

# FastAFGCN: A Memory-Learn GNN-based Approximate Solver using ONNX Runtime

Lars Malmqvist  
The Tech Collective  
Denmark  
lama@thetechcollective.eu

**Abstract**—FastAFGCN is an approximate abstract argumentation solver designed for memory efficiency. It computes the credulous or skeptical acceptance of arguments using Graph Neural Network (GNN) models deployed via the ONNX runtime. The solver first computes the grounded extension using a memory-lean algorithm without dense matrices. If the argument’s status is not determined by the grounded extension, it leverages pre-trained and quantized GNN models for inference, applying task-specific thresholds. This approach builds upon previous work on GNN-based approximation but prioritizes runtime efficiency and reduced memory footprint by using ONNX and simplifying runtime feature computation.

**Index Terms**—abstract argumentation, GNN, ONNX, approximation, memory efficiency, ICCMA

## I. INTRODUCTION

Abstract Argumentation Frameworks (AFs) provide a powerful formalism for non-monotonic reasoning by modeling arguments and their attack relations as directed graphs [1]. Determining the acceptability of arguments under various semantics often involves computationally hard problems, many being NP-hard or beyond [1], [2]. This complexity motivates the development of approximate solvers, particularly for large-scale frameworks.

FastAFGCN is an approximate solver that leverages the predictive power of Graph Neural Networks (GNNs), building on previous work like AFGCN [3], [4] and recent findings on GNN-based approximation [5]. However, FastAFGCN introduces significant architectural changes focused on runtime efficiency and reduced memory consumption. It utilizes pre-trained GNN models that have been converted to the Open Neural Network Exchange (ONNX) format and run using the efficient ONNX Runtime.

Key features include:

- A memory-efficient implementation for calculating the grounded extension.
- Deployment of pre-trained, quantized GNN models via ONNX Runtime.
- Simplified runtime feature generation, reducing computational overhead and dependencies.
- Use of task-specific decision thresholds adjustable via a configuration file.

This approach aims to provide fast approximations while minimizing the memory resources required, making it suitable for environments with tighter constraints.

## II. APPROXIMATION APPROACH USING ONNX

Graph Neural Networks, particularly Graph Convolutional Networks (GCNs) [6], [7], have shown promise for approximating argumentation semantics [3], [5]. FastAFGCN adopts this GNN-based approach but modifies the runtime deployment strategy for efficiency.

Instead of loading a full deep learning framework (like PyTorch) at runtime, FastAFGCN uses pre-trained GNN models that have been exported to the ONNX format. These models are based on GCN architectures similar to previous work [3], [4], incorporating techniques like residual connections and dropout [8], trained using standard methods like Adam optimization [9] with Focal Binary Cross-Entropy loss. The key difference lies in the deployment. Using ONNX Runtime allows for optimized inference across different hardware platforms with a significantly smaller footprint compared to the original training framework. The models used are also quantized (indicated by ‘\_int8.onnx’ filenames), further reducing size and speeding up inference.

The runtime process is as follows:

- 1) The input AF is parsed, storing edges efficiently.
- 2) The grounded extension is computed using an algorithm optimized to avoid dense adjacency matrices, operating in space proportional to the number of nodes and edges.
- 3) If the target argument’s status is determined by the grounded extension (i.e., it is IN), the solver outputs “YES” immediately.
- 4) Otherwise, task-specific thresholds are loaded from a ‘thresholds.json’ file (defaulting to 0.5 if not specified).
- 5) If inference is needed, simple, randomly initialized node features are generated on-the-fly using PyTorch.
- 6) The appropriate pre-trained ONNX model for the specified task (e.g., ‘DS-PR\_int8.onnx’) is loaded via ONNX Runtime.
- 7) Inference is performed using the graph’s edge index and the generated random features as input. This yields logits for all arguments.
- 8) The pre-defined threshold is applied to the logit corresponding to the target argument to produce a “YES” or “NO” decision.

