

GRASP: Accelerating Shortest Path Attacks via Graph Attention*

Zohair Shafi¹, Benjamin A. Miller¹, Ayan Chatterjee¹, Tina Eliassi-Rad¹, Rajmonda S. Caceres²
^{{shafi.z,miller.be,chatterjee.ay,teliassirad}@northeastern.edu,rajmonda.caceres@ll.mit.edu}

¹ Northeastern University, Boston, MA, USA

² MIT Lincoln Laboratory, Lexington, MA, USA

ABSTRACT

Recent advances in machine learning (ML) have shown promise in aiding and accelerating classical combinatorial optimization algorithms. ML-based speed ups that aim to learn in an end to end manner (i.e., directly output the solution) tend to trade off run time with solution quality. Therefore, solutions that are able to accelerate existing solvers while maintaining their performance guarantees, are of great interest. We consider an APX-hard problem, where an adversary aims to attack shortest paths in a graph by removing the minimum number of edges. We propose the **GRASP** algorithm: **G**raph **A**ttention **A**ccelerated **S**hortest **P**ath **A**ttack, an ML aided optimization algorithm that achieves run times up to 10x faster, while maintaining the quality of solution generated. GRASP uses a graph attention network to identify a smaller subgraph containing the combinatorial solution, thus effectively reducing the input problem size. Additionally, we demonstrate how careful representation of the input graph, including node features that correlate well with the optimization task, can highlight important structure in the optimization solution.

KEYWORDS

Attacking shortest paths in graphs, ML-based optimization on graph problems, Graph attention

1 INTRODUCTION

ML algorithms have had great success in learning a variety of inference tasks, and more recently, they have been shown to accelerate and improve classically difficult combinatorial optimization problems. Following Bengio et al's survey [1], we observe that there are essentially two categories for ML-based combinatorial optimization: one that uses ML in an end to end fashion, and another that uses ML to aid classical algorithms. Recent work by Veličković et al. [11],[12] falls in the first category where the authors make progress by aligning the ML model to the structure of an algorithm—for example, dynamic programming—and employing imitation learning to mimic each step of the algorithm. Gasse et al. [2] fall into the second category of using ML to learn offline, expensive to compute heuristics that are used in MIP solvers. Our contributions focus on the second category – i.e., ML aided approaches to classical algorithms – since the form of alignment discussed in end to end solutions are not

*Distribution Statement A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering. © 2023 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

straightforward for problems with more complex components. We consider one such algorithm by Miller et al., PATHATTACK [9], where an adversary wants to force traffic from a pair of nodes to take a particular route by removing a minimum number of edges in the graph. Minor approximation errors can accumulate along components of the algorithm, leading to sub-optimal solutions. Our goal is to speed up the PATHATTACK algorithm while ensuring that the quality of solutions does not decrease. To achieve this goal, we examine alternate ways of incorporating ML into combinatorial optimization. To that end, we propose the **GRASP** algorithm: **G**raph **A**ttention **A**ccelerated **S**hortest **P**ath **A**ttack, that achieves run times up to 10x faster, while maintaining the quality of solution generated by PATHATTACK. GRASP uses a graph attention network to identify a smaller subgraph containing the combinatorial solution, thus effectively reducing the input problem size. We demonstrate the speed up achieved by GRASP across a variety of synthetic and real-world graphs with varying topologies, graph characteristics and sizes. We also discuss the impact of node features on various topologies.

2 PROBLEM DEFINITION AND METHODS

PATHATTACK [9] provides an approximate solution to *the Force Path Cut problem*, where the objective is to find an edge set within a budget constraint whose removal makes a particular path the shortest. This problem is APX-hard. However, it can be recast as an instance of the Weighted Set Cover problem, which enables an approximation algorithm with a logarithmic approximation factor in the worst case. A brief overview of the PATHATTACK algorithm is shown in Figure 1. The algorithm has two main components: a constraint generation routine and a optimization component that utilizes a traditional linear solver. The size of the linear constraint matrix used by PATHATTACK is $(2|E| + |P| + 1) \times |E|$, where P is the set of constraint paths identified by the constraint generation procedure and E is number of edges. Thus, considering a subset of the graph—and thus a much smaller edge set—enables a substantial speedup of the algorithm.

