Parallelization of Truncated Search Tree (TruST) Algorithm on GPU

Xuan Zhang, Leo Franco Soto, Dushyant Singh Udawat

IE533, UIUC

Abstract –

Subgraph Matching is a field of graph theory where we find the best matching of a query graph or a template graph into a data graph. The Truncated Search Tree is an inexact graph matching algorithm that heuristically searches for the best matching in a graph. The TruST Algorithm is an inexact, heuristic search algorithm, that provides parameters that can be tweaked to run the program faster or slower and with more or less accuracy. The algorithm has not been parallelized formally on the Graphics Processing Unit (GPU) and this paper attempts to do that. We are using cupy and numba.cuda libraries on python to parallelize the code.

1. Introduction –

A. Graph Matching –

Graph matching is a subfield of graph theory that aims at finding the best matching subgraph of a template graph in a data graph. There are various types of graph matching, two of the easiest ways to classify them are -

i. Exact Matching

The structure of this subgraph in the data graph and that of the template graph are isomorphic to each other. In other words, they have the same topology of nodes and edges in between them.[1]

ii. Inexact matching

While exact graph matchings are desirable, it is not always possible an exact match. Therefore, an alternative is choosing a heuristic method that finds inexact graph matching which is not optimal but very close. By tweaking coefficients/parameters a higher accuracy / low runtime can be obtained. [1]

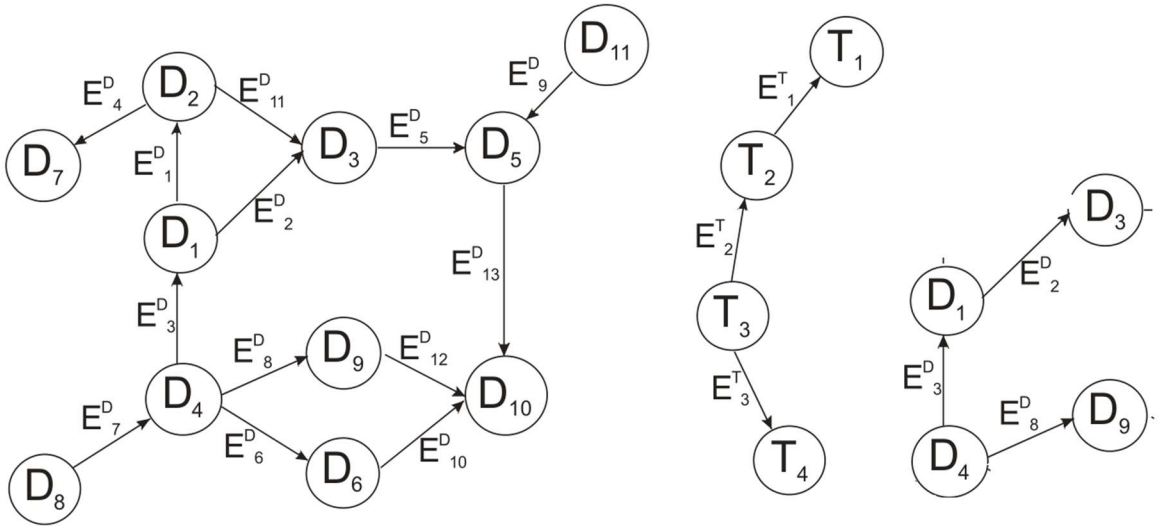


Figure 1. Graph Matching example from left – data graph, template graph, an exact matching.

Graph matching finds its application in various fields such as social networks, identification of anomalies in a big systems (e.g. supply chain, logistics)[4]. Because of its high application in big graphs, generally the graph matching algorithm needs to be extremely fast and be able to produce diverse results which are high in accuracy or matching score; the TruST algorithm aims to achieve that. The inputs in the graph matching problem are the template graph and the data graph.

B. TruST Algorithm/ Problem Formulation -

The TruST Algorithm is discussed in this section, an inexact graph matching algorithm where we traverse through the data graph and template graph and try to find the best matching close to the template graph. It works in the following phases –

1. Similarity Score Calculation

   The template graph and data graph being provided, finding the similarity score between each node of template graph and data graph is required so that the TruST algorithm can be complete. This similarity score can be calculated using various methods; the one used for the TruST algorithm is the Levenshtein string distance. The Levenshtein string distance is an algorithm that uses matrices to find the distance between two strings by approximating how much one string must be changed to convert it to the other. The values on each node of the template graph and the values of the data graph are taken, compared, then stored in a similarity score table. So we

can find a similarity score between the node on the template $v_i \in V_T$ and the node on the data graph $v_i \in V_D$, denote it by $S(v_i, v_j)$[4].

2. 1-Hop score calculation

   Once the similarity score calculation has been done, one-hop scores between each of the nodes are required. This 1-Hop score is a representation of not only the similarity between those two nodes but also the similarity between their 1-Hop neighborhood. It is important to take into account the neighborhoods because the topology of the template graph and data graph also counts towards matching. In doing this, the probability of getting a better graph matching is increased. In the later stages of the heuristic search(TruST), only the 1-Hop score is used. By mixing the similarity scores between two nodes and the similarity between their neighborhoods, a new score can be obtained and used for the TruST algorithm[4]. The way we calculate similarity score and 1-Hop scores is as follows –

   i. Similarity scores can be obtained in the form of $S(v_i, v_j)$ or S_{ij} using the Levenshtein string distance.
   ii. Then, find all the neighbors of node *i* and all the neighbors of node *j* and find the best bipartite matching between them. Let's denote the best bipartite matching score as W_{ij}. Finding bipartite matching is time-consuming and therefore other heuristic methods could also be used. For example, taking the average of node scores of all the neighbors of *i* and *j*.
   iii. Therefore, $OH_{ij} = 0.5*S_{ij} + 0.5*W_{ij}$. This OH_{ij} is passed into phase 3.

