Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Making Classical Planning Domains Available for Deep Neural Network Training

Bachelor Thesis

Radovan Tomala

Programme: Open Informatics
Branch of study: Artificial Intelligence and Computer Science
Supervisor: Ing. Michaela Urbanovská

Prague, May 2021

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Tomala Radovan** Personal ID number: **483662**

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

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Making Classical Planning Domains Available for Deep Neural Network Training

Bachelor's thesis title in Czech:

Zpřístupnění domén klasického plánování pro učení neuronových sítí

Guidelines:

The student will build on the work introduced in "Learning Classical Planning Transition Functions by Deep Neural Networks" (M. Urbanovská, 2020). The task will be selecting suitable classical planning problem domains in order to use them with the proposed Deep Neural Network (DNN) architectures. That includes creating generators for each of the domains for training the "expansion network" and the "heuristic network". Next task will be comparing performance of the DL approaches with methods used in classical planning. To do so, the student has to implement a classical planning framework which allows them to use the trained networks and deploy experiments with both the trained DNNs and classical planning methods.
1. Study state of the art and classical planning domains available and select two to three suitable domains.
2. Implement data generators for the selected domains in order to train "expansion network" and "heuristic network" architectures.
3. Train all necessary networks for each of the selected domains.
4. Implement or reuse an existing implementation of a classical planning solver with possible integration of the "expansion network" and "heuristic network" for each of the domains as well as with a transition function and heuristic functions commonly used in classical planning.
5. Deploy experiments with the trained DNNs and methods used in classical planning and compare their performance in terms of coverage, length of found solutions and number of expanded states.

Bibliography / sources:

[1] Urbanovská, M. (2020). Learning Classical Planning Transition Functions by Deep Neural Networks [Master's Thesis, Czech Technical University in Prague]. ČVUT DSpace
[2] Masataro Asai, Alex Fukunaga: Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary.AAAI 2018: 6094-6101
[3] 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. 1-77
[4] Christopher M. Bishop, Nasser M. Nasrabadi: Pattern Recognition and Machine Learning. J. Electronic Imaging16(4):049901(2007)

Name and workplace of bachelor's thesis supervisor:

Ing. Michaela Urbanovská, Department of Computer Science, FEE

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

Date of bachelor's thesis assignment: **16.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Michaela Urbanovská	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

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

. _____
Date of assignment receipt

_____
Student's signature

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 19.5.2021

.....
Radovan Tomala

Abstract

Abstract

This thesis further expands the topic of replacing standard transition and heuristic functions in classical planning algorithms by deep neural networks. Three new domains were implemented: Peg solitaire, Painting robots and Multiagent painting robots. The implementation consisted of proposing a graphical domain representation and implementing solvers and data generators for deep neural network learning. Transition and heuristic neural networks were trained on the generated data and then used to replace the standard transition and heuristic functions in the solvers. Experiments were conducted to compare the performance of the standard and neural network solvers. The results show that the peg solitaire domain is too complex for current network architectures to solve. In the case of single and multiagent painting robots, the performance in terms of length of the solution and average number of expanded states is comparable between neural networks and standard baseline functions.

Keywords: Classical planning, Convolutional neural networks, Deep learning, Planning domains.

Abstrakt

Tato prace rozv j tema nahrazovan standardn ch prechodovych a heuristickych funkc v klasickem planovan hlubokymi neuronovymi s temi. Byly naimplementovany tri nove domeny: Peg solitaire, Painting robots a Multiagentn painting robots. Implementace pozestavala z navrhu gra cke reprezentace domen a implementace resicu a generatoru dat pro ucen hlubokych neuronovych s t . Na vygenerovanych datech byly natrenovany prechodova a heuristicka neuronova s t, kterymi byly nahrazeny standardn prechodove a heuristicke funkce v resicech. Byly provedeny experimenty na porovnan vykonu standardn ch a neuraln ch resicu. Vysledky ukazaly, ze domena Peg solitaire je pr lis komplexn a soucasne architektury s t ji neum vyresit. V pr pade Painting robots a Multiagentn ch Painting robots jsou delky resen a pocet expandovanych stavu porovnatelne se standardn mi funkcemi.

Kl cova slova: Klasicke planovan , Konvolucn neuronove s te, Hluboke ucen , Planovac domeny.

Acknowledgements

I would like to thank my supervisor, Ing. Michaela Urbanovska for help, support and guidance during my work on this thesis.

List of Tables

List of Figures

Listings

Contents

Introduction

Deep neural networks are hot. They classify. They regress. They beat you in chess. They can approximate any continuous function. Nowadays, deep neural networks dominate the research in the eld of machine learning. They provide excellent results in problems ranging from computer vision to unsupervised reinforcement learning.

Classical planning lives in a completely di erent realm of the arti cial intelligence eld - symbolic AI. As its name suggests, the goal of the classical planning problem is to create a plan - a sequence of steps needed to achieve a particular goal state. Classical planning has its applications in logistics, manufacturing or even in robotics.

Classical planning relies heavily on functions. Namely step-generating transition and heuristic functions. These functions represent an intersection where the worlds of neural networks and classical planning collide. Both transition and heuristic functions can be replaced by deep neural networks. However, there are very few domains that have been adapted to this approach. The main focus of this thesis is to implement new domains and test the performance of neural network solvers on classical planning problems.

The best domains for this approach are the ones that can be represented visually, in a form of an image. Images can be easily processed using convolutional neural networks. This approach was used in [1] to implement the maze and Sokoban domains. This thesis will build on [1], using the same network architectures on new domains. Speci cally, peg solitaire, painting robots and multiagent painting robots domains were selected for implementation.

Thesis goals

The goals of this thesis consist of providing general background information on classical planning and deep neural networks, implementing domains and conducting experiments to compare the performance of solvers and heuristics.

The goal of domain implementation is to propose a domain representation suitable for deep neural network learning and implementing solvers and data generators. Data generators should generate datasets used in neural network training and datasets of problem

instances used in experiments.

The experiments should compare solvers based on the average solution length, average number of expanded states and other characteristics that can help to compare the e ectiveness of the solvers.

Chapter 1

Problem de nition

The main problem solved in this thesis is a part of a bigger problem of replacing standard transition and heuristic functions in classical planning by deep neural networks. This problem was already solved for two domains in [1]. In that work, Sokoban and maze domains were implemented and adapted for deep neural network learning. In this thesis, the goal is to implement three new domains and conduct experiments comparing the performance of the learned functions with standard transition and heuristic functions.

The implementation of domains consists of implementing a solver of a classical planning problem with domain-speci c transition and heuristic functions. Data generators generating domain instances and training data for transition and heuristic functions also need to be implemented. Testing new neural network architectures is not a goal of this thesis, therefore the networks proposed and implemented in [1] will be used. The domains selected for implementation in this thesis are the peg solitaire domain, painting robots domain and its multiagent variant.

The experiments will be performed on a datasets that were not used during training. They will be performed with multiple domain-speci c con gurations, such as instance size. All combinations of the deep learned and standard functions will be compared to determine if the deep learned functions provide satisfactory results. Heuristic networks will be compared with the selected standard heuristic functions to determine which perform better.

1.1 Peg Solitaire domain

Peg Solitaire is a one-player puzzle game consisting of a board and a number of pegs. The game starts with pegs placed on the board with at least one empty square. A peg can be removed after it is jumped over by another peg. The jump is possible when there is an empty square next to the neighboring peg. The jumping peg is placed on an empty

square and jumped-over peg is removed from the board. The goal of the game is to end up with only one peg left on the board.

