5.88	1.0	26.0	10.72	2.0	34.0	1.0	5.88	1.0	26.0	10.86	2.0	39.0	1.0
hcbfs	hff	5.88	1.0	26.0	7.86	2.0	37.0	1.0	5.88	1.0	26.0	7.68	2.0	28.0	1.0
hcbfs	lmcut	5.88	1.0	26.0	7.86	2.0	37.0	1.0	5.88	1.0	26.0	7.68	2.0	28.0	1.0
hcbfs	nn	5.88	1.0	26.0	6.88	2.0	27.0	1.0	5.88	1.0	26.0	6.88	2.0	27.0	1.0
mh-gbfs	eucl, hff	5.92	1.0	26.0	6.88	1.0	26.0	1.0	5.92	1.0	26.0	6.88	1.0	26.0	1.0
mh-gbfs	eucl, lmcut	5.92	1.0	26.0	6.88	1.0	26.0	1.0	5.92	1.0	26.0	6.88	1.0	26.0	1.0
mh-gbfs	eucl, nn	5.92	1.0	26.0	6.88	1.0	26.0	1.0	5.92	1.0	26.0	6.88	1.0	26.0	1.0
mh-gbfs	hff, eucl	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	hff, lmcut	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	hff, nn	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	lmcut, eucl	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	lmcut, hff	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	lmcut, nn	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	nn, eucl	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	nn, hff	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0
mh-gbfs	nh, lmcut	5.88	1.0	26.0	5.88	1.0	26.0	1.0	5.88	1.0	26.0	5.88	1.0	26.0	1.0

Table 4.8: Planning experiments for multi-goal maze domain

4.4.3 Planning Experiments for Multi-agent Maze Domain

Results of planning experiments for multi-agent maze domain are shown in Table 4.9.

Using the expansion network results in the same coverage as in the case of regular state-transition network. Therefore, we can assume, that the expansion network works correctly, since every problem was solved.

Using the heuristic network does not bring any significant improvement for any single-heuristic search. The only heuristic outperformed is usually the zero heuristic. In the multi-heuristic search, the one case, where heuristic network benefits the results is when using is as a tie-breaker with the h^{FF} heuristic, with the default state-transition function. That decreases both lengths of found solutions and expanded states.

4.4.4 Planning Experiments for Sokoban Domain

Planning experiments for Sokoban domain are shown in Table 4.10. The number of entries is smaller due to computational resources. We did not get any results for LM-cut and H^{FF} heuristics.

Search	Heuristic	Default expansion							Expansion network						
		Path len			Expanded states				Path len			Expanded states			
		avg	min	max	avg	min	max	cvg	avg	min	max	avg	min	max	cvg
gbfs	none	93.92	3.0	231.0	237.0	5.0	649.0	1.0	62.06	5.0	136.0	170.28	8.0	367.0	1.0
gbfs	eucl	12.56	1.0	40.0	35.28	1.0	234.0	1.0	12.26	1.0	40.0	24.04	1.0	164.0	1.0
gbfs	hff	10.5	1.0	30.0	15.28	1.0	43.0	1.0	10.34	1.0	30.0	14.72	1.0	51.0	1.0
gbfs	nn	15.42	1.0	38.0	171.12	1.0	505.0	1.0	11.42	1.0	34.0	26.94	1.0	96.0	1.0
hcbfs	none	8.9	1.0	18.0	485.14	5.0	934.0	1.0	8.9	1.0	18.0	350.08	4.0	684.0	1.0
hcbfs	eucl	8.92	1.0	18.0	83.08	2.0	294.0	1.0	8.92	1.0	18.0	73.36	2.0	281.0	1.0
hcbfs	hff	8.9	1.0	18.0	50.98	2.0	273.0	1.0	8.9	1.0	18.0	45.3	2.0	211.0	1.0
hcbfs	nn	9.2	1.0	20.0	168.74	2.0	906.0	1.0	9.16	1.0	18.0	92.98	2.0	413.0	1.0
mh-gbfs	eucl, hff	12.3	1.0	40.0	30.56	1.0	230.0	1.0	12.3	1.0	40.0	30.56	1.0	230.0	1.0
mh-gbfs	eucl, nn	12.36	1.0	40.0	33.88	1.0	230.0	1.0	12.12	1.0	40.0	25.5	1.0	164.0	1.0
mh-gbfs	hff, eucl	10.46	1.0	30.0	11.88	1.0	45.0	1.0	10.46	1.0	30.0	11.84	1.0	45.0	1.0
mh-gbfs	hff, nn	9.7	1.0	24.0	12.94	1.0	59.0	1.0	9.66	1.0	24.0	11.38	1.0	36.0	1.0
mh-gbfs	nn, eucl	15.42	1.0	38.0	171.12	1.0	505.0	1.0	11.42	1.0	34.0	26.94	1.0	96.0	1.0
mh-gbfs	nn, hff	15.42	1.0	38.0	171.12	1.0	505.0	1.0	11.42	1.0	34.0	26.94	1.0	96.0	1.0