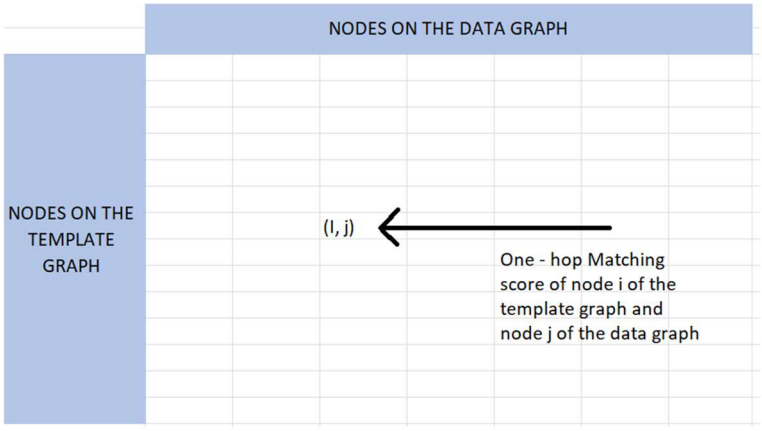


Figure 2. A representation of One-hop score table

3. Tree Traversal.

The third and the most important part of the TruST algorithm is the tree traversal. Input to this stage is the one-hop score that we got. The graph can be represented by an attributed structure as:

$$G = (V, E, A_v, A_e)$$

where V a set of nodes, E a set of edges or arcs, A_v represents the set of node attributes, and A_e represents the set of arc attributes. We have the one hop scores in the form of OH_{ij}. The TruST algorithm that is usually proposed does breadth-first search[3], and for each level keep the top *k* 1-Hop score branches then continue searching[4]. The TruST algorithm created is a beam-search algorithm that utilizes a parametric method to control the state growth. After the 1st stage, we take the *k* best one-hop-scores from the table and then perform the second iteration on them. This is a matching. The second iteration involves looking at all the neighbors of the items in matching and trying to find the best one to match. Only the corresponding matching is considered and therefore, template topologies are bound to be adhered. This search continues until either all the nodes of the template graphs are matched to data graph nodes, or terminated when the user wants. For maximum accuracy, the depth of the search is the number of nodes in the template graph denoted as:

$$\delta = |N_T|$$

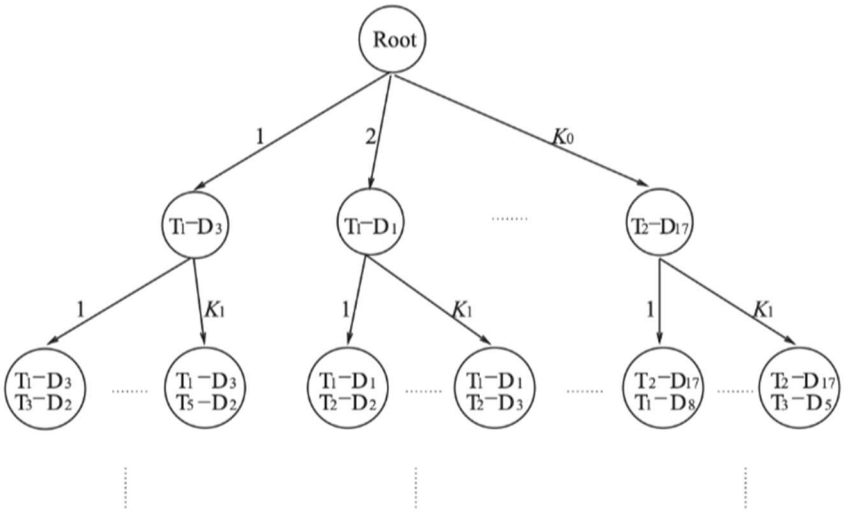


Figure 3. An example of Tree Traversal

4. Parallelization

In this section, we talk about the need for parallelization and architecture for it. The subgraph matching problem is generally applied to pick graphs where traversal is very time-consuming.

Generally, these graphs have a number of nodes from a couple thousand to forty-million or more;[1] therefore, there's a need for a heuristic approach as well as a parallel approach to speed things up. The parallelization is achieved on GPU using Compute Unified Device Architecture (CUDA) architecture in Python programming language. Libraries used are numba.cuda and cupy[8][9]. These are the specific libraries created to utilize the power of NVIDIA's GPU using Python. This section will discuss the salient features of parallelization and CUDA by NVIDIA.

When a computer program makes use of certain code again and again during processing, it is observed that that specific part of the code does not interfere with itself and can be executed parallelly[2]. NVIDIA and AMD provide commercial-grade parallel processing units, GPUs, which can be coded for any computation. We will specifically discuss CUDA, which has been developed by NVIDIA to facilitate the programming of its GPUs[8][9].

CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows the software to use certain types of graphics processing unit (GPU) for general-purpose processing, an approach called general purpose computing on GPUs (GPGPU)[2]. CUDA is designed to work with programming languages such as C, C++, and Fortran[2]. The program is coded on the Local machine (aka host), the host transfers the data to the GPU and then GPU processes (aka device) all that data in parallel. After the processing, the data is transferred back to the host. The data transfer between host and GPU are time consuming steps. [2]

