Constrained Shortest Path Search with Graph Convolutional Neural Networks

Kevin Osanlou[1,3], **Christophe Guettier**[1], **Andrei Bursuc**[2], **Tristan Cazenave**[3], **Eric Jacopin**[4],
[1] Safran Electronics & Defense
[2] Safran SA
[3] LAMSADE, Université Paris-Dauphine
[4] MACCLIA, Ecoles de Saint-Cyr Coëtquidan
kevin.osanlou@safrangroup.com christophe.guettier@safrangroup.com andrei.bursuc@safrangroup.com
tristan.cazenave@lamsade.dauphine.fr eric.jacopin@st-cyr.terre-net.defense.gouv.fr

Abstract

Planning for Autonomous Unmanned Ground Vehicles (AUGV) is still a challenge, especially in difficult, off-road, critical situations. Automatic planning can be used to reach mission objectives, to perform navigation or maneuvers. Most of the time, the problem consists in finding a path from a source to a destination, while satisfying some operational constraints. In a graph without negative cycles, the computation of the single-pair shortest path from a start node to an end node is solved in polynomial time. Additional constraints on the solution path can however make the problem harder to solve. This becomes the case when we need the path to pass through a few mandatory nodes without requiring a specific order of visit. The complexity grows exponentially with the number of mandatory nodes to visit. In this paper, we focus on shortest path search with mandatory nodes on a given connected graph. We propose a hybrid model that combines a constraint-based solver and a graph convolutional neural network to improve search performance. Promising results are obtained on realistic scenarios.

1 Introduction

Autonomous unmanned ground vehicle (AUGV) operations are constrained by terrain structure, observation abilities, embedded resources. Missions must be executed in minimal time, while meeting objectives. This is the case, for examples, in disaster relief, logistics or area surveillance, where maneuvers must consider terrain knowledge. In most of applications, the AUGV ability to maneuver in its environment has a direct impact on operational efficiency.

AUGV integrates several perception capabilities (on-line mapping, geolocation, optronics, LIDAR) in order to update its situation awareness on-line. This knowledge is used by various on-board planning layers to maintain mission goals, provide navigation waypoints and dynamically construct platform maneuvers. Resulting actions and navigation plans are used for controlling the robotic platform. Once the plans are computed, the AUGV automatically manages its trajectory and follows the navigation waypoints using control algorithms and time sequence. Such autonomous system architectures are challenging, especially because some mission data (terrain, objectives, available resources) are known off-line and others are acquired on-line. The planning problem also involves technical actions (observations, measurements, communications, etc.) to realize on some mandatory waypoints along the navigation plan [Guettier and Lucas, 2016].

Figure 1 presents the *eRider*, a UGV that has high mobility abilities, and is able to perform cross-over maneuvers on difficult terrain for exploring disaster areas.

The paper focuses on the automatic planning algorithms, and more specifically, on the ability to guide the problem solving with machine learning techniques.

For such problems, classical robotic systems integrate A* algorithms [Hart *et al.*, 1968], as a best-first search approach in the space of available paths. For a complete overview of static algorithms (such as A*), replanning algorithms (D*), anytime algorithms (e.g. ARA*), and anytime replanning algorithms (AD*), see [Ferguson *et al.*, 2005]. Representative applications to autonomous systems can be very realistic, as reported in [Meuleau *et al.*, 2009] and [Meuleau *et al.*, 2011].

Algorithms stemming from A* can handle some heuristic metrics but can become complex to develop when dealing with several constraints simultaneously like mandatory waypoints and distance metrics. Our approach combines a Constraint Programming (CP) method, with novel machine learning techniques. CP provides a powerful baseline to model and

Figure 1: The *eRider*, developed by SAFRAN, is an all-purpose optionally piloted vehicle that can patrol a given area, observe at long range or carry goods for various off-road applications in difficult environments. These specific vehicles can be either piloted or turned instantaneously into AUGV.

solve combinatorial and / or constraint satisfaction problems (CSP). It has been introduced in the late 70s [Laurière, 1978] and has been developed until now [Hentenryck *et al.*, 1998; Ajili and Wallace, 2004; Carlsson, 2015], with several real-world autonomous system applications, in space [Bornschlegl *et al.*, 2000; Simonin *et al.*, 2015], aeronautics [Guettier *et al.*, 2002; Guettier and Lucas, 2016] and defense [Guettier *et al.*, 2015].

Convolutional neural networks (CNNs) have proven to be very efficient when it comes to image recognition [Krizhevsky *et al.*, 2012]. They use a variation of multilayer perceptrons designed to require minimal preprocessing, and are capable of detecting complex patterns in the image. In this paper, we are dealing with graphs representing maneuvers or off-road navigation. Instead of CNNs, the paper focuses on graph convolutional neural networks (GCNNs), a recent variant for learning complex patterns in graph data. We show that GCNNs are capable of making decisions to solve path-related problems. In this work we combine this type of machine learning algorithms with constraint solving methods for planning.

The paper is organized as follows. The first section (§ 2) describes AUGV mission, environment and applications. The following section (§ 3) presents the CP approach, problem formalization and resolution. Section (§ 4) describes the graph-based learning algorithm and section (§ 5) the data generation schema for training. Last sections (§ 6) and (§ 7) discuss results, highlight further work and conclude. Related works are provided along the different sections.

2 Context and problem presentation

In modern AUGV architectures, path planning is performed on the fly, responding to an operator demand (mission update), a terrain obstacle, or free space discovery. In logistics, such operator demands correspond to movement requests for pick-up and delivery, while in area surveillance or disaster relief, the platform must recon some specific areas.

Figure 2 shows a flooded area and possible paths to assess disaster damages. Possible paths are defined using a graph representation, where edges and nodes represent respectively ground mobility and accessible waypoints. With the given

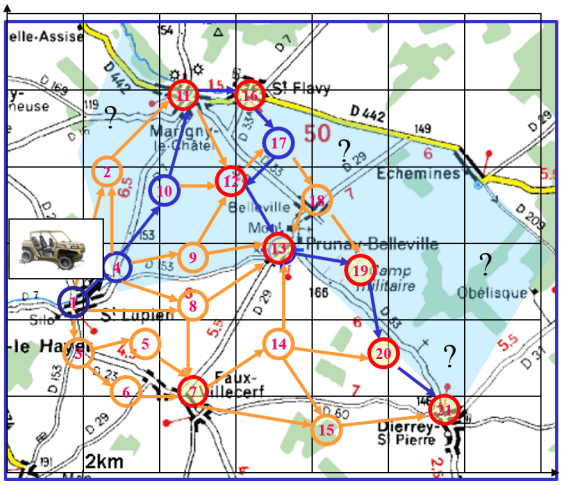

Figure 2: Search and rescue mission. Potential paths for a manned vehicle and a AUGV throughout a flooded area between the Seine and the Vanne rivers in the area of Troyes, France. The blue area represents the expected flooded area. Red circles represent possible waypoints while blue circles represent mandatory ones. Orange edges represent trafficability and the blue ones a potential optimal solution.

applications, graphs are defined during mission preparation, by terrain analysis and situation assessment. The UGV system responds to an operator demand by finding a route from a starting point to a destination, and by maneuvering through mandatory waypoints. During the mission, some paths may not be trafficable or new flooded areas to investigate may be identified.

