

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**  
**SINGAPORE**

School of Computer Science and Engineering

CZ2001: ALGORITHMS

Semester 1 2020/2021

## Project 2: Graph Algorithms

Group members:

AIDE ISKANDAR BIN ABDUL RAHIM

CHEW KIT WAYE ANDREL

CHIEN YONG QIANG

GOH YUN BO, WAYNE

JIN HAN

Lab Group: SSP7 Group 1

## Introduction

Graphing algorithms are implemented in many areas and industries, from warehousing logistics, to smart vacuum robots, games, and even in emergency operations. For the healthcare industry, every moment is crucial for determining the fate of lives. Ambulances with the most optimal route to hospitals play an important role. While every city's road network and conditions are distinct, an effective graphing algorithm can shorten the time to the nearest hospitals.

A\* Search combines both the performance of Breadth First Search (BFS) and Greedy Search algorithms, by finding the fast and shortest path to the hospital. In this project, the graph algorithm was programmed in Python, and processed using CUDA programming, along with capable hardware. As part of an empirical study, the time taken for the algorithm to execute with varying total numbers of hospitals and numbers of nearest hospitals looked for was measured and compared.

## Information Processing

Given a real world network with roughly 3 million edge lists and 1 million nodes, undirected and unweighted, using conventional algorithm methods to convert an edge list into an adjacency list or matrix might not be feasible and will lead to out of memory error. We explored CUDA programming[1] where CUDA cores can be found on NVIDIA Graphic cards (GPU). Modern GPUs contain thousands of CUDA cores that enable us to perform a few billions operations within seconds [Fig. 1]. Furthermore, it is more efficient to process large datasets on GPU as compared to a CPU. Hence, using GPU accelerated algorithms allows the program to perform operations in parallel, also drastically reducing the time complexity of an algorithm [Fig. 2,4]. However, time complexity varies on different GPU models and the amount of CUDA cores. In our test bench, we are running on 2 Nvidia RTX 3090 on SLI [Fig.3].