There are more variations of this problem. For example, triangular boards where pegs can jump in diagonal directions. These variations will not be regarded in this thesis. It is clear that this problem can be viewed as a problem of classical planning. The goal is to plan a sequence of jumps to end up on a board with one peg.

1.2 Painting robots domain

The problem solved in the painting robots domain consists of one or multiple robots moving on the grid. Robots can paint any square of the grid with some color. In this thesis, robots are only allowed to paint with one color, but a multicolored variant of this problem also exists. Robots can move freely on the grid, however, they cannot step on a square that has already been painted. The goal is to plan a sequence of steps that the robots have to make to paint the provided picture on the grid.

Chapter 2

Background

This chapter is focused on providing a general background on the fields explored in this thesis. The main goal is to define and explain the problems of classical and satisficing planning and the algorithms used to solve them, neural networks and their architectures used in this thesis.

2.1 Classical planning

The main goal of this thesis is to implement new domains of classical planning problems for deep neural network learning. First, the planning problem will be defined, followed by the definition of classical planning and the STRIPS representation of the classical planning problems.

2.1.1 The planning problem

The goal of the planning problem is to construct a sequence of steps leading from the initial state to the goal state. As stated in [2] planning problem is characterized by the description of initial and goal states. The agent has a set of actions it can perform in every state. A sequence of actions leading from the initial to the goal state is called a plan. The environment in which the agent operates is called a domain.

2.1.2 The problem of classical planning

Classical planning problem is a special case of a planning problem, where the environment is fully observable, deterministic, finite, static, and discrete in time, action, objects, and effects [3].

The problem and domain can be represented in various ways. One of the most common is the STRIPS representation described in [4]. When defining a problem instance in

STRIPS, the domain is de ned together with each instance of the problem. Both the domain and problem are de ned using a set of conditions. Each state is described by a set of conditions or facts, that hold for the current state.

The other widely used representation is PDDL [5]. With PDDL, the domain and problem are de ned separately. PDDL is based on the First Order Logic. PDDL representation is commonly used and the problems in this representation are accepted by the majority of planners. However, in this thesis, following [1], this representation will be replaced by an own graphical domain representation suitable for neural network training.

2.1.3 Satis cing planning

With satis cing planning, the goal is not to nd the best solution, but any possible solution. This grants freedom in planner implementation. Even though well-known planners use problems de ned in standardized formats, planning algorithms can be implemented for alternative problem representations. Since the classical planning problem involves the agent moving between states using actions, a multiple of state-space search algorithms can be used to nd a path between initial and goal states.

Searching the state space

The state space search is a powerful way to nd a solution to many problems. There are many algorithms used to perform state space search, however, the basic idea is always the same.

When searching the state space, every achieved state is called a node. Search algorithm expands the node to nd the states that an agent can reach by using one of its actions. The algorithm keeps expanding nodes until the goal state is found.

```
function generalsearch(initial_state)
    nodes = initial_state
    while nodes is not empty
       current_node = pop(nodes)
       if is_final_state(current_node) then return current_node
       next_states = get_next_states(node)
       push(nodes, nextstates)
   end
   return fail
end
```

Listing 2.1: General state space search algorithm

The order in which the nodes are expanded is varied between algorithms. The two most common algorithms - Breath First Search (BFS) and Depth First Search (DFS) [6] are di erent only in the order of the expanded nodes. BFS expands all nodes in the current depth, and then their ancestors in the next level. Thus, the nodes variable in Listing 2.1 would be a queue. DFS, on the other hand expands a node on a deeper level immediately and returns to a higher lever only if the solution was not found in the current branch of the search tree. In this case, the nodes variable in Listing 2.1 would be a stack.[3] However, for some problems these simple algorithms do not perform well and more advanced algorithms are needed.

Heuristics

While searching the state space, the order of expanded nodes is crucial to nd the problem solution e ciently. General algorithms like BFS often search and expand nodes that are too far away from the solution.

For example, the problem of nding the shortest road between Prague and Brno. BFS would continuously expand all roads around Prague in a circular manner until it has found Brno. This means every road in a 180 km radius around Prague would have to be expanded rst. Even cities as far as Dresden would be discovered before Brno, despite the fact that Dresden lies in a completely di erent direction from Prague than Brno. To avoid situations like this, additional information can be provided to the algorithm to expand the nodes that are more likely to lead to the solution.

This piece of additional information is called a heuristic. In general, a heuristic is a value that can be computed for a state and it is an estimate of the distance from the state to its closest goal state. This value can then be used to make informed decisions while expanding new states. Making informed decisions may lead to expanding less states. The function that can compute a heuristic is called a heuristic function.

Heuristic function usually computes a heuristic by solving a simpli ed or modi ed problem. One example of simpli cation is relaxation of the problem. This means that some of the problem constraints are disregarded and removed. In the case of nding the shortest road between Prague and Brno, one possible heuristic function could disregard the need for travelling on roads. The function would then compute crow- y distance between the current position and Brno. This value can then be used to expand nodes that have smaller crow- y distance to Brno rst.

Crow- y distance in our example is actually called Euclidian distance heuristic and can be used in problems of path nding. Manhattan distance heuristic can also be used if the agent operates on a grid and can only move in 4 directions. However, not all heuristic functions are problem speci c. Advanced heuristics such as LM-Cut or h^{max} [7] exist and

can be used in any problem of classical planning.

Heuristic functions can have many properties. One of the most important heuristic function properties is admissiblity.

De nition 2.1.1 (Admissible heuristic). "A heuristic function h is said to be admissible if h(n) h (n) 8 n ." [8]

In De nition 2.1.1, h(n) is the heuristic value andh (n) is an actual true cost required for getting from the current state to the end state. This means that the admissible heuristic never overestimates the actual cost. Use of an admissible heuristic is important in A* search algorithm, because it guarantees the optimality of the algorithm.

The other important property is goal awareness. As the name suggests, this heuristic can determine if the state is a goal state.

De nition 2.1.2 (Goal-aware heuristic) A heuristic function h is goal-aware ifh(n) = 0 8 goal states n .

The heuristic neural networks used in this thesis may not be admissible. Even if they are trained on the dataset based on an admissible heuristic, there is no way to ensure that the heuristic network will not overestimate the cost. For this reason, not only A* but also the greedy best rst and pruning search algorithms will be explained in detail. These algorithms are used in the practical part of this thesis.

Best rst greedy search

Greedy search uses the heuristic as the only criterion and expands the nodes with the smallest heuristic rst. This approach does not guarantee a globally optimal solution, however, it is not a problem in the case of satis cing planning.

Greedy search can be implemented by using a priority queue. Each discovered node is evaluated by a heuristic function and inserted into the priority queue, with the heuristic value serving as the nodes priority. The algorithm then expands nodes in the order of their heuristic values. Pseudocode for this algorithm is shown in Listing 2.2.

```
function greedy_search(initial_state)
    nodes = new priority_queue()
    nodes.push(initial_state, 0)
    while nodes is not empty
       current_node = nodes.pop()
       if is_final_state(current_node) then return current_node
       next_states = get_next_states(node)
       for each state in nextstates
           h = compute_heuristic(state)
           nodes.push(state, h)
       end
   end
   return fail
end
```

Listing 2.2: Greedy search algorithm

The main shortcoming of greedy search is the complete disregard for the cost of the path leading to the current state. Greedy search only considers the heuristic value. This of course means that the optimal solution is not guaranteed, the found solution can be overly complex, and the time needed to find a solution might be long, especially if the heuristic does not provide enough information.