These constraints depend on both environment situation and mission objectives:

- Environment situation: new muddy areas can occur, affecting cross-over duration between two waypoints or increasing the risk of losing platform control.
- Mission objectives: mandatory waypoints are imposed by the user and can correspond, for instance, to observation spots.

In figure 2, the UGV starts from its initial position (blue circle) and must visit waypoints $\{W_1, W_2, W_3, W_4\}$. In the disaster relief scenario, areas in blue are flooded and the disaster perimeter must be evaluated by the vehicle. All nodes circled in red have to be visited, as refugees and casualties are likely to be found there. The main criterion to minimize is the global traverse duration that meets all visit objectives. A typical damage assessment would require up to 10 mandatory nodes to visit. Using the UGV system, the remote operator can update the situation and set objectives accordingly, which usually yields replanning events. To achieve high operational efficiency, path planning must be solved *just in time* to execute maneuvers smoothly. However, such planning problems rapidly become NP-hard and state-of-the-art solvers may involve heavy processing loads, whereas a solution is required right away.

Our proposed approach is to learn problem invariants from the terrain structure in order to accelerate a model-based planner. This can be done by training a neural network on several

problem instances. Using feedback from the neural network, the model-based planner can efficiently solve new problem instances. Learning from terrain data can be performed off-line, such that the knowledge acquired can be used on-line to accelerate the on-board planner. However, critical missions such as disaster relief necessitate rapid vehicle deployment and it is not possible to pre-compute problem instances for heavy datasets.

In this paper we consider two scenarios, leading to two evaluation benchmarks. The first scenario, b_1, is a fine grain maneuver on a muddy terrain of small size (referred as benchmark *'maneuver'*), while the second one, b_2, is a coarse mapping (referred as benchmark *'exploration'*) of a disaster risk on a shore environment nearby a city.

3 Problem formalization and resolution approach

This section presents the global constraint programming approach to solve complex planning problems. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a connected graph. A typical instance I of the kind of path-planning problem we consider is defined as follows:

$$I = (s, d, M)$$

where:

- $s \in \mathcal{V}$ is the start node in graph $\mathcal{G}$,
- $d \in \mathcal{V}$ is the destination node in graph $\mathcal{G}$,
- $M \subset \mathcal{V}$ is a set of distinct mandatory nodes that need to be visited at least once, regardless of the order of visit.

In order to solve instance I, one has to find a shortest path from node s to node d that passes by each node in M at least once. There is no limit to how many times a node can be visited in a path. Since the graph is connected, there is a solution path to every existing instances. Let A be the adjacency matrix of graph $\mathcal{G}$, used in (§4) by our neural network, defined as follows:

$$A_{vv'} = \begin{cases} 0, & \text{if there is no edge from node } v \text{ to node } v' \\ w_{vv'}, & \text{otherwise, and the weight of the edge is } w_{vv'} \end{cases}$$

Let A' be the cost matrix of graph $\mathcal{G}$, used by our path-planning solver, defined as follows:

$$A'_{vv'} = \begin{cases} \infty, & \text{if there is no edge from node } v \text{ to node } v' \\ w_{vv'}, & \text{otherwise, and the weight of the edge is } w_{vv'} \end{cases}$$

3.1 Constraint Programming for Navigation and Maneuver Planning

In our approach, planning is achieved using Constraint Programming (CP) techniques, under a model-based development approach. In CP, it is possible to design global search algorithms that guarantee completeness and optimality. A CSP, formulated within a CP environment, is composed of a set of variables, their domains and algebraic constraints, that are based on problem discretization. With CP, a declarative formulation of the constraints to satisfy is provided which is decoupled from the search algorithms, so that both can be worked out independently. The CSP formulation and search algorithms exposed in the paper are implemented with the CLP(FD) domain of SICStus Prolog library [Carlsson, 2015]. It uses the state-of-the-art in discrete constrained optimization techniques Arc Consistency-5 (AC-5) [Deville and Van Hentenryck, 1991; Van Hentenryck *et al.*, 1992] for constraint propagation, managed by CLP(FD) predicates. With AC-5, variable domains get reduced until a fixed point is reached by constraint propagation. The search technique is hybridized by statically defining the search exploration structure using probing and learning on multiple problem instances. This approach, named *probe learning*, relies on several instances of a problem to build up the search tree structure.

3.2 Planning model with Flow Constraints

The set of flow variables $\varphi_u \in \{0, 1\}$ models a possible path from $start \in X$ to $end \in X$, where an edge u belongs to the navigation plan if and only if a decision variable $\varphi_u = 1$, 0 otherwise. The resulting navigation plan, can be represented as $\Phi = \{u |\ u \in U,\ \varphi_u = 1\}$. From an initial position to a requested final one, path consistency is enforced by flow conservation equations, where $\omega^+(x) \subset U$ and $\omega^-(x) \subset U$ represent respectively outgoing and incoming edges from vertex x. Since flow variables are $\{0, 1\}$, equation (1) ensures path connectivity and uniqueness while equation (2) imposes limit conditions for starting and ending the path:

$$\sum_{u \in \omega^+(x)} \varphi_u = \sum_{u \in \omega^-(x)} \varphi_u \leq N \tag{1}$$

$$\sum_{u \in \omega^+(start)} \varphi_u = 1, \quad \sum_{u \in \omega^-(end)} \varphi_u = 1, \tag{2}$$

These constraints provide a linear chain alternating pass-by waypoint and navigation along the graph edges. Constant N indicates the maximum number of times the vehicle can pass by a waypoint. With this formulation, the plan may contain cycles over several waypoints. Mandatory waypoints are imposed using constraint (3). The total path length is given by the metric (4), and we will consider the path length as the optimization criterion to minimize in the context of this paper:

$$\forall i \in M \sum_{u \in \omega^+(i)} \varphi_u \geq 1 \tag{3}$$

$$D_v = \sum_{v'v \in \omega^-(v)} \varphi_{v'v} w_{vv'} \tag{4}$$

3.3 Global search algorithm

The global search technique under consideration guarantees completeness, solution optimality and proof of optimality. It relies on three main algorithmic components:

- Variable filtering with correct values, using specific labeling predicates to instantiate problem domain variables. AC being incomplete, value filtering guarantees search completeness.

- Tree search with standard backtracking when variable instantiation fails.
- Branch and Bound (B&B) for cost optimization, using minimize predicate.