Fig. 1

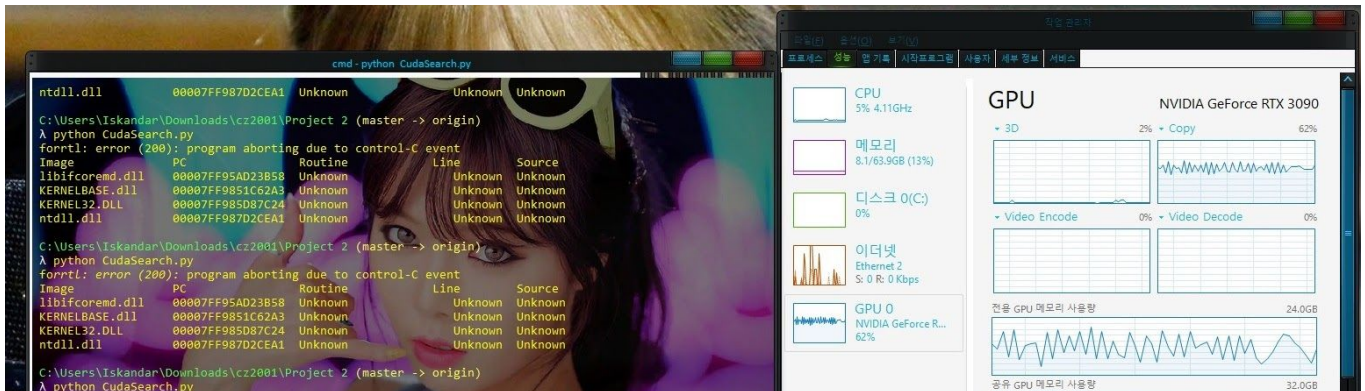


Fig. 2

```
4      r = [[0 for i in range(size)] for j in range(size)] # n^2
5
6      for row,col in arr1: #n
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
```

```
#Initialize a [0,0,0,...,0] with a size of n in parallel
arr = cp.zeros(size, dtype="int32")
cur = 0

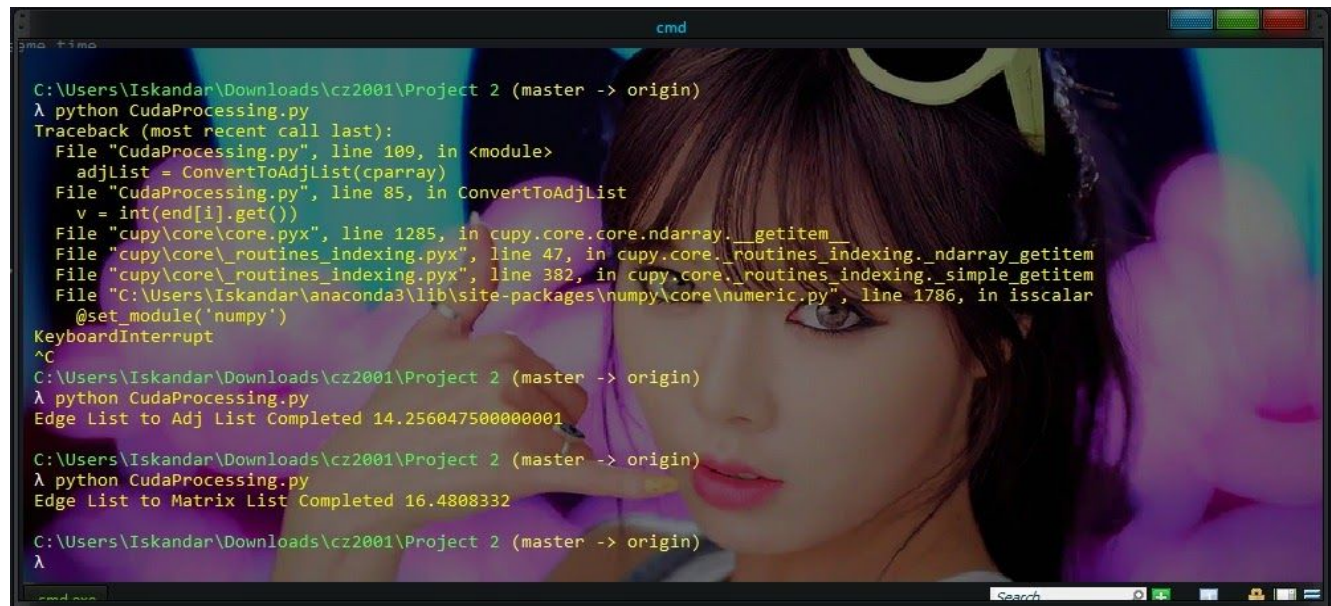
cp.cuda.Device({0,1}).use()
with open("matrixList.txt", 'w') as f:
```

Fig. 3



Using CUDA for processing, we have converted the edge list from the real network graph into both an Adjacency List and Matrix List [Fig. 4]. The data processing of converting from an edge list of a graph to an adjacent list was completed in 14.256 seconds, while the processing of the edge list of the same graph to a matrix list was completed in 16.481 seconds. Compared to using a CPU for data processing, a GPU utilises multiprocessing with a wide amount of cores, that can allow parallel processing. If a CPU was used, the conversion takes up an average of half a day to finish running. Using a GPU also has lesser caches than a CPU. It significantly increases the processing speed especially for instances in a real roadmap graph where over 3 million edge lists and 1 million nodes were processed.

Fig. 4



#### Convert Edge list to Adjacency List and Matrix

Convert to:	Scenarios: $G = (V, E)$ and $C = \text{Cuda Cores}$ ( $V = \text{Vertice}$ , $E = \text{Edge}$ )	Conventional algorithm	GPU accelerated algorithm
Adjacency List	$E = [[1,2], [2,3], [2,4], \dots, [V-1, V]]$ $\text{List} = [[1,2], [2,3,4], \dots [V, V-1]]$	$O(V+E)$	$O(V + \frac{E}{C} + \log C)$

Adjacency Matrix	$E = [[1,2], [2,3], [3,4], \dots, [V-1, V]]$ Matrix = $[[0,0,1, \dots, 0], [0,1,0, \dots, 1]]$	$O(V^2)$	$O(V^2/C + \log C)$
------------------	---	----------	---------------------

## Analysis of Algorithm

While the processing was accomplished using a GPU, the instructions were done using a CPU, to adapt to most users' hardware usage of PCs. Using the A\* Search, the time complexity of the algorithm can be analysed as shown below.

