A LAZY LOVÁSZ ALGORITHM

A.1 CLASSIC LOVÁSZ ALGORITHM

Here we illustrate the steps of the classic Lovász method (Lovász, 1985) for the coherence of this research. The first two steps of the classic Lovász method are the same as the CLE method. For the third step, the algorithm invokes a depth-first search (DFS) to detect all the cycles in graph F. Assume DFS returns a list of cycles $\{C_i\}_{i=1}^{L}$ ($L \geq 1$). For each cycle C_i in graph F, the method would detect all the vertices that is connected to the cycle C_i, noted as H_i ($C_i \cap H_i = \emptyset$). We denote those vertices non-reachable from C_i as K_i, which is defined as:

$$K_i := \{v | v \in V, v \notin C_i \cup H_i\} \tag{14}$$

Let $d(v_k, x_c)$ be the shortest distance from $v_k \in K_i$ to $x_c \in C_i$, where the path could include vertices and edges not in F. Let β_i be the *shortest entering distance* from outside (i.e., K_i) into cycle C_i. We have:

$$\beta_i := \min\{d(x_k, x_c) | x_k \in K_i, x_c \in C_i\} \tag{15}$$

Let U_i be the set of vertices to be contracted:

$$U_i := \{v | \min_{x_c \in C_i} d(v, x_c) \leq \beta_i\}$$

We then shrink every U_i, which is a region of distance β_i centered at C_i, into one single vertex u_i. The self-connected edges to u_i will be deleted. We denote the contracted graph as G' and update the weight of edges related to u_i. For every incoming edge that is previously connected to the set of vertices U_i, the edge weight would be adjusted as:

$$A(v, u_i) = \min_{x_u \in U_i} A(v, x_u) + d_i(x_u, C_i) - \beta_i \tag{16}$$

All the rest edges stay the same. Afterward, the classic Lovász approach iteratively updates over graph G' until there are no cycles. Finally, the Lovász method performs the same expansion step as in the CLE method to find the exact edges that form MWA in the original graph G.

B IMPLEMENTATION AND RUNNING TIME ANALYSIS

B.1 UNDER $\mathcal{O}(n)$ GPU PROCESSORS

We dedicate this part to showing the difference between the existing Lovász-based algorithm. The main difference is that our approach trade-offs the time efficiency for hardware limitations that only $\mathcal{O}(n)$ processors are available. While Lovász (1985) requires $\mathcal{O}(n^3)$ processors for the best time efficiency.

Step 1: Edge Preprocess. The first step is to find the minimum incoming edge for each vertex. In Lavá method, every vertex launches a processor to check all the incoming edges, which takes $\mathcal{O}(n)$ time. Instead, Lovász (1985) uses the idea of parallel divide-and-conquer, that $\mathcal{O}(n)$ processors with $\mathcal{O}(\log n)$ time to find the minimum edge for a vertex.

If we limit the size of processors to $\mathcal{O}(n)$, then Lovász (1985) would take more time than ours. The classic Lovász method still can process every vertex using $\mathcal{O}(\log n)$ time but need to process vertices sequentially. The total time under this hardware constrained setting for Lovász (1985) would be $\mathcal{O}(n \log n)$.

Subtract Min Incoming Edges. We subtract the value of the minimum incoming edge from all the incoming edges of every vertex. In our method, the j^{th} processor sequentially subtracts a_j (in equation 7) for all the incoming edges of the j^{th} vertex. So our method takes $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ processors. Instead, Lovász (1985) launches a processor for every edge for subtraction, which takes $\mathcal{O}(1)$ time with $\mathcal{O}(n^2)$ processors.

If we limit the size of processors to $\mathcal{O}(n)$, Lovász (1985) takes the same time complexity as ours. Specifically, Lovász (1985) processes all the incoming edges for one single vertex in parallel and then sequentially goes through all the vertices. Thus, the overall time is $\mathcal{O}(n)$.