Designing a good search technique consists in finding the right variables ordering and value filtering, accelerated by domain or generic heuristics. A static probe provides an initial variable selection ordering, computed before running the global branch and bound search. Note that in general probing techniques [Sakkout and Wallace, 2000], the order can be redefined within the search structure [Ruml, 2001]. Similarly, in our approach, the variable selection order provided by the probe can still be iteratively updated by the labeling strategy that makes use of other variable selection heuristics. Mainly, first fail variable selection is used in addition to the initial probing order. These algorithmic designs have already been reported with different probing heuristics [Guettier and Lucas, 2016], such as A* or meta-heuristics such as Ant Colony Optimization [Lucas *et al.*, 2010], [Lucas *et al.*, 2009]. In our design, the search is still complete, guarantying proof of optimality, but demonstrates efficient pruning. Instead of these heuristics techniques, we choose to train with multiple instances the probing mechanism that provides a tentative variable order to the global search.

4 Neural network training

In this section, we present how a neural network can be trained on a particular graph. The aim is to let the neural network learn to approximate the behavior of a model-based planner on the graph. To this end, we first use our solver to compute solutions for several random instances and use them as training data. Then we train the neural network for solving these instances by using the previously generated solutions as supervision. We want to leverage the powerful mechanism of neural networks to guide our path planner. We first provide a brief introduction to neural networks and their main concepts.

4.1 Neural Networks

In recent years, neural networks (NNs), in particular deep neural networks, have achieved major breakthroughs in various areas of computer vision (image classification [Krizhevsky *et al.*, 2012], [Simonyan and Zisserman, 2014], [He *et al.*, 2016], object detection [Ren *et al.*, 2015], [Redmon *et al.*, 2016], [He *et al.*, 2017], semantic segmentation [Long *et al.*, 2015]), neural machine translation [Sutskever *et al.*, 2014], computer games [Silver *et al.*, 2016], [Silver *et al.*, 2017] and many other fields. While the fundamental principles of training neural networks are known since many years, the recent improvements are due to a mix of availability of large image datasets, advances in GPU-based computation and increased shared community effort.

Deep neural networks enable multiple levels of abstraction of data by using models with millions of trainable parameters coupled with non-linear transformations of the input data. It is known that a sufficiently large neural network can approximate any continuous function [Funahashi, 1989], although the cost of training such a network can be prohibitive. With this in mind, we attempt to train a neural network to approximate the behavior of a model-based planner.

In spite of the complex structure of a NN, the main mechanism is straightforward. A *feedforward neural network*, or *multi-layer perceptron (MLP)*, with L layers describes a function $f(\mathbf{x};\boldsymbol{\theta}) : \mathbb{R}^{d_\mathbf{x}} \mapsto \mathbb{R}^{d_\mathbf{y}}$ that maps an input vector $\mathbf{x} \in \mathbb{R}^{d_\mathbf{x}}$ to an output vector $\mathbf{y} \in \mathbb{R}^{d_\mathbf{y}}$. $\mathbf{x}$ is the input data that we need to analyze (*e.g.* an image, a signal, a graph, etc.), while $\mathbf{y}$ is the expected decision from the NN (*e.g.* a class index, a heatmap, etc.). The function f performs L successive operations over the input $\mathbf{x}$:

$$h^{(l)} = f^{(l)}(h^{(l-1)};\theta^{(l)}), \qquad l = 1,\dots,L \tag{5}$$

where $h^{(l)}$ is the hidden state of the network (*i.e.* features from intermediate layers) and $f(h^{(l-1)};\theta^{(l)}) : \mathbb{R}^{d_{l-1}} \mapsto \mathbb{R}^{d_l}$ is the mapping function performed at layer l; $h_0 = \mathbf{x}$. In other words, $f(\mathbf{x}) = f^{(L)}(f^{(L-1)}(\dots f^{(1)}(\mathbf{x})\dots))$. Each intermediate mapping depends on the output of the previous layer and on a set of trainable parameters $\theta^{(l)}$. We denote by $\boldsymbol{\theta} = \{\theta^{(1)},\dots,\theta^{(L)}\}$ the entire set of parameters of the network. The intermediate functions $f(h^{(l-1)};\theta^{(l)})$ have the form:

$$f^{(l)}(h^{(l-1)};\theta^{(l)}) = \sigma\left(\theta^{(l)}h^{(l-1)} + b^{(l)}\right), \tag{6}$$

where $\theta^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ and $b^{(l)} \in \mathbb{R}^{d_l}$ are the trainable parameters and the bias, while $\sigma(\cdot)$ is an *activation* function which is applied individually to each element of its input vector to introduce non-linearities. Intermediate layers are actually a combination of linear classifiers followed by a piecewise non-linearity. Layers with this form are termed *fully-connected layers*.

NNs are typically trained using labeled training data, *i.e.* a set of input-output pairs $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1,\dots,N$, where N is the size of the training set. During training we aim to minimize the training loss:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i), \tag{7}$$

where $\hat{\mathbf{y}}_i = f(\mathbf{x}_i;\boldsymbol{\theta})$ is the estimation of $\mathbf{y}_i$ by the NN and $\ell : \mathbb{R}^{d_L} \times \mathbb{R}^{d_L} \mapsto \mathbb{R}$ is the loss function. ℓ measures the distance between the true label $\mathbf{y}_i$ and the estimated one $\hat{\mathbf{y}}_i$. Through *backpropagation*, the information from the loss is transmitted to all $\boldsymbol{\theta}$ and gradients of each θ_l are computed w.r.t. the loss. The optimal values of the parameters $\boldsymbol{\theta}$ are then found via stochastic gradient descent (SGD) which updates $\boldsymbol{\theta}$ iteratively towards the minimization of $\mathcal{L}$. The input data is randomly grouped into mini-batches and parameters are updated after each pass. The entire dataset is passed through the network multiple times and the parameters are updated after each pass until reaching a satisfactory optimum. In this manner all the parameters of the NN are learned jointly and the pipeline allows the network to learn to extract features and to learn other more abstract features on top of the representations from lower layers.

CNNs [Fukushima and Miyake, 1982], [LeCun *et al.*, 1995] are a generalization of multilayer perceptrons for 2D

data. In convolutional layers, groups of parameters (which can be seen as small fully-connected layers) are slided across an input vector similarly to filters in image processing. This reduces significantly the number of parameters of the network since they are now *shared* across locations, whereas in fully connected layers there is a parameter for element of the input. Since the convolutional units act locally, the input to the network can have a variable size. A convolutional layer is also a combination of linear classifiers (6) and the output of such layer is 2D and is called *feature map*. CNNs are highly popular in most recent approaches for computer vision problems.

