Optimal Solving of Constrained Path-Planning Problems with Graph Convolutional Networks and Optimized Tree Search

Kevin Osanlou[1,2] Andrei Bursuc[3] Christophe Guettier[1] Tristan Cazenave[2] Eric Jacopin[4]
[1]Safran Electronics & Defense [2]LAMSADE, Paris-Dauphine University [3]valeo.ai
[4]CREC Saint-Cyr, Coetquidan School Campus
{kevin.osanlou,christophe.guettier}@safrangroup.com andrei.bursuc@valeo.com
tristan.cazenave@lamsade.dauphine.fr eric.jacopin@st-cyr.terre-net.defense.gouv.fr

***Abstract*— Learning-based methods are growing prominence for planning purposes. However there are fewer approaches for learning-assisted constrained path-planning on graphs, even though there are multiple downstream practical applications. This is the case for constrained-path planning for Autonomous Unmanned Ground Vehicles (AUGV), typically deployed in disaster relief or search and rescue applications. In off-road environments, the AUGV must dynamically optimize a source-destination path under various operational constraints, out of which several are difficult to predict in advance and need to be addressed on-line. We propose a hybrid solving planner that combines machine learning models and an optimal solver. More specifically, a *graph convolutional network* (GCN) is used to assist a *branch and bound* (B&B) algorithm in handling the constraints. We carry out experiments on realistic scenarios and show that GCN support enables substantial speedup and smoother scaling to harder problems.**

I. INTRODUCTION

Automated path-planning is an area of interest in AI with a wide panel of applications. The ability to efficiently plan an optimal path in a geometric graph that meets a set of requirements is becoming increasingly critical in a world where autonomy is starting to prevail. The requirements are typically a set of constraints imposed on the solution path which make it more difficult to compute. In the case of autonomous unmanned ground vehicles (AUGV), terrain structure is represented by a geometric graph, and maneuvers must consider terrain knowledge. Disaster relief, logistics, or area surveillance are a few among many applications for which online constrained path-planning algorithms enable autonomous mobility using perception and control functionalities. The ability of the AUGV to efficiently come up with an optimal path for a given mission has a direct impact on operational efficiency, highlighting the importance of an efficient path-planner.

For such problems, classical robotic systems integrate A* algorithms [1] 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*), we refer the reader to Ferguson et al. [2]. Representative applications to autonomous systems can be very realistic, as reported in [3] and [4]. Tree search algorithms stemming from A* require the specification of a planning domain where constraints are modeled, and can be more or less efficient depending on the affinity of the search heuristic with the planning context. In this work, we focus on the branch and bound (B&B) tree search algorithm instead [5]. More specifically, our study focuses on the performance gain when it is coupled with machine learning techniques.

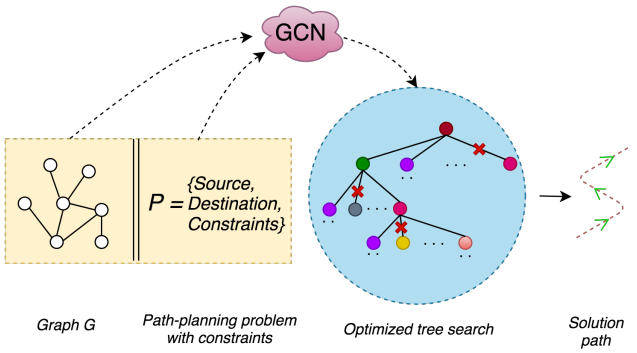

Fig. 1. Proposed framework for solving a path-planning problem with constraints in a graph with a GCN-assisted solver. The GCN takes as input the graph and the problem, and provides relevant information to speed up a tree search, after which an optimal solution path can be built.

Convolutional neural networks (CNNs) have proven to be very efficient when it comes to image recognition [6]. 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 which represent either maneuvers or off-road navigation. Instead of CNNs, the paper focuses on graph convolutional neural networks (GCNs), a recent variant for learning complex patterns in graph data. Several structures of GCNs have emerged ([7], [8], [9]). One of the main reasons GCNs are preferred over CNNs for graph processing is that nodes and edges are attributed relevant features. CNNs are not able to learn features of an equivalent quality from an image of the graph. Moreover, CNNs are not invariant to node permutations, an issue GCNs do not share.