Table 4.9: Planning experiments for multi-agent maze domain

Search	Heuristic	Default expansion							Expansion network						
		Path len			Expanded states				Path len			Expanded states			
		avg	min	max	avg	min	max	cvg	avg	min	max	avg	min	max	cvg
gbfs	none	111.24	5.0	351.0	3495.22	26.0	62388.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
gbfs	eucl	31.1	5.0	64.0	484.1	8.0	10038.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
gbfs	nn	27.32	5.0	54.0	1328.62	11.0	19092.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
hcbfs	none	23.34	5.0	45.0	3919.6	18.0	81238.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
hcbfs	eucl	23.34	5.0	45.0	2148.8	11.0	43823.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
hcbfs	nn	24.3	5.0	48.0	1723.16	11.0	38089.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
mh-gbfs	eucl, nn	29.44	5.0	62.0	369.82	5.0	4587.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0
mh-gbfs	nn, eucl	27.98	5.0	54.0	1360.78	11.0	19092.0	1.0	Inf	Inf	Inf	5.0	1.0	13.0	0.0

Table 4.10: Planning experiments for Sokoban maze domain

As expected, the expansion network did not work very well as the transition function, as the coverage of all configuration equals to 0.0. Maximum number of expanded valid states for a problem is 13, which means, that some of the successor states were found correctly. However, not enough were found in order to find a solution to any of the problems.

Now we focus on results from experiments with regular state-transition function. Coverage for all the configurations is 1.0, which means, that all problem instances were solved. For GBFS, we can see that using heuristic network leads to obtaining the shortest paths. However, we do not outperform any other heuristic in the number of expanded states. For HCBFS, path lengths differ by one on average, but the number of expanded states is significantly lower.

In multi-heuristic search, we have configuration with heuristic network and Euclidean heuristic in both orders. The *eucl, nn* configuration returns slightly longer paths, however, the maximum number of expanded states is the lowest from the whole table. That shows, that using heuristic network as a tie-breaking heuristic returns impressive results.