4.2 Graph Convolutional Networks

Graph Convolutional Neural Networks (GCNNs) are generalizations of CNNs to non-Euclidean graphs. GCNNs are in fact neural networks based on local operators on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which are derived from spectral graph theory. The filter parameters are typically shared over all locations in the graph, thus the name $convolutional$. In the past two years there has been a growing interest for transferring the intuitions and practices from deep neural networks on structured inputs to graphs [Gori *et al.*, 2005], [Henaff *et al.*, 2015], [Defferrard *et al.*, 2016], [Kipf and Welling, 2016]. [Bruna *et al.*, 2013] and [Henaff *et al.*, 2015] bridge spectral graph theory with multi-layer neural networks by learning smooth spectral multipliers of the graph Laplacian. Then [Defferrard *et al.*, 2016] and [Kipf and Welling, 2016] approximate these smooth filters in the spectral domain using polynomials of the graph Laplacian. The free parameters of the polynomials are learned by a neural network, avoiding the costly computation of the eigenvectors of the graph Laplacian. We refer the reader to [Bronstein *et al.*, 2017] for a comprehensive review on deep learning on graphs.

We consider here the approach of [Kipf and Welling, 2016]. The GCNNs have the following layer propagation rule:

$$h^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}h^{(l)}\theta^{(l)}\right), \tag{8}$$

where $\tilde{A} = A + I_N$ is the adjacency matrix of the graph with added self-connections such that when multiplying with $\tilde{A}$ we aggregate features vectors from both a node and its neighbors; I_N the identity matrix. $\tilde{D}$ is the diagonal node degree matrix of $\tilde{A}$. $\sigma(\cdot)$ is the activation function, which we set to $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$. The diagonal degree $\tilde{D}$ is employed for normalization of $\tilde{A}$ in order to avoid change of scales in the feature vectors when multiplying with $\tilde{A}$. [Kipf and Welling, 2016] argue that using a symmetric normalization, *i.e.* $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, ensures better dynamics compared to simple averaging of neighboring nodes in one-sided normalization $\tilde{D}^{-1}A$.

The input of our model is a vector $\mathbf{x}$, where $h^{(0)} = \mathbf{x}$, containing information about the problem instance, including the graph representation. We describe next the chosen encoding for vector $\mathbf{x}$.

4.3 Vector encoding of instances

For every instance $I = (s, d, M)$ of a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we associate a vector $\mathbf{x}$ made up of triplets features

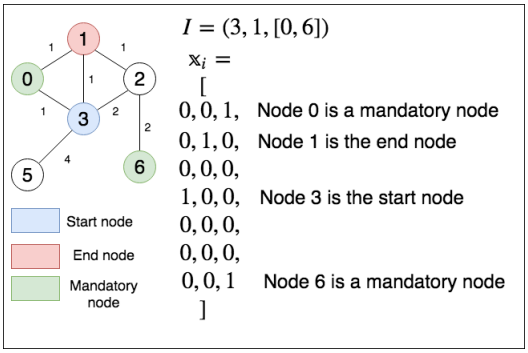

Figure 3: Graph with seven nodes. The instance I defined in the figure requires finding an optimal path from node 3 to node 1 which passes by node 0 and 6 at least once. $\mathbf{x}$ is the vector associated with instance I. The number written next to an edge is its cost.

per node in $\mathcal{G}$, making up for a total of $3\,|\mathcal{V}|$ features. The three features for a node $j \in \mathcal{V}$ are:

- A *start node* feature

$$s_j = \begin{cases} 1, & \text{if node } j \text{ is a start node in instance } I \\ 0, & \text{otherwise} \end{cases}$$

- An *end node* feature

$$e_j = \begin{cases} 1, & \text{if node } j \text{ is the end node in instance } I \\ 0, & \text{otherwise} \end{cases}$$

- A *mandatory node* feature

$$m_j = \begin{cases} 1, & \text{if node } j \text{ is a mandatory node in instance } I \\ 0, & \text{otherwise} \end{cases}$$

Vector $\mathbf{x}$ is the concatenation of these features:

$$\mathbf{x} = (s_1, e_1, m_1, s_2, e_2, m_2, ..., s_{|\mathcal{V}|}, e_{|\mathcal{V}|}, m_{|\mathcal{V}|}) \tag{9}$$

Figure 3 illustrates an example of an instance I in a graph and the associated vector $\mathbf{x}$.

4.4 Neural network architecture

We define a neural network f that uses a sequence of graph convolutions followed by a fully connected layer. The idea is to have the neural network take as input any instance I on the graph $\mathcal{G}$, and output a probability $\hat{\mathbf{y}}$ over which node should be visited first from the start node in an optimal path that solves I. The weights of the neural network $\boldsymbol{\theta}$ are tuned in the training phase for this purpose. We train the neural network on instances that have already been processed by the solver. Here, the solver serves as a *teacher* to the neural network and the neural network learns to approximate the solutions given by the solver.

The output $\hat{\mathbf{y}}$ of the neural network is a vector of size $|\mathcal{V}|$. We format the input vector $\mathbf{x}$ as following: we reshape $\mathbf{x}$ into a matrix of size $(|\mathcal{V}|, 3)$, which contains the *start*, *end* and *mandatory* features of every node by rows. This matrix becomes the input for f, which aims to extract and aggregate local information over layers starting from the input. The output of the convolutions, a matrix, is flattened into a vector by concatenating its rows. A fully connected layer then

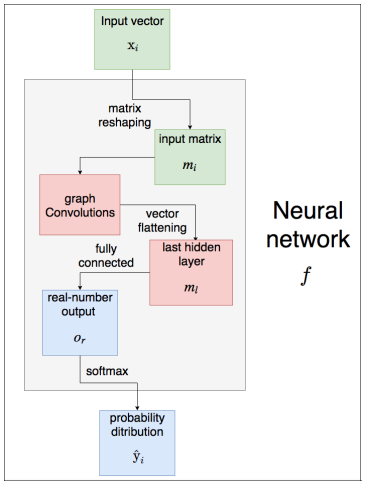

Figure 4: **Architecture of the neural network f:** f takes as input the vector $\mathbf{x}_i$ of an instance I_i and outputs a probability over all nodes in the graph, suggesting which node should be visited first.

links the flattened vector to a vector z of real-numbers of size $|\mathcal{V}|$. Finally we use a softmax function to obtain a probability distribution. Formally, the softmax function is given by:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{k=1}^{|\mathcal{V}|} e^{z_k}} \tag{10}$$

This formula ensures that: $\forall i \in \{1, 2, ..., |\mathcal{V}|\}$, $\text{softmax}(\mathbf{z})_i > 0$ and $\sum_{i=1}^{|V|} \text{softmax}(\mathbf{z})_i = 1$.

Once the training of the neural network is concluded, we expect it to return the next node to visit for any instance I in the same manner as the solver.

5 Data generation

This section provides a global schema for data generation, either for the learning purpose, or for experimenting the global search strategy. We also give the main processing principles to conduct the learning phase.

