N-Puzzle using AI Algorithms on GPU

KAUSTAV VATS, ANUBHAV JAISWAL and ARHSDEEP SINGH, Indraprastha Institute of Information Technology

1. INTRODUCTION

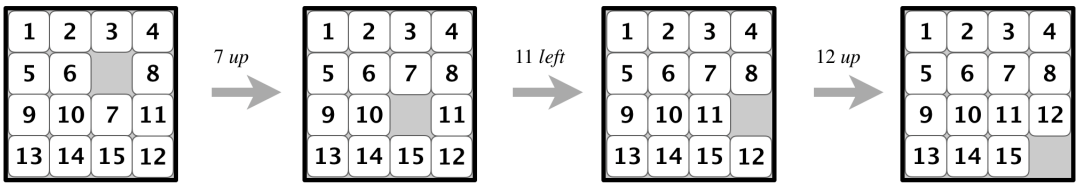

Fig. 1. Example of a 15-Puzzle Problem

The N-Puzzle problem consist of $\sqrt[2]{N+1}$ size square tiles numbered from 1 to N-1 with one tile missing. The objective of the problem is to arrange tile in some numerical order.

Intuitively solving this problem requires the algorithm to visit all the possible states and select the final solution state. Visiting states requires recursion and backtracking which is easy to implement in a serial code(Non threading implementation), Thus implementing these graph algorithms on the GPU is a challenging task that we wish to accomplish.

There are 4 algorithms in this field that we are implementing - **BFS**, **DFS**, **A*** and **IDA*** algorithm.

2. LITERATURE REVIEW

2.1 Accelerating Large Graph Algorithms on the GPU Using CUDA [2]

Graph algorithms require recursion and implementation of queues and stacks that might lead to an hindrance in the performance hence this paper describes how to implement a parallel version of the BFS and other graph algorithms. It defines a new way to store the adjacency list of the graph making parallel execution easier and causing less overheads.

2.2 Massively Parallel A* Search on a GPU [3]

This paper explains three parallel things required to make the A* algorithm work in parallel. It describes the parallel Computation of the Heuristic function, parallel implementation of priority queue and detecting Node duplication on GPU.

3. IMPLEMENTATION AND ANALYSIS OF THE ALGORITHMS

N-Puzzle is a graph traversal problem. Where each state contains a structure of a representation of the puzzle in a form of 2D Array. Each state can have at max 4 neighbouring state, depending upon

the position of the blank tile. If blank tile is on the corners there will be 2 possible state. When blank tile is on the edges there are 3 possible state and 4 for rest of the positions.
Big issue with N-Puzzle problem is that, search space is not defined. It's an exploratory search problem, which means that the algorithm can be massively parallelized.

3.1 Breadth First Search Algorithm

Breadth first search is the most commonly used algorithm for solving graph traversal search space problems. Bfs start traversing from a start node and traverse the graph layerwise "Fig. 3" thus exploring the neighbour nodes. Bfs also handles already visited nodes and visits each node only once.
Serial implementation of bfs uses queue to store neighbours. Then it pops the neighbour from the queue and add it's neighbour if not visited to the queue again. In the breadth first fashion, it first visits all neighbours of the current node and so on.

