A* Algorithm for Dynamic Graphs on GPU

A Project Report

submitted by

LOKESH KOSHALE

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2020

THESIS CERTIFICATE

This is to certify that the thesis titled **A* Algorithm for Dynamic Graphs**, submitted by **CS15B049**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Rupesh Nasre
Research Guide
Professor
Dept. of CSE
IIT Madras, 600036

Place: Chennai

Date: 19th March 2020

ACKNOWLEDGEMENTS

First and foremost, I want to extend my sincere gratitude towards my advisor and guide prof. Rupesh Nasre. This project wouldn't have been possible without his invaluable direction. From the beginning, he has given me the freedom to choose the topic, work independently, pick the applications, and even to make a research poster for the ISC-HPC conference. He has shown immense trust in my limited ability. He is always calm and soothing and has always corrected me for the repetitive mistakes, I made. I have learned a great deal not only about academics but also about life. And in the most difficult time during the project he gave me with the moral support I needed, I can't thank him enough for that.

My special word of thanks should also go to prof Krishna Nandivada V, who has reviewed my mid-project report and has provided invaluable inputs for the project. I also want to thank him for providing guidance and support during the pace lab presentation. I would always remember my fellow peers and lab mates for the fun-time we spent together and for the discussions. I would also like to thank my friends, V. Pradeep Naidu and Gaurav Arora who have supported me throughout the project.

Without a family an individual is incomplete, I feel a deep sense of gratitude towards my parents whose infallible love and support is my greatest strength. Words can not express how grateful I am for all the sacrifices they have made for me. They will always inspire me and drive me towards a good human being.

Dozens of people has helped and taught me immensely in IITM, I want to thank them all.

ABSTRACT

A* is one of the widely used shortest path approximation algorithms. It is applied to a diverse set of problems due to its performance and accuracy. In several real-life applications such as wireless networks and robotics, the underlying graph changes continuously with time. D*, Focused D*, and D* Lite are search algorithms which solve path planning problems with free space assumption. In these, the cost of an edge can change while the optimal path is being explored. In this work, we propose A* algorithm on GP-GPUs for the graphs that are subjected to an update operation, where an update operation refers to insertion or deletion of an edge. Instead of re-executing the A* algorithm from the start after each update, our algorithm processes only the affected subgraph. The updates in the graph are batched and sent to GPU to process in parallel. Compared to the repeated A* algorithm, our algorithm achieves considerably better performance. We have observed an increase in speedup, which is proportional to the number of updates in the graph. For the temporal graphs in the SNAP dataset, our dynamic A* achieves 25–40× speedup over the static A* search. A* has various applications, one of them is an energy-efficient routing algorithm for wireless sensor networks, where the graph is inherently dynamic in nature. In energy-efficient routing protocol (EERP) our algorithm achieves 24–55× performance improvement over static A*. We have also extended A* maze solver algorithm to use the proposed algorithm for dynamic mazes which achieves 10–14× speedup. With the growing use of Automated Guided Vehicles (AGV) in supply chain environments, routing has become one of the major problems wherein A* is used for shortest time routing for AGV. We integrated our proposed algorithm for finding the K-shortest paths, and achieved over 10× performance improvement, and the speedup is directly proportional to K, the number of paths to be retrieved.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BFS	Breadth First Search algorithm
DFS	Depth First Search algorithm
GA*	GPU A* search algorithm (32)
A*	A* search algorithm (12; 35)
WSN	Wireless Sensor Networks
AGV	Automated Guided Vehicle
DG_2A*	Dynamic Graph's GPU A* algorithm
DGA*	Dynamic Graph's GPU A* algorithm
LPA*	Lifelong Planning A* algorithm
PQ	Priority queues

CHAPTER 1

Introduction

Graphs are used to model and simulate a wide variety of relations and processes in physical, biological, and information systems. In graph theory, one of the well-studied problems is the shortest path problem. The shortest path problem is to find the shortest or optimal path between the vertices of the graph. The shortest path problem is also referred to as *Single Pair Shortest Path* problem and is classified in:

- *Single Source Shortest Path*: which is to find the shortest paths from *source* vertex to all other vertices.
- *Single Destination Shortest Path*: which is to find the shortest paths from all vertices to a single *destination* vertex in the directed graph.
- *All Pair Shortest Path*: which is to find the shortest path between every pair of vertices.

Many algorithms are used to solve the above-described problems most significant ones are **Dijkstra's algorithm** (5) which solves single-source shortest path problem with non-negative weights, **Bellman-Ford algorithm** (3) (10) which also solves the single-source shortest path problem and also includes negative weights. **Floyd-Warshall algorithm** (9) which solves all pair shortest path problems and **A* algorithm** (12) which solves single-pair shortest path problems using heuristics to speed up the search.

Why the shortest path has such importance in Graph theory is because it reveals some important properties and relations of the model. Like a Graph representing all possible chemical reactions, the shortest path might be the one which has the least energy associated with it or in a Network Graph, and it might be the best routing path available.

In the following sections, we will discuss in detail the A* algorithm which is one of the widely used algorithms to solve *single-pair shortest path* problem.

1.1 What is A* Algorithm?

A* Algorithm is a heuristic-based search algorithm where we find the optimal path (the path with the lowest cost) from the source node to destination node, the algorithm guarantees to find the optimal path if the heuristic values are not overestimated for the graph. A* solves the pathfinding problem for the graphs with non-negative weights, and negative weights will cause incorrect results if cycles are present in the graph.

Because of its optimality, completeness, and effectiveness A* search algorithm is regarded as a standard for search algorithms. Many state-of-the-art path planning algorithms, i.e., D* lite (15), LPA* (16) are variants of the A* search algorithm. A* algorithm also as an optimization algorithm in many problems can be represented as graphs, i.e, DEE with A* (19), COStar (22).

1.2 Dynamic Graphs

Graphs are continuously changing in many applications of the graph algorithms, including Routing and Communication, Analysis and Simulation of complex systems, and Path planning. Dynamic Graphs are the graphs which are subjected to discrete changes such as insertion and deletion of edges. In this section, we give an overview of the field.

Dynamic Graphs are subjected to a series of *updates* where an *update* operation is insertion or deletion of an edge. Dynamic Graph-based problems can be classified in the two categories (14):

1. Partially Dynamic Problems: Graphs that are subjected to update operations which are either Insertion of edges or Deletion of edges but not both.
2. Fully Dynamic Problems: Graphs that are subjected to an intermixed sequence of updates where an update can be Insertion or Deletion of an edge.

A typical Dynamic graph problem supports query operations about properties of the graphs such as, weather two nodes are connected or not or what is the shortest path between given two nodes.

1.3 Why on GPU?

Scientific discovery and business analytics drive an insatiable demand for more computing resources. Graphs are inseparable part of many state-of-the-art analytic and modeling algorithms. Many applications such as weather forecasting, computational fluid dynamics simulations, and path planning for unmanned rovers utilizes graph algortithms and it requires an order of magnitude more computational power than is currently available, with more complex algorithms that need more compute power to run.

NVIDIA graphics processing units (GPUs) were originally designed for running games and graphics workloads that were highly parallel in nature. Because of high demand for FLOPS and memory bandwidth in the gaming and graphics industry, GPUs evolved into a highly parallel, multithreaded, manycore processor with enormous computational horsepower and high memory bandwidth. This started the era of GPGPU: general purpose computing on GPUs that were originally designed to accelerate only specific workloads like gaming and graphics.

GPU-accelerated applications offload the time-consuming routines and functions (also called hotspots) to run on GPUs and take advantage of massive parallelism. The rest of the application still runs on the CPU. This is also called a hybrid computing model.

With the rise in hybrid computing, both processors co-existed but they were still fundamentally different. Figure 1.1 shows the fundamental difference between CPUs and GPUs.

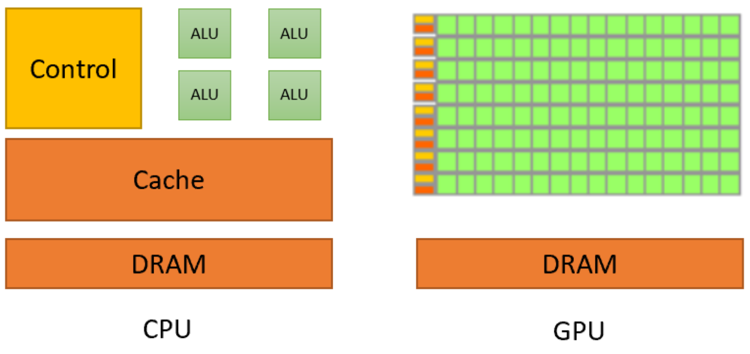


Figure 1.1: Difference between CPU and GPU

A* algorithm is used as an optimization algorithm in many of its applications which belongs to domain of scientific modeling and simulations. These large applications requires large graphs which has nodes and edges in order of millions. One such example is using A* algorithm along with DEE (Dead End Elimination) algorithm in structural designing of novel proteins, the graph grows exponentially with the length of the amino acids and its side chains. Thus we have proposed parallel algorithms on GPU so that A* algorithm can efficiently run on large dynamic graphs.

1.4 Our Contributions

A* algorithm 2.1 is a crucial ingredient of several AI and non-AI methods. Due to the scale of these networks, A* has been effectively parallelized on multi-core CPUs, may-core GPUs 2.2, as well as on distributed systems. Most of these applications in real-life undergo dynamic updates (1.2). In such situations, it is better to compute the new cost only for the part of the network that has changed. Our key contributions towards formulating A* algorithm for dynamic graphs are:

1. To find the new optimal path after the graph is subjected to a series of updates, we have proposed algorithms that propagate the new cost value to the affected nodes and finds the optimal path.
2. We have proven crucial properties of the dynamic computation, which allowed us to implement synchronization efficiently, improving the overall speedup of the A* algorithm.
3. For fully dynamic updates, we have proposed two different methods to find the new optimal path and also have shown the advantages and disadvantages of each.
4. We have also presented different case studies of applying the proposed algorithm in the existing application such as wireless sensor networks, game trees, pathfinding, and k-shortest path problem.
5. We have also developed a framework and library freely available at GitHub[1] which can be extended and applied to any application which utilizes the A* algorithm.

[1]url: `https://github.com/lkoshale/DA_STAR`

1.5 Organization of this thesis

This document is divided into seven chapters. Chapter 1 presents the problem of optimal pathfinding and discusses the related work from the literature. It also contains the necessary background details required for further chapters, i.e. A* algorithm and its variants and Dynamic Graphs. Chapter 3 introduces the partially dynamic problem of pathfinding in the graph, which is only subjected to insertion of edges. It presents the critical property related to the insertion of edges and the proposed algorithm for finding the new optimal path. In Chapter 4 the pathfinding with deletion of edges from the graph has been discussed. It also presents vital property related to the deletion of edges and builds the propagation algorithm upon them and states out the differences between the processing of insertion and deletion of edges to find the new optimal path. Chapter 5 introduces the fully dynamic setting where insertion and deletion of edges can be at the same time in any manner. It presents two different algorithms to find the new optimal path when the graph is subjected to fully dynamic updates and shows the performance comparison between them. In chapter 7 the applications of A* algorithm in a dynamic setting and how to integrate the proposed algorithm have been discussed. The applications include energy-efficient routing in wireless sensor networks, solving multi-path dynamic mazes, and shortest time routing for an automated guided vehicle using the k-shortest path algorithm. It also contains the analysis of the performance for each of the applications. When a graph is subjected to a set of updates, it might be the case that heuristic which is used to guide the A* search has also been modified. Chapter 8 gives a brief introduction to this problem and presents the possible methods that can be used to solve it. Chapter 9 concludes all the discussion and presents some key findings with the possible future work.

CHAPTER 2

Background and Related Work

In this chapter we will cover A* algorithm and its sequential and parallel implementations in detail. We will also discuss about D* family of algorithms which are heavily used in robotics and motion planning. Also we will discuss briefly about how to store dynamic graphs on GPU.

2.1 A* Algorithm

A* Algorithm is a heuristic-based search algorithm where we find the optimal path (the path with the lowest cost) from the source node to destination node, the algorithm guarantees to find the optimal path if the heuristic values are not overestimated for the graph. A* solves the pathfinding problem for the graphs with non-negative weights, and negative weights will cause incorrect results if cycles are present in the graph. Therefore we assume that *weight(e)* $> 0 \; \forall \; e \in \mathbb{E}$.

Each node $x \in \mathbb{N}$ has:

- $h(x)$: The heuristic value of node x, the approximate cost of reaching destination node from node x
- $g(x)$: The cost of reaching node x from the source node.
- $f(x)$: $g(x) + h(x)$, it is the approximate cost of reaching destination node from source node via node x.

2.1.1 Sequential A* Algorithm

The pseudocode 2.1 shows A* algorithm. The algorithm maintains an open_list for the nodes it has to expand further. The open_list is implemented as a min priority queue for nodes where priority is based on $f(nodes)$. To find the min cost path from source to destination, given $h(x) \; \forall x \in \mathbb{N}$, $f(source)$ is computed first, $f(source) = g(source) + h(source)$, as the cost of reaching the *source* node itself is 0, $g(source) = 0$ and $f(source) = h(source)$ and *source* is then inserted in the open_list. Till open_list is not empty, the *node* with minimum f value is extracted from open_list. If the extracted *node* is *destination*, *source* to *destination* path has been found and it is returned. Otherwise for each children of the *node*, $f(child)_{node \rightarrow child}$ is computed which is approximate cost of reaching *source* to *destination* via the edge $node \rightarrow child$, if $f(child)_{node \rightarrow child}$ is less than $f(child)$ and $child \notin$ open_list, *child* is inserted into the open_list and $f(child)$ is set to $f(child)_{node \rightarrow child}$. If $f(child)_{node \rightarrow child}$ is less than $f(child)$ and $child \in$ open_list then *child* is removed from the open_list and $f(child)$ is set to $f(child)_{node \rightarrow child}$ and it is inserted into open_list. If all nodes have been exhausted and the optimal path has not been found, then failure is returned.

Algorithm 2.1: A* Algorithm

```
   Input: Graph, source, destination
   Output: Least cost path from source to destination
 1 begin
 2     $open\_list = \{\}$
 3     $g(source) = 0$
 4     $f(source) = g(source) + h(source)$
 5     $insert(open\_list, source)$
 6     while not_empty(open_list) do
 7         node = $extract\_min(open\_list)$
 8         if node == destination then
 9             return path
10         end
11         for each child of node do
12             $g(child) = g(node) + weight(node, child)$
13             cost = $g(child) + h(child)$
14             if $child \notin open\_list$ and $f(child) > cost$ then
15                 $f(child) = cost$
16                 $insert(open\_list, child)$
17             end
18             if $child \in open\_list$ and $f(child) > cost$ then
19                 $remove(open\_list, child)$
20                 $f(child) = cost$
21                 $insert(open\_list, child)$
22             end
23         end
24     end
25     return failure
26 end
```

2.1.2 Parallel A* Algorithm for GPU

GPUs are designed for massive parallelism and are based on the SIMT (Single Instruction Multiple Thread) execution model, which is an extension of the SIMD(Single Instruction Multiple Data) execution model. They have been widely used to accelerate numerous applications. In the paper "Massively Parallel A* Search on a GPU", Zhou and Zheng (32) has proposed A* algorithm for GPU. The bottleneck for A* algorithm 2.1 is the sequential nature of the priority queue which is used to implement the open_list, the authors have proposed to use multiple priority queues in GPU A* algorithm 2.2 so that multiple threads can extract and expand multiple nodes simultaneously in the A* Algorithm.

The pseudocode 2.2 shows GA*(GPU A*) Algorithm. The Algorithm maintains a set of K priority queues, initially all the priority queues are initialized and cost of *source* node is computed then *source* node is inserted in one of the priority queues. Then the algorithm loops till all of the priority queues are empty or we have found the path from source to destination. In parallel K minimum cost *nodes* are extracted from each of the priority queues and if the *node* is the destination and all the nodes in the K-priority queues have cost higher than *destination* then the path is returned. Otherwise for each *child* for all the extracted nodes $f(child)_{node \to child}$ is computed and if $f(child)_{node \to child}$ is less than $f(child)$ then $f(child)$ is set to $f(child)_{node \to child}$. Also if $child \notin$ any of K-priority queues then it is inserted in the insert_list. Since many threads would try to access $f(child$ thus there is need of synchronization, it is implemented by locking the *child* before updating $f(child)$ and then unlocking it afterwards. There can be *child* nodes who are already present in any of the K-priority queues thus updating f of those nodes will result in incorrect heap structure thus heap structure of each priority queues is checked after modifying f. As a *child* node can have multiple parents and thus might get inserted multiple times in insert_list, to make sure that we are inserting only single copy of such *child* nodes duplicate *child* nodes are dropped from insert_list. Then all nodes $\in$ insert_list are added to K-priority queues. In the paper (32) authors have used parallel hashing methods to obtain this, we have implemented this by having a flag array for all nodes and marking those nodes which need to be added next in priority queues and so even if multiple nodes have marked the same *child* node it is added just once in the K-priority queue.

The other noticeable difference than using multiple priority queues compared to the sequential A* algorithm is the end condition when the optimal path is returned. In sequential A* algorithm, the optimal path is returned when the extracted node is the destination node. In contrast, in parallel A* algorithm, there is an additional check for making sure that there is no node in any of the parallel priority queues that has a lower cost than that of the *destination* node. This is because while expanding many nodes in parallel it is possible to reach the destination with the path that is not the optimal path.

By increasing the number of parallel priority queues the degree of parallelism can be increased and this results in better usage of the GPU's computational power. However, there is a limit as the degree of parallelism cannot be increased too much as more number of priority queue will lead to extracting multiple states in parallel thus there is an overhead on the number of expanded states. Figure 2.1 shows how the execution time varies with varying number of priority queues. As we increase the number of priority queues from one we see an exponential decrease in run time, it is because more and more cores of the of the GPU can be utilized. As we increase more, we reach a plateau because even if we increase the number of priority queues, there is a limit on how many states can be processed in parallel imposed from the structure of the graph.

2.1.3 Property of A* search

One of the vital property of sequential A* search is that if it is known that a node is visited, then we can surely say that any node which has a lower cost than current node has already been visited.

Lemma 2.1.1. *If we have found a node d which has cost f_d then all the nodes $i \in V$ which have cost $f_i < f_d$ are already visited(expanded).*

Proof. Suppose we found the node d with cost f_d and there exists a node n such

Algorithm 2.2: GA*: GPU A* Algorithm

```
Input: Graph, $source$, $destination$, $K$
Output: Least cost path from source to destination
begin
    $g(source) = 0$
    $f(source) = g(source) + h(source)$
    for $i \leftarrow 0$ to $K$ do
        $open\_list_k = \{\}$
    end
    $insert(open\_list_0, source)$
    insert_list = {}
    while $open\_list_i$ is not empty,$i \leftarrow 0$ to $K$ do
        for $k \leftarrow 0$ to $K$ in parallel do
            node$_k$ = $extract\_min(open\_list_k)$
            if node$_k$ is destination and $f(destination) < f(x)$
              $\forall x \in open\_list$ then
                return path
            end
            for each child of node$_k$ do
                lock(child)
                $g(child) = g($node$_k) + weight($node$_k, child)$
                cost = $g(child) + h(child)$
                if cost $< f(child)$ then
                    $f(child) = cost$
                    if $child \notin open\_list_i$, ,$i \leftarrow 0$ to $K$ then
                        $insert($insert_list$, child)$
                    end
                end
                unlock(child)
            end
        end
        for $k \leftarrow 0$ to $K$ in parallel do
            /* heapify all the nodes which do not satisfy the priority
               queue order                                              */
            maintain_priority_queue($open\_list_k$)
        end
        remove_duplicate_nodes(insert_list)
        for $l \leftarrow 0$ to $K$ in parallel do
            $i \leftarrow l$
            while $i <$ size(insert_list) do
                $node$ = insert_list[i]
                $insert(open\_list_k, node)$
                $i = i + K$;
            end
        end
    end
    return failure
end
```

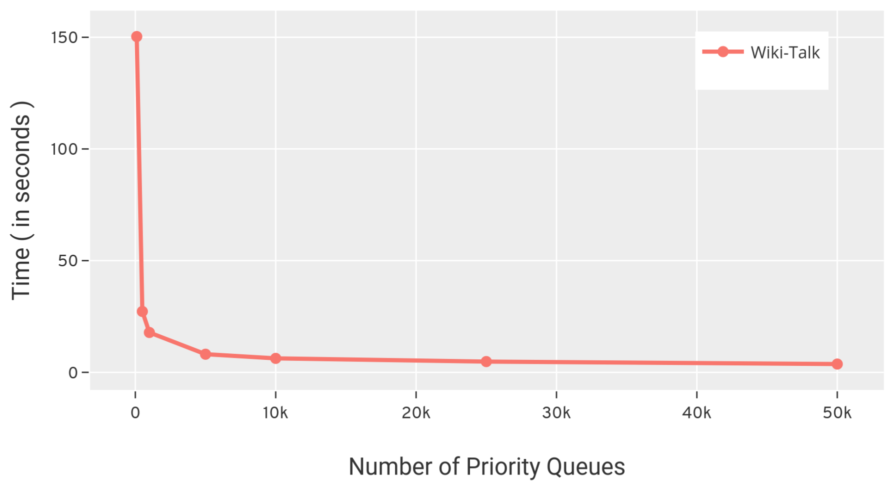


Figure 2.1: Execution Time vs Number of Priority Queue for GPU A* algorithm

that node n is not visited, i.e., $n \notin closed\_list$ and $f_n < f_d$.

If $n \notin closed\_list$ then either node $n \in open\_list$ or n is not explored yet which means $f_n = \infty$.

> If $n \in open\_list$ and we know that $f_n < f_d$, in A* to expand a node we always chose the node with least f (as in line 7 of A* Algorithm 2.1). So n should be chosen before d and which implies that n is already visited(expanded). This contradicts our assumption that n is not visited.
>
> If $f_n = \infty$, we have found the node d so $f_d \neq \infty$ which implies $f_n > f_d$ which contradicts our assumption that $f_n < f_d$.

Hence proved. □

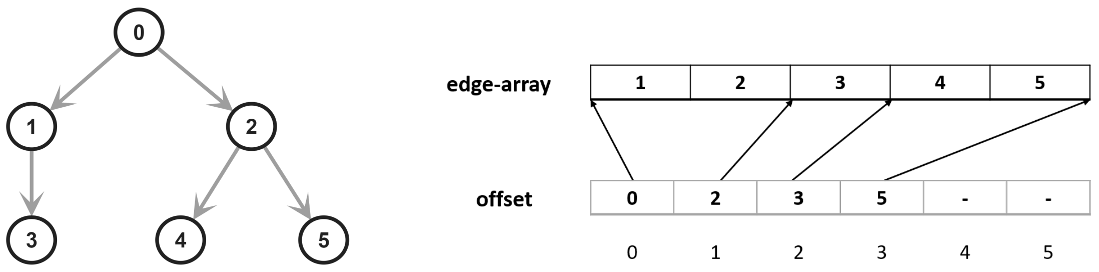
Corollary 2.1.1.1. *If some nodes are not visited at the end of A*, then those nodes will have cost greater than or equal to the cost of destination.*

In parallel A* algorithm we cannot guarantee that **Lemma** 2.1.1 always holds for all nodes of the graph, as many nodes are expanded in parallel, a node with the higher cost can be expanded before some other node with the lower cost which has not been reached by other thread. So to make sure that **Lemma** 2.1.1 always holds for the *destination* node additional condition has been added as when to terminate the parallel A* algorithm 2.2 [line 12-13]. So corollary 2.1.1.1 always holds for parallel A* algorithm.

2.2 Graph Representation

In the GPU's graphs are generally represented in **CSR** (Compressed Sparse Row) (31) format, as it is more space-efficient compared to the matrix representation and is also better suited for GPU than the adjacency list. An adjacency list will require more number of memory copies from CPU to GPU for the same graph compared to the CSR format.

In **CSR** (31) format the edges $\in \mathbb{G}$, are stored in a single *edge-array*, and an additional *offset array* is required which points to the first neighbour of the vertices. A sample Graph and its CSR representation is shown in the Figure 2.2b.



(a) Example Graph (b) CSR representation of Example graph

Figure 2.2: An example graph and it's CSR representation

To store the Dynamic Graphs we have used **diff-CSR** (23) proposed by Gaurav Malhotra and Rupesh Nasre et al. In diff-CSR the evolving graph is denoted by using a version number as sequence like $\mathbb{G}^0, \mathbb{G}^1, \mathbb{G}^2, \ldots$. $\mathbb{G}^0$ represents the initial graph in CSR format, while $\mathbb{G}^i, i \in \mathbb{N}$, represents the additional diff-CSR arrays which contains the inserted edges in the CSR format, which are not present in $\mathbb{G}^0..\mathbb{G}^{i-1}$ during ith update of the graph. An example dynamic graph with updates and it's CSR and diff-CSR representation is shown in Figure 2.3.

2.3 Related Work

A* (12) search algorithm was published by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute in 1969. It was developed for *Shakey* the robot so that it can plan its own actions. The concept of **admissibility** and **consistency** of *heuristic* functions was first introduced by Peter Hart with the

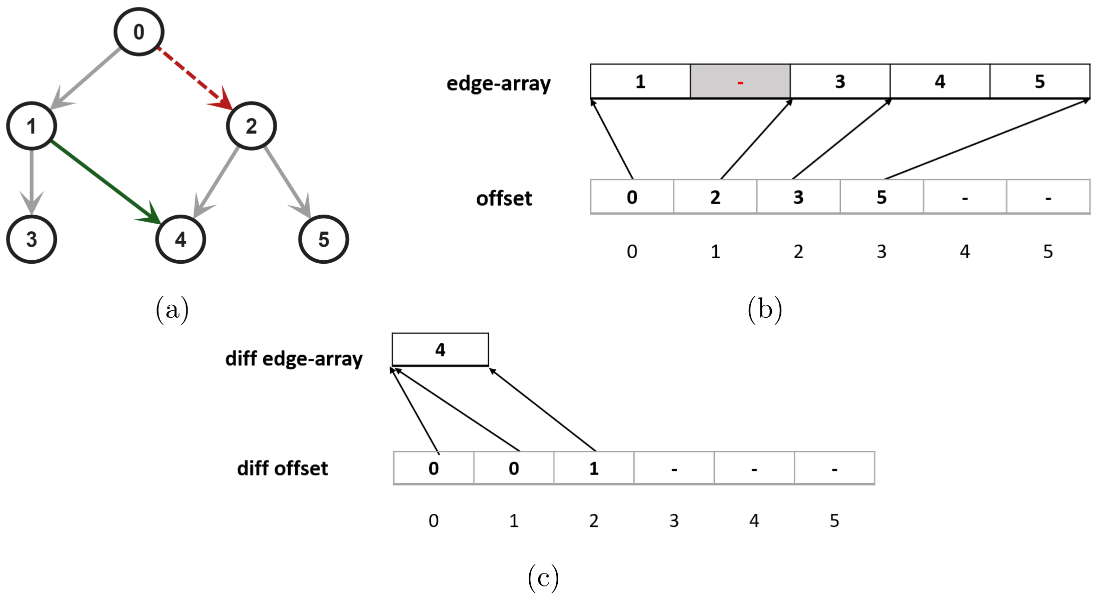


Figure 2.3: (a) Example Graph with deleted edge $0 \rightarrow 2$ and inserted edge $1 \rightarrow 4$ (b) CSR representation (c) diff CSR representation

A* algorithm. A* algorithm achieves better performance by using the previously known information (heuristics) to guide its search. It is different from greedy best first search algorithm as it also takes into account $g(x)$, actual cost of reaching node x. It can be seen as an extension of Dijkstra's algorithm. Figure 2.4 shows Dijkstra's algorithm and A* algorithm for forward search of goal($destination$) from start($source$).

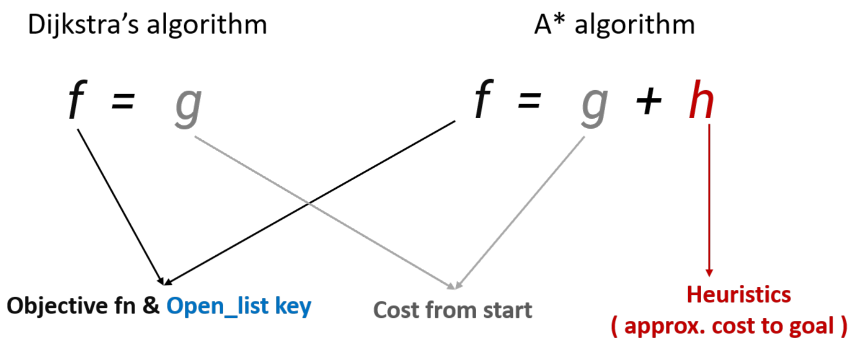


Figure