5.1 Number of instances

Suppose the connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has $|V| = n$ nodes. Let P_n be the set of all existing instances for graph $\mathcal{G}$. The number of instances $C_n = |P_n|$ that exist is given by the following formula:

$$C_n = 2!\binom{n}{2}\sum_{p=0}^{n-2}\binom{n-2}{p} \tag{11}$$

where:

- $2!\binom{n}{2} = n(n-1)$ is the number of existing combinations of source-destination pairs, taking ordering into account.
- $\sum_{p=0}^{n-2}\binom{n-2}{p}$ is, for every possible source-destination pair, the number of possible sets of mandatory nodes to visit. It can be simplified to 2^{n-2}

Therefore, equation (11) can be rewritten as:

$$C_n = 2^{n-2}n(n-1) \tag{12}$$

The number of instances for a graph with $n = 20$ is equal to 99 614 720. It is reasonable to assume that calculating the solutions of all instances becomes too time-consuming.

5.2 Instance generator

In order to generate instances $I_i = (s_i, d_i, M_i)$, a generator function is built. It returns a set of instances R, all of which are solvable since the graph $\mathcal{G}$ is connected. These instances are sampled out of the set of all possible instance configurations $P_n = \{I_1, I_2, ..., I_{2^{n-2}n(n-1)}\}$ such that they cover P_n as evenly as possible . For our experiments, we built 2 different connected graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ containing respectively 15 and 22 nodes, based on the two benchmarks b_1 and b_2. Although these graphs are undirected, our approach remains applicable to directed graphs. However, the problem instances generated must remain realistic in terms of mission data. To be close to some 'realistic' instances, we first generate a shortest path length among all pairs of starting to ending nodes (s_i, d_i) within the graph. We then apply a decimation ratio (typically 90%), keeping the 10% set of instances that have the longest paths. For each resulting pair (s_i, d_i), we then generate multiple random instances, with an increasing cardinality for the set of mandatory waypoints (typically from 1 to 10 mandatory waypoints). Each instance I_i generated in R is fed to our solver, providing an optimal p_i. The shortest path (Dijkstra) is set as probe, and for processing convenience, we set a 'time out' at 3 seconds for each instance. If no optimal solution can be proven, the instance is not considered for further learning process (although some suboptimal solutions may be found). Using this process, the number of instances generated and resolved are reported in Table 1.

Table 1: Number of instances resolved under 3 seconds by the solver out of the generation process, and used for the learning phase.

Mandatory waypoints #:	0	1	2	4	6	8	10
Instances for benchmark *maneuver* (b_1)							
generated (1256):	42	212	246	252	252	252	
optimally solved (651):	42	212	185	121	61	30	
Instances for benchmark *exploration* (b_2)							
generated (2503):	69	368	410	414	414	414	414
optimally solved (554):	69	265	168	42	9	1	0

Only 651 (over 1256 generated) and 554 (over 2503) are solved optimally for respectively benchmarks b_1 and b_2. Indeed, easy instances are optimally solved more often than difficult ones. Given the number of nodes in the graphs, and applying (11), we are working with less than 0.001% of the total instances. As a consequence, most of the learning is performed on a small number of easy instances. For each instance $I_i = (s_i, d_i, M_i)$, let $|M_i|$ be the number of mandatory nodes from M_i. The corresponding optimal path p_i found by our solver is a path that passes through all the nodes in $M_i = \{m_{i1}, m_{i2}, ..., m_{i|M_i|}\}$, *i.e.*: $p_i = \{s_i, v_{i1}, v_{i2}, ..., v_{iq}, d_i\}$, $m_{ij} \in \{v_{i1}, v_{i2}, ..., v_{iq}\}\ \forall j \in 1, 2, ..., |M_i|$.

Since our neural network model predicts the next node to visit for a given instance I, our data is reprocessed before the training phase. We use the following lemma to reprocess the data:

Lemma 1. *Let $I = (s, d, M)$ be a problem instance, and $p = \{s, v_1, v_2, v_3, ..., v_q, d\}$ an optimal path that solves I. It*

comes that:

- $\{v_1, v_2, v_3, ..., v_q, d\}$ *is an optimal solution for the instance* $(v_1, d, M \backslash \{v_1\})$
- $\{v_2, v_3, ..., v_q, d\}$ *is an optimal solution for the instance* $(v_2, d, M \backslash \{v_1, v_2\})$
- ...
- $\{v_q, d\}$ *is an optimal solution for the instance* $(v_q, d, M \backslash \{v_1, v_2, .., v_q\})$

More specifically: $\{v_i, v_{i+1}, ..., v_q, d\}$ *is an optimal solution for the instance* $(v_i, d, M \backslash \{v_1, v_2, ..., v_i\}) | \forall i \in 1, 2, ..., q$

Proof. Suppose that $\{v_1, v_2, v_3, ..., v_q, d\}$ is not an optimal solution for $(v_1, d, M \backslash \{v_1\})$. There is therefore a path p' that starts from v_1, ends in d, that visits every node in $M \backslash \{v_1\}$ with a lower cost than the path $\{v_1, v_2, v_3, ..., v_q, d\}$. Therefore $p = \{s, v_1, v_2, v_3, ..., v_q, d\}$ is not optimal for the original problem p since it does not take that shorter path, which is contradictory. It results that $\{v_1, v_2, v_3, ..., v_q, d\}$ is optimal for $(v_1, d, M \backslash \{v_1\})$. The same reasoning is applied recursively. □

5.3 Data processing

Let (I_i, p_i) be an instance-solution pair that was previously generated, such that $I_i = (s_i, d_i, M_i)$ and $p_i = \{s_i, v_{i1}, v_{i2}, v_{i3}, ..., v_{iq_i}, d_i\}$. This pair, which we call root pair, is split into several pairs $(I_{i,j}, p_{i,j}), \forall j \in 1, 2, ..., q_i$ in the same way as in Lemma 1. This guarantees that for each instance $I_{i,j}$, $p_{i,j}$ is an optimal solution path. For each newly obtained pair ($I_{i,j}$, $p_{i,j}$), we store $(I_{i,j}, t_{i,j})$ in a dataset d, where $t_{i,j} \in \mathcal{V}$ is the first node visited in path $p_{i,j}$ after the start node. The same process is applied for every root pair (I_i, p_i) which was stored. The dataset is shuffled to compensate for the correlation resulting from splitting root pairs (I_i, p_i) into pairs $(I_{i,j}, p_{i,j})$, which are children instances and whose solutions enable the solving of the parent instance.