A* search

Opposite to the greedy search, A* does consider the cost of the path leading to the current node. A* does not use the heuristic value as the only criterion when searching for the solution. In A*, the heuristic only shifts the search in the direction of the expected solution and the cost of the path is still an essential factor.

Implementation of A* is very similar to greedy search. Priority queue is used to expand the best nodes first. Here, however, the priority is the sum of the heuristic and the cost of the path leading to the current state. Pseudocode is shown in Listing 2.3.

If the heuristic used in A* algorithm is admissible and goal-aware, the found solution will be optimal. This makes sense, because the admissible heuristic does not overestimate the actual cost of the path leading to the solution. Therefore, expanding nodes in the order of their actual cost summed with the admissible approximation of the cost leading to the goal state will cause the optimal solution to be found first. This would not hold if the heuristic was not admissible. Suboptimal solution could be found first if the cost of the optimal solution was overestimated by the heuristic function. The more rigorous

proof can be found in [8].

```
function a_star_search(initial_state)
    nodes = new priority_queue()
    nodes.push(initial_state, 0)
    while nodes is not empty
       current_node = nodes.pop()
       if is_final_state(current_node) then return current_node
       next_states = get_next_states(node)
       for each state in nextstates
           h = compute_heuristic(state)
           g = compute_cost(state)
           nodes.push(state, h + g)
       end
   end
   return fail
end
```

Listing 2.3: A* algorithm

Pruning search

When the space state is very broad, even algorithms such as greedy or A* search can have problems with finding a solution. Both of these algorithms can be improved by including pruning.

Pruning means cutting off the entire branches of the search tree that are less likely to contain a solution. This can be achieved by computing the next states for one depth of the tree. Then these states can be sorted by the heuristic. Finally, states that are less likely to lead to the solution will be removed from the queue. This will lead to a massive reduction in the width of the search tree, and consequently to faster discovery of the solution. This type of search is suitable for satisficing planning because if the pruning cuts away a branch with the optimal solution, it is not a problem. The pseudocode is shown in Listing 2.4.

The threshold determining which states are likely to lead to the solution has to be carefully selected by experimentation. Setting the threshold too low can lead to pruning away a branch with a solution, which in some cases can mean losing the solution altogether. Setting it too high might not bring any noticeable improvement in performance.

```
function prune_solve(initial_state, pruning_constant)

    Initialising first generation
    generation = [initial_state]

    while generation length is> 0

        Compute the next states of each state in the current
           generation sorted by heuristic
        next_states = get_next_states_sorted_by_heuristic(
           generation)

        The new generation consists of the first
           pruning_constant elements of nextstates

        generation = new_states[1:pruning_constant]

        if generation contains end state
            return end state

    end
end
```

Listing 2.4: Pruning solver pseudocode

2.2 Neural networks

Neural networks are a powerful machine learning tool. In general, the neural network consists of connected layers of neurons. Arti cial neuron is a mathematical model of the real neuron. Neurons are interconnected and each connection has its associated weight. Each neuron is activated by a nonlinear activation function. In the simplest type of network, neurons are ordered into layers, where neurons from one layer are connected to the following layer. The rst layer of the network is called the input layer, the last is called the output layer. Layers between them are called hidden layers.

To train a neural network, a large labeled (in case of supervised learning) dataset is needed. This dataset is then used to learn the weights between neurons using the backpropagation algorithm. [9].

There are countless ways in which neurons can be connected into the neural network.

The scheme of how neurons are connected into layers that form a network is called neural network architecture. There are many types of architectures. For example, feedforward, convolutional or attention network, which are also used in the practical part of this thesis.

2.2.1 Deep feedforward networks

Deep feedforward networks are one of the most commonly used deep learning models. They are called feedforward because all the information flows only in the forward direction - between the input and output layer. This means there are no connections where the network inputs the data to itself.

Feedforward network can approximate any continuous function with only one hidden layer. This fact is stated in The Universal Approximation Theorem [10] . The goal of training feedforward network is to learn the weights between neurons which result in the best function approximation.

Gradient based learning and gradient descent

As stated above, training the neural network means finding the best values of the parameters. First, the proper loss function has to be chosen to correctly measure how well the model performs. The value of the loss function can then be minimised by using standard gradient-based algorithms by computing the gradient w.r.t. parameters.

The most common gradient-based optimization method is the gradient descent algorithm. Gradient descent computes the gradient of a function in the current point. Gradient is the direction in which the function values grow the most. The algorithm subtracts the gradient multiplied by the step length from the current point to get the next point, which means moving in the opposite direction of the biggest growth. This process is repeated until the stopping condition is satisfied [11]. The step length is crucial for the performance and convergence of the gradient descent algorithm. Too small step size will cause very slow convergence, on the other hand, if the step size is too big, the algorithm can diverge from the solution. The divergence can be avoided by constantly checking that with every step taken the function value decreases, $F(x_{i+1}) < F(x_i)$. If $F(x_{i+1}) > F(x_i)$ the step size needs to be decreased, for example, halved, and the test should be repeated until it is safe to take a step leading to a point with a smaller value. When training neural networks, the step size is often called learning rate.

Gradient descent can be easily adapted for neural network learning. Vanilla gradient descent for neural networks computes the gradient of the loss function on the entire dataset and uses it to adjust the parameters. This can be very time consuming if the dataset is very large. The algorithm can be adjusted by computing the gradient for each dataset

instance. This adjusted algorithm is called stochastic gradient descent and is usually much more e cient than vanilla gradient descent[12]. Stochastic gradient descent pseudocode is shown in Listing 2.5.

```
function sgd(dataset, modelparameters, lr)

    while convergence condition not met
        for each instance in dataset
            gradient = compute_gradient(parameters, instance)
            parameters = parameters= lr * gradient
        end

    end

    return parameters
end
```

Listing 2.5: Stochastic gradient descent pseudocode

Gradient descent is not the only gradient-based algorithm used to train neural networks. Algorithms such as Adam, Adagrad or Adadelta are all more advanced algorithms that adaptively change the learning rate to converge faster and speed up the learning process.

Backpropagation

Backpropagation is an algorithm used to compute the gradient of the loss function w.r.t. model parameters. The process of neural network learning consists of two parts. First is forward pass, where the information ows from the input layer to the output layer. The value of the loss function is then computed on the output layer and the backward pass starts.

The idea behind the backward pass is to compute the gradient by using the chain rule. Neural networks can be viewed as composite functions, because each layer of the network is a function that transforms input into output. The gradient is computed in each layer as $f(g(x))' = f'(g(x))g'(x)$, where g is the layer before f in the backward pass. Backpropagation algorithm computes gradient on the layer and then passes it backward to the previous layer where the value is used to compute the gradient of that layer. This way, the gradient backpropagates from the output layer back to the input layer and can be used in a gradient-based optimization algorithm to nd the minimum of the loss function. [13]

2.2.2 Convolutional neural networks

Convolutional neural networks (CNN) are a famous class of neural network architectures. They are widely used in computer vision problems and image classi cation. Regarding images, feedforward networks are not very suitable. Feeding an image into a feedforward network would not provide satisfactory results because an image is a very complex input with a lot of dependencies between pixels. CNN, on the other hand, does not use fully interconected layers.

CNN uses image convolution. It applies a number of convolution lters to the image. The parameters of the convolution function as parameters of the network that need to be learned. The learned lter is then applied to the image. This ensures that CNN can classify the object in the image regardless of its position. This property of CNNs is called the shift invariance. Training a CNN can be also achieved by using gradient-based algorithm and backpropagation.[14]

In [15] new Transformer architecture using attention mechanism was proposed and shown good performance. Attention mechanism works with CNNs and serves to highlight more important parts of the data. This architecture is also used in this thesis as the heuristic neural network.