Our proposed approach, GRASP, uses a learned model to help reduce the input problem size – i.e., given a large graph, GRASP finds a subgraph that contains all paths relevant to the task of attacking the shortest paths between a given source and target. Such a subgraph is highlighted in Figure 1(A) in green, where nodes in grey are of no impact to the problem, but add additional complexity. This subgraph is passed into the PATHATTACK algorithm to arrive at a solution (1(B)), thereby ensuring that the final output does not compromise on the quality of the solution, while achieving a speed up in run time. If the sampled graph does not yield a valid solution (i.e., p^* is not the shortest path), this is because a path that competes with p^* to be shortest was removed by the subgraph

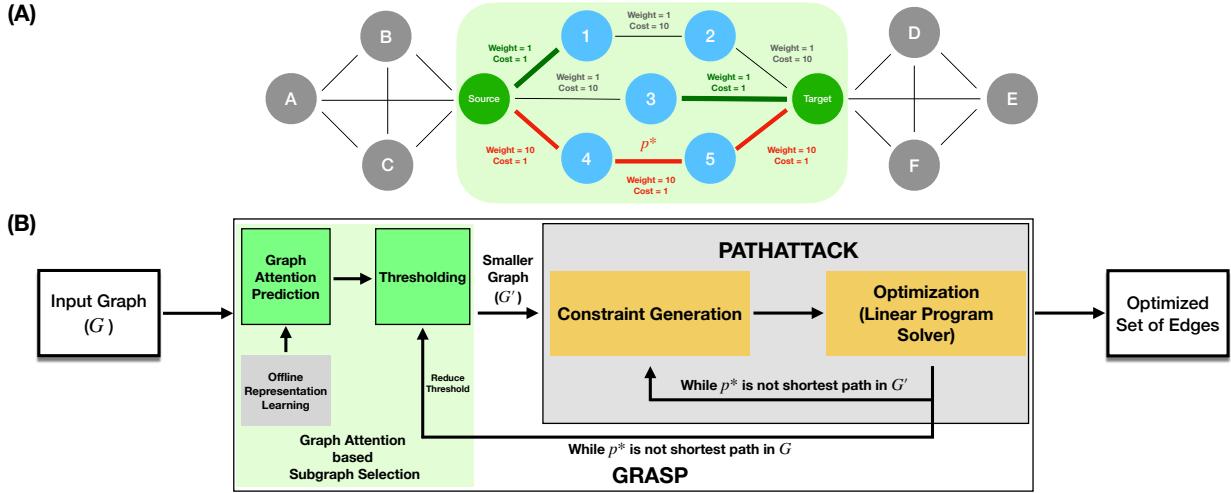


Figure 1: An overview of PATHATTACK. (A) The Force Path Cut problem: Given a source and target, we aim to make p^* (edges in red) the shortest path by deleting the minimum cost of edge set (edges in green). (B) Our proposed solution, GRASP, identifies the subgraph (highlighted in green) that contains all paths of interest, i.e., paths between source and target that are shorter than p^* and passes it to PATHATTACK. It can iteratively increase the subgraph size by reducing the threshold until p^* is the shortest path returned by PATHATTACK. Observe how nodes in grey do not contribute to the solution, but add to the problem size.

selection procedure. We then reduce the selection threshold and run PATHATTACK with a larger subgraph.

2.1 Graph Embeddings Highlight Optimization Solution Structure

The field of Graph Representation Learning has demonstrated the important role of learned features in improving many inference tasks on graphs. We explore if learned features can similarly reveal important structure about the PATHATTACK solution space. In particular we are interested to understand if edges that fall in the optimal solution are clustered or easy to separate in some learned feature space. This type of structure can then be utilized to reduce the dimensionality of the optimization space.