In this paper, we study a GCN-based approach for constrained path-solving. We focus on exact resolution methods for different path-related problems in a specific graph. We run experiments on realistic AUGV scenarios for which we consider mandatory pass-by nodes as type of constraint. This makes path-planning similar to the traveling salesman problem (TSP). The TSP is an NP-hard problem for which there exist efficient approximate solvers [10], however it remains a challenge for exact approaches ([11], [12], [13]).

We make the following contributions. First, we define a GCN architecture suited for the considered problem. Second, we propose a self-supervised training strategy for the GCN. We then provide a framework which combines the GCN with basic depth-first branch and bound (B&B) algorithm. Lastly, results exhibit accelerated solving on realistic benchmark problems.

II. Context and Problem Formalization

We consider a weighted connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, A)$, where $\mathcal{V}$ is the set of vertices, $\mathcal{E}$ the set of edges and A the adjacency matrix of the graph. While in this work we choose to deal with realistic AUGV scenario graphs which are undirected (§VII), our approach is equally applicable to directed graphs. Let I be a path-planning problem instance, defined as follows:

$$I = (start, dest, C)$$

- $start \in \mathcal{V}$ is the index of the start node in $\mathcal{G}$,
- $dest \in \mathcal{V}$ is the index of the destination node in $\mathcal{G}$,
- C is a set of constraints that need to be satisfied.

Solving I optimally means finding a path p, *i.e.* a sequence of nodes (or edges), which starts from $start$, ends in $dest$, satisfies all constraints in C and minimizes the total weight of the edges included in p. We can consider various types of constraints in C. In this work, we experiment with constraints related to mandatory nodes, which require for the solution path to include a given set of nodes $M \in \mathcal{V}$. In the next sections, we will refer to a path-planning problem instance $(start, dest, M)$ simply as an instance. A solution path is required to include every node in M at least once. Since the order of visit is not imposed for M, this problem can be assimilated to a TSP variant.

III. Path-building with Graph Convolutional Networks

In this section, we present our approach for training a neural network on a particular graph. We aim to leverage the learning capacity of a network to approximate the behavior of a model-based planner on the graph.

A. Neural Networks

Neural Networks (NNs) enable multiple levels of abstraction of data by using models with trainable parameters coupled with non-linear transformations of the input data. It should be possible to attempt NN training in order to approximate the results 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 $\mathbf{x}$:

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

where $h^{(l)}$ is the hidden state of the network and $f^{(l)}$ is the mapping function performed at layer l and parameterized by trainable parameters $\theta^{(l)}$ and bias $b^{(l)}$, and piece-wise activation function $\sigma(\cdot)$; $h^{(0)} = \mathbf{x}$.

Convolutional Neural Networks (CNNs) [14], [15] are a popular architecture for 2D data. They generalize MLPs by sliding groups of parameters across an input vector similarly to filters in image processing, leveraging fewer parameters and parallel computation. Hidden states in CNNs preserve the number of dimensions of the input, *i.e.* 2D when images are used as input, and are called *feature maps*.

B. Graph Convolutional Networks

Graph Convolutional Networks (GCNs) are generalizations of CNNs to non-Euclidean graphs [16]. GCNs 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*.

We consider here the approach of Kipf and Welling [8]. The GCNs have the following layer propagation rule:

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

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 is 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 $\tilde{A}$. In [8], the authors 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 $\tilde{D}^{-1} A$.

C. Problem Instance Encoding

The input of our model is a vector $\mathbf{x}$ containing information about the problem instance, including the graph representation. For every instance $I = (start, dest, M)$ of a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we associate a vector $\mathbf{x}$ made up of triplets features from each node in $\mathcal{G}$, making up for a total of $3 \times |\mathcal{V}|$ features. The three features for a node $j \in \mathcal{V}$ are:

- *start node* feature $st_j = 1$ if node j is start node in instance I, otherwise 0
- *end node* feature $ed_j = 1$ if node j is end node in instance I, otherwise 0
- *mandatory node* feature $my_j = 1$ if node j is a mandatory node in instance I, otherwise 0.