Chapter 3

Implementation

This chapter regards the process and details of implementation. This includes specifying the languages and tools used to implement new domains as well as the description of the implemented solvers and data generators.

3.1 Languages and tools

All code used in this thesis was written in Julia programming language in version 1.4.2. Most of the code uses standard Julia libraries, however, for some functions additional libraries were used.

JLD library was used for storing and loading the generated datasets produced by data generators. This library has been chosen because it provides a simple way of storing and reading data arrays in les.

Flux.jl framework was used for neural network training. Flux.jl is a powerful neural network framework for Julia language. It provides methods for building and training neural network models and is on par with other famous frameworks such as Tensor ow.

The code was written in Visual Studio Code editor with Julia language extension. Remote - SSH extension was used to connect to the university server josef.felk.cvut.cz. Code was developed on this server because it provides powerful graphics cards that sped up the training of the models.

3.2 Neural network models

Implementation of neural network models was not a goal of this thesis. Used neural network architectures were implemented and explained in detail in [1]. There are two types of networks used - the heuristic and the transition network.

3.2.1 Transition network

The purpose of the transition network is to replace the standard transition function of the solver. This function is used to generate the next states from the current state. The transition network architecture works with a graphical representation of the image and uses convolutional layers.

Transition network uses the architecture described in [1]. There are three convolutional layers and one residual connection connecting the input and the output of the convolutions. Then there are two convolutional layers with 1x1 kernel to adjust the number of channels. This network takes one-hot representation with three channels as an input and outputs one-hot representation of the probabilities of the next states that are learned. The architecture is shown in Figure 3.1.

Figure 3.1: Transition Network Architecture proposed in [1]

3.2.2 Heuristic networks

The role of the heuristic network is to replace the standard heuristic function of the solver. Just the same as the transition network, the heuristic network also works with a graphical representation of the domain. The heuristic network expects one-hot graphical representation as an input and returns a scalar heuristic value as an output. Two di erent heuristic network architectures were used in the experiments.

The rst heuristic network is purely convolutional without attention. It uses four convolutional layers and four fully connected layers. The network architecture is described in Figure 3.2. This architecture is scale-free, which means that an instance of any size can be processed without the need to change the architecture.

Figure 3.2: Purely convolutional Heuristic Network Architecture proposed in [1]

The second heuristic network uses the attention architecture explained in Subsection 2.2.2. It uses five attention masks that are applied at the beginning of the network. The rest of the network consists of four convolutional layers and one fully connected layer. The used architecture is described in Figure 3.3.

Figure 3.3: Attention Heuristic Network Architecture proposed in [1]

For experiments in the Peg solitaire domain, both architectures were compared. In the painting robots domain, only the attention network was used during the experiments.

3.3 Peg Solitaire domain implementation

As stated in the problem definition, Peg Solitaire is a one-player game where pegs are removed from the board by jumping over one another. The goal is to end up with only one last peg on the board.

3.3.1 Domain representation

The domain is fully represented by the state of the board. The board is represented as a 2D array of $\{0, 1, 2\}$, where 0 represents an empty square, 1 represents a square with a peg and 2 represents a nonplayable square. For example, the classic 33-square cross-shaped board representation is shown in Listing 3.1

```
board = [
        2 2 2 2 2 2 2 2 2;
        2 2 2 1 1 1 2 2 2;
        2 2 2 1 1 1 2 2 2;
        2 1 1 1 1 1 1 1 2;
        2 1 1 1 0 1 1 1 2;
        2 1 1 1 1 1 1 1 2;
        2 2 2 1 1 1 2 2 2;
        2 2 2 1 1 1 2 2 2;
        2 2 2 2 2 2 2 2 2;
    ]
```

Listing 3.1: Board representation example

In the rest of this chapter, the board state will be displayed in a heatmap format. Black tiles represent empty squares (0), red tiles represent a peg (1) and yellow squares represent a nonplayable square (2). Board from Listing 3.1 is shown in Figure 3.4

Figure 3.4: Heatmap board representation

3.3.2 Solver

Two solvers were implemented for this domain. The rst solver is an implementation of the A* algorithm. The second one is an implementation of the pruning search. These algorithms are explained in detail in Chapter 2. Both of these algorithms need transition and heuristic functions which are domain-speci c. The pruning search algorithm was chosen to perform the experiments because it showed better results during implementation.

The pruning search algorithm was implemented to search by generations. The initial state is a rst generation. All the following states form the second generation. Second generation next states form the third generation and so on. Each generation is sorted by the heuristic and all but the rst n states are removed from the generation, where is a pruning constant.

During the search, it is possible to end up in the same state multiple times. Because of this, every visited state is stored in a set. If the discovered state is already in the visited set, it is not inserted into the queue.

Additional check for solvability had to be added to the A* algorithm to stop the search if it took too long. This was added for the purposes of data generation explained later in this chapter. The search is stopped if the number of expanded states is above the set threshold. The threshold was chosen by computing an average number of expanded states over a set of di erent solvable problem instances and multiplied by ten.

Pruning search algorithm does not need this check, because if the pruning constant is set reasonably low, the search will terminate in a reasonable time in the case of this speci c domain. The reason for this comes from the rules of the domain where each move must remove one peg from the board. Because of this, each generation contains states with the same number of pegs on the board. With each generation, the number of pegs on the board decreases by one. In the second to last generation, there are only boards with two pegs. If these states contain a board with two pegs next to each other, the solution will be in the next generation and the search will terminate with the found solution. If such states are not present in the generation, the last generation will be empty and the search will terminate without solution.

Transition function

Transition function receives a state as an input and returns all possible next states that can be reached from the input state in a single move. Implementation of the transition function rst nds all squares without a peg. Next, for each empty square, it checks if there is a square with a peg next to it. If yes, it then checks if that peg can be jumped over by some other peg in its neighbouring region and the correct direction. If the condition is met, a new state is created and added to the result of the function.

Heuristic functions

Three heuristic functions were implemented. The simplest one is the number of pegs remaining on the board. This heuristic was considered mainly because it is trivial to implement. The function just computes the sum of the array representing the state.

The next implemented heuristic was the number of detached pegs on the board. Detached peg does not have any neighboring pegs and cannot be removed in one jump. The less detached pegs the board has, the better chance there is to lead to the final state.

The third implemented heuristic is the number of attacking pegs. Attacking peg can jump over another peg and remove it from the board. This can also be viewed as a number of next possible states.

The third heuristic showed the best results overall and is used in both solvers. This heuristic is also used as a baseline in the experiments.

3.3.3 Instance generator

To perform the experiments, a range of problem instances - boards needed to be generated. Implementing a generator which produces solvable boards proved to be a difficult task.

The first version of the generator used a brute force approach - randomly generating a board configuration and checking its solvability using the solver. The generator generated only the configuration of pegs and nonplayable fields and then randomly assigned one empty field. On a 9x9 board, there are 2^{81} possible configurations of pegs and nonplayable fields and 81 ways to place the empty field. The vast majority of these configurations are unsolvable and thus worthless. Chances of the generator to produce a solvable board with this approach are low. The time needed to generate a reasonable number of boards would be too long.

The current version of the generator also relies on randomness. However, instead of randomly assigning every single field of the board, the generator randomly assigns sections of the board. The 9x9 board is split into 9 3x3 sections. The generator randomly assigns each section from the set of predefined 3x3 board segments. These segments were defined to not contain any configuration which would make the board unsolvable. Generating boards with this approach is much faster than the first approach but still quite slow. During a few days, a total of 20000 solvable boards were generated. Some examples are shown in Figure 3.5.