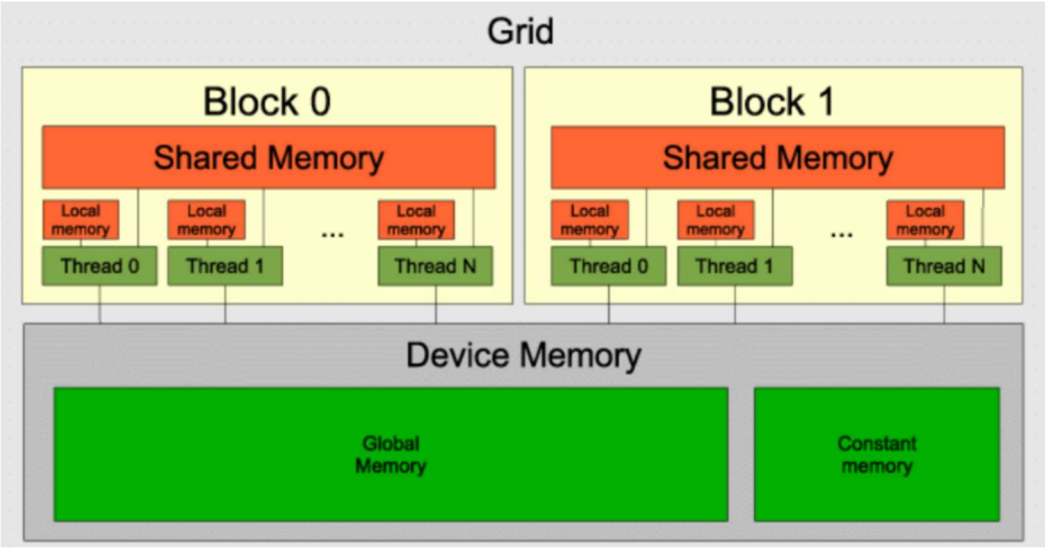


Figure 4. Memory organization in a GPU

The Memory architecture in the GPU is also different as compared to the CPUs. Figure 4 explains the memory architecture. Every device has one global memory, when data is copied from the host, it gets stored in the device's global memory. The GPUs have different blocks which share the shared memory among themselves. It is very fast to access the shared memory as compared to the global memory. And every block has many threads, which has a local memory, which can only be accessed in that thread[2]. Generally, optimization of memory allocation is an important step in parallelizing a particular algorithm.

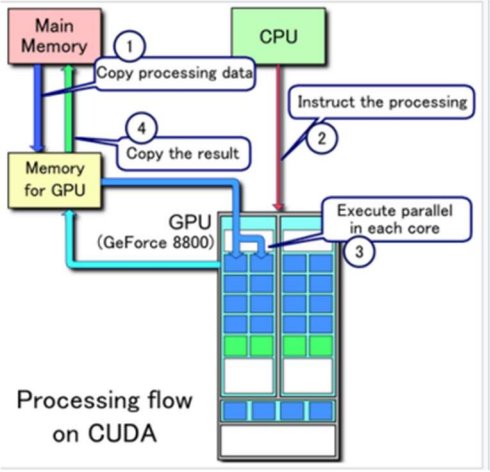


Figure 5. Operation Processing Flow on GPU

Numba.cuda allows the programmer to use the GPU easily. The user must define a kernel in Python and then use it to launch a kernel[8]. The launch of the kernel specifies the number of blocks and threads in that specific kernel. Cupy is also used to create arrays directly on the GPU[9]. Since the TruST algorithm requires many generated arrays, cupy is necessary for those operations. This will save time as the array is directly created on the GPU compared to having to transform the array from CPU to GPU type.

2. Methodology/Approach

In this section, we will discuss how we are parallelizing the TruST algorithm. We aim to parallelize all the three sections of the Trust Algorithm. So, we will go through the details of strategy in the following section.

a. **Parallelization of Similarity Scores Calculation**

   This step is highly parallelizable. We could run every single node-node pair on a single thread of GPU. We pass the whole node values array inside the kernel and find the index of the template graph node and the index of the data graph node to process. Retrieving its data, it can be passed through the Levenshtein string distance function. The function returns a value that will be stored in the similarity score matrix, which will have a size of (number of nodes in Template Graph, number of nodes in Data Graph) at the index *i* (selected index of data graph) and column index *j* (selected index of template graph). We will use the result of similarity scores table in the one-hop scores calculations and trust searching algorithm phase of the program.

b. **Parallelization of One-hop-Score Calculations**

   The CPU version of our code first go through all the neighboring nodes of the data graph part of the matching and the corresponding template graph part of the matching and creates a 2D array 't' to store all the values of similarity scores (between neighbors). Later, this 2d array is passed into the 'MAX_ALGORITHM_ALGO' function to find the best bipartite matching. This score is used in conjunction with the similarity score to give out the one-hop scores between the Data node – template node pair.

   We parallelize this algorithm over number of nodes in the data graph and the number of nodes in the template graph, thereby reducing time consumed by a factor of n*m. This is a theoretical boost-up and the actual boost-up attained will not be close to this value because of several reasons. We will be comparing our GPU implementation with the version of CPU implementation.

   The 'MAX_ALGORITHM_ALGO' function which calculates the bipartite graph matching between the neighbors of the candidate nodes has a recursive part to it and in the CPU function, we added it as a subfunction to the 'MAX_ALGORITHM_ALGO' function. However, in Cuda, subfunctions are not allowed and therefore we changed the recursive part of the bipartite matching to an artificial stacked function and then performed our calculations on it in the GPU. Obviously, there are shortcomings to this implementation, and we will discuss them in discussion section.

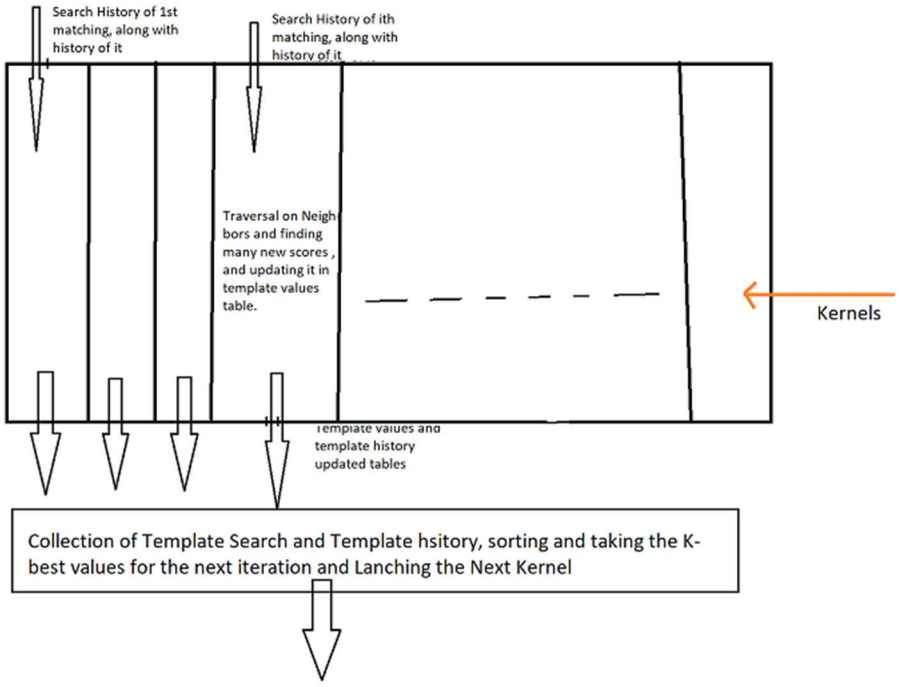
c. **Parallelization of TruST Algorithm**

In the CPU version of TruST algorithm, we have a tree search where we are starting with k0 trees and then increasing the size of the tree to the maximum search size that we need (the variable Beta). Therefore, this algorithm is parallelizable on the search size of the tree. We launch "search size" many threads in 1-Dimensional grid and then perform traversal on each of them. The data structure is as follows –

1. A template value array variable was created to store the current values of matching of all the tree branches.
2. A template history array was created to store the value of all the previously matched nodes in that particular tree branch (every branch corresponds to a different thread and therefore every thread has its own history).