We obtain the input $\mathbf{x} \in \mathbb{R}^{|\mathcal{V}| \times 3}$ by stacking node features:

$$\mathbf{x} = [[st_1, ed_1, my_1], [st_2, ed_2, my_2], ..., [st_{|\mathcal{V}|}, ed_{|\mathcal{V}|}, my_{|\mathcal{V}|}]] \quad (3)$$

D. Neural network architecture and training

We define a neural network f that consists of a sequence of multiple graph convolutions followed by a fully connected layer. This GCN takes as input any instance I over the graph $\mathcal{G}$, and outputs a probability vector $\hat{\mathbf{y}} \in \mathbb{R}^{|\mathcal{V}|}$. $\arg\max(\hat{\mathbf{y}})$ corresponds to the next mandatory to visit from the $start$ node in an optimal path that solves I. The hidden states of the graph convolution layers $h^{(l)} \in \mathbb{R}^{|\mathcal{V}| \times N_{hidden}}$ consist of N_{hidden} higher-dimensional features for each node in the graph . After the final convolutional layer, the feature matrix is flattened into a vector by concatenating its rows and connected to a fully connected layer that maps it to a vector $z \in \mathbb{R}^{|\mathcal{V}|}$. We use the *softmax* function to convert it into probabilities.

NNs trained in a supervised manner use labeled training data, *i.e.* a set of input-output pairs $(\mathbf{x}_i, \mathbf{y}_i)$ sampled from a large training set. Here, $\mathbf{x}_i$ is an instance I and $\mathbf{y}_i$ is the index of the next mandatory node for $\mathbf{x}_i$ computed by any graph solver. We train the GCN on instances that have already been processed by the solver, which serves here as a *teacher*. The network learns to approximate the solutions given by the solver. To this end, we train the network using the *negative log-likelihood loss* typically used for multi-class classification and stochastic gradient descent (SGD).

E. Mandatory Node Ordering

For a given input instance, the GCN computes a mandatory node prediction at a time. In order to make it compatible with instances with varying amounts of mandatory nodes we make few adjustments. Given an instance with multiple mandatory nodes $(start, dest, M)$, we perform multiple recursive GCN calls to get the next mandatory node predictions q_i. More specifically, after a prediction q_i is computed, we generate a sub-instance where the start node becomes the current predicted mandatory node q_i and where the list of mandatory nodes contains the remaining nodes except q_i, *i.e.* $(q_i, dest, \tilde{M}_i = \tilde{M}_{i-1} \setminus q_i)$, where $\tilde{M}_0 = M$ and $q_0 = start$. We use the recursive calls during both training and testing. An interesting side-effect of this strategy is that it ensures an implicit balancing of the training samples by difficulty, as we generate sub-instances ranging from challenging (large $|M|$) to trivial (small $|M|$).

IV. Self-supervised learning

We present a self-supervised learning strategy aimed at training the GCN for a particular graph. First, we define a planning domain to solve instances. Second, we introduce a modified version of the A* algorithm for optimal data generation, and train the GCN on the data generated. In this section, we refer to an instance as a planning state $s = (start, dest, M)$. An end state is a termination instance, *i.e.* an instance for which the destination node has been reached and all mandatory nodes have been visited. We denote the termination instances as $F_i = (i, i, \emptyset), i \in \{1, 2, ..., |V|\}$. There are exactly as many termination instances as there are nodes in $\mathcal{G}$. We respectively define the *successors* and *predecessors* of a state $s = (start, dest, M)$ as $succ(s)$ and $pred(s)$ in Table I.

TABLE I
TRANSITION RULES TO SUCCESSORS AND PREDECESSORS.

$s = (start, dest, M)$	
Successor state s'	Predecessor state s'
$s' = (start', dest', M')$	$s' = (start', dest', M')$
$(start, start') \in \mathcal{E}$	$(start', start) \in \mathcal{E}$
$dest' = dest$	$dest' = dest$
$M' = M \backslash \{start'\}$	$M' = M$ $\triangleright$ [1]*
-	or
-	$M' = M \cup \{start\}$ $\triangleright$ [2]*
s is not an end state	s' is not an end state
Transition cost: $(start, start')$	Transition cost: $(start', start)$

[1]* only if $start' \notin M$
[2]* only if $start \neq dest$

Transition costs from a state to a neighboring state is the cost of the edge in the graph linking the start nodes of both states. With these rules, the destination node remains always the same. Therefore, we run the backwards version of A* from every termination instance F_i as initial state (using $pred$ as rule of succession). For each state s visited by A*, a path p is built from s to F_i which the algorithm considers the shortest. We define $g(s)$ as the cost of p, $a(s)$ as the next state visited after s in p, and $d(s)$ as the first mandatory node of s that is visited in p.