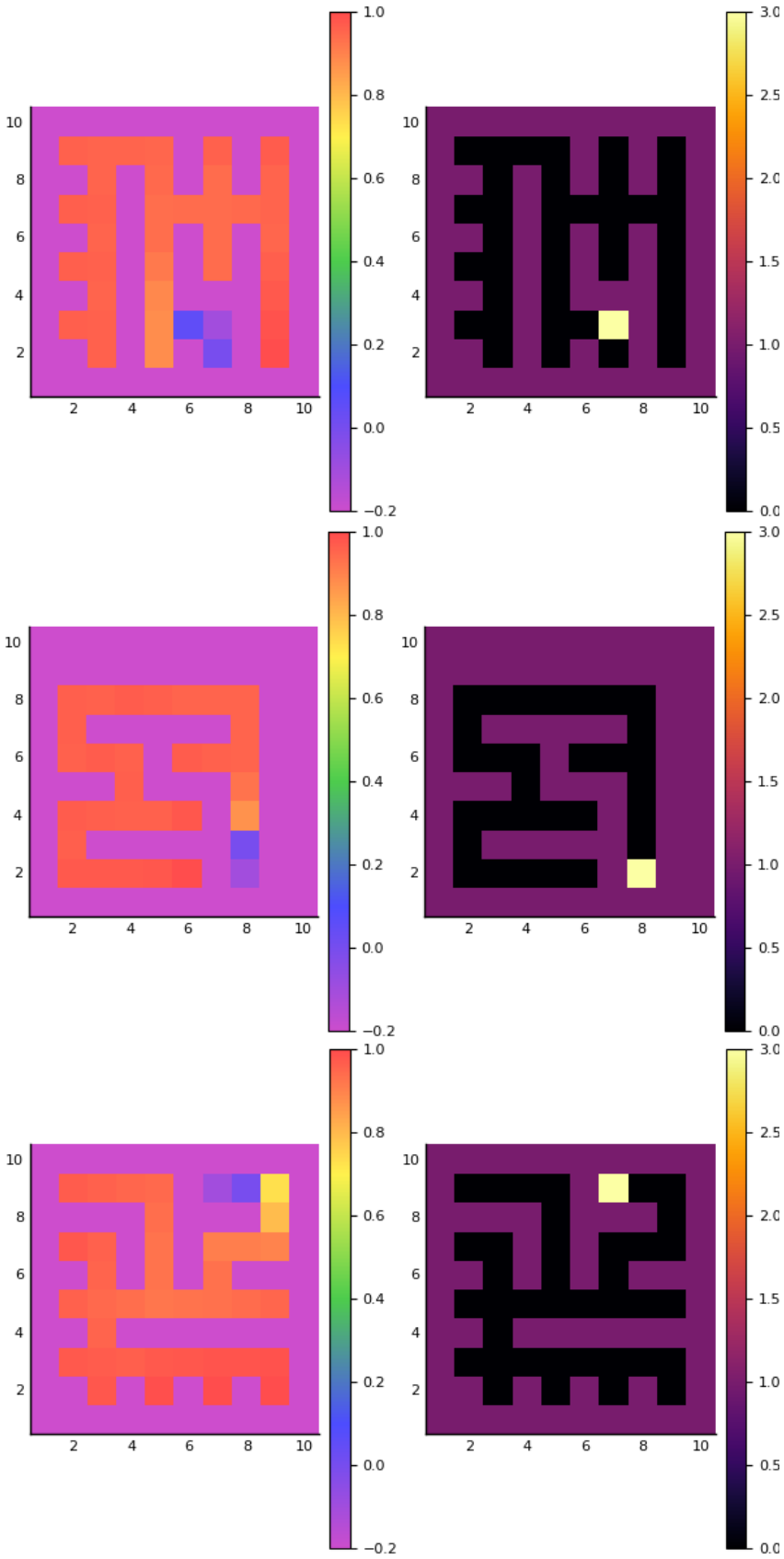

Figure 4.1: Example of visualization of heuristic network output for maze domain
Plots on the left show visualization of output values of the heuristic network for each of the cells. Plots on the right are generate mazes with a goal placed in the position denoted by yellow.