5.4 Supervised learning

Following the creation of dataset $\mathcal{X}$, we train the neural network so that it can learn to approximate the behavior of our solver, and correctly predict the next step that should be taken in an optimal path for a given instance I. We take 80% of the data in dataset $\mathcal{X}$ for our training set. The validation set is given by the remaining 20% of the data in $\mathcal{X}$. We generate a separate set of instances for testing purposes in (§6.2). Let f be the function for the neural network defined previously. f takes as input a vector instance $\mathbf{x}_i$ and outputs a distribution vector $\hat{\mathbf{y}}_i$ which is the probability distribution over all nodes in the graph of the next node to visit: $f(\mathbf{x}_i; \boldsymbol{\theta}) = \hat{\mathbf{y}}_i$ where $\boldsymbol{\theta}$ are the weights of the neural network. We define the loss function $\mathcal{L}$ as the average of the logarithmic loss of the probabilities predicted by the neural network for each problem instance I_i with the actual label target node t_i. $\mathcal{L}$ is defined as follows:

$$\mathcal{L}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} -t_{ij} log(f(\mathbf{x}_i; \boldsymbol{\theta})_j) \tag{13}$$

- m is the number of training examples in the training set,
- n is the number of nodes in the graph,
- $\mathbf{x}_i$ is the vector of the problem instance I_i,
- t_{ij} is the variable that indicates whether for the problem instance I_i, the node j was the next optimal node to visit: $t_{ij} = 1$ if so, else $t_{ij} = 0$,
- $f(\mathbf{x}_i; \boldsymbol{\theta})_j$ is the probability the neural network outputs to visit node j for the problem instance I_i.

We ran the training on the data obtained from graphs $\mathcal{G}_1$ and $\mathcal{G}_2$. Training curves are available in the Appendix. The neural networks trained for each graph generalize relatively well on unknown instances (the neural network for graph $\mathcal{G}_1$ achieves 96% accuracy on its validation set, while the neural network for graph $\mathcal{G}_2$ achieves 92% accuracy on its validation set).

6 Experiments and results

In this section, we evaluate to which extent the neural network can help solve new instances. When solving an instance I, the neural network is used at search start, and given as input the corresponding vector $\mathbf{x}$. The output $\hat{\mathbf{y}}$ is used to create a preference ordering over all existing nodes in the graph, and corresponds to the preference of the next node to visit from the start node of the instance I. This order suggested by the neural network is provided to the solver at the root of the search tree, and children nodes of the root node will be visited in the order they appear in the preference ordering. Figure 5 depicts this process. The neural network is not used again to create preference orderings of root nodes of the resulting subtrees. We motivate this design choice by the fact that a large amount of operations is required for a feedforward pass. Probing with the neural network at every choice point would make the solving too slow and impractical.

6.1 Implementation details

We consider the following architecture for the neural network: 2 graph convolution layers with 10 hidden units, and a fully connected layer. We apply dropout on the units of the last graph convolution layer with a *keep rate* of 0.9. We use batch normalization [Ioffe and Szegedy, 2015] for every layer in the graph convolutions with decay of moving average $\varepsilon = 0.9$. The neural network is trained using a variant of the stochastic gradient descent (SGD) called *Adam* [Kingma and Ba, 2015]. *Adam* uses adaptive learning rates for each variable of the neural network to decrease the number of steps required to minimize the loss function. We set the learning rate to $\eta = 10^{-4}$ and train the neural network with mini-batches of data of size 32. To reduce overfitting, we do *early stopping*: the training is stopped once the performance on the validation set stops improving. Training takes less than an hour for each graph on an Nvidia Tesla V100 GPU. The neural network was implemented in Python using Tensorflow.

6.2 Performance evaluation

We generate instances for two test benchmarks, a maneuver benchmark, associated with graph $\mathcal{G}_1$, and an exploration

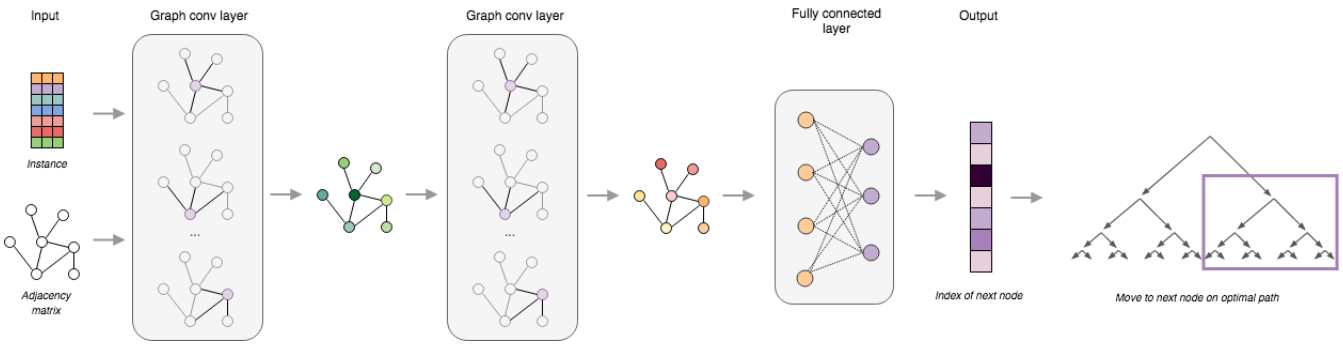

Figure 5: **Processing pipeline for path planning using GCNs:** The GCN takes as input the adjacency matrix with costs and the instance. Graph convolutional layers process each node in the graph and its neighbors. In the hidden layers, new features are generated for each node in the graph. In the last layer, the features are passed through a fully-connected layer and a softmax. The softmax layer indicates the next node in the optimal path.

benchmark, associated with graph $\mathcal{G}_2$. The first benchmark comprises 1008 instances, the second one 2208 instances. Note that these instances are generated with our instance generator, and therefore may contain anywhere from 0 to 10 mandatory nodes. We solve the instances using our original model-based planning solver without neural network support to obtain *reference* performance. Then, we evaluate solving performance on the same instances using *neural net probing*: a modified version of the solver based on an initial variable ordering (eg: probing) for the search tree by the neural network that was trained over data provided in (§ 5). In both cases, we only keep results where the proof of optimality could be achieved. Results are reported in Table 2, showing stable improvements on all datasets of instances.

Table 2: Number of instances resolved with proof of optimality. Comparison between the reference version and the neural network probing one (under 3 seconds 'time out').

Mandatory waypoints #:	3	5	7	9
Solving instances for benchmark *maneuver* (b_1)				
reference:	167	99	43	15
neural net probing:	220	185	131	89
Solving instances for benchmark *exploration* (b_2)				
reference:	104	29	4	1
neural net probing:	210	65	18	4

Table 3 summarizes the number of instances solved by both solvers under 3 seconds with more detailed search features. Average number of backtracks and solving time on *maneuver* are significantly lower, denoting an efficient pruning of the search tree. The situation is different for *exploration* with 10% more backtracks. Given that the graph for *exploration* contains more nodes than the graph for *maneuver*, this result is explained by more complex instances solved optimally with neural network probing, with an accordingly high number of backtracks. The reference solver, on the other hand, was unable to solve those instances, thus not taking into account the number of backtracks.