Termination Criterion. This step is to find if there is a directed cycle. In our method, j^{th} processor maintain a distance dictionary they can reach j^{th} vertex and overall it would takes $\mathcal{O}(\log n)$

iterations. At t^{th} iteration, the j^{th} processor checks for the parent vertex that are 2^t step away from the current vertex. See figure 5 as an example. Note that in this specific problem, except the root, every vertex has exactly one parent vertex with one single incoming edge. In total, our method takes $\mathcal{O}(\log^2 n)$ time with $\mathcal{O}(n)$ processors.

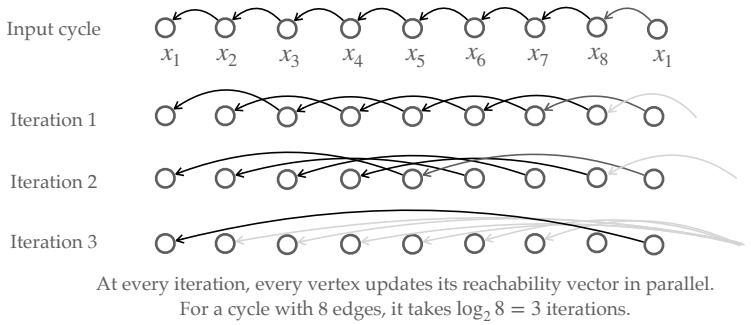

Figure 5: Illustration of checking cycle.

In Lovász (1985), fix j^{th} vertex, $\mathcal{O}(n^2)$ processors are launched in parallel for every edges to check if vertex x_j can reach any new vertex through the edge and it would takes $\mathcal{O}(\log n)$ iterations. In summary, Lovász (1985) takes $\mathcal{O}(\log n)$ time with $\mathcal{O}(n^3)$ processors.

If we limit the size of processors to $\mathcal{O}(n)$, then Lovász (1985) cannot be more efficient than our method for this step. The classic method would process edges sequentially for every vertex, taking $\mathcal{O}(n^2 \log n)$ time with $\mathcal{O}(n)$ processors. In another approach, Lovász (1985) would process only adjacent edges for every vertex, which uses the same time and processors as ours.

Note that in the sequential setting with 1 processors, we can use the DFS method to recursively find the parent of the current vertex. Since there are only $n-1$ edges, DFS only needs $\mathcal{O}(n)$ time.

C Detecting WCCs

C.1 Depth First Search for Detecting WCCs

Classic algorithms for detecting the connected components are mainly based on DFS. For example, Tarjan's algorithm is applied to find strongly connected components (i.e, the cycle C on graph F), and thereafter a separate DFS is applied with any vertex on the cycle to determine H. Note that Tarjan's algorithm internally is based on DFS. For the dense graph, it takes take $\mathcal{O}(n^2)$ time. The proposed approach also takes $\mathcal{O}(n^2)$ time. But the actual execution of our approach is on GPU, which is highly optimized for all the matrix computations and avoids transferring the matrix moving from CPU to GPU or vice versa. Thus our approach can have a more efficient empirical running time than the DFS-based algorithms in practice.

C.2 Binary Matrix Multiplication for Detecting WCCs

After "step 1: edge pre-process", the obtained graph F contains $n-1$ edges. In the corresponding adjacency matrix $F \in \{0,1\}^{n \times n}$, $F_{ij} = 1$ implies the edge $x_i \to x_j$ exists. The matrix power in Eq. 8 is defined as: is there a two-step path from x_i to x_j using edges only in F?

$$(F \times F)_{ij} = \begin{cases} 1 & \text{there exists } x_k \in F, \text{s.t. } x_i \to x_k \to x_j \\ 0 & \text{otherwise} \end{cases}$$

which has the same meaning using the element-wise logic operators: $(F \times F)_{ij} = \bigvee_{1 \le k \le n} (F_{ik} \wedge F_{kj})$. An intuitive idea is to define:

$$S = \bigvee_{k=1}^{n-1} F^k = F \vee F^2 \vee \ldots \vee F^{n-1} \tag{17}$$

$S_{ij} = 1$ means there is a path from vertex x_i to vertex x_j in *less than* $n-1$ steps. Since there are $n-1$ edges in graph F, the last element in the equation of S is F^{n-1}. Directly computing S using the logical-AND operator and logical-OR operator requires $\mathcal{O}(n)$ to compute every F^k for $1 \leq k \leq n$, which is not time efficient. Therefore we propose Eq. 9 by observing the following equivalence: $(F \vee F^2)_{ij} = 1$ if x_i can reach x_j with 1 or 2 steps.