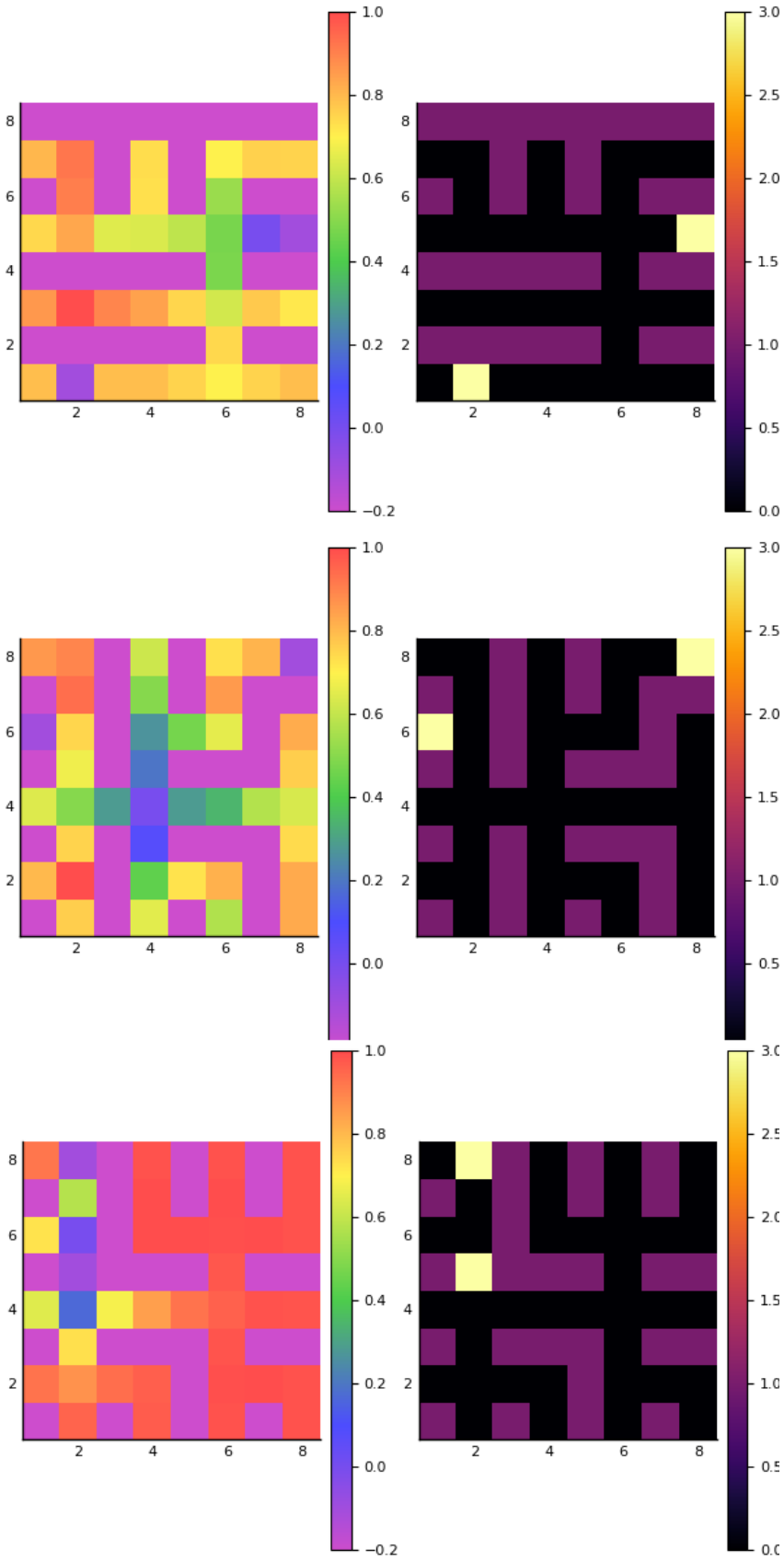

Figure 4.2: Example of visualization of heuristic network output for multi-goal maze domain

Plots on the left show visualization of output values of the heuristic network for each of the cells. Plots on the right are generate mazes with a goals placed in the positions denoted by yellow.