We first explore unsupervised representation learning of edges in the original graph using the Deep Graph Infomax (DGI) [14] algorithm. We embed a variety of synthetic graphs (~ 1000 nodes) into low-dimensional space (number of dimensions used in our experiments is 32) and generate edge embeddings by averaging its node embeddings. We also incorporate additional path information into the embeddings that we hypothesize might play a role in highlighting any structure relevant to the optimization task. More specifically, we use the following three sets of features:

- Structural Features - following Ghasemian et al. [3], these include local and global features such as Degree, Clustering Coefficient, Katz Centrality, Page Rank, Eigenvector Centrality, Structural Holes Constraint, Average Neighbor Clustering, number of Ego Net edges and Node Betweenness.
- Flow between Source and Target nodes, i.e., we compute the max flow between the source and target nodes using the weights on edges as the capacity. The flow value on each edge is then used as a feature.

- Personalized Page Rank (PPR) vectors for each node along the desired path (p^*). Observe that this gives us a node feature vector for each node, which is of the length of the number of edges of p^* . More concretely, if p^* has 5 edges, then every node in the graph gets a node feature vector of dimensions $\mathbb{R}^{5 \times 1}$. We pad these vectors up to a maximum length of 64 to account for varying lengths of p^* .

Figures 2 and 3 show the results of embedding two different graph topologies, a lattice and a Barabási-Albert graph, respectively, into 32 dimensions using DGI. Highlighted in red are edges deleted by PATHATTACK, in green are edges along the desired path p^* and in blue are all the remaining edges. Each of the 4 sub plots were generated using different sets of node features as discussed above. Figure 2,3 (A) are generated using no node features; 2, 3 (B) use structural features, 2, 3 (C) use flow values as node features and 2, 3 (D) use personalized page rank (PPR) vectors for each node along p^* as node features.

We observe that not all graph embeddings are equally effective in highlighting the structure of the PATHATTACK solution edges. In particular, embeddings that use the flow information (for the lattice graph) and the p^* related PPR scores (for the Barabási-Albert graph) lead to a separation of the solution edges from the rest of the graph. These observations lead to the main intuition of our ML-aided optimization algorithm. We can use carefully engineered graph representations to identify a much smaller subgraph that contains the solution edges, effectively reducing the dimensionality of the optimization space and allowing us to accelerate the solution identification run-time.

2.2 Problem Size Reduction via GRASP

To learn the subgraph of interest, we use supervised variants of graph neural networks. More specifically, we use Graph Attention

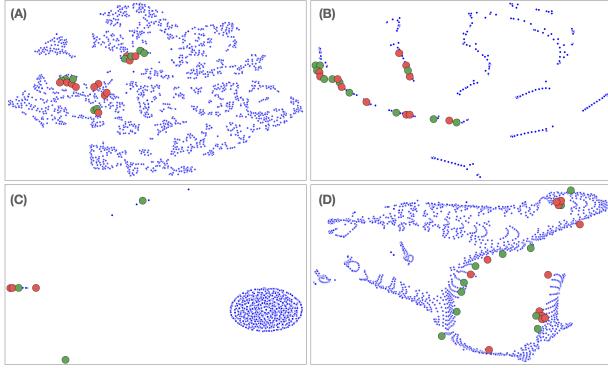


Figure 2: Visualizing edge embeddings for a 900-node lattice graph (30×30). Shown in red are edges that PATHATTACK deleted, shown in green are edges that are part of p^* , with blue being all other edges. The graph was embedded into 32 dimensions using DGI. Node features used were (A) no node features (B) structural features (C) flow values (D) Personalized Page Rank vectors. Observe that for the lattice graph, using flow values as node features (C) leads to the deleted edges (red) separating out from the rest of the edges.