Let $\mathbf{I}$ be the identity matrix. Following the definition in Eq. 8, the element $((\mathbf{I} \vee F) \vee F)_{ij}$ is computed as:

$$\begin{aligned}
((\mathbf{I} \vee F) \times F)_{ij} &= \bigvee_{1 \leq k \leq n} ((\mathbf{I} \vee F)_{ik} \wedge F_{kj}) \\
&= \bigvee_{1 \leq k \leq n} ((\mathbf{I}_{ik} \wedge F_{kj}) \vee (\mathbf{I}_{ik} \wedge F_{kj})) \\
&= \left(\bigvee_{1 \leq k \leq n} (\mathbf{I}_{ik} \wedge F_{kj}) \right) \vee \left(\bigvee_{1 \leq k \leq n} (F_{ik} \wedge F_{kj}) \right) \\
&= \left(\bigvee_{1 \leq k \leq n} (F_{ij}) \right) \vee \left(\bigvee_{1 \leq k \leq n} (F_{ik} \wedge F_{kj}) \right) \\
&= F_{ij} \vee F^2_{ij}
\end{aligned} \tag{18}$$

Thus $(\mathbf{I} \vee F) \vee F$ computes the same thing as $F \vee F^2$. Another benefit is that we can combine the whole series. For example, let $n = 4$,

$$\begin{aligned}
\bigvee_{k=1}^{4} F^k = F \vee F^2 \vee F^3 \vee F^4 &= \left(F \vee F^2\right) \vee \left(F^2 \vee F^3\right) \vee \left(F^3 \vee F^4\right) \\
&= ((\mathbf{I} \vee F) \vee F)) \vee \left((\mathbf{I} \vee F) \vee F^2)\right) \vee \left((\mathbf{I} \vee F) \vee F^3)\right) \\
&= (\mathbf{I} \vee F) \bigvee_{k=1}^{3} F^k \\
&= (\mathbf{I} \vee F)^2 \bigvee_{k=1}^{2} F^k \\
&= (\mathbf{I} \vee F)^3 F
\end{aligned}$$

The mentioned divide-and-conquer trick works as follow: We can first compute $S_1 = (\mathbf{I} \vee F)$ at round 1 and then compute $(\mathbf{I} \vee F)^2$ by doing $S_2 = S_1^2$. In the next step, we compute $(\mathbf{I} \vee F)^4$ by doing $S_3 = S_2^2$. In fact, k-th round computes $S_k = S_{k-1}^2 = (\mathbf{I} \vee F)^{2^k}$. Therefore, we state that $\mathcal{O}(\log n)$ round is needed to compute the whole series in Eq. 9 and is empirically more efficient to compute, compared to the computation involved in the definition in Eq. 17.

C.3 Matrix Inverse for Detecting WCCs

Numerically, the logic-AND operator computes the same thing as the add operator. Another reason we need to transfer to add operator is that we can apply matrix inverse property on top of it. Let $S = \sum_{k=1}^{n-1} F^k$, where $S_{ij} > 0$ (previously we use $S_{ij} = 1$) implies vertex x_i can reach vertex x_j within $n-1$ steps.

Directed computing S using the above divide-and-conquer trick would be slower because arithmetic operations are slower than logic operations. For example, arithmetic multiplication is slower than bit-wise level logical-OR operations. Instead, we further extend at most $n-1$ steps to infinite steps, where vertex x_i can take infinite steps to reach vertex x_j in graph F. Let $S = \sum_{k=1}^{\infty} F^k$, we have

$$\begin{aligned}
S + \mathbf{I} &= \mathbf{I} + F + F^2 + F^3 + \dots \\
(S + \mathbf{I})F &= \quad F + F^2 + F^3 + \dots
\end{aligned}$$

Thus we obtain $S = (\mathbf{I} - F)^{-1} - \mathbf{I}$. However, the power series on the RHS almost certainly does not converge numerically. So that the matrix inverse does not exist.