3.3.4 Transition network data generator

The transition network determines which states can be generated from the input state. To train this network, proper training data had to be generated.

The training process expects the data to be represented as pairs of inputs and labels. However, generating such labels from full boards and training the network from them would be ineffective. Since a convolutional neural network is used, the shift invariant property of such network can be used to train the model with smaller inputs. In other

Figure 3.5: Examples of generated boards (Yellow squares represent non-playable edges, red squares represent pegs and black squares represent empty squares)

words, the model can be trained using only data about speci c parts of the board and possible moves on them. The convolution kernel then detects these patterns wherever on the board.

The input training data consists of 4x4 board sections. These are generated as every possible combination of pegs and empty tiles. This is generated by iterating numbers from zero to $2^{4\ 4}$. Each number is then converted to a binary array, which is then transformed to a 4x4 2D array.

Then, to each one of the combinations, the mask of nonplayable tiles is applied. Masks are de ned to represent edges of the board. The set of masks is de ned by hand. There is a total of 40 masks.

To generate the labels, the transition function is used. If there are no possible moves from the input, it is disregarded and removed from the data set. There is a total of 3833856 examples in the data set. Example from this data set is shown in Figure 3.6

After the data set is generated in the representation described in Section 3.3.1 one last step is to convert them to one-hot representation. This dataset is then used to train the transition neural network.

Figure 3.6: Example of feature - label data set

3.3.5 Heuristic network data generator

Heuristic function is crucial to correct functionality of the solver. Training a neural network to compute a heuristic from the input image requires the labels to be a scalar value.

To achieve this, a sample of 100 boards were randomly selected from the set of boards generated by the instance generator. The described solver was adjusted to return a list of every position encountered during search. This list also includes positions cut by pruning. Positions were then evaluated by the heuristic function. The heuristic values were saved as the labels and input boards were converted to one-hot representation. Training data in this format can be used to train the heuristic neural network.

In [1] heuristic data was generated by optimally solving the problem and using the actual costs of the paths leading to the goal state as labels. This approach was not possible here, because of the structure of the domain. The rules of peg solitaire state that on each move, one peg must be removed from the board. The goal is to end up with a board with only one peg left. These rules split all states in two categories. The first category are the states where achieving the solution is possible and the cost of the solution is the number of pegs on the board minus one (the number of pegs that need to be jumped over to end up on a board with only one peg). The second category are states where achieving the solution is impossible because the configuration of the pegs on the board can no longer lead to the board with only one peg left. The role of the heuristic in this domain is to distinguish between these two categories and eliminate states from the second category from the search as soon as possible. The number of attacking pegs heuristic does this well because the states with more attacking pegs will generate more following states and are more likely to be solvable. For this reason, the values of this heuristic were used as labels

to train the heuristic network.

The other possible way to train this network is to try to teach it to classify the states into solvable and unsolvable categories. Unsolvable states would then be removed from the search. This approach was not implemented and can be explored in further research.

3.3.6 Neural network training

Both networks were trained on the generated datasets. Transition network dataset was batched to 10 000 batches of 100 instances. Purely convolutional heuristic network dataset was batched to 5000 batches of 40 instances. Both trainings consisted of 50 epochs. Attention heuristic network was trained on the same dataset batched to 10 000 batches of 100 instances.

3.3.7 Neural network solver

Neural network solver uses the same pruning search algorithm as the standard solver, however, the heuristic and transition functions are replaced by a new function which uses neural networks. Transition function uses the network output as a guide to determine new states. Heuristic function uses the neural network output directly.

Solver uses a function to get the next states from output of the transition network. The output is in a form similar to one-hot representation, but the values can range between zero and one. This output represents the probability distribution of the pegs on the new board. Solver takes the input board in the standard array representation described in Solver section. Then it checks every tile without peg and compares it with the network output. If the probability of peg on the same position in the network output is greater than 0.5, the function then checks which peg got jumped over and generates the next state. This is more e cient than the classic transition function which checks if there are any pegs that can be jumped over for every tile without a peg.

3.4 Painting robots domain implementation

The painting robots domain consists of one or more robots on a grid. Their goal is to paint the provided picture on the grid, but they can not step on a square that has already been painted.

3.4.1 Domain representation

The domain was implemented in two representations. Similarly to peg solitaire, the grid is represented as a 2D array of zeroes and ones, where a zero is not a painted square and

one is a painted square. Both representations use a grid represented in this way.

The di erence is in the representation of robot positions on the grid. For the purposes of the solver, it is much more convenient to have the robot position stored as an array of their coordinates on the grid. This allows the nonneural transition and heuristic functions to be less complex.

The neural networks, however, need the representation to be an image, or in this case, a 2D array. The representation for neural networks represents robots on the grid as a number two. The whole problem instance is then represented by a 2D array of 0, 1, 2. For the purposes of neural network training, this representation is then converted to a one-hot representation.

The conversion between these two representations is implemented by a simple function. The graphical representation for the neural network is shown in Figure 3.7.

Figure 3.7: Example of a problem de ned in the representation for neural network. Black squares are empty, red squares represent the image that needs to be painted and the yellow squares represent a robot. The goal is to paint a percentage symbol with robot starting at position (1,1)

3.4.2 Solver

The solver uses the implementation of the greedy best- rst search algorithm explained in Section 2.1.3. Same as in the peg solitaire domain, the visited states are also stored in a set to avoid expanding the same state multiple times and slow the search down. The search is also stopped if the number of expanded states is above the threshold, which is computed in the same fashion as in the peg solitaire domain - average number of expanded

states computed on solvable solutions multiplied by 10. The solver is able to solve both single and multiagent versions of this problem.

The solver expects the final image that is supposed to be painted as an input. Together with the image, an array of the initial robot position, or positions in a multiagent variant, is expected as an input. The solver then constructs an empty grid in the size of the image and starts the search for the plan.

Transition function

The transition function is much simpler than the one in the peg solitaire domain. For each robot, the function checks all four directions around the robot. It then creates new states by placing robots on the unpainted squares around it. Two new states are created for each robot move. One with the original robot position left unpainted, and the other painted by the robot.

Heuristic function

Two heuristic functions were implemented. The first trivial one counts the number of the remaining squares that need to be painted. This heuristic provides enough information for best- first greedy search to be effective. This heuristic is admissible, because the number of squares that need to be painted can never be greater than the number of steps that the robot has to make to paint them.

The other implemented heuristic is the distance between robot and the furthest square that needs to be painted. This heuristic was tested during implementation and did not provide improvements over the first implemented heuristic.

The first heuristic is used in the solver and will be used during experimentation. Values of this heuristic are used as labels for the heuristic neural network.

3.4.3 Instance generator

The instance generator generates instances in representation for the solver. It generates the grid and the array of robot positions separately.

The grid, or the image that needs to be painted, is generated randomly. Random 2D array of zeroes and ones is generated and saved. All of these grids are solvable. If we let the robot go row by row and paint the squares that need to be painted, it would eventually solve the problem. Hopefully, the solver is able to provide more efficient plans.

An array of robot positions is also generated randomly. It consists of random Cartesian coordinates in the range of the board size. The position of the robot does not affect the solvability if it starts on the blank grid. The solver always creates a new blank grid when

starting to search for the plan. This means that all generated robot positions are valid and will not create unsolvable instances.