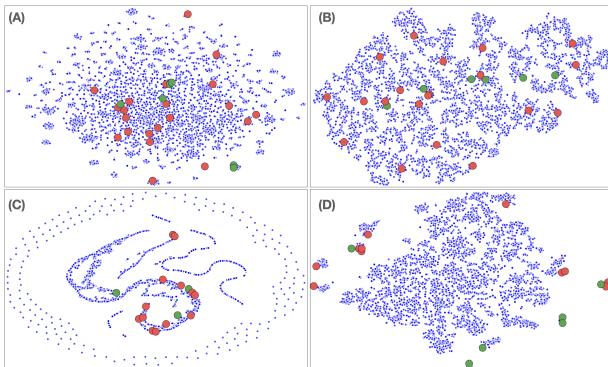


Figure 3: Visualizing edge embeddings for a 1000-node Barabási-Albert (BA) graph. Shown in red are edges that PATHATTACK deleted, shown in green are edges that are part of p^* , with blue being all other edges. The graph was embedded into 32 dimensions using DGI. Node features used were (A) no node features (B) structural features (C) flow values (D) Personalized Page Rank (PPR) vectors. For the BA graph, using PPR vectors as node features (D) leads to the deleted edges (red) separating out from the rest of the edges.

Networks (GAT) [13]. We pass in each individual set of features discussed above as node features, as well as the concatenation of all three sets of features, with the results of each discussed in Section 3. GRASP starts by picking nodes in the 95th percentile of the GAT predictions as the initial subgraph and passes the resultant graph into PATHATTACK. It checks if the resultant path is the same as p^* , if not, reduces the percentile threshold (thereby generating a larger subgraph) and repeats the process until the resultant path that PATHATTACK returns is p^* .

Table 1: Parameters ranges for the synthetic graphs. Each synthetic graph instance contained about $n = 1000$ nodes and about $|E| \approx n \log n \approx 7000$ edges. The parameters p stands for probability of edge creation for ER graphs, m stands for number of edges to attach from a new node to existing nodes for BA graphs and k is the number of nearest neighbors each node is joined to in a ring topology for WS graphs with p_r being the probability of rewiring each edge.

Graph	Parameter(s)	Range
Erdős-Rényi (ER)	p	0.01–0.017
Barabási-Albert (BA)	m	5–9
Watts-Strogatz (WS)	k, p_r	11–15, 0.02

2.2.1 Experimental Setup. We consider both synthetic and real graphs for training and testing the GRASP algorithm. The synthetic graph topologies include the Lattice, Barabási-Albert, Erdős-Rényi and Watts-Strogatz models. Each graph instance generated contained $n = 1000$ nodes and average edge density of $n \log n$. Edge densities were chosen to keep the graphs sparse, a domain which the PATHATTACK algorithm finds difficult to solve. We trained a different graph attention model for each type of graph topology, using 25 instantiations, where the source and target nodes are chosen randomly each time, with p^* being the 800th shortest path. Table 1 shows the distribution of all the varying graph parameters. Real world graphs were however, chosen to span a wide range of edge densities, sizes and characteristics to demonstrate the scalability of GRASP. Details of the real graphs chosen are shown in Table 2. The CPG graph has a grid like topology with AS and LBL graphs being computer networks with topologies akin to scale free graphs. The Wikispeedia graph is a dense graph with a large number of triangles. Weights and costs on all edges were set to 1 and each graph was trained for 1000 epochs, with an early stopping criteria of 250 steps.

For the testing setup, we generate 15 graphs of each topology type and report averaged results across these runs. For real graphs, we test on 15 randomly drawn triplets of source, target and p^* 's. We also compare against a greedy baseline algorithm that finds the shortest path between source and target and deletes the edge with the minimum cost. This process is repeated till p^* is the shortest path. We refer the interested reader to the paper by Miller et al. [9] for more details on PATHATTACK and the baseline algorithm (amongst baselines discussed in [9], this one prioritises shorter run times).