We introduce a hyper-parameter $\alpha > 0$, to solve the numerical convergence problem. We re-define $S = \sum_{k=1}^{\infty} \alpha^k F^k$. Following the same analysis, we obtain

$$S = (\mathbf{I} - \alpha F)^{-1} - \mathbf{I} \tag{19}$$

If we choose a small α , the infinite series on RHS would converge. In experiment, we set $\alpha = 0.9$. Solving the matrix inverse is generally not recommended, since numerically the solver tends to incur large errors for the large matrix. We show the numeric stability in Fig. 3(c) and empirical running time in Fig. 3(d).

To conclude, the matrix-inverse approach is another approach for computing strongly connected components (C) and weakly connected components ($C \cup H$). See the following example for a detailed overview of the result.

Example 2. Using Fig. 2(a) left figure as an example. The corresponding adjacency matrix F and matrix $S = (\mathbf{I} - \alpha F)^{-1} - \mathbf{I}$ are:

$$F = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ r \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & r \end{array} \\ \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \quad S = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ r \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & r \end{array} \\ \begin{bmatrix} \mathbf{1.01} & \mathbf{1.01} & \mathbf{1.01} & \mathbf{1.01} & 0 \\ \mathbf{1.10} & \mathbf{1.10} & \mathbf{1.10} & \mathbf{1.01} & 0 \\ 0 & 0 & 0 & \mathbf{1.01} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array},$$

Since $S_{11} > 0$ and $S_{22} > 0$, vertices x_1 and x_2 are on the cycle. In fact, the two vertices are on the same cycle by the following analysis. By equation 10, 1) the set of vertices that x_1 can reach from and to can be determined by

$$S \wedge S^\top = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ r \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & r \end{array} \\ \begin{bmatrix} \mathbf{1.01} & \mathbf{1.01} & 0 & 0 & 0 \\ \mathbf{1.10} & \mathbf{1.10} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

so we obtain $C = \{x_1, x_2\}$. 2) the set of vertices that can be reached from the cycle but not on the cycle are $H = \{x_3, x_4\}$. 3) the set of vertices that are non-reachable from the cycle $K = \{r\}$.

C.4 Other Approaches for Detecting Cycles

There exist more advanced algorithms for detecting WCCs on GPU, that can have faster theoretical running time but are non-trivial to be implemented (Barnat et al., 2011; Azad & Buluç, 2019; Zhang et al., 2020). There are line of work in *parallel* DFS (Träff, 2013; Naumov et al., 2017) that requires to pre-process the general directed graph into directed acyclic graph.

D Proof of Theorem 1

Here we present detailed proof of the upper bound on the number of contractions that our Lavá method takes.

Proof. 1) According to the steps of Lavá, there exists at least one super vertex in the cycle. Otherwise, this cycle would be contracted in the previous rounds, which contradicts the assumption.

2) We then prove by contradiction that there are at least exists two contracted vertices. Suppose we have only one super vertex u_i in the contracted graph. In the original graph, this cycle corresponds to a path starting at U_i and ending at U_i. No vertex on this path is included in U_i. Based on the edge weight update rule in Eq. 13, we have

$$0 = A'(x_j, x_k) = A(x_j, x_k) + d_i(x_k) - \beta_i \geq d_i(x_j) - \beta_i.$$

where $d_i(x_j)$ and $d_i(x_k)$ is the shortest entering distance from x_j and x_k to C_i correspondingly. In the worst case, x can pick $x_j \rightsquigarrow x_k \rightsquigarrow C_i$ as the shortest path. In any other case, x_j can find a shorter one. So the RHS is always smaller than or equal to the LFS. This implies that $x_j \in U_i$ by the definition of U_i (in Eq. 12), which forms a contradiction. This proves that there exists *at least two* super vertices in the cycle.