3.4.4 Transition network data generator

The robot can move only to squares that are right next to the current robot position. Thanks to this, a 3x3 area is sufficient to reliably train the network to perform transition. Data for this dataset was generated using the representation for a neural network, where the robot position is represented by number two.

The generator first generates all possible configurations of painted and unpainted squares. This is the same problem as generating squares with and without pegs in peg solitaire. Generator uses the same algorithm to generate this configuration. In this case, the segment is smaller, so only the numbers 0 to $3^3 - 1$ are iterated and converted to a binary array.

Then for each configuration, a robot is added to each one of the 9 squares. In the multiagent case, two robots are added to some of the instances, so the network can learn that multiple robots can not stand on the same square.

Standard transition function is then used to generate labels. Generated inputs and labels are then converted to one-hot representation and stored as pairs of inputs and labels. An example from this dataset is shown in Figure 3.8.

3.4.5 Heuristic network data generator

Heuristic used to train the heuristic neural network is the number of squares that are not yet painted but need to be. This heuristic is not affected by a robot position. This is an advantage regarding training and generating data for neural network, because the position of the robot can be left out, leaving a simpler dataset.

The generator generates a number of random grids with painted and not painted squares. These are then evaluated by the standard heuristic function. Grids are then transformed to one-hot representation. The generated dataset consists of pairs of one-hot grids and heuristic values.

3.4.6 Neural network training

Both networks were trained on the dataset generated by the implemented data generators. The dataset for transition network batched to 10000 batches of 100 instances. The heuristic network was trained on the dataset batched to 1000 batches of 70 instances.

Figure 3.8: Example from the transition dataset. There are four possible next moves for the input on the left. Black squares are empty, red are painted and the robot is yellow. Pairs are then converted to one-hot representation and used to train the transition neural network.

3.4.7 Neural network solver

The neural network solver is the same as the standard solver. It uses the heuristic network output directly as a heuristic.

The transition network outputs results in the form of a tensor of probabilities. To extract new states from this representation, a function was implemented. The function collects all positions with a probability of a robot higher than 0.2 and constructs new states with the robot in these positions. The 0.2 threshold might seem low, but experimentation showed that in some cases, the valid robot position probabilities were lower than 0.5. The solver was constantly tested for generating invalid robot positions with this threshold, but an invalid robot position was never generated.

To ensure that the solver works correctly, functions that control the output and process of the search were also implemented. If the transition function generates a state that is not reachable from input, the search is stopped and an exception is thrown. However, such a case did not occur during testing or experiments. When the solution is found, it is also checked to ensure it is correct.

Chapter 4

Experiments

One of the main goals of this thesis is to compare the performance of the classical methods and neural networks. This can be achieved by conducting a range of experiments. This chapter regards the methodology and results of the conducted experiments.

The goal of the experiments is to compare the implemented classical functions and trained neural networks. Average time, length of the solution, the number of expanded states and coverage will be compared. Coverage is computed as a percentage of correctly found solutions. All experiments were conducted on the university server josef.felk.cvut.cz.

Experiments compare 4 versions of the solvers. Solver with standard transition function and standard heuristic function, the solver with standard transition function and heuristic network, the solver with transition network and standard heuristic and nally, the solver with transition and heuristic networks.

For every domain, there will be one classical heuristic used in the solvers with the standard heuristic and blind heuristic. Blind heuristic is zero for every state. Performance of the heuristic networks will be compared to those two.

4.1 Peg Solitaire domain experiments

Peg solitaire domain uses a pruning search solver. This solver sorts the states by heuristic, and then continues to expand only the rstn states. The pruning constant n is provided to the solver. This solver is very sensitive to the heuristic. For this reason, to compare the performance of the heuristic, experiments were run for 4 pruning constants: 1000, 500, 250, 150.

The experiment problem instances are 150 board con gurations that were checked by a solver with attacking pegs heuristic and pruning constant 1000 to be solvable. However, lowering the pruning constant or the change of the heuristic may lead to the loss of the solution. For this reason, the solvers are also compared by coverage.

Solver \ Pruning const.	1000	500	250	50
Standard tran. + standard heur.	7.53	3.45	1.80	0.38
Standard tran. + blind heur.	1.47	0.72	0.39	0.08
Standard tran. + conv. network	29.20	15.03	7.81	1.68
Standard tran. + att. network	102.07	53.03	27.48	6.06
NN tran. + any heur.	incorrect	incorrect	incorrect	incorrect

Table 4.1: Peg Solitaire: Average time in seconds the solver needs to terminate search by either nding a solution or expanding all possible states

Solver \ Pruning const.	1000	500	250	50
Standard tran. + standard heur.	100	93.3	86.67	57.33
Standard tran. + blind heur.	12.66	11.33	4.66	2
Standard tran. + conv. network	8	3.33	2	0.67
Standard tran. + att. network	8	6	4.66	2
NN tran. + any heur.	incorrect	incorrect	incorrect	incorrect

Table 4.2: Peg Solitaire: Coverage in %

Solvers were not compared by the length of the solution. In the case of peg solitaire, all solutions of one instance have to have the same length. On each move, exactly one peg must be removed from the board. The goal is to end up with one peg on the board. For this reason, the length of the solution must be the number of pegs on the board - 1 for each problem instance.

During experimentation, the transition neural network showed to be faulty by not generating all the next possible states. Because of this, all solvers with transition neural network were excluded from the experiments. In [1] similar situation happened with the Sokoban domain due to its complexity. In the peg solitaire domain there is no acting entity, so generating all possible successor states becomes a lot more complex than in the case of painting robots domain.

Solver \ Pruning const.	1000	500	250	50
Standard tran. + standard heur.	23979	12337	6329	1330
Standard tran. + blind heur.	16873	9220	4608	1088
Standard tran. + conv. network	15651	8322	4360	1079
Standard tran. + att. network	14641	7884	4115	975
NN tran. + any heur.	incorrect	incorrect	incorrect	incorrect

Table 4.3: Peg Solitaire: Average number of expanded states. Computed only for instances, that the solver can solve.

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	0.96	1.65	1.93
Standard tran. + blind heur.	0.91	2.63	0.42
Standard tran. + NN heur.	3.22	3.9	7.37
NN tran. + standard heur.	2.98	4.79	6.67
NN tran. + blind heur.	5.39	3.59	4.27
NN tran. + NN heur.	8.01	9.29	9.21

Table 4.4: Single agent painting robots on instances checked by solver with standard transition: Average time in seconds computed only on instances where solver found the solution

4.2 Painting robots domain experiments

The performance of painting robot solvers and heuristics was tested on a set of six datasets. The first three datasets consisted of 50 instances that were randomly generated by the instance data generator in three sizes - 7x7, 8x8 and 9x9. These instances were checked to be solvable in a reasonable time by the solver with the standard transition function. The next three datasets also contain 50 7x7, 8x8 and 9x9 instances. These, however, were checked to be solvable using a solver with the heuristic transition network.

This decision was made because during implementation, both standard and neural network transition functions provided correct results, however, the performance was different for different types of instances. By investigating this problem, it was shown that this is actually the problem of the heuristic. Both transition functions provide the next possible moves in different order. Because the heuristic, which counts the number of squares that need to be painted, does not take into account the robot position, this causes the solution to be found more quickly with different orders of the next states. Since no new heuristic was implemented, the experiments were conducted on two sets of datasets to objectively compare the solvers with neural network transition and standard transition function.