3 RESULTS

Figure 4 shows runtime performance across the different graph topologies. For this experiment, we considered PPR vectors as node features considering their success in learning using graph embeddings relative to the optimization task. Subplots (A), (B) and (C) show results for synthetic graphs with subplots (D), (E) and (F) showing results for real graphs. Across Figures 4(C) and (F), we see that run times for GRASP is shorter than PATHATTACK, with the largest reduction being 10x for Lattice graphs (C). We also show run times that include the time taken to compute the node features and

show that it is still shorter than running PATHATTACK. Figures 4(A) and (D) show the reduction in problem size across graphs. We see that reduction in problem sizes are largest for grid like graphs (Lattice and CPG), with reductions being smallest for ER graphs. Figures 4(B) and (E) show the number of edges cut, where we see GRASP cutting similar number of edges as PATHATTACK, sometimes outperforming PATHATTACK.

An interesting case is that of the CPG graph where the percentage of reduction is large, with the number of edges cut being fewer when using GRASP as compared to PATHATTACK, but the run time being larger. We posit the larger run times are due to the graph being small (~ 300 nodes), where it might make more sense to simply run the original PATHATTACK algorithm.

We compare the impact of the different sets of node features in Figure 5. Results for synthetic graphs are shown in Figure 5 (A) and (B). We observe that for this set of synthetic topologies, using PPR only (red) yields the highest reduction in the problem size, as well as lowest run times. However, examining Figures 5 (C) and (D), we see that this is not the case for real world graphs. Here, we observe that structural features (purple) lead to higher reduction in problem size in 3 out of the 4 graphs tested. Interestingly, the concatenation of all features (blue) does not out perform any one set. Since we run the resultant subgraph through the PATHATTACK algorithm, we always ensure solution quality. However, picking the right set of node features has a large impact in the reduction of the problem size and the resultant run times, and depends on the underlying topology of the graph.

The experiments on real world graphs demonstrate GRASP across graphs with varying sizes, graph characteristics and topologies. We also highlight the scalability by running GRASP trained on a BA ($m = 7$) graph with 1000 nodes and tested on BA graphs of varying sizes in Figure 6.

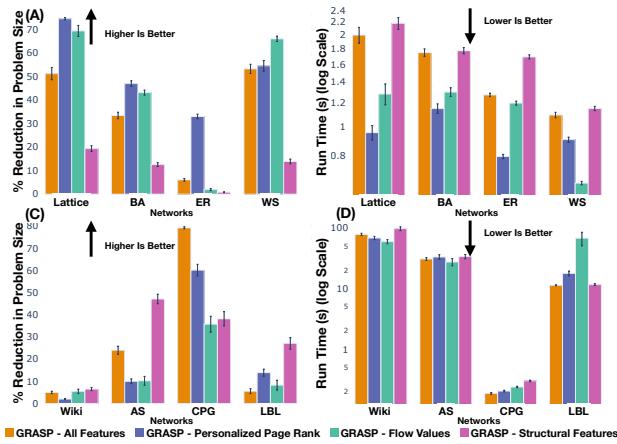


Figure 5: Figures (A) and (B) show results for synthetic graphs, with (C) and (D) showing results for real graphs from Table 2. For synthetic graphs, using PPR as node features seems to have the highest reduction in problem size and lowest run times. However, for real graphs, using structural features seems to be the best choice. Problem size reduction is dependent on picking the right set of features, which in turn depends on the topology of the graph.

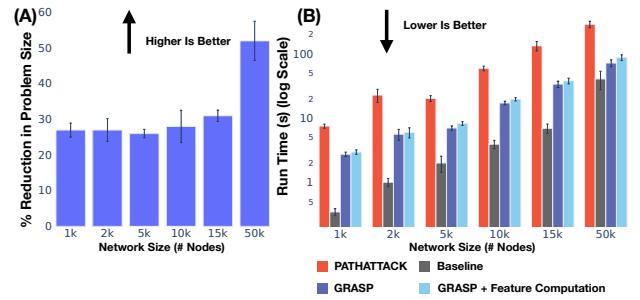


Figure 6: We demonstrate the scalability of GRASP trained on a BA graph with 1000 nodes and tested across BA graphs ($m = 7$) of varying sizes. Figure (A) shows reduction in problem size and (B) shows reduction in run times, where we see that GRASP scales well with input graph size.