3) By induction on the rounds, the t-th round at least contracts 2^t vertices in the original graph G. Therefore, Lavá takes at most $\mathcal{O}(\log n)$ rounds. □

E Closed Form for Computing $\log Z_\theta(x)$

The Laplacian matrix L of graph G is defined as:

$$L_{ij}(x) = \begin{cases} -\exp(\phi_\theta(x_i, x_j)) & \text{if } x_i \neq x_j \\ \sum_{x_k \neq x_i} \exp(\phi_\theta(x_i, x_k)) & \text{otherwise} \end{cases} \tag{20}$$

Define $\det(L)$ be the determinant of a matrix L. Let $\det_+(L)$ to denote the determinant of matrix L with the last row and column removed. By the matrix tree theorem (Durfee et al., 2017), $\log Z_\theta(x)$ can be computed in a closed form as: $Z_\theta(x) = \det_+(L)$. So we can easily obtain:

$$\log Z_\theta(x) = \log \det_+(L((x))) \tag{21}$$

We can use this exact solution to evaluate the wellness of the perturbation-based solution in Eq. 5.

F Experiment Settings

F.1 Implementations

For programming convenience, all the baselines and our implementations are based on Python3.9. We acknowledge that implementation in low-level programming language would use less time. Here we want to evaluate the time growth pattern that every algorithm would take, which is orthogonal to the choice implementation language. Our code sample can be found at the following link:

G Additional Experiments

Please find our (anonymous) online code repository at:

`https://anonymous.4open.science/r/lazy_lovasz-0475/`

The CLE method used in the experimental analysis is from the standard implementation in AllenNLP package[1].

The set of baselines in Fig. 3(d):

- Tarjan-DFS (cpu) from the scipy sparse graph library[2]. The implementation use an iterative version of Tarjan's algorithm to find the strongly connected components of a directed graph. The algorithmic complexity is for a graph with n edges and $n-1$ vertices is $\mathcal{O}(n)$. The GPU version is still under heavy development and is not available for time benchmark[3].
- The "matrix inverse (cpu)" is computed from Numpy linear algebra API[4]. The "matrix inverse (gpu)" is computed with Cupy linear algebra API[5].

[1] https://github.com/allenai/allennlp/blob/main/allennlp/nn/chu_liu_edmonds.py

[2] docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.connected_components.html

[3] docs.cupy.dev/en/stable/reference/generated/cupyx.scipy.sparse.csgraph.connected_components.html

[4] numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

[5] docs.cupy.dev/en/stable/reference/generated/cupy.linalg.solve.html

- The "BMM (cpu)" is implemented with Numpy's logical operations[6]. and the "BMM (cpu)". The "BMM (gpu)" is implemented with Cupy's logical operations[7], which is because PyTorch does not implement low precision matrix multiplication.

G.1 GRAPH-BASED DEPENDENCY PARSING

Dependency parsing considers the syntactic structure of a sentence, which describes the grammatical relations among words Jurafsky & Martin (2009). The notion of non-projectivity considers languages with flexible word order, like German, Dutch, and Czech McDonald et al. (2005). Words in sentences are formulated as vertices a in graph, which avoids the short and long-range accuracy difference when representing sentences as sequences McDonald & Nivre (2011).

In what follows, $x = [\texttt{root}, x_1, \ldots, x_n]$ represents an input sentence with `root` being the dummy root. Let T denote the ground-truth dependency tree for sentence x. The annotation tree T is composed of n edges, where every $(v_{x_i}, v_{x_j}) \in T$ is a directed dependency relation from the head word x_i to modifier word x_j. Figure 6 shows the non-projective dependency analysis as a tree alongside its universal part-of-speech (UPOS) tags for the example "A hearing is scheduled on the issue today". The learning task is to learn a neural model given annotated training data $\{x_i, T_i\}_{i=1}^N$. In testing, we need to predict a dependency tree with the optimal score for a given input x'.