On the dataset checked by the classical transition solver, the average time, solution length and number of expanded states were evaluated. Coverage was also compared between the solvers. The search is stopped after the number of expanded states is larger than the threshold of 20000. This threshold corresponds to the average number of expanded states multiplied by ten. All combinations of solvers with standard and neural network transition function and standard, neural network and blind heuristic were compared. On the dataset checked by the solver with transition neural network, only solvers with the standard heuristic were compared, because other heuristics were sufficiently compared on the first dataset. All averages were computed only on the solutions that were found in the expanded states limit.

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	41.2	**53.42**	66.72
Standard tran. + blind heur.	69.93	103.65	131.5
Standard tran. + NN heur.	45.89	58.72	72.22
NN tran. + standard heur.	**40.96**	54.02	**65.63**
NN tran. + blind heur.	123.16	169.34	212.13
NN tran. + NN heur.	45.116	57.05	70.20

Table 4.5: Single agent painting robots on instances checked by solver with standard transition: Average solution length computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	1077.94	1590.52	1845.16
Standard tran. + blind heur.	**838.70**	1958.28	**1117.64**
Standard tran. + NN heur.	1802.34	2096.60	3134.65
NN tran. + standard heur.	1333.47	1921.05	1945.27
NN tran. + blind heur.	1938.36	**1260.45**	1379.63
NN tran. + NN heur.	2332.54	2563.32	1950.03

Table 4.6: Single agent painting robots on instances checked by solver with standard transition: Average number of expanded states computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	**100**	**100**	**100**
Standard tran. + blind heur.	94	92	68
Standard tran. + NN heur.	94	92	92
NN tran. + standard heur.	90	76	66
NN tran. + blind heur.	86	70	60
NN tran. + NN heur.	88	68	58

Table 4.7: Single agent painting robots on instances checked by solver with standard transition:Coverage in %

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	**0.51**	**2.99**	**2.68**
NN tran. + standard heur.	3.16	4.30	8.60

Table 4.8: Single agent painting robots on instances checked by solver with transition neural network: Average time in seconds computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	**40.81**	53.47	**66.65**
NN tran. + standard heur.	41.28	**52.18**	67.08

Table 4.9: Single agent painting robots on instances checked by solver with transition neural network:Average solution length computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	**1036.92**	2630.39	**1978.28**
NN tran. + standard heur.	1508.86	**1493.24**	2627.14

Table 4.10: Single agent painting robots on instances checked by solver with transition neural network:Average number of expanded states computed only on instances where solver found the solution

4.3 Multiagent painting robots experiments

Experiments on the multiagent painting robots domain were conducted on three datasets consisting of twenty instances of size 7x7, 8x8 and 9x9. The problem instances all contained two robots placed in the opposite corners of the grid.

Datasets of sizes 7x7 and 8x8 were generated randomly, the 9x9 dataset was created by hand to contain "pretty" images of symbols and letters. All instances were checked to be solvable in the 10000 expanded states limit.

The experiments follow the same structure as in the case of the single agent variant. All combinations of neural and standard functions in a solver are compared.

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	84	72	64
NN tran. + standard heur.	**100**	**100**	**100**

Table 4.11: Single agent painting robots on instances checked by solver with transition neural network: Coverage in %

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	4.91	0.56	0.15
Standard tran. + blind heur.	1.65	2.63	1.06
Standard tran. + NN heur.	1.30	1.22	1.30
NN tran. + standard heur.	4.94	20.71	5.25
NN tran. + blind heur.	6.55	23.33	45.97
NN tran. + NN heur.	5.88	13.99	7.82

Table 4.12: Multiagent painting robots: Average time in seconds computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	52.9	59.86	61.7
Standard tran. + blind heur.	153.0	204.17	291.1
Standard tran. + NN heur.	50.11	63.54	61.7
NN tran. + standard heur.	52.9	72.5	81.0
NN tran. + blind heur.	407.6	796.5	1141.94
NN tran. + NN heur.	54.56	70.72	81.0

Table 4.13: Multiagent painting robots: Average solution length computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	1402.8	758.933	270.6
Standard tran. + blind heur.	1085.15	2193.0	830.45
Standard tran. + NN heur.	709.29	572.9	270.6
NN tran. + standard heur.	1402.8	3601.6	864.26
NN tran. + blind heur.	1279.2	2815.0	2948.61
NN tran. + NN heur.	1281.5	2697.27	847.26

Table 4.14: Multiagent painting robots: Average number of expanded states computed only on instances where solver found the solution

Solver \ Instance size	7x7	8x8	9x9
Standard tran. + standard heur.	100	75	100
Standard tran. + blind heur.	100	90	100
Standard tran. + NN heur.	85	55	100
NN tran. + standard heur.	100	100	100
NN tran. + blind heur.	50	10	90
NN tran. + NN heur.	80	55	95

Table 4.15: Multiagent painting robots: Coverage in %

Chapter 5

Result discussion

The experiments show different results for different domains. It is clear just by looking at the coverage of the peg solitaire domain, that the solvers using neural networks failed and performed terribly in comparison with the standard solver. Both single and multi-agent painting robots, however, show more optimistic results.

5.1 The failure of the Peg solitaire domain

The peg solitaire failed both in terms of transition and heuristic neural networks. The transition network did not function at all and the use of heuristic networks resulted in coverage not exceeding 8% as seen in Table 4.2. Even the blind heuristic provided better coverage than any of the heuristic networks. Blind heuristic in the case of pruning solver implemented in this thesis means that the states that are pruned are not sorted in any particular order, and the solution is essentially found by chance.

When the coverage is this low, the comparison based on the number of expanded states shown in Table 4.3 is not relevant. The comparison based on the average time in Table 4.1 shows that the solvers with heuristic networks took significantly more time than solvers with standard heuristics. Especially the attention network solver took the longest to find the solution. This is might be caused by the computational complexity of this network.

5.1.1 Possible causes of the transition network failure

The first big issue with peg solitaire neural network solvers is the transition neural network. It does not work properly. Outputs of this network were thoroughly investigated with the following findings.

Transition network can currently generate valid next states, however it can not generate all of them. The most representative example of this phenomenon is shown in Figure 5.1. It is clear that this extremely simple problem can be solved because there is a peg

that can be jumped over resulting in only one peg left on board. However, the transition network does not pick up this move and returns no valid next moves.

However, if the same instance is ipped by 180 degrees, the transition network correctly returns the next step. This behavior was observed with multiple network con gurations - both 3x3 and 5x5 kernel sizes, various number of channels in convolutional layers and various numbers of training epochs.

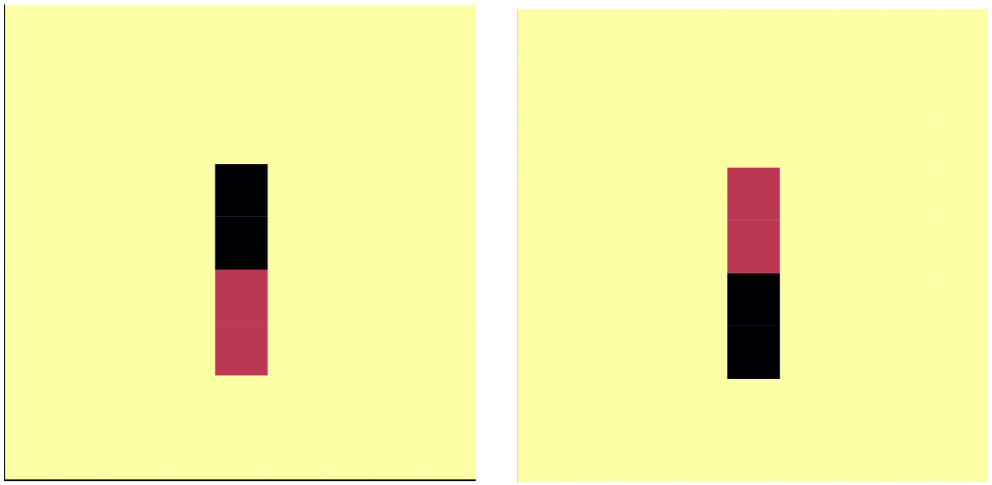

Figure 5.1: Board where neural network does not pick up valid move (left) and where it does (right)

The rst obvious cause might be insu cient training data. The network was trained on 4 by 4 grids with all possible combinations of pegs. This seemed to su ce from a theoretical perspective, but might not be enough for the network to function properly.