Structure of Execution –

1. We launch the kernel delta times, one time for each of the levels of the search. This means that we pass the current level as an input to the kernel function.
2. The kernel knows the level of search, it knows the searching history using the template history variable and the matching value using the template value variable. It will go through the template history variable up to a depth of the current level and then find the neighbors of all the node-node matchings in that particular graph matching. Then perform the calculation for the best graph options that we have. It then stores all this data into a template value array that has (no of threads * no_of_further_traversal_thread) shape.
3. After the execution of the kernel, we get our data back in the form of a template value and template history. The data in the template value array is then sorted and accordingly we also sort and stores the template history table which contains information about the new matching, this way we save information of new matching that are going to go into the next stage.



Figure(6). Working of the Trust Tree Traversal Algorithm in Parallel.

3. Experiments Section

a. Data Generation

We are generating random synthetic data for the data graph using random. For the template graph we could choose to make our own template graph or make that randomly based on the number of nodes and the number of edges it can have. Some classical graph template motifs can also be generated manually in our program. However, we will be sticking to randomly generated graphs as they provide more variety of results.

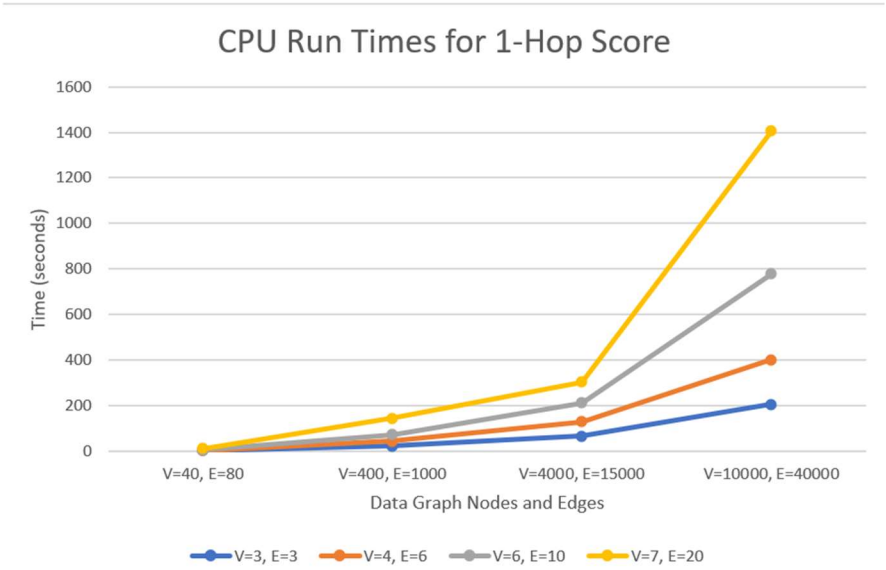
b. Experimental data collection

We use different number of nodes and different number of edges on both the data graph and the template graph. we then record the time taken by one hop score implementation on the GPU and that on the CPU. this is followed by the recording of time taken for execution of trust tree searching algorithm on CPU and on GPU. We collect these times and show our findings.

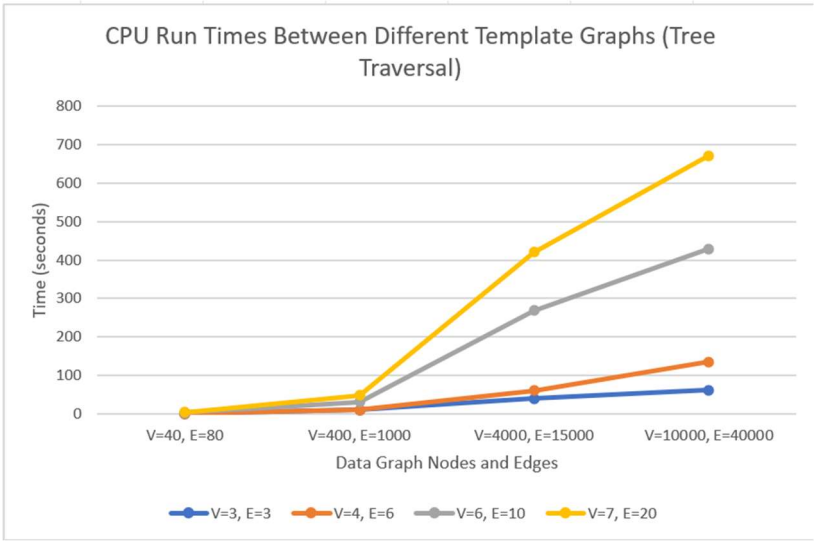
System Configuration –

1. GPU used – Tesla K80, Compute Capability 3.7, 24GB Memory, 4992 cores.
2. CPU used – Intel® Xeon® 2.3 GHz, RAM – 12GB.