4 RELATED WORK

ML has been used to approximate classical algorithms or combinatorial problems in various forms. The Neural Algorithmic Reasoning (NAR) framework proposed by Veličković et al. [12] exploits the fact that graph neural networks have similar structure to classical dynamic programming problems to learn a variety of commonly used classical algorithms [11]. Numeroso et al. [10] take the NAR framework a step further and propose Dual Algorithmic Reasoning that aims to jointly optimize for the primal and dual of a given problem. Specifically, they look at training models to find solutions for both the min cut and the max flow at the same time and show that it leads to better solutions.

In terms of NP-hard problems, Khalil et al. [4] propose a reinforcement learning approach that utilizes graph embeddings to learn a greedy policy that incrementally builds a solution. Closely related to our method is work by Li et al. [8], where a GCN is used as a heuristic function for a greedy search algorithm for the MIP solver. More specifically, the input graph is reduced to a smaller graph based on probability maps generated from the GCN that encode the likelihood of each node being in the optimal solution. Gasse et al. [2] use a GNN to learn expensive to compute branch-and-bound heuristics for MIP solvers. The learned models are then used at run time to achieve speed ups while maintaining the quality of the solution.

In this context, we can categorize the use of ML in combinatorial optimisation as belonging to either an end to end approach, or one where the learned model is used to aid an existing classical algorithm. We point the interested reader to the survey by Bengio et al. [1] that contains a deeper dive into the different categorisations and their details.

5 CONCLUSION AND FUTURE WORK

We consider the Force Path Cut problem, where an adversary's aim is to cut a minimum number of edges in order to force traffic from a pair of nodes through a specific path. This is an APX-hard problem. The PATHATTACK algorithm [9] approximately solves this problem by using a linear programming formulation of the Weighted Set Cover problem. We present GRASP, an ML-based optimization method that speeds up run time while maintaining

Table 2: Real-world graphs used and their properties. For each graph, we list the average degree ($\langle k \rangle$), standard deviation of the degree (σ_k), average clustering coefficient (κ), transitivity (τ), number of triangles (Δ), and number of components (φ).

Graphs	Nodes	Edges	$\langle k \rangle$	σ_k	κ	τ	Δ	φ
Chilean Power Grid (CPG) [5]	347	444	2.559	1.967	0.086	0.087	40	1
Lawrence Berkley Lab (LBL) [6]	3,186	15,553	9.763	40.702	0.048	0.001	1821	10
Wikispedia (Wiki) [15]	4,592	119,882	52.213	78.601	0.195	0.158	550,545	2
Autonomous System (AS) [7]	10,670	22,002	4.124	31.986	0.296	0.009	17,144	1