Apart from the location of the starting and ending nodes, the edges can affect the effectiveness of the algorithm. The best case is found when the target node is found right beside the starting node, with a complexity of  $O(E)$ . The worst case is when no path can be found, or when the target node cannot be reached. It has a time complexity of  $O(|V|+|E|)$ . Lastly, the average case is where the target node is found a few edges away from the starting node, with a complexity of  $O(\frac{1}{E} + \frac{V+E}{V})$ .

Best Case:  $O(E)$  - target node beside start node

Worst Case:  $O(|V|+|E|)$  - no path found / node cannot be reached

Average Case:  $O(\frac{1}{E} + \frac{V+E}{V})$  - number of edges from the current node

Other algorithms include BFS, which uses the First In First Out (FIFO) approach and Depth First Search which uses the Last In First Out (LIFO) approach. Greedy Search approximates your distance from the ending node, through a heuristic and calculations.

However, we have chosen A\* Search instead of other algorithms. Unlike DFS, A\* Search is optimal. Furthermore, whilst Greedy Search just explores all possible paths, A\* Search will try to look for a better path by using a heuristic function which gives priority to nodes that are supposed to be better than others, resulting in better performance. A\* Search also usually expands less nodes compared to BFS, hence outperforming BFS in terms of speed.

## Empirical Study

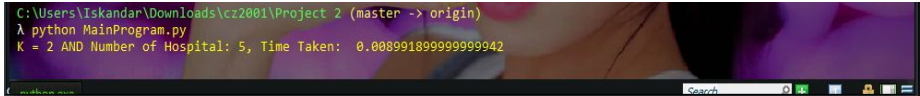
In order to test the effects of varying the number of hospitals (h) and the number of nearest hospitals looked for (k) on the A\* Search algorithm, we have conducted empirical study on a random graph, generated from NetworkX [2].

The results are summarised as follows:

### A\* Search Algorithm

Time Taken /ns	Random Graph
Number of Hospitals = 5 k = 1	0.0094793 
Number of Hospitals = 10 k = 1	0.0175139 



Number of Hospitals = 5 k = 2	0.0089919 
----------------------------------	---

When the number of hospitals (h) increased twofold from 5 to 10, the time taken by the algorithm to execute roughly doubled as well. Please note that the graph is generated randomly, execution time will be slightly different after each execution.

In the event of a major accident or natural disaster, the top few nearest hospitals from each node on the roadmap will have to be found and distance computed in the shortest time possible. In this case, the number of nearest hospitals looked for (k) will have to increase. When k increased from 1 to 2 as shown above, there was no significant change in the time taken by the algorithm. Thus, k has minimal effect on this algorithm, enabling it to remain fast and efficient even during crises.

The designed A\* search algorithm is able to find the closest k hospitals, quickly by eliminating the hospital furthest away while pre-processing, to shorten the time for searching.

## Conclusion

After conducting an empirical study, we have concluded that the chosen algorithm, A\* search, is the most effective algorithm capable of finding the shortest route to hospitals with a fast speed. With the use of a real road network graph, the number of nodes and edges are substantially huge. A fast algorithm such as A\* search can be better improved with efficient hardware, such as through CUDA programming on a GPU with faster data processing speeds.

## Statement of Contribution

Coding	Iskandar, Yong Qiang, Wayne
Report & Slides	AndreI, Iskandar, Yong Qiang, Jin Han

## References

- [1] NVIDIA, CUDA Toolkit Documentation. [Online].  
Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [2] NetworkX, Random Graph Generator. [Online].  
Available: <https://networkx.org/documentation/networkx-1.10/overview.html>
- [3] SNAP Stanford, Road Network Graph. [Online].  
Available: <https://snap.stanford.edu/data/index.html>
- [4] CuPy, API Accelerated with CUDA Documentation. [Online].  
Available: <https://docs.cupy.dev/en/stable/index.html>