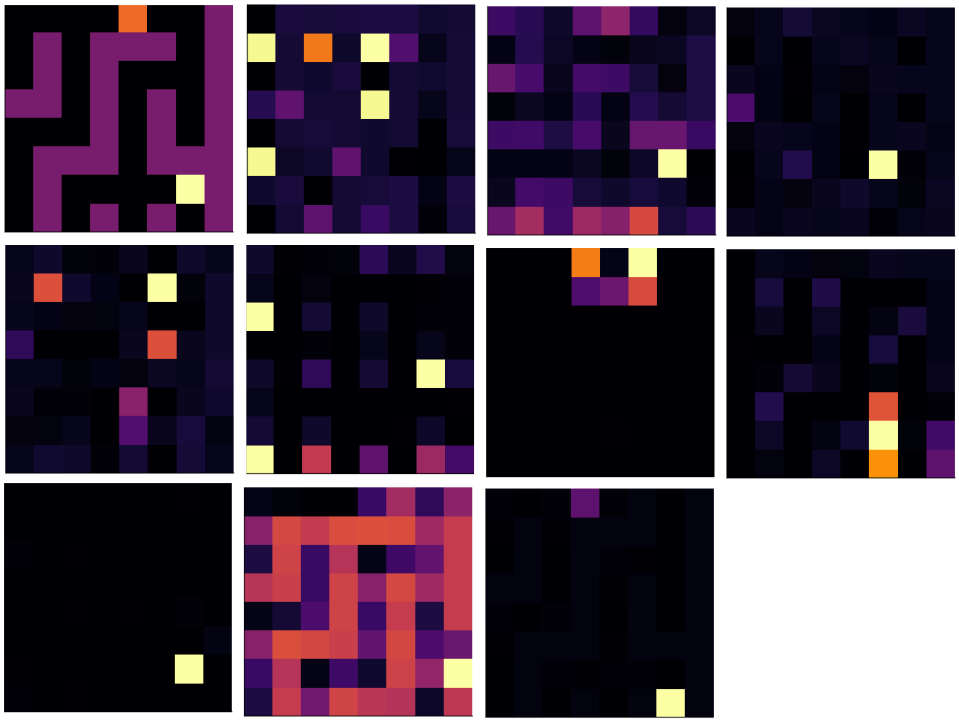

Figure 4.3: Attention mask visualization for maze domain
The top-left image is the problem instance on the input. The rest of the images are attention masks, from **1-10-5-100** heuristic network, applied on the input maze.

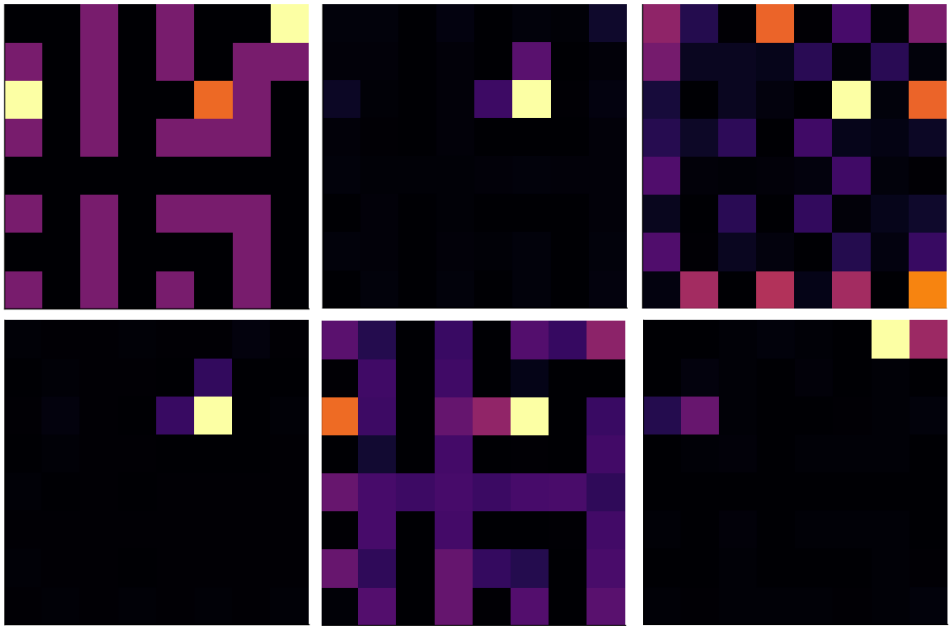

Figure 4.4: Attention mask visualization for multi-goal maze domain
The top-left image is the problem instance on the input. The rest of the images are attention masks applied on the input maze.