Furthermore we perform the following changes to A*. First, when a shorter path is found to a state s' while developing a state s, *i.e.* $g(s') > g(s) + c(s, s')$, the values of $a(s')$ and $d(s')$ are also updated along with $g(s')$ to take into account the shorter path. Secondly, we set the heuristic function h to 0. Since A* is run backwards from a termination state F_i, we are not aiming for the algorithm to reach a defined state in particular, but seek to reach as many states as possible. This ensures that when a state s is taken from the OPEN priority list of states left to develop, an optimal path from s to F_i had already been found. Choosing $d(s)$ as the next mandatory node to visit thus enables optimal solving. Consequently we can add the pair $\langle s, d(s) \rangle$ to the training set. We provide the pseudo-code in Algorithm 1.

The data generated by the algorithm is added to the training set and shuffled. The GCN is then trained on this set with supervised learning. Results show that training on the "synthetic" data generated with A* enables the GCN to generalize well on instances that A* did not process. We argue this is because the distribution obtained with A* is related to the path length of resolved instances. In fact, graph patterns already explored in short path solutions are incrementally included into longest ones.

V. Depth-First Branch and Bound Tree Search

It is possible to resolve an instance within a combinatorial search tree, rather than in the path-planning domain

Algorithm 1 Backwards A* for reverse instance solving

```
function ComputePaths()
    while elapsed time < timeout do
        remove state s from the front of OPEN;
        if length(mand(s)) > 0 then                          ▷ [3]*
            insert the pair < s, d(s) > to the training set
        for all s' ∈ pred(s) do                              ▷ [3]*
            if g(s') > g(s) + c(s, s') then
                g(s') = g(s) + c(s, s')
                a(s') = s
                if mand(s) = mand(s') then
                    d(s') = d(s)                             ▷ [4]*
                else
                    d(s') = startNode(s)                     ▷ [3,5]*
                insert s' into OPEN with value g(s')         ▷ [6]*
function main()
    for all i ∈ {1, 2, ..., |V|} do
        for all s ∈ S do
            g(s) = ∞
        g(E_i) = 0, a(E_i) = ∅, d(E_i) = ∅
        OPEN = ∅
        insert F_i into OPEN with value g(E_i)               ▷ [6]*
        ComputePaths();
```

[3]* $length(x)$: returns length of list x; $mand(x)$: returns mandatory nodes of instance x; $startNode(x)$: returns the start node of instance x; $pred(x)$: returns all possible predecessors of instance x

[4]* s' is created from s without adding its start node to the list of mandatory nodes

[5]* s' is created from s by adding its start node to the list of mandatory nodes

[6]* $h = 0$

defined in §IV. To this end, we compute the shortest source-destination paths for every pair of nodes $\langle i, j \rangle$ in $\mathcal{G}$ using Dijkstra's algorithm, as well as the associated path cost. Solving an instance $I = (start, dest, M)$ then becomes equivalent to finding the optimal order in which the mandatory nodes in M are visited for the first time. The solution path associated with an order $o = (m_1, m_2, m_3, ..., m_{q-1}, m_q)$ of the mandatory nodes can be built by concatenating the shortest path from $start$ to m_1, from m_1 to m_2, from m_2 to m_3, ... , from m_{q-1} to m_q, and from m_q to $dest$. Its total cost is the sum of the cost of each shortest path used to build it. Tree search algorithms can be used to look for an optimal order of the mandatory nodes by searching a particular tree. For an instance I, we define the root of this tree as the $start$ node, every leaf node as the $dest$ node, and every intermediate tree node as a mandatory node in M, such that a path from the root of the tree to a leaf defines an order in which to visit the mandatory nodes in M. The cost of of transitioning from a node v to a child node v' in the tree is the cost of the shortest path in $\mathcal{G}$ between the pair $\langle v, v' \rangle$. In the following, we refer to this tree as the mandatory search tree. Since we are using here only mandatory node constraints, the combinatorial optimization problem associated with this tree displays a similar structure with the TSP. However, it differs in the choice of the start and destination nodes which are fixed.

The branch and bound algorithm (B&B) is a popular tree search algorithm that is well known for its computational efficiency ([5]). In this work, we consider a depth-first B&B search algorithm. When developing a node inside the tree, the algorithm checks if each branch is expected to host a better solution than the best solution found so far. Should that not be the case for a given branch, the branch is cut, and the algorithm will not develop nodes further down the branch. This is done by using a lower bound and an upper bound. The lower bound is the sum of the total cost from the root node to the current node and a heuristic function h that approximates the remaining cost from the current node to the best achievable solution in branches below. This heuristic function should return a value as close as possible to this remaining cost (to cut as frequently as possible), while staying smaller (for the algorithm to remain optimal).