The other potential cause is the combination of network architecture and the domain. The peg solitaire domain does not have an agent. The agent in this case is the entity moving the pegs on the board, but is not present in the domain representation. The proposed transition neural network performed well on domains containing an agent moving in an environment. For example, in the maze domain in [1] or even in the painting robots domain implemented in this thesis. New network architecture performing well on domains without an agent is an interesting topic that can be explored in further research.

5.1.2 Possible causes of the heuristic networks failure

The main issue of the heuristic networks is the choice of the heuristic they were trained on. The number of attacking pegs heuristic, which is actually the number of the next possible states, performed very well with the pruning solver. However, this heuristic might be too complex for the neural network to learn properly. Even the standard function computing this heuristic is complex. This is the most probable reason why the heuristic networks

did not perform well. New heuristic network architectures for more complex heuristics is a topic that can be explored in further research.

5.2 The (partial) success of the Painting robots domains

The experiment results show that the solvers in both domains function properly. The coverage in the case of single-agent domain is never lower than 58 % as seen in Table 4.7 in the case of the solver with transition and heuristic neural networks on a 9 by 9 grid. The multiagent domain coverage is never lower than 55 % in the case of two solvers on an 8 by 8 grid in Table 4.15. This grid size, however, has a generally lower coverage. This might be caused by the instances being more complex.

The transition networks work properly in these domains. All next states are generated and there are no wrong moves made. It was demonstrated that the lower coverage of single-agent domains solvers with transition network was caused by the dataset, which was generated and checked by the solver with the standard transition function. The performance of the solver with the standard transition function on the dataset checked by the solver with transition network is comparable as shown in Tables 4.8 - 4.11.

The heuristic neural network also provides satisfactory results in both domains. The average solution lengths of the heuristic network are comparable to the standard heuristic. Blind heuristic provides plans that are too long. These results can be seen in Tables 4.5 and 4.13. This shows that the network was able to learn the heuristic and use this information to provide a plan with a solution length that is much better than using the blind heuristic.

The number of expanded states is slightly larger for solvers using the heuristic network in the single agent domain. In the multiagent domain, the heuristic network solvers have a lower number of expanded states than the solvers using the standard heuristic. The blind heuristic is the best for both domains in terms of the number of expanded states, the price, however, is a large length of a solution. This can be seen in Tables 4.6 and 4.14.

The main disadvantage of transition and heuristic networks is the time they need to compute the result. Same as in the peg solitaire domain, in both single and multiagent painting robot domains, neural solvers were the slowest, as shown in Tables 4.4 and 4.12. This is caused by the computational complexity of these networks. However, some more advanced heuristics, such as LM-Cut are also very computationally demanding and the difference in time complexity might be smaller.

Chapter 6

Conclusion

The main goal of this thesis - to implement new domains of classical planning for neural network learning was ful lled. The peg solitaire and single and multiagent painting robots domains were implemented.

Implementation consisted of creating a domain representation, creating data generators for neural network training and implementing solvers. Transition and heuristic networks were trained on the data generated by data generators.

With trained networks implemented into the solvers, experiments were conducted to compare the performance of the solvers. The experiments compared solvers based on coverage, time, length of the found solution and the number of expanded states.

The experiment results showed that the peg solitaire domain was too complex for the used neural network models to solve. Transition network did not work properly, and the heuristic network did not provide any improvement over the blind heuristic. On the other hand, neural network solvers in the painting robot domains provided satisfactory results comparable to the standard methods. The downside of neural network solvers is the time needed to nd the solution, which is larger than the standard methods.

Outcomes shown in this thesis further expand the ndings in [1] that standard transition and heuristic functions in classical planning problems can be replaced by deep neural networks and perform well in comparison with standard methods. This thesis also shows that the current network architectures used for transition and heuristic networks are not able to solve more complex domains or domains without an agent. The best example is the peg solitaire domain where the networks were not able to learn the transition and heuristic functions on this domain properly.

Results of this thesis show that there is still room for improvement and further research. The research of this eld is still ongoing in the classical planning group at the faculty. The optimistic results in the case of painting robots domain prove that this eld is worth expanding further.

Appendix A

Source code

A compressed folder containing the source codes and model parameters is attached to this thesis. Training data for the networks and generated instances are not included due to their size. The folder has the following structure:

- painting_robots
  - { experiments.jl - scripts used in experiments
  - { heur_att_network.jl - script for training heuristic network
  - { heur_data_generator.jl - huristic training data generator
  - { robots_expansion_run.jl - script for training transition network
  - { robots_problem_generator.jl - instance data generator
  - { robots_solver.jl - solver for painting robots domain
  - { robots_step_gen.jl - transition network data generator
  - { tests.jl - unit tests
  - { parameters - folder containing neural network models parameters
- peg_solitaire
  - { boards.jl - instance data generator
  - { experiments.jl - scripts used in experiments
  - { heur_att_network.jl - script for training attention heuristic network
  - { heur_conv_run.jl - script for training convolutional heuristic network
  - { heur_gen.jl - huristic training data generator
  - { peg_expansion_run.jl - script for training transition network
  - { pruning_solver.jl - pruning solver for peg solitaire domain

- solver.jl - A* solver for peg solitaire domain
- step_generator.jl - transition network data generator
- step_masks.jl - masks used in transition function data generation
- parameters - folder containing neural network models parameters

Bibliography

[1] U. Michaela, *Učení přechodových funkcí v klasickém plánování pomocí hlubokých neuronových sítí*. Ceske vysoke ucen technicke v Praze. Vypocetn a informacn centrum., 2020.

[2] J. A. Hendler, A. Tate, and M. Drummond, \Ai planning: Systems and techniques", *AI magazine*, vol. 11, no. 2, pp. 61{61, 1990.

[3] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. 2002.

[4] 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.

[5] M. Fox and D. Long, \Pddl2. 1: An extension to pddl for expressing temporal planning domains", *Journal of artificial intelligence research*, vol. 20, pp. 61{124, 2003.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. 2009.

[7] F. Pommerening and M. Helmert, \Incremental lm-cut", in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 23, 2013.

[8] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Reading, MA, 1984.

[9] Y. Bengio, I. Goodfellow, and A. Courville, *Deep learning*. MIT press Massachusetts, USA: 2017, vol. 1.

[10] G. Cybenko, \Approximation by superpositions of a sigmoidal function", *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303{314, 1989.

[11] H. B. Curry, \The method of steepest descent for non-linear minimization problems", *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258{261, 1944.

[12] J. Zhang, \Gradient descent based optimization algorithms for deep learning models training", *arXiv preprint arXiv:1903.03614*, 2019.

[13] 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.

[14] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, \Handwritten digit recognition with a back-propagation network", in *Advances in neural information processing systems*, 1990, pp. 396{404.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, \Attention is all you need", *arXiv preprint arXiv:1706.03762*, 2017.