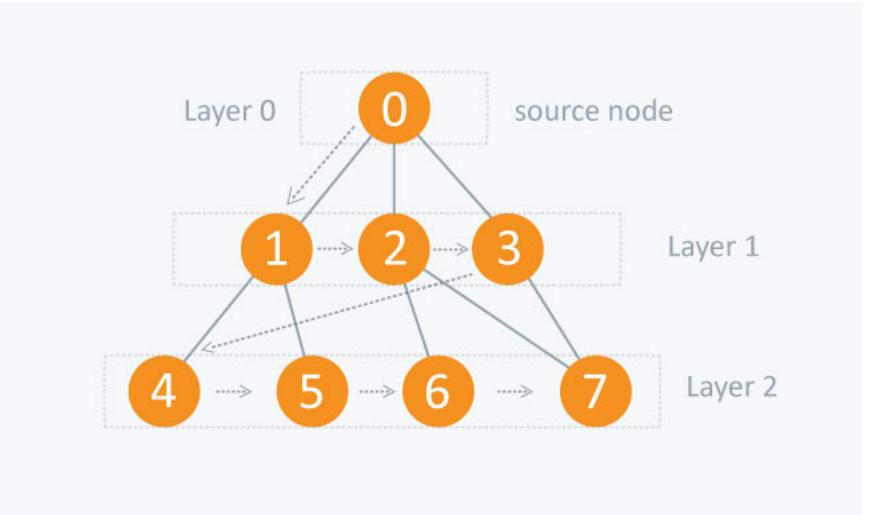

Fig. 2. Example of a bfs approach

Trying to parallelize the serial approach is has two difficulties: A Recursive BFS will affect the performance and stack overflow is likely to occur cause our state space is of (N)! Where N is the size of the puzzle. Maintaining a queue would require atomic operations on the queue which again would lead to communication overhead among the threads Hence the authors of the paper Accelerating Large Graph Algorithms on the GPU Using CUDA have suggested a different approach to store as well as query the graph.

We will query the graph level by level and use the adjacency list to store the representation of the graph.

To store the graph we usually use adjacency List as it is space efficient but every vertex might have a different number of edges hence the size of the list for each vertex varies and hence is difficult to manage and allot threads for computation. Thus we modify the adjacency list and store it in a single array where edges of a vertex are stored contiguously. So we have an array of Edges and we will have a structure of vertices that stores the starting position in the Edges array from where its edges start and the number of nodes it is attached to.

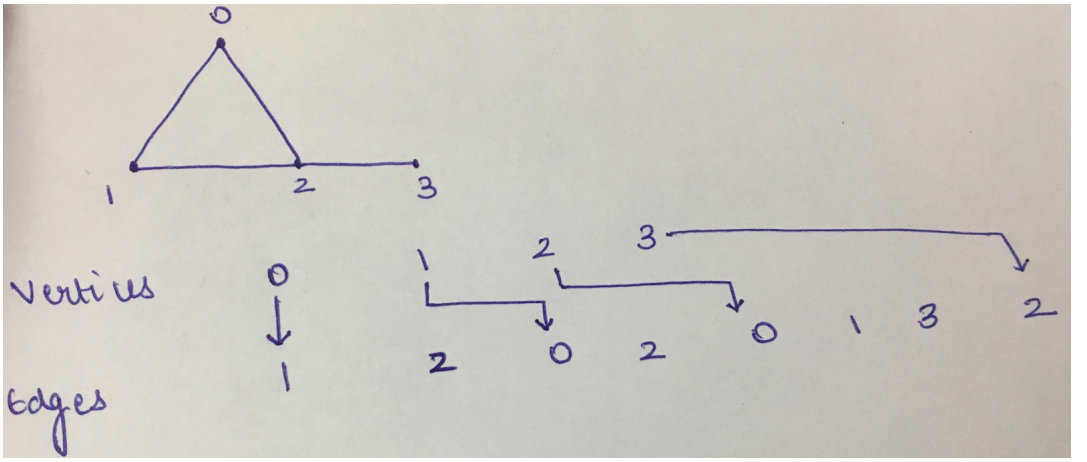

Fig. 3. Parallel Bfs approach

In “Fig. 3” vertex 0 starts from 0th index and has length 1, vertex 1 starts from 1st index and has length 2 and so on.

Along with the above array and structure, we maintain another visited array, cost array, and level array. Visited array prevents from visiting the node again, cost array stores the minimum cost path and level array tells whether this vertex lies on the level we are currently exploring.

We take advantage of the parallelism in the level wise architecture of the graph where vertices on the same level can be queried parallelly.

Current Implementation of the parallel code

The implementation for parallelly executing bfs in cuda has been referenced from[**?**].