Higher performance could be obtained by training on more random instances. However, this would require solving more instances with the initial solver to generate training data, which brings a significant additional computational cost.

Table 3: Global search features for proof of optimality, comparison between the reference version and the neural network probing one (under 3 seconds 'time out').

Benchmarks	*Maneuver*	*Exploration*
Number of instances resolved		
reference	324	138
neural net probing	625	297
Average solving time for optimality		
reference	504	1044
neural net probing	394	1203
Average number of backtracks for optimality		
reference	55787	26065
neural net probing	35176	26504

7 Conclusion

In order to solve AUGV path planning problems with mandatory waypoints, we introduced an algorithm capable of significantly accelerating the performance of a constraint-based solving method. The algorithm requires the solving of multiple random instances to enable the training of the neural network, which is then used to accelerate the solver itself on new instances. The approach is efficient, even with small training datasets, as required by application preparation requirements. The performance obtained is realistic on two representative AUGV planning benchmarks. A drawback is that the neural network can only be used to accelerate the solver on the same geometric graph from which the random instances were solved, making it impractical if a solution is required right away on a previously unknown graph. Nevertheless, if the graph is known in advance, it becomes a better alternative to pre-computing the solution to all existing instances, as the number of instance configurations grows exponentially with the size of the graph.

8 Acknowledgments

We would like to thank Nicolas Fouquet for lending us a workstation for the duration of our experiments.

References

[Ajili and Wallace, 2004] F. Ajili and M. Wallace. Constraint and integer programming : toward a unified methodology. *Operations*

research/computer science interfaces series, 2004.

[Bornschlegl *et al.*, 2000] E. Bornschlegl, C. Guettier, and J-C. Poncet. Automatic planning for autonomous spacecraft constellations. In *Proceedings of the 2nd International NASA Workshop on Planning and Scheduling for Space*, San Francisco, California, 2000.

[Bronstein *et al.*, 2017] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[Bruna *et al.*, 2013] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[Carlsson, 2015] M. Carlsson. *SICSTUS Prolog user's manual*, 2015.

[Defferrard *et al.*, 2016] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

[Deville and Van Hentenryck, 1991] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In *Proceedings of the 12th International Joint Conference on Artificial intelligence (IJCAI)*, volume 1, pages 325–330, 1991.

[Ferguson *et al.*, 2005] David Ferguson, Maxim Likhachev, and Anthony (Tony) Stentz. A guide to heuristic-based path planning. In *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[Fukushima and Miyake, 1982] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[Funahashi, 1989] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[Gori *et al.*, 2005] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 729–734. IEEE, 2005.

[Guettier and Lucas, 2016] C. Guettier and F. Lucas. A constraint-based approach for planning unmanned aerial vehicle activities. *The Knowledge Engineering Review*, 31(5):486–497, 2016.

[Guettier *et al.*, 2002] C. Guettier, B. Allo, V. Legendre, and J-C. Poncet. Constraint model-based planning and scheduling with multiple resources and complex collaboration schema. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, 2002.

[Guettier *et al.*, 2015] C. Guettier, W. Lamal, I. Mayk, and J. Yelloz. Design and experiment of a collaborative planning service for netcentric international brigade command. In *IAAI*, 2015.

[Hart *et al.*, 1968] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science and Cybernetics*, 4(2):100–107, 1968.

[He *et al.*, 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[He *et al.*, 2017] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2980–2988. IEEE, 2017.

[Henaff *et al.*, 2015] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

[Hentenryck *et al.*, 1998] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *J. Log. Program.*, 37(1-3):139–164, 1998.

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[Kingma and Ba, 2015] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference for Learning Representations*, 2015.

[Kipf and Welling, 2016] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[Krizhevsky *et al.*, 2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[Laurière, 1978] J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[LeCun *et al.*, 1995] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[Long *et al.*, 2015] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[Lucas *et al.*, 2009] F. Lucas, C. Guettier, and P. Siarry. Hybridisation of constraint solving with an ant colony algorithm for online vehicle path planning. In *Proceedings of the 4th Workshop on Planning and Plan Execution for Real-World Systems, ICAPS'09*, 2009.

[Lucas *et al.*, 2010] F. Lucas, C. Guettier, P. Siarry, A. de La Fortelle, and A-M. Milcent. Constrained navigation with mandatory waypoints in uncertain environment. *International Journal of Information Sciences and Computer Engineering (IJISCE)*, 1:75–85, 2010.

[Meuleau *et al.*, 2009] N. Meuleau, C. Plaunt, D. Smith, and T. Smith. Emergency landing planning for damaged aircraft. In *Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference*, 2009.

[Meuleau *et al.*, 2011] N. Meuleau, C. Neukom, C. Plaunt, D.E. Smith, and T. Smithy. The emergency landing planner experiment. In *21st International Conference on Automated Planning and Scheduling*, 2011.

[Redmon *et al.*, 2016] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[Ren *et al.*, 2015] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with

region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[Ruml, 2001] W. Ruml. Incomplete tree search using adaptive probing. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, volume 1, pages 235–241. Morgan Kaufmann Publishers Inc., 2001.

[Sakkout and Wallace, 2000] H. El Sakkout and M. Wallace. Probe backtrack search for minimal perturbations in dynamic scheduling. *Constraints Journal*, 5(4):359–388, 2000.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[Silver *et al.*, 2017] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[Simonin *et al.*, 2015] G. Simonin, C. Artigues, E. Hebrard, and P. Lopez. Scheduling scientific experiments for comet exploration. *Constraints*, 20:77–99, 2015.

[Simonyan and Zisserman, 2014] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[Sutskever *et al.*, 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[Van Hentenryck *et al.*, 1992] P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 10 1992.

A Appendix: Learning curves

A.1 Graph $\mathcal{G}_1$

$|V| = 15$

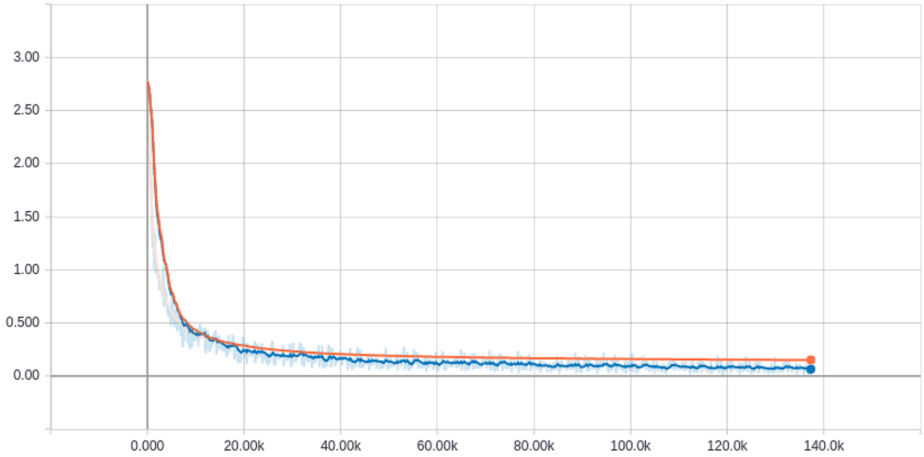

Figure 6: *The loss function curve during training. The X-axis is the iteration step. The Y-axis is the average logarithmic log loss. The blue curve is the loss on a batch of training examples from the training set, the orange curve is the loss for the cross-validation set. The curves are smoothed by a coefficient $\mu = 0.8$. We observe that the loss of the cross-validation set follows the loss of the training set as the training steps go on, while staying only slightly superior. This means the model, trained on instances of the training set, generalizes well on instances that are not part of the training set.*