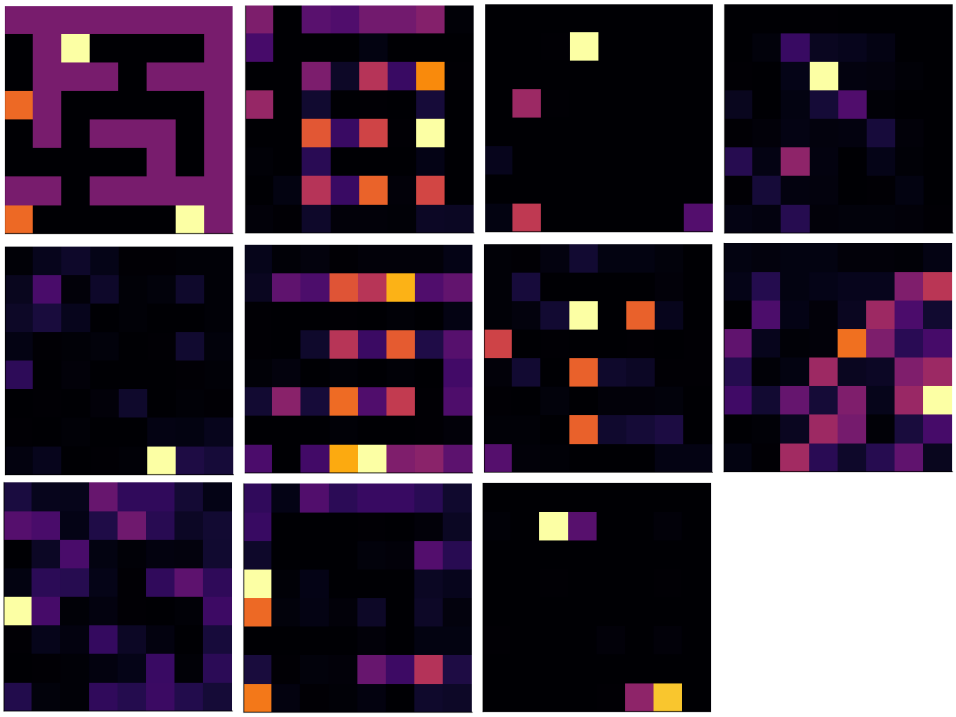

Figure 4.5: Attention mask visualization for multi-agent maze domain
The top-left image is the problem instance on the input. The rest of the images are attention masks applied on the input maze.

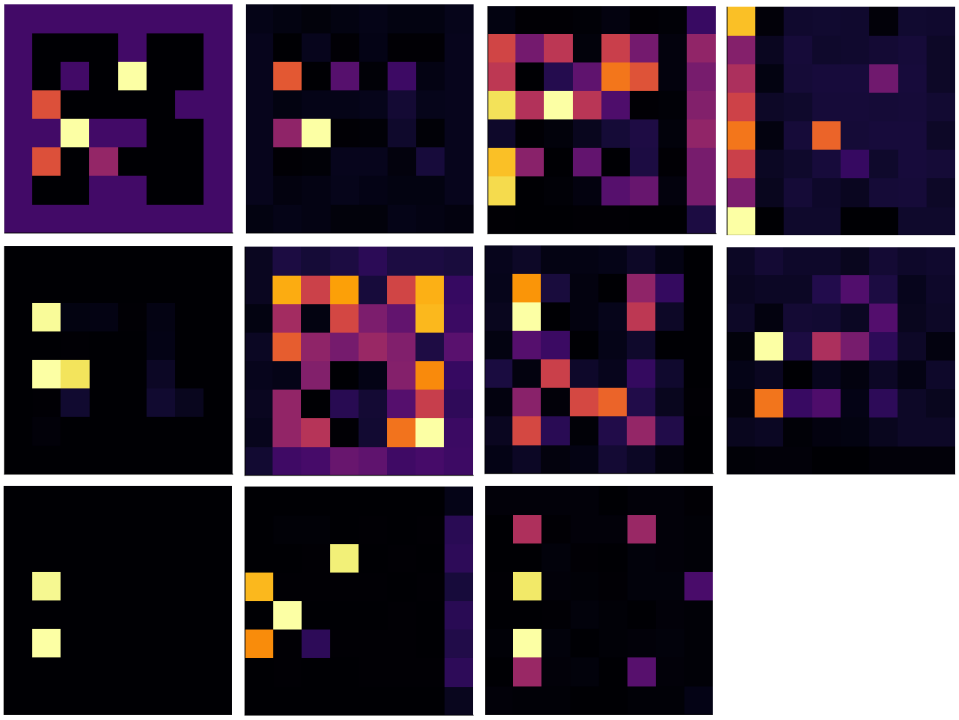

Figure 4.6: Attention mask visualization for Sokoban domain
The top-left image is the problem instance on the input. The rest of the images are attention masks applied on the input map.