### A. Input Features

Unlike the previous AFGCN v2 solver, which utilized complex, pre-computed graph-based features (like centrality

measures, PageRank, coloring), FastAFGCN employs a simpler approach at runtime. The ONNX model is fed only the graph structure (edge index) and basic node features that are randomly initialized at runtime. This design choice significantly reduces the computational cost and library dependencies during the inference phase, contributing to the solver’s speed and memory efficiency, assuming the GNN model was trained to work effectively with such minimal features or implicitly learns relevant structural properties. The grounded extension information is used before invoking the GNN, acting as a fast path rather than an input feature to the network itself.

### III. IMPLEMENTATION

#### A. Design of the Solver

The solver consists of a Python script (`‘solver.py’`) and a Bash wrapper (`‘solver.sh’`). The Python script implements the core logic using `‘numpy’` for the efficient grounded computation and basic operations, `‘onnxruntime’` for loading and running the inference models, and `‘torch’` minimally for generating the random input features required by the ONNX model. The emphasis is on minimizing memory usage, notably by avoiding dense matrix representations where possible and delaying tensor creation until immediately before the ONNX call.

The underlying GNN models are trained offline, using frameworks like PyTorch and graph libraries, on datasets derived from sources like the ICCMA competitions [4], similar to previous versions. These trained models are then converted to quantized ONNX format (`‘_int8.onnx’`) for deployment.

The Bash wrapper (`‘wrapper.sh’`) provides a command-line interface conforming to ICCMA standards, parsing arguments and invoking the Python script (`‘solver.py’`) with the necessary parameters (`‘-filepath’`, `‘-task’`, `‘-argument’`).

The ONNX inference step inherently calculates acceptability probabilities for all arguments in parallel, leveraging the efficiency of the GNN and ONNX Runtime. However, the solver only outputs the result for the single argument specified in the query, as per standard decision task requirements. A safety net is included to output “NO” in case of timeouts or interruptions.

#### B. Competition Specific Information

FastAFGCN is submitted for the approximate track. It does not participate in exact or other tracks.

The solver implements functionality for the following semantics and problem types:

- Credulous Complete (DC-CO)
- Skeptical Preferred (DS-PR)
- Credulous Stable (DC-ST)
- Skeptical Stable (DS-ST)
- Credulous Semi-stable (DC-SST)
- Skeptical Semi-stable (DS-SST)
- Credulous Ideal (DC-ID)

Support for other tasks would depend on the availability of corresponding trained `‘.onnx’` model files.

The solver is called via the wrapper script:

```
./solver.sh -p <task> -f <file> -a
<argument_id>
```

Example: `./solver.sh -p DS-PR -f testaf1.af -a 4`

### REFERENCES

- [1] G. Charwat, W. Dvořák, S. A. Gaggl, J. P. Wallner, and S. Woltran, “Methods for solving reasoning problems in abstract argumentation - a survey,” *Artificial Intelligence*, vol. 220, pp. 28–63, 2015.
- [2] S. Woltran, “Abstract argumentation – all problems solved?” *ECAI 2014*, pp. 1–93, 2014.
- [3] L. Malmqvist, “Afgcn: An approximate abstract argumentation solver,” *ICCA 2021*, 2021. [Online]. Available: <http://argumentationcompetition.org/2021/downloads/afgcg.pdf>
- [4] —, “Approximate solutions to abstract argumentation problems using graph neural networks,” 2022. [Online]. Available: <https://theses.whiterose.ac.uk/32152/6/thesis.pdf>
- [5] L. Malmqvist, T. Yuan, and P. Nightingale, “Approximating problems in abstract argumentation with graph convolutional networks,” *Artificial Intelligence*, vol. 336, p. 104209, 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0004370224001450>
- [6] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 9 2019. [Online]. Available: <http://arxiv.org/abs/1609.02907>
- [7] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A comprehensive survey on graph neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2020.
- [8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: <http://jmlr.org/papers/v15/srivastava14a.html>
- [9] D. P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization,” *ICLR 2015*, pp. 1–15, 2015. [Online]. Available: <https://arxiv.org/pdf/1412.6980.pdf>