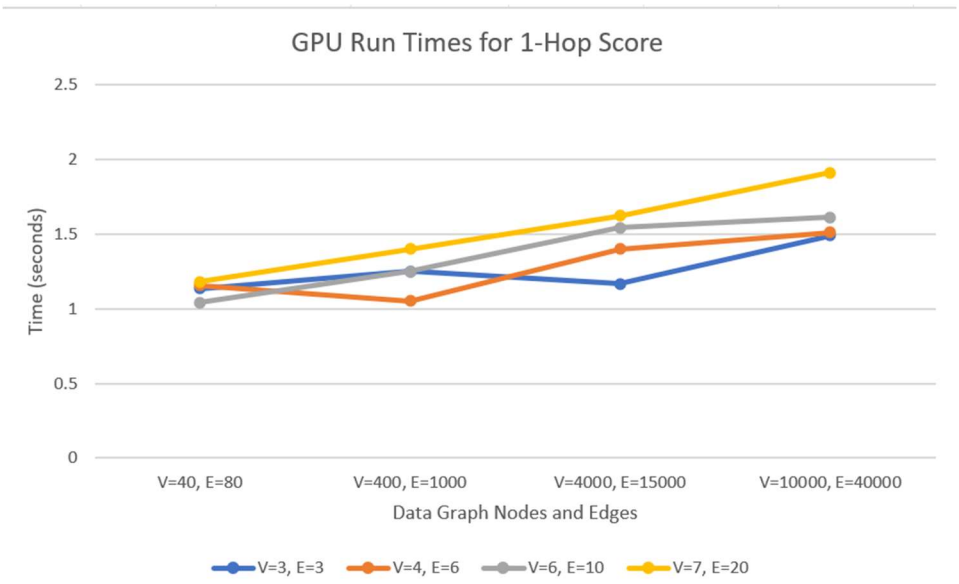
c. Results



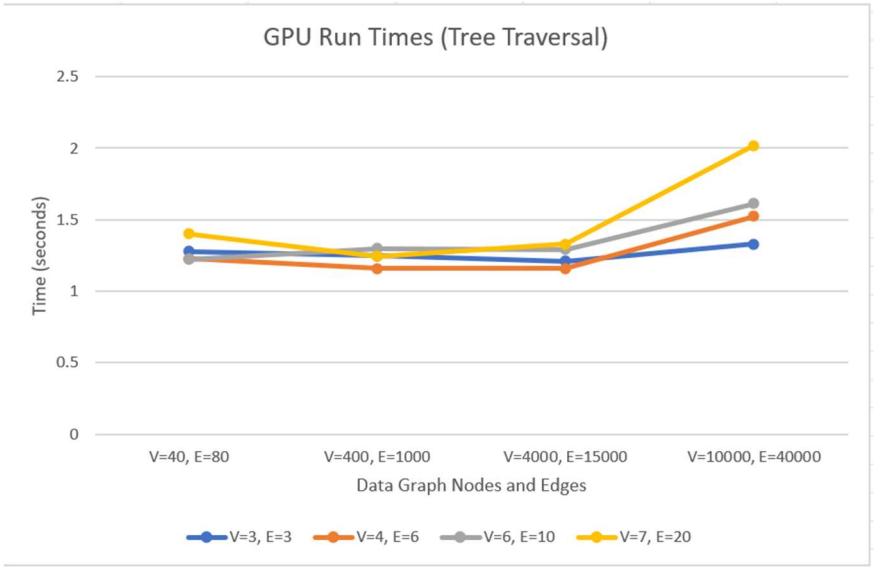
Figure(7). The Comparison of the Performance of CPU on One-Hop Scores on varying Data sizes.



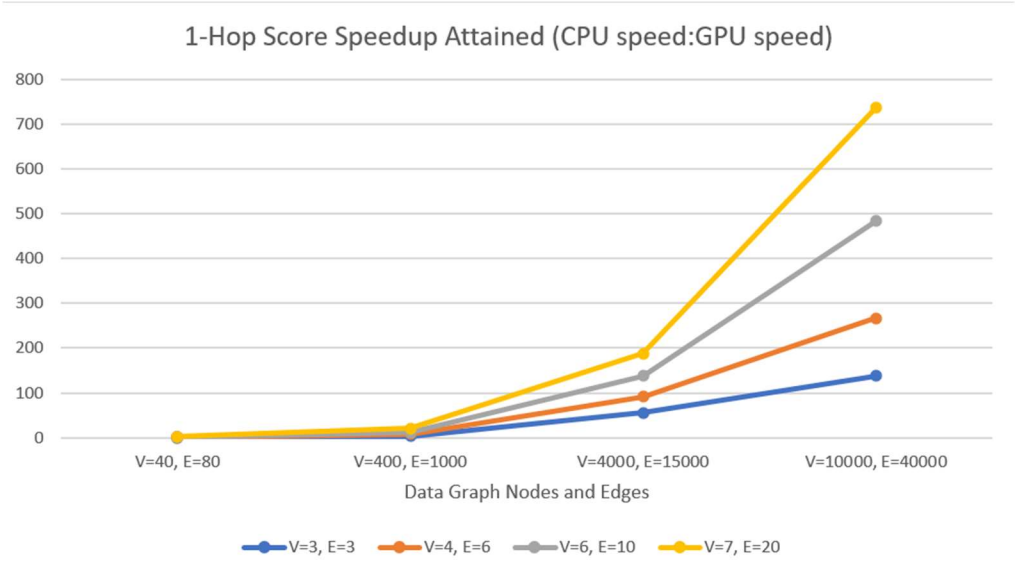
Figure(8). The Comparison of the Performance of CPU on Tree Traversal on varying Data sizes.



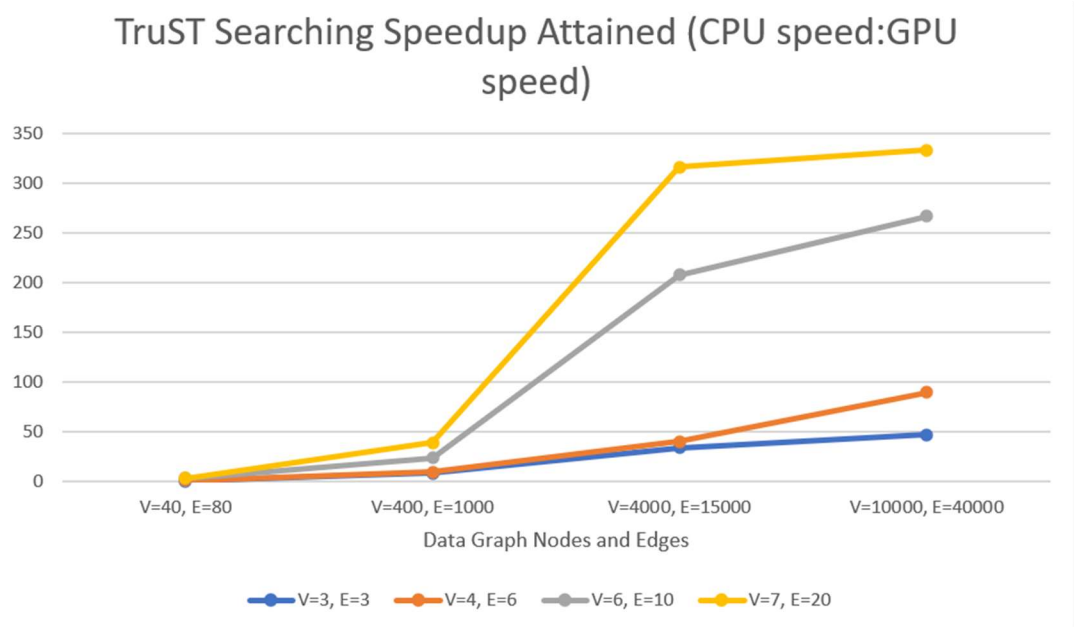
Figure(9). The Comparison of the Performance of GPU on One-Hop Scores on varying Data sizes.