Chapter 5

Result Discussion

We stated several hypotheses over the course of this work. First is, that it is possible to use expansion network as a transition function for all stated domains. Next one is, that it is possible to use heuristic network as a heuristic function in all stated domains and that it will outperform zero and Euclidean heuristic.

We created all possible networks in order to use them in a classical planning state-space search algorithm for all stated domains. However, not all the networks provided us with successful results. One example of that is the expansion network for the Sokoban domain. As shown in Table 4.10, when using expansion network as a state-transition function, no problems were solved. That was expected, since results from Table 4.2 show, that none of the trained networks was capable of generating only correct successor states.

On the other hand, using expansion network for all other three domains provided us with solutions to all problems in maze, multi-goal maze and multi-agent maze domains. The coverage of experiments using expansion network is the same as when using regular state-transition function.

Using heuristic network as a heuristic function also shows several interesting results. Heuristic network managed to outperform zero heuristic in every single-heuristic search across all domains.

In maze domain, the heuristic network outperformed zero and h^{FF} heuristic, and performed as well as Euclidean heuristic, with the difference of 0.4 average path length in case of GBFS. The only heuristic function, which provided significantly better results for this domain is LM-cut.

The greatest success was the multi-goal maze domain. In HCBFS, it outperformed all other heuristics. In GBFS, it outperformed zero and Euclidean heuristic and had the exact same results as h^{FF} and LM-cut.

In the multi-agent domain, the only outperformed function is zero heuristic. This can be caused by the higher complexity of this problem domain compared to the others.

In Sokoban domain, heuristic network managed to outperform only zero heuristic. A noticeable success is the amount of expanded states in HCBFS, which was a lot lower for the heuristic network, than for the other heuristic functions.

This shows us, that in different domains, heuristic network can be used as a heuristic estimate, which gives good amount of information to the search. Results of search with heuristic network sometimes outperformed classical planning heuristic functions and many times performed just as well.

Chapter 6

Conclusion

Main goal of this thesis was showing, that a combination of symbolic AI and model free machine learning can be created by replacing parts of a classical planning algorithm by neural networks.

We implemented an expansion network for three maze domains and one for the Sokoban domain. We used this network, instead of a traditional state-transition function. We implemented four heuristic networks, one for each domain. We then used these networks as heuristic functions in planning experiments and achieved results comparable to those achieved by classical planning methods. All of these results were obtained by experimental evaluation of the implemented solutions.

In the future, there is still more to consider. There's a possibility of expanding to more domains, scaling the approaches and continuing ideas, that weren't fully closed. The results we obtained show, that it is a possible direction, which deserves to be noticed and explored further.

Appendix A

Attachments

Enclosed to this thesis is an archive with trained networks and Julia source files. We included all trained networks used in Section 4.4.

Due to the size, we didn't include generated data sets used to train the networks. However, we did provide all the scripts necessary to generate the data.

Next, we provide the implementation of MAC reasoning network and files used to train all the expansion and heuristic networks.

Here's description of the structure of the archive.

- expansion_networks
  - maze8_resnet_composer_nogoal_ma_1.bson - maze expansion network pt. 1
  - maze8_resnet_model_nogoal_ma_1.bson - maze expansion network pt. 2
  - sokoban_64_5_2_250_composer.bson - Sokoban expansion network pt. 1
  - sokoban_64_5_2_250_model.bson - Sokoban expansion network pt. 2
- heuristic_networks
  - ma_maze_model - heuristic network for multi-agent maze
    - att1-10.bson - attention layers
    - model.bson
  - maze_model - heuristic network for maze
    - 1-10-1-100
      - att1-10.bson - attention layers
      - model.bson
    - 1-10-5-100
      - att1-50.bson - attention layers