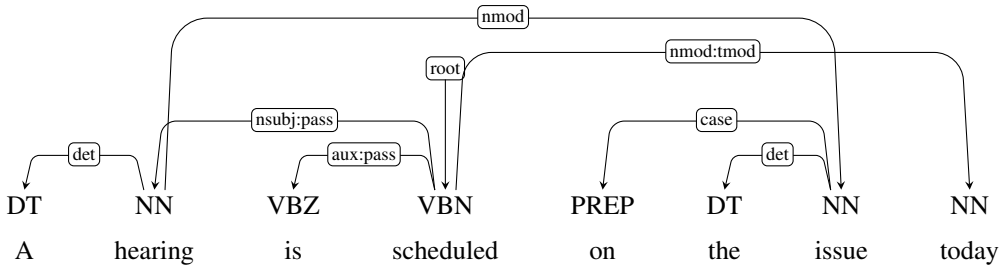

Figure 6: A non-projective dependency tree with fixed `root` for sentence along with part-of-speech tags. Directed arcs represent the head-modifier relation between words, and the labels on the arcs denote the dependency type. The task is to assign every sentence a labeled grammar tree for understanding the structure and composition of sentences.

Datasets & Evaluation Metrics. For the choice of data, we select 4 popular languages in the Universal Dependency Dataset (de Marneffe et al., 2021)[8], the statistics of which are shown in Table 2. For the evaluation of the models, we report Labeled Attachment Score (LAS) for labeled cases and Unlabeled Attachment Score (UAS) for the unlabeled case.

Table 2: Statistics dependency parsing datasets of multiple languages. The table includes the number of sentences in the training, development, and testing datasets.

	Du-ALP	En-GWT	Fr-GSD	Cr-SET
Train	12269	14187	2914	14450
Dev	718	1400	707	1476
Test	596	426	769	416

Model Settings. The deep neural network used in this application follows the NeruoMST model Ma & Hovy (2017). Our implementation and analysis are conducted on top of their code repository [9]. Our neural architecture is composed of 3 layers Bi-LSTM layers with dimension 512, which serve to transform the input sentence into a sequence of vectors. We use Fasttext pretrained embeddings[10]

[6] numpy.org/doc/stable/reference/routines.logic.html
[7] docs.cupy.dev/en/stable/reference/logic.html
[8] universaldependencies.org/
[9] https://github.com/XuezheMax/NeuroNLP2
[10] https://fasttext.cc/docs/en/crawl-vectors.html

for the initialization of the embedding layer. Then the biaffine attention mechanism is used for computing the edge weight of the score Dozat & Manning (2017).

For the $\log Z_\theta$ approximation task, we ask the model to predict $M = 1000$ MWA with the perturbed adjacency matrix and then computes the empirical average based on Eq. 5:

$$\overline{\log Z_\theta}(x) = \frac{1}{M}\sum_{i=1}^{M}\max_{T'\in\mathcal{T}(x)}\sum_{(x_i,x_j)\in T}\phi_\theta(x_i,x_j)+g'_{ij}, \qquad g'_{ij}\sim \text{Gumbel}(0,1) \tag{22}$$

if u is drawn form the uniform distribution between $(0, 1)$, then $-\log(-\log(u))$ follows the Gumbel distribution. Here we draw i.i.d. gumbel variable g'_{ij} from its distribution we obtain a adjacency matrix A'_θ based on the new edge weight $\phi_\theta(x_i, x_j) + g'_{ij}$ for all $x_i, x_j \in x$. We then compute the noisy MWA. After M iterations, we compute the averaged score of the noisy MWA and thus obtain the approximated value $\overline{\log Z_\theta}(x)$. To quatify how well is the noisy estimation, we compute the distance between the exact solution by Eq. 21 and our estimation:

$$\text{apx. err.} = \frac{1}{N}\sum_{k=1}^{N}|\log\det{}_+(L((x^k))) - \overline{\log Z_\theta}(x^k)| \tag{23}$$

This metric can help us to reveal the learning process of arborescence using the Gumbel-max trick.

G.2 Synthetic Dense Graph

Synthesised Dataset Generation. We present the statistics of the round of contractions taken by the CLE method for graphs of different sizes. For this synthetic experiment, we fix the number of vertices $|V|$ from 2^5 to 2^{11} and assign edge weights using uniform distribution between $\in [-1, 1]$ for every pair of edges. We generate 100 dense graphs for a fixed $|V|$ with random weights and evaluate the number of contractions the CLE approach takes.