Figure(10). The Comparison of the Performance of GPU on Tree Traversal on varying Data sizes.



Figure(11). Speedups on One-Hop Scores



Figure(12). Speedups on Tree-Traversal

	Data Graph		Template Graph		Execution time		GPU Execution time		Speedups attained CPU:GPU	
Sr No.	No. of Nodes	No. of Edges	No. of Nodes	No. of Edges	One-hop score	Trust Tree Traversal	One-hop score	Tree Traversal	One-hop score	Trust Searching
1	40	80	3	3	0.3672	0.1760	1.1356	1.2801	0.3234	0.1375
2	40	80	4	6	0.7339	0.6426	1.1564	1.2267	0.6346	0.5238
3	40	80	6	10	1.2012	3.2430	1.0435	1.2254	1.1511	2.6464
4	40	80	7	20	2.2612	3.7501	1.1820	1.4012	1.9131	2.6763
5	400	1000	3	3	4.4040	9.7514	1.2527	1.2494	3.5155	7.8050
6	400	1000	4	6	8.8727	10.5813	1.0522	1.1607	8.4324	9.1165
7	400	1000	6	10	14.4211	30.6109	1.2505	1.3005	11.5326	23.5373
8	400	1000	7	20	28.9680	47.8257	1.4024	1.2437	20.6554	38.4556
9	4000	15000	3	3	64.8889	40.3240	1.1668	1.2093	55.6119	33.3455
10	4000	15000	4	6	128.6486	60.6072	1.4021	1.5253	91.7519	39.7337
11	4000	15000	6	10	212.7749	268.3556	1.5429	1.2927	137.9077	207.6006
12	4000	15000	7	20	304.1698	420.0750	1.6223	1.3290	187.4924	316.0935
13	10000	40000	3	3	204.8660	61.9532	1.4900	1.3295	137.4928	46.6005
14	10000	40000	4	6	402.4207	135.4928	1.5113	1.5232	266.2702	88.9531
15	10000	40000	6	10	779.2258	429.3690	1.6135	1.6124	482.9273	266.2929
16	10000	40000	7	20	1406.2339	672.1200	1.9123	2.0194	735.3451	332.8340

Figure (13). Run Times of the Algorithm on GPU and CPU.

4. Discussion

As it can be seen, we were able to get a boost up of upto 800x while using the GPU on large Graphs. Figure(13) shows the collected data and Figures (7)-(12) shows the inferred results. We can see that the execution time of the CPU increase as the size of the data graph and template graph increase, whereas the same for the GPU doesn't change much because of the parallelization. This is as expected from the parallelization perspective. We will now discuss some of the strategies that we have implemented, their limitations, and how they can be improved in the future.

a. Adjacency matrix

   In the tree searching part, we have now added an adjacency matrix which has reduced space requirements by a lot. This has in turn allowed us to launch more threads per block, thereby extending the amount of capability we have to use the GPU. In the absence of adjacency matrix implementation, we would have had to search through all the edges of the graph which we do not have to do now. However, adjacency matrix implementation also has its disadvantages, the most major disadvantage is that the number of edges in any node is not constant and therefore while saving it we cannot just allocate it a single chunk of memory. the solution that we implemented in the GPU is that we gave a maximum number of neighbors every node can have and store that array in the GPU global memory. if the average degree of any note is 4, then the maximum degree of one of the nodes could easily go to 40. Therefore, we will have to use 40 vacant spaces for every one of the nodes to be able to store all the data and adjacency matrix form. while we have done it, it will still be better to limit the number of neighbors to a particular value and store only the highest 20 or 10 neighbors. this has even better capability of reducing the space.

b. Recursive functions

   The recursive function in the Max algorithm Algo function was eliminated using an artificial stack that keeps track of a lot of values. However, this means that we have to store a lot of values on the local array. But since there is only 64 KBs available on GPU local memory (or around that number depending upon the GPUs), when the size of the graph increases, this can push the local memory consumption to go beyond the 64 KB requirement, and therefore can crash the program. That is why in our code, we have only used one thread per block to execute our code. However, better ways to manage the local memory should be explored in this regard.

c. Memory efficiency tradeoff

In the trust tree searching algorithm, we have a very large local array allocation, while the local arrays are fast to process, the fact that they are consuming too much space do not allow us to launch many threads per block, thereby, limiting the scalability of the algorithm. In our case, this specific array was the similarity values of all the neighbors. We chose to keep that in global memory, end retrieve the information as and when required. this allowed us to launch more threads per block, but limited the speed of the code execution. this tradeoff will always remain and has to be carefully balanced.

d. Hashing

   In the CPU version of our code we had used, hashing to avoid multiple entries. However, in GPU code because hashing function has not been implemented, there was presence of redundant matching in our final result. This can be avoided in future work by using some hashing strategy inside the GPU kernel itself.

e. Implementation on big graphs or Social Networks

   If any local array or shared memory array in the GPU has its size or shape dependent upon the size of the graph, then that solution is not scalable. Therefore, in our opinion, we do not think that it is wise to use any local array that will increase its size based on the number of nodes or edges the data graph has. However, it might be acceptable to have a local variable on the GPU threads that uses the size of the template graph. The reason for that is that most of the graph searching problems will have limited size of template graph.