Next, we define the heuristic function h that we use. Let $v \in M$ be a mandatory node in the mandatory search tree, and R the set of remaining mandatory nodes left to the leaf node $dest$, *i.e.* the nodes in M that haven't been included between the root node $start$ and v. Let $D = \{v\} \cup R \cup \{dest\}$. We define two functions, min_1 and min_2, that respectively return for a node x in D the lowest shortest path cost in $\mathcal{G}$ from x to any node in $D \backslash \{x\}$, and the second lowest such cost. We build the heuristic function h by considering the remaining nodes left. For each node left, we consider the weight of all edges connecting it to other nodes in D and add the first and second smallest such weights, with the exception of v and $dest$ for which only the first smallest weight is added, and divide the total by 2:

$$h(v) = \frac{1}{2}(min_1(v) + min_1(dest) + \sum_{r \in R} min_1(r) + min_2(r))$$

Recent progress in learning-assisted tree search has shown that machine learning can be used to narrow the search space in very large domains to allow for efficient solving. Inspiring results have been shown in the game of Go ([17], [18]), where a very good solution, which is not required to be optimal, is found in record time. The search tree is pruned extensively by the neural network without optimality requirements. On the other hand, in a context where finding an optimal solution is critical, the neural network cannot be used for pruning. Instead, it could be used to determine a preferential order of processing for children nodes. However, using the neural network on every node of the search tree is unfeasible as it can actually make the search longer due to the larger computational overhead. Consequently, we propose to keep the GCN out of the tree search procedure. For a given instance, we use the GCN recursively in order to obtain a suggested order of visit of the mandatory nodes (III-E), from which we build the associated solution path by concatenating the shortest paths. This is done in negligible time compared to the tree search. The cost of the solution path found in such *probing* manner [19] is then used as an initial upper bound for the B&B algorithm. In section §VII we carry out experiments to evaluate the influence of the upper bound obtained with the GCN on search performance.

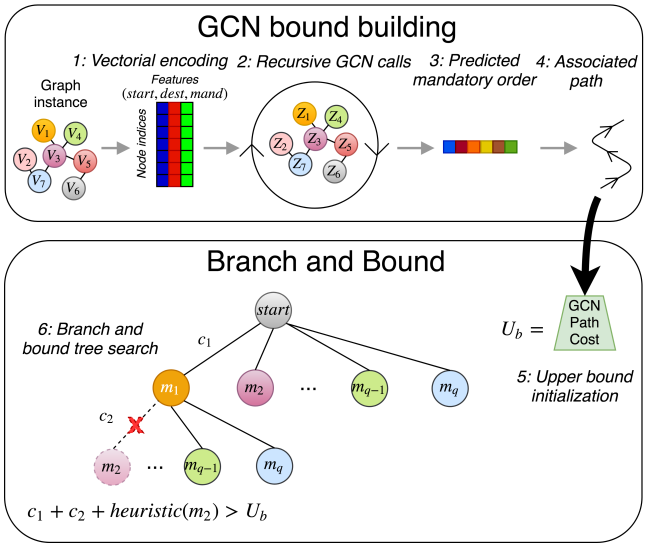

Fig. 2. GCN-assisted branch and bound algorithm pipeline. The GCN is used recursively to build an order of visit for the mandatory nodes. The order is converted into a path using the previously computed shortest path pairs, and the cost of the path is used as an initial upper bound for the algorithm. Here, the upper bound of the GCN allows for a level 2 early cut.

VI. Related Work

In the past few years there has been a growing interest for transferring the intuitions and practices from neural networks on structured inputs to graphs [20], [21], [22], [8]. [21] bridges spectral graph theory with multi-layer neural networks by learning smooth spectral multipliers of the graph Laplacian. Then [22] and [8] 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 [16] for a comprehensive review on learning on graphs.