We have currently implemented the parallel BFS on 4-Puzzle and checked whether we are able to reach the solution and if we are then what is the cost. The main part of the implementation was to make the modified adjacency list of the graph. We have assumed each node to be a permutation of the puzzle and thus each node has a maximum of 4 neighbors. We have first given a unique id to each permutation and then proceeded with creating neighbors and storing them in the adjacency list. We have done this for all the 4! Permutations

3.2 Depth First Search Algorithm

Depth first search algorithm is a recursive algorithm, which basically uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
Here backtrack means when you are moving forward there’s no state available to explore further, so you move backwards on the same path to find nodes to traverse.

3.3 A* Algorithm

A* algorithm is very similar to Breadth first search algorithm. A* uses a heuristic function to decide which state need to be explored first and which is closer to final solution. Whereas Bfs explore each and every node, A* only explore those node which have a less total cost. Heuristic tries to minimize the sum of distances between the initial state and the current state the final state and the current state. Ref “Fig. 4”

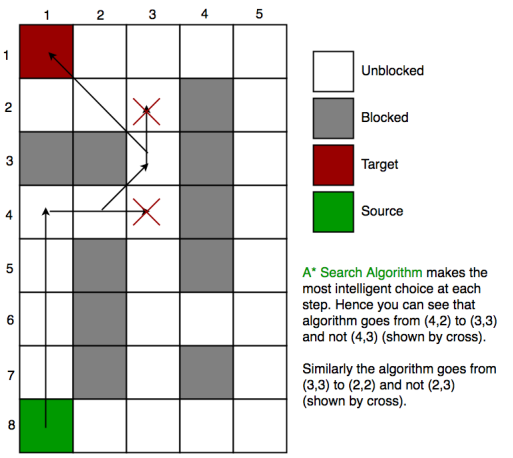

Fig. 4. A* algorithm approach

3.4 Iterative Deepening A* Algorithm

On similar lines, IDA* limits some nodes from visiting, if they have a cost of visiting greater than a threshold. It follows a Dfs approach however, only visits to a threshold of depth. This depth is calculated as the heuristic distance between the initial state and the final state. Ref “Fig. 5”

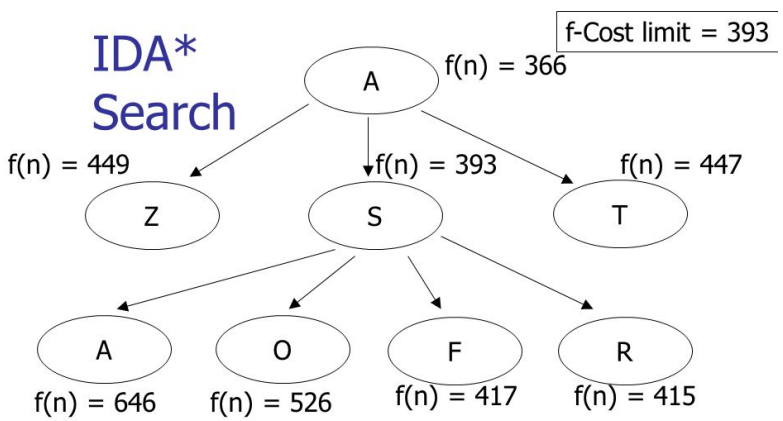

Fig. 5. IDA* algorithm approach

4. RESULTS

We observed that serial implementation of A* and IDA* mostly perform best as compared to Serial implementation of Bfs, Dfs Algorithm. Below graph can give much better comparison between all algorithms. Ref “Fig. 6” We have completed serial implementation of all four algorithms in our project and bfs cuda implementation. We have implemented generic code for different value of N and different complexity of Start state.