5. Conclusion

In this paper, we used the GPUs to accelerate the TruST Searching Algorithm. We parallelized one hop score calculation and trust tree traversal, and attained a speedup of 20x-200x and 10x-200x respectively. we discussed our implementation of the parallel code. we discussed the relationship that the execution time of trust tree traversal have with different sizes of data graph in template graph. in the discussion section, we explained the limitations of our work and suggested some ways to overcome those limitations.

6. References

1. Gross, Geoff A., and Rakesh Nagi. "Precedence Tree Guided Search for the Efficient Identification of Multiple Situations of Interest - AND/OR Graph Matching." *Information Fusion*, vol. 27, 2016, pp. 240-54.

2. NVIDIA, et al. "CUDA Toolkit Documentation." *Nvidia*, 24 Mar. 2022, docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 30 Apr. 2022.
3. Zhang, Weixiong. *Depth-First Branch-and-Bound versus Local Search: A Case Study*. American Association for Artificial Intelligence, 2000.
4. Gross, Geoff, et al. "A Fuzzy Graph Matching Approach in Intelligence Analysis and Maintenance of Continuous Situational Awareness." *Information Fusion*, vol. 18, 2014, pp. 43-61.
5. Sambhoos, Kedar, et al. "Enhancements to High Level Data Fusion Using Graph Matching and State Space Search." *Information Fusion*, vol. 11, no. 4, 2010, pp. 351-64.
6. Abi-Chahla, Fedy. "Nvidia's CUDA: The End of the CPU?" *Tom's Hardware*, 18 june 2008, www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html. Accessed 30 Apr. 2022.
7. Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing, Volume 38, Issue 3, March 2012, Pages 157-174.
8. Lam, Siu Kwan, et al. *Numba: A LLVM-Based Python JIT Complier*. Association for Computing Machinery, 2015.
9. *CuPy: NumPy* & SciPy for GPU. cupy.dev. Accessed 30 Apr. 2022.
10. High performance DFS-based subgraph enumeration on GPUs , Vibhor Dodeja, 2021, http://hdl.handle.net/2142/110559.