Applications of these types of networks are starting to emerge. Recent works suggest GCNs are capable of making key decisions to solve either path-planning problems in graphs [23], or even STRIPS planning tasks [24]. Regarding path-related optimization problems, Li *et al.* [23] tackle the maximum independent set (MIS) problem with a solver that combines a GCN-guided tree search and local search optimization. The input to their solver is a graph, and the output is a highly optimized solution, which is not necessarily optimal. They report the limitation that their GCN can only be used for the task it was trained for. Kool *et al.* [25] propose a reinforcement learning framework to solve the travelling salesman problem (TSP) and variants of the vehicle routing problem (VRP) approximately. While they prefer an encoder-decoder architecture over GCNs, they achieve better results than previous learning-based approaches. Related works focus on approximate solving of a given task (*e.g.* the MIS problem) on different graphs, while this work aims to solve different tasks optimally in the same graph.

VII. Experiments

A. Benchmarks and baselines

We run experiments to evaluate the impact of the GCN's upper bound on the B&B method. Since the proof of optimality is necessary in our context, we focus only on small-scale problems for which optimal solving is possible in reasonable time. We consider four different graphs $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_3$ and $\mathcal{G}_4$, with respectively 15, 23, 22 and 23 nodes. These graphs represent realistic AUGV crisis scenarios in which aid has to be provided to key points in operational areas. More details on the graphs are available in [26], from which the scenarios have been built. We generate 1512, 2928, 2712 and 2712 random instances for each graph respectively. In order to remain close to some 'realistic' instances, we generate the instances as follows: using the shortest source-destination paths computed previously, we apply a decimation ratio (typically 80%) to keep only the 20% source-destination pairs that have the longest shortest paths. For each resulting pair $\langle start, dest \rangle$ kept, we generate multiple random instances with an increasing cardinality for the set of mandatory nodes, ranging from 5 to 12.

We consider 4 different baseline solvers on the benchmark instances generated to compare solving performance. First, we use a solver based on dynamic programming (DP) which searches, for a given instance, the mandatory search tree to determine the optimal order of visit of the mandatory nodes, from which the optimal path can be built. The B&B algorithm is also used to search the mandatory search tree, both with and without the upper bound provided by the GCN. Lastly, we solve instances using forward A* applied on the planning domain described in section §IV, with the minimum spanning tree (MST) as heuristic function. The MST heuristic is computed for an instance $(start, dest, M)$ by considering the complete graph $\mathcal{G}'$, which comprises only the $start$, $dest$ and mandatory nodes M. All pairs of nodes (v, v') in $\mathcal{G}'$ are connected by an edge which has a weight equal to the cost of the shortest path from v to v' in $\mathcal{G}$. The MST heuristic value is obtained by adding the following three values: the total weight of the MST of all mandatory nodes M in $\mathcal{G}'$, the minimum edge weight in $\mathcal{G}'$ from the $start$ node to any node in the MST, and the minimum edge weight in $\mathcal{G}'$ from any node in the MST to the $dest$ node.

B. Implementation details

We set the run-time of A* to 10 hours per graph for data generation. We use 3 graph convolutional layers of width 100. During training, we apply batch normalization [27] with decay of moving average $\varepsilon = 0.9$, dropout with a *dropout rate* of 0.1, and train the GCN with *Adam* [28]. We set the learning rate to $\eta = 10^{-4}$. We train models on a Tesla P100 GPU using over 1.5M training examples generated by A*, for $\sim$5 hours. We carry out the benchmark tests in this section on a laptop with an Intel i5 processor and 8GB of RAM. We highlight that our approach requires GPU only for the training of the GCN, which can be done offline. Problem instances can be then solved online on a CPU.

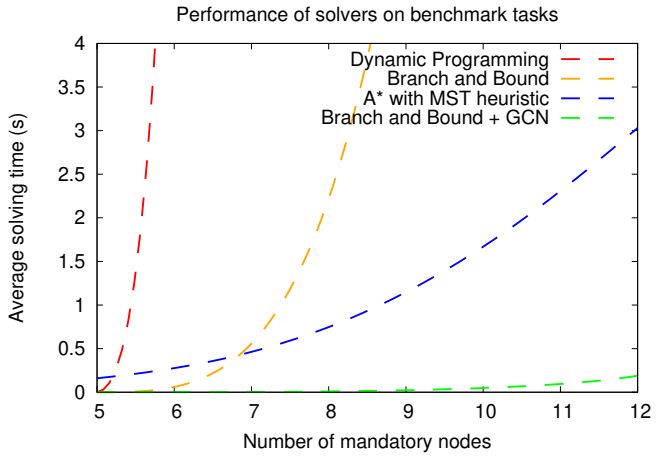

Fig. 3. Comparison of the performance of different solvers on benchmark instances generated for graph $\mathcal{G}_2$. The X axis represents the number of mandatory nodes of the instances, the Y axis the average solving time. We limit the Y axis to a range [0,4] to obtain a better comparison scale.

C. Results

We summarize results for $\mathcal{G}_2$ in Figure 3. We detail the experiments for graph $\mathcal{G}_3$ and $\mathcal{G}_4$ in Tables II and III. We include solving time only if it is measurable by CPU clock time. Figures for all graphs show a similar trend. We note that for instances with seven mandatory nodes and more, best-first algorithms such as A* applied on the planning domain defined in §IV become more suited than depth-first algorithms such as B&B applied on the combinatorial search tree associated with the mandatory constraints. Indeed, although each planning state requires more processing time in order to account for the specifics of the planning domain and the graph, overall far fewer planning states are visited than mandatory search tree nodes, giving A* a significant edge over depth-first DP and B&B. We note, however, that when the upper bound of the GCN is used, the B&B algorithm is able to outperform A* on all instances, even the most complex ones, and scales with the number of mandatory nodes more smoothly. Some information obtained within the mandatory search tree for instances with 11 mandatory

TABLE II

EXPERIMENTS FOR GRAPH $\mathcal{G}_3$. LEGEND: T/O = 5 MINUTES TIMEOUT.

Mandatory #:	5	7	9	11	12
DP					
Avg. node visits:	11.7K	876K	98.6M	T/O	T/O
Avg. time (s):	-	0.26	30.52	T/O	T/O
B&B					
Avg. node visits:	418	4,94K	146K	11,1M	120M
Avg. time (s):	-	-	0.07	5.39	71.1
A*, h=MST					
Avg. state visits:	26.4	58.5	141	342	503
Avg. time (s):	0.14	0.29	0.73	1.84	2.70
B&B + GCN					
Avg. node visits:	148	1,24K	10,8K	161K	642K
Avg. time (s):	-	-	0.01	0.08	0.34

TABLE III

EXPERIMENTS FOR GRAPH $\mathcal{G}_4$. LEGEND: T/O = 5 MINUTES TIMEOUT.

Mandatory #:	5	7	9	11	12
DP					
Avg. node visits:	11.7K	876K	98.6M	T/O	T/O
Avg. time (s):	-	0.24	30.53	T/O	T/O
B&B					
Avg. node visits:	545	7.58K	193K	11,8M	122M
Avg. time (s):	-	-	0.09	5.28	70.8
A*, h=MST					
Avg. state visits:	41.78	96.8	226	787	1,09K
Avg. time (s):	0.15	0.44	0.96	3.18	4.36
B&B + GCN					
Avg. node visits:	233	2.46K	29.1K	235K	859K
Avg. time (s):	-	-	0.02	0.11	0.56

nodes, displayed in table IV, provide more insight regarding these results. The average number of nodes processed in the subtree of each child node of the root node is given, as well as the average score of the best known solution after the subtree is processed.

TABLE IV

INFORMATION FROM THE MANDATORY SEARCH TREE.

Root node child #	-	1	2	3	...
B&B					
Avg. node visits	-	11M	50K	30K	...
Avg. best sol. score	∞	10.35K	9940	9781	...
B&B + GCN					
Avg. node visits	-	74K	11K	7K	...
Avg. best sol. score	9890	9300	9264	9249	...

Since our B&B algorithm is depth-first, processing the entire subtree under the first child node of the root node when no initial upper bound is known is highly computationally expensive. Indeed, no cut can be made until a leaf node is reached, and even then, the solution found is very likely to be costly compared to the optimal solution, thus the updated upper bound would still not allow for frequent cuts, until a good part of the subtree has been processed. On the other hand, if a good upper bound is known in advance, which is generally the case for the one given by the GCN in our experiments, the algorithm does not suffer from this issue, and early cuts can be made.

VIII. DISCUSSION AND FURTHER WORKS

We experiment with path-planning problems defined by three features: the start node, destination node, and mandatory nodes. By using the upper bound output by the GCN, we accelerate optimal depth-first solving of the search tree associated with the mandatory constraints. We show that the speedup achieved is significant and successfully competes with A*. This is the case even with a scenario setting where handling constraints within the planning domain is more appropriate than extracting and solving them separately. Also, our attempts to guide A* search with a GCN heuristic achieved worse results than the MST heuristic. The reason is

due to the best-first approach for which the GCN is unable to provide a suitable heuristic.