        - model.bson
    - mg_maze_model - heuristic network for multi-goal maze
        - att1-5.bson
        - model.bson
    - sokoban_model - heuristic network for Sokoban
        - att1-50.bson
        - model.bson
- planners - planners for all domains used in planning experiments
    - solver_ma_maze.jl
    - solver_maze.jl
    - solver_mg_maze.jl
    - solver_sokoban.jl
- sokoban_level_generator
    - sokoban_all_step_generator.jl - data generator for Sokoban expansion network
    - sokoban_generator.jl - generator of solvable Sokoban levels
    - sokoban_solver.jl
    - templates.jl
- custom_mazes.jl - hand-written mazes
- heur_att_load.jl - loader for heuristic networks
- heur_att_run.jl - script used to train heuristic network for maze domain
- heur_att_run_ma.jl - script used to train heuristic network for multi-agent domain
- heur_att_run_mg.jl - script used to train heuristic network for multi-goal domain
- heur_att_run_sokoban.jl - script used to train heuristic network for Sokoban domain
- heur_mac_network.jl - script for training MAC network
- MAC_network.jl - implementation of MAC network
- maze8_expansion.jl - expansion network for maze domains
- mazegen_prim.jl - data generator for maze domain

- multimazegen.jl - data generator for multi-goal and multi-agent maze domains
- sokoban_expansion_run.jl - script used to train expansion network for Sokoban domain
- utils_heur.jl - utility functions

Bibliography

[1] M. Asai and A. Fukunaga, "Classical Planning in Deep Latent Space: From Unlabeled Images to PDDL (and back).", in *NeSy*, 2017.

[2] ——, "Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary", in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[3] C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez, "Learning to rank for synthesizing planning heuristics", *arXiv preprint arXiv:1608.01302*, 2016.

[4] P. Gomoluch, D. Alrajeh, A. Russo, and A. Bucchiarone, "Learning Neural Search Policies for Classical Planning", *arXiv preprint arXiv:1911.12200*, 2019.

[5] M. Ghallab, D. Nau, and P. Traverso, *Automated planning and acting*. Cambridge University Press, 2016.

[6] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving", *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[7] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[8] J. Pearl, "Heuristics: intelligent search strategies for computer problem solving", 1984.

[9] G. Röger and M. Helmert, "The more, the merrier: Combining heuristic estimators for satisficing planning", in *Twentieth International Conference on Automated Planning and Scheduling*, 2010.

[10] S. Richter and M. Westphal, "The LAMA planner: Guiding cost-based anytime planning with landmarks", *Journal of Artificial Intelligence Research*, vol. 39, pp. 127–177, 2010.

[11] J. Hoffmann, "FF: The fast-forward planning system", *AI magazine*, vol. 22, no. 3, pp. 57–57, 2001.

[12] F. Pommerening and M. Helmert, "Incremental LM-cut", in *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[13] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[16] B. C. Csáji, "Approximation with artificial neural networks", *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 24, p. 48, 2001.

[17] K. Hornik, "Approximation capabilities of multilayer feedforward networks", *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[18] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function", *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[20] I. S. M. Mohamed, "Detection and tracking of pallets using a laser rangefinder and machine learning techniques", PhD thesis, Master's thesis, European Master on Advanced Robotics Plus (EMARO+ ..., 2017.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need", in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] A. Graves, M. Liwicki, H. Bunke, J. Schmidhuber, and S. Fernández, "Unconstrained on-line handwriting recognition with recurrent neural networks", in *Advances in neural information processing systems*, 2008, pp. 577–584.

[24] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks", in *2013 IEEE international conference on acoustics, speech and signal processing*, IEEE, 2013, pp. 6645–6649.

[25] D. A. Hudson and C. D. Manning, "Compositional attention networks for machine reasoning", *arXiv preprint arXiv:1803.03067*, 2018.

[26] H. Geffner, "Model-free, model-based, and general intelligence", *arXiv preprint arXiv:1806.02308*, 2018.

[27] R. C. Prim, "Shortest connection networks and some generalizations", *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[28] J. Taylor and I. Parberry, "Procedural generation of sokoban levels", in *Proceedings of the International North American Conference on Intelligent Games and Simulation*, 2011, pp. 5–12.

[29] J. Culberson, "Sokoban is PSPACE-complete", 1997.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[31] *FluxML Github Repository*, 2020. [Online]. Available: `https://github.com/FluxML/Flux.jl`.

[32] X. Wei, I. A. Bârsan, S. Wang, J. Martinez, and R. Urtasun, "Learning to Localize Through Compressed Binary Maps", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 316–10 324.

[33] M. G. Kendall, "A new measure of rank correlation", *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.