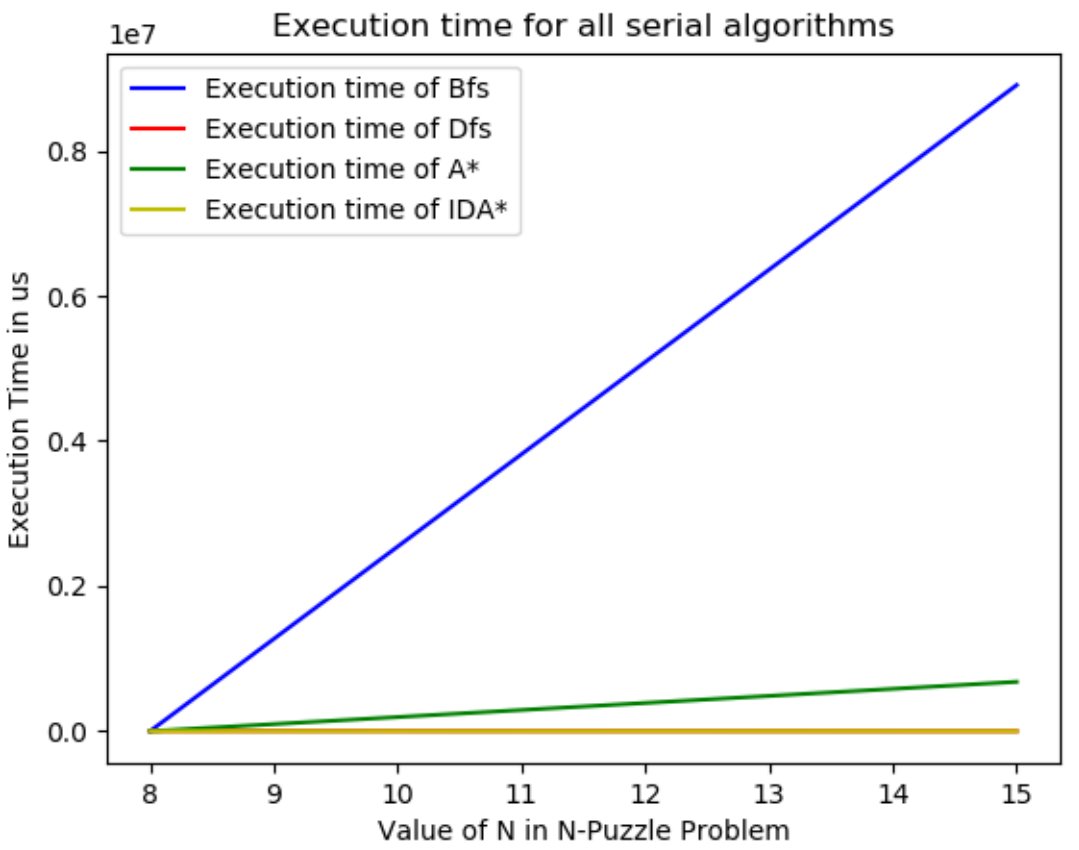

Fig. 6. Execution time Graph

N-Puzzle Size	Execution time for different algorithms			
	Bfs	***Dfs***	***A****	***IDA****
8	58 us	2429 us	14 us	10 us
15	8905442 us	-	680001 us	321 us

5. NEXT MILESTONES

(1) Analysis of time for execution of BFS Algorithm.
(2) Analysis of time for execution of DFS Algorithm.
(3) Analysis of time for execution of A* Algorithm.
(4) Parallel implementation of hashing algorithms proposed.
(5) Analysis of time for execution of IDA* Algorithm.
(6) Final Implementation of all algorithms
(7) Comparison of the performance all the implementations on varying value of N.
(8) Comparison for different complexity of N-Puzzle start state.

6. REFERENCES

[1] https://github.com/siddharths2710/cuda_bfs/blob/master/cuda_bfs/kernel.cu
[2] https://www.nvidia.co.uk/content/cudazone/CUDABrowser/downloads/Accelerate_Large_Graph_Algorithms/HiPC.pdf
[3] https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/download/9620/9366