Code Snippets

```
def GPU_search_best_fitting_graph_matrix_type(self, template_graph):
    self.TRUST_clear_start_matrix(template_graph.n)
    # searching_history: [lvl, instance num, 3]
    #                    where each instance = [father instance index, data_node_index, template_node_index]
    # searching_history: Corresponding value of "newest instance"
    # THE FATHER INDEX: WHERE THE INSTANCE COMING FROM.
    #                   Say if we use instance searching_history[3][7(instance 7)] to update the searching_history[4][2]
    #                   Then searching_history[4][2][0] = 7,
    #                        searching_history[4][2][1] = A,
    #                        searching_history[4][2][2] = a

    GPU_SEARCH_RANGE = cuda.to_device(SEARCH_RANGE)
    GPU_one_hop_score_table = cp.array(self.one_hop_score_table)
    GPU_template_graph_adj_matrix = cp.array(template_graph.adj_matrix)
    GPU_data_graph_adj_matrix = cp.array(self.adj_matrix)
    searching_history = cp.zeros([template_graph.n + 2, SEARCH_RANGE + 1, 3], dtype=int)
    searching_value = cp.zeros([template_graph.n + 2, SEARCH_RANGE + 1], dtype=float)

    tmp_value = np.zeros(SEARCH_RANGE * ADJ_MAX * template_graph.n + 1, dtype=float)
    tmp_history = np.zeros((SEARCH_RANGE * ADJ_MAX * template_graph.n + 1, 3), dtype=int)

    #for j in range(1, template_graph.n + 1):
    j = 1
    for i in range(1, self.n + 1):
      tmp_value[(i - 1) * template_graph.n + j - 1] = self.one_hop_score_table[i][j]
      tmp_history[(i - 1) * template_graph.n + j - 1] = (0, i, j)
    tmp_value= cp.array(tmp_value, dtype = float)
    tmp_history = cp.array(tmp_history, dtype = int)
    #print(type(tmp_value))
    #print(type(tmp_history))

    self.GPU_search_update_results(1, tmp_value, tmp_history, searching_history, searching_value)


    for current_lvl in range(1, template_graph.n):
      # self.hashing_table = set()
      GPU__trust_searching_matrix_type[SEARCH_RANGE,1](self.n, template_graph.n, current_lvl, GPU_one_hop_score_table, G
      #DATA_NODE_SIZE : int , TEMPLATE_NODE_SIZE : int ,current_lvl, GPU_one_hop_score_table, GPU_template_graph_adj_mat
      self.GPU_search_update_results(current_lvl + 1, tmp_value, tmp_history, searching_history, searching_value)
      #print(searching_value[current_lvl])

    self.GPU_TRUST_output_result_matrix(template_graph.n, searching_value, searching_history)
```

```
def GPU_search_update_results(self, cur_level, tmp_value, tmp_history, searching_history, searching_value):
  nn = cp.size(tmp_value)
  p = cp.argsort(tmp_value)[max(nn - SEARCH_RANGE, 0):nn]
  nn = cp.size(p)
  GPU_search_update_history_position[nn,1](cur_level, p, tmp_value, tmp_history, searching_history, searching_value,
```

```
@cuda.jit
def GPU__trust_searching_matrix_type(DATA_NODE_SIZE : int, TEMPLATE_NODE_SIZE : int, current_lvl, GPU_one_h
  matching_info = cuda.local.array(shape=50, dtype=numba.int64)
  #  print("LEVEL:", current_lvl)
  check_index = cuda.grid(1)
  searching_index = cuda.grid(1)
  tmp : int = 0
  for p in range(current_lvl, 0, -1):
    check_index, x, y = searching_history[p][check_index]
    if x == 0 or y == 0:
        return
    matching_info[y] = x
    for j in range(1, TEMPLATE_NODE_SIZE + 1):
        if matching_info[j] != 0:
            continue
        for i in range(1, DATA_NODE_SIZE + 1):
            check_flag = True
            for k in range(1, TEMPLATE_NODE_SIZE + 1):
                if matching_info[k] == i or (matching_info[k] != 0 and GPU_template_graph_adj_matrix[j][k]
                    check_flag = False
                    break
            if check_flag:
                # matching_info[j] = i
                # if self.check_hashing_matrix(matching_info, template_graph.n):
                #     print("YES")
                tmp_value = searching_value[current_lvl][searching_index] + GPU_one_hop_score_table[i][j]
                s = int(searching_index * 50 * TEMPLATE_NODE_SIZE + tmp)
                new_value[s] = tmp_value
                new_history[s][0] = int(searching_index)
                new_history[s][1] = int(i)
                new_history[s][2] = int(j)
                tmp += 1
                if tmp >= 50 * TEMPLATE_NODE_SIZE:
                  return
                #self.updating_new_instance_matrix_base(new_value, current_lvl + 1, searching_index,i, j)
```

```
@cuda.jit
def GPU_calculate_hop_score_i_j(gpu_data_n, gpu_word_distance, gpu_data_graph_adj_matrix, gpu_templa
 #   print("matching....")
    id_x, id_y = cuda.grid(2)
    id_x += 1
    id_y += 1

    if id_x > gpu_data_n:
      return
    n = gpu_data_graph_adj_matrix[id_x][0]
    m = gpu_template_graph_adj_matrix[id_y][0]
    #print(n, m)
    max_nm = max(n, m)
    if max_nm == 0:
      gpu_one_hop_score_table[id_x][id_y] = 0.5 * gpu_word_distance[id_x][id_y]
    else:
      cnt = 0.
      cnt_n = 0
      # w = cuda.local.array(shape = (52, 52), dtype=numba.float64)
      # for x in range(1, n + 1):
      #  p = gpu_data_graph_adj_matrix[id_x][x]
      #  for y in range(1, m + 1):
      #    q = gpu_template_graph_adj_matrix[id_y][y]
      #    w[x][y] = gpu_word_distance[p][q]

      lx = cuda.local.array(80, dtype = numba.float64)
      ly = cuda.local.array(80, dtype = numba.float64)
      visited_x = cuda.local.array(80, dtype=numba.boolean)
      visited_y = cuda.local.array(80, dtype=numba.boolean)
      link_y = cuda.local.array(80, dtype=numba.int64)
      for i in range(1, max_nm + 1):
        lx[i] = ly[i] = 0
      for i in range(1, max_nm + 1):
          for j in range(1, max_nm + 1):
              lx[i] = max(lx[i], gpu_word_distance[gpu_data_graph_adj_matrix[id_x][i]][gpu_template_

      for x in range(1, max_nm + 1):
        while True:
          lack_GPU = 100000
          for i in range(1, n + 1):
            visited_x[i] = 0
            visited_y[i] = 0
```

```
@cuda.jit
def GPU_search_update_history_position(cur_level, p, tmp_value, tmp_history, searching_history, searching_value, nn):
  i = cuda.grid(1)
  searching_value[cur_level][nn - i - 1] = tmp_value[p[i]]
  if tmp_history[p[i]][1] != 0 and tmp_history[p[i]][1] != 0:
    #print(type(searching_history))
    searching_history[cur_level][nn - i - 1][0] = tmp_history[p[i]][0]
    searching_history[cur_level][nn - i - 1][1] = tmp_history[p[i]][1]
    searching_history[cur_level][nn - i - 1][2] = tmp_history[p[i]][2]

@cuda.jit
def GPU__trust_searching_matrix_type(DATA_NODE_SIZE : int, TEMPLATE_NODE_SIZE : int, current_lvl, GPU_one_hop_score_table, GPU_template_g
  matching_info = cuda.local.array(shape=50, dtype=numba.int64)
  adj_check = cuda.local.array(shape=50, dtype=numba.int64)
  check_index = cuda.grid(1)
  searching_index = cuda.grid(1)
  tmp : int = 0
  for p in range(current_lvl, 0, -1):
    check_index, x, y = searching_history[p][check_index]
    if x == 0 or y == 0:
      return
    matching_info[y] = x
  for tt in range(1, TEMPLATE_NODE_SIZE + 1):
    if matching_info[tt] != 0:
      for i_index in range(1, GPU_data_graph_adj_matrix[matching_info[tt]][0] + 1):
        i = GPU_data_graph_adj_matrix[matching_info[tt]][i_index]
        already_matched = False
        for check_y in range(1, TEMPLATE_NODE_SIZE + 1):
          if matching_info[check_y] == i:
            already_matched = True
            break
          adj_check[check_y] = 0
        if already_matched: continue
        check_sum = 0
        for check_x_index in range(1, GPU_data_graph_adj_matrix[i][0] + 1):
          check_x = GPU_data_graph_adj_matrix[i][check_x_index]
          for check_y in range(0, TEMPLATE_NODE_SIZE + 1):
            if check_x == matching_info[check_y]:
              adj_check[check_y] = 1
              check_sum += 1
        for j in range(1, TEMPLATE_NODE_SIZE + 1):
          if matching_info[j] != 0: continue
          check_flag = True
```

```python
# for simplicity we assume it's a totally undirected graph for now
class Graph:
    # n, m are node_size, edge_size
    def __init__(self, n : int, m : int, node_list: List[Node], edge_list: List[Edge]):
        # here the node in the edge_list is corresponding to the node in the node_list
        self.n = 0
        self.m = m
        self.hashing_table = set()
        self.node_list : Dict[str, GraphNode] = {}
        self.heap_size = 0
        self.H: List[(int, int)] = []
        self.heap_info_list = [{}]
        # adj_matrix and indexing is 1-based
        self.adj_matrix = np.zeros([n + 2, ADJ_MAX + 2], dtype=int)

        self.word_distance_table = np.zeros((n + 2, ADJ_MAX + 2), dtype= float)
        self.searching_history = np.zeros([0], dtype= int)
        self.searching_value = np.zeros([0], dtype= float)

        self.node_name_list = ["-1"]
        self.one_hop_score_table_old = None
        self.one_hop_score_table_CPU = None
        self.one_hop_score_table = None

        # build_node
        for node in node_list:
            self.node_name_list += [node.name]
            if self.node_list.get(node.name) is None:
                self.n += 1
                self.node_list[node.name] = GraphNode(node, self.n)
        # build_edge
        self.edge_list : Dict[str, GraphEdge] = {}
        for edge in edge_list:
            self.build_edge(edge)

        self.one_hope_score_dict: Dict[str, float] = {}
```