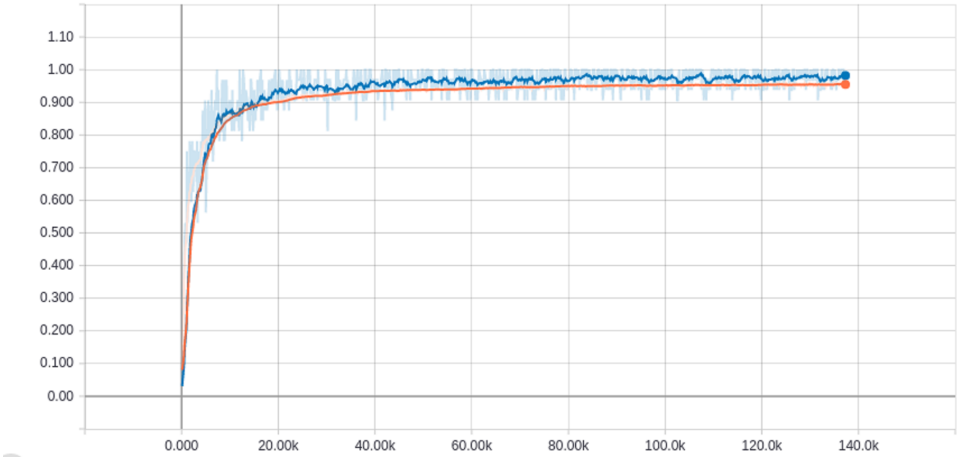

Figure 7: *The accuracy curve during training. The X-axis is the iteration step. The Y-axis is the prediction accuracy. The blue curve is the accuracy on a batch of training examples from the training set, the orange curve is the accuracy for the cross-validation set. The curves are smoothed by a coefficient $\mu = 0.8$. We observe that the accuracy of the cross-validation set follows the accuracy of the training set as the training steps go on, while staying only slightly inferior. This means the model, trained on instances of the training set, correctly predicts instances that are not part of the training set. By the end of the training phase, the model achieves an accuracy of 96% on the cross-validation set.*

A.2 Graph $\mathcal{G}_2$

$|V| = 22$

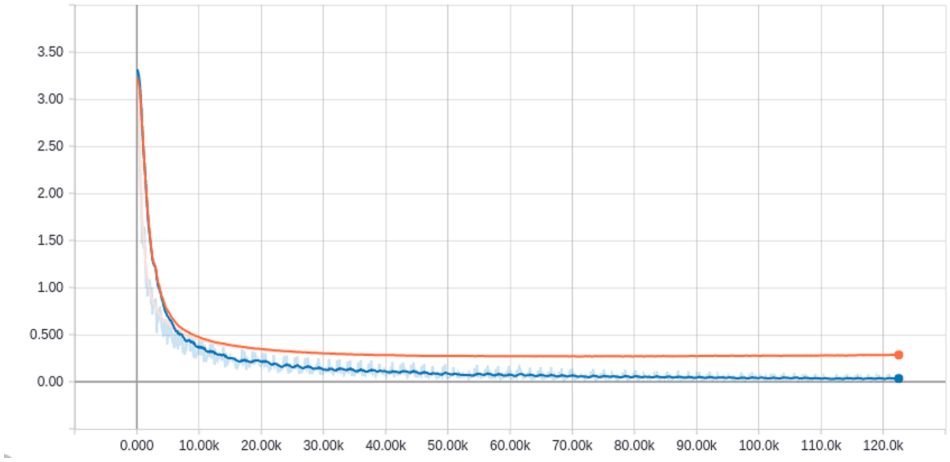

Figure 8: *The loss function curve during training. The X-axis is the iteration step. The Y-axis is the average logarithmic log loss. The blue curve is the loss on a batch of training examples from the training set, the orange curve is the loss for the cross-validation set. The curves are smoothed by a coefficient* $\mu = 0.8$*. We observe that the loss of the cross-validation set follows the loss of the training set as the training steps go on, while staying only slightly superior. This means the model, trained on instances of the training set, generalizes well on instances that are not part of the training set.*

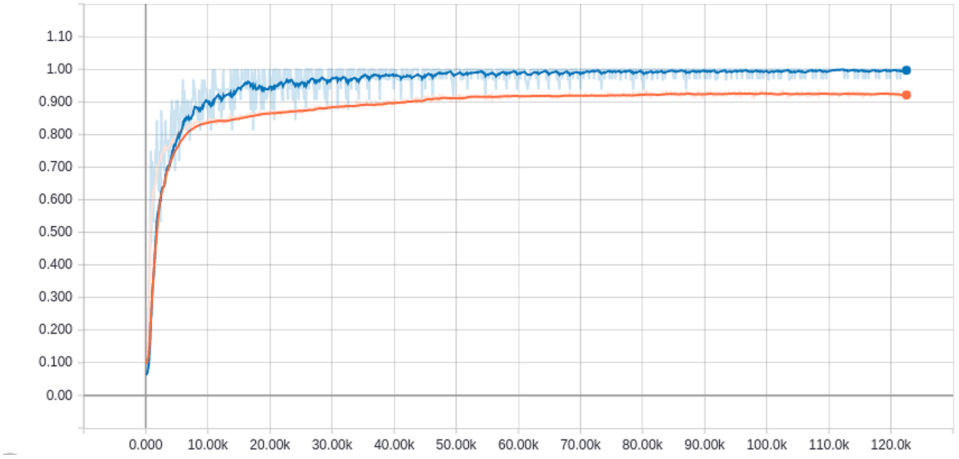

Figure 9: *The accuracy curve during training. The X-axis is the iteration step. The Y-axis is the prediction accuracy. The blue curve is the accuracy on a batch of training examples from the training set, the orange curve is the accuracy for the cross-validation set. The curves are smoothed by a coefficient* $\mu = 0.8$*. We observe that the accuracy of the cross-validation set follows the accuracy of the training set as the training steps go on, while staying only slightly inferior. This means the model, trained on instances of the training set, correctly predicts instances that are not part of the training set. By the end of the training phase, the model achieves an accuracy of 92% on the cross-validation set.*