Moreover, the proposed framework can include additional types of constraints for path-planning problems. Each new constraint type results in additional features on nodes, and potentially also on edges [29]. In this case, the GCN predicts the next node to visit, and not the next mandatory node. Recursive GCN calls are made until a solution which satisfies all constraints is found. The approach can easily be combined with state-of-the-art constraint propagation techniques [30]. In the same manner, the solution cost is used as an initial upper bound for a depth-first search of a combinatorial tree associated with the constraints.

Learning-wise, the more constraint types exist for a given problem scenario, the wider the GCN learning domain will be. Further work will especially focus on this limitation to relate the exhaustiveness of the training phase with the variety of constraint types. Also, the proposed approach requires neural network offline training for a given graph (e.g. problem scenario). It can then be used online for path re-planning purposes, as the AUGV drives through the graph, with interesting computational performances.

IX. Conclusion

We introduced a method combining graph neural networks (GCN) and branch and bound (B&B) tree search to handle constraints in path-planning, successfully accelerating optimal solving. A relevant self-supervised strategy has been developed, based on A*, which provides appropriate data for GCN training. The heuristic information computed by the GCN enables better scaling of the B&B algorithm onto more complex problems. Results exhibit solving times that outperform A* with a handcrafted heuristic function based on a minimum spanning tree. Various AUGV applications can benefit from such an approach, especially when known terrains are given and path or itineraries must be computed on the fly. We also hope this line of work will serve to highlight the merits of learning with GCNs in optimal path-planning problems.

References

[1] 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*, vol. 4, no. 2, pp. 100–107, 1968.

[2] D. Ferguson, M. Likhachev, and A. T. 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.

[3] 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.

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

[5] P. M. Narendra and K. Fukunaga, "A branch and bound algorithm for feature subset selection," *IEEE Transactions on Computers*, vol. C-26, pp. 917–922, 1977.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[7] J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun, "Spectral networks and locally connected networks on graphs," in *International Conference on Learning Representations*, 2014.

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[9] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *International Conference on Learning Representations*, 2016.

[10] J. Monnot, V. T. Paschos, and S. Toulouse, "Approximation algorithms for the traveling salesman problem," *Mathematical methods of operations research*, vol. 56, no. 3, pp. 387–405, 2003.

[11] Y. Caseau and F. Laburthe, "Solving small tsps with constraints." in *ICLP*, vol. 97, 1997, p. 104.

[12] T. Volgenant and R. Jonker, "A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation," *European Journal of Operational Research*, vol. 9, pp. 83–89, 01 1982.

[13] G. Laporte, "The traveling salesman problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, vol. 59, no. 2, pp. 231–247, 1992.

[14] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[15] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[16] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[17] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[18] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[19] K. Osanlou, C. Guettier, A. Bursuc, T. Cazenave, and E. Jacopin, "Constrained shortest path search with graph convolutional neural networks," in *Workshop on Planning and Learning (PAL-18)*, 2018.

[20] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 2. IEEE, 2005, pp. 729–734.

[21] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," *arXiv preprint arXiv:1506.05163*, 2015.

[22] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, 2016, pp. 3844–3852.

[23] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Advances in Neural Information Processing Systems*, 2018, pp. 536–545.

[24] T. Ma, P. Ferber, S. Huo, J. Chen, and M. Katz, "Adaptive planner scheduling with graph neural networks," *CoRR*, vol. abs/1811.00210, 2018. [Online]. Available: http://arxiv.org/abs/1811.00210

[25] W. Kool and M. Welling, "Attention solves your tsp," *arXiv preprint arXiv:1803.08475*, 2018.

[26] C. Guettier, "Solving planning and scheduling problems in network based operations," in *Proceedings of Constraint Programming (CP)*, 2007.

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

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference for Learning Representations*, 2015.

[29] W. B. W. Vos, "End-to-end learning of latent edge weights for graph convolutional networks," Master's thesis, University of Amsterdam, 8 2017.

[30] C. Guettier and F. Lucas, "A constraint-based approach for planning unmanned aerial vehicle activities," *The Knowledge Engineering Review*, vol. 31, no. 5, pp. 486–497, 2016.