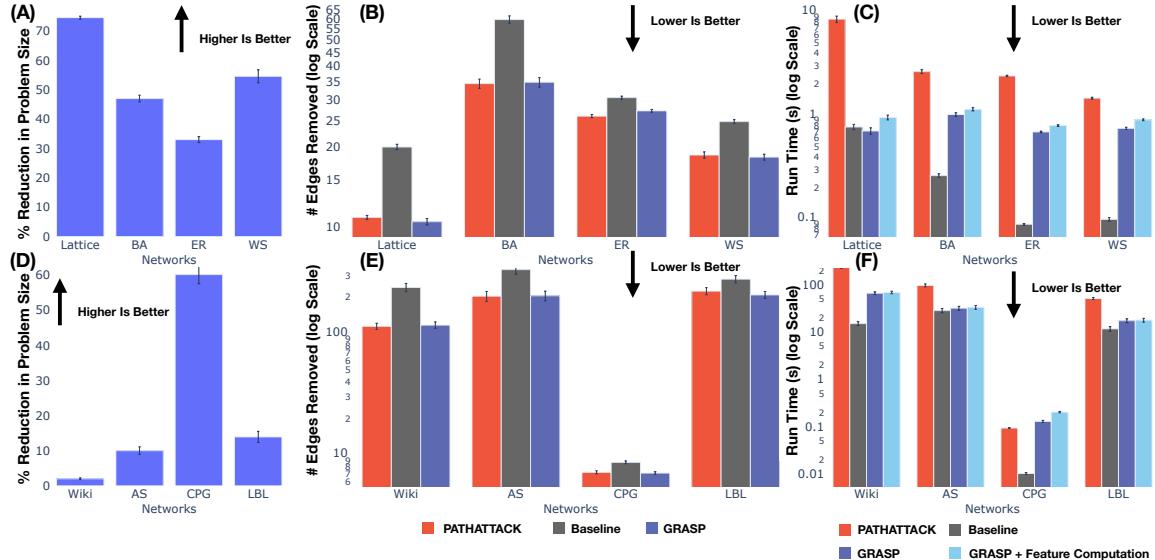


Figure 4: The first row of figures (i.e., A, B, and C) show results on synthetic graphs. The second row of figures (i.e., D, E, and F) show results on the real-world graphs listed in Table 2. Figures (A) and (D) show the percentage reduction in input problem size with GRASP compared to the original input graph used by PATHATTACK and baseline. Figures (B) and (E) show the number of edges removed. Observe that GRASP and PATHATTACK remove fewer edges than the baseline. Figures (C) and (F) show that run times for GRASP are shorter across graphs when compared to PATHATTACK. (The exception is CPG, which is too small to benefit from the edge selection procedure.) GRASP achieves the same performance as PATHATTACK and typically runs twice as fast, while the baseline runs faster with significant additional cost in the number of edges removed.

the quality of solutions generated. We highlight the importance of various node features and how they impact run time and solutions generated. We plan to extend this work to speed up general set cover problems by (1) reducing the input problem space and (2) running the classical algorithms on the reduced input space.

REFERENCES

- [1] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2021. Machine learning for combinatorial optimization: A methodological tour d’horizon. *European Journal of Operational Research* 290, 2 (2021), 405–421.
- [2] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. 2019. Exact combinatorial optimization with graph convolutional neural networks. In *NeurIPS*.
- [3] Amir Ghasemian, Homa Hosseini Mardi, Aram Galstyan, Edoardo M. Airoldi, and Aaron Clauset. 2020. Stacking models for nearly optimal link prediction in complex networks. *PNAS* 117, 38 (2020), 23393–23400.
- [4] Elias Khalil, Hanju Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *NeurIPS*.
- [5] Heetae Kim, David Olave-Rojas, Eduardo Alvarez-Miranda, and Seung-Woo Son. 2018. In-depth data on the network structure and hourly activity of the Central Chilean power grid. *Scientific Data* 5, 1 (2018), 180209.
- [6] Lawrence Berkeley National Laboratory. 2013. Lawrence Berkeley National Laboratory Network Data. <https://www icir org/enterprise-tracing/download.html>.
- [7] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*. 177–187.
- [8] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *NeurIPS*.
- [9] Benjamin A Miller, Zohair Shafi, Wheeler Ruml, Yevgeniy Vorobeychik, Tina Eliassi-Rad, and Scott Alfeld. 2022. Attacking Shortest Paths by Cutting Edges. *arXiv preprint arXiv:2211.11141* (2022).
- [10] Danilo Numeroso, Davide Bacciu, and Petar Veličković. 2023. Dual algorithmic reasoning. In *ICLR*.
- [11] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. 2022. The CLRS algorithmic reasoning benchmark. In *ICML*. 22084–22102.
- [12] Petar Veličković and Charles Blundell. 2021. Neural algorithmic reasoning. *Patterns* 2, 7 (2021), 100273.
- [13] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *ICLR*.
- [14] Petar Veličković, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. 2019. Deep graph infomax. In *ICLR*.
- [15] Robert West, Joelle Pineau, and Doina Precup. 2009. Wikispedia: An online game for inferring semantic distances between concepts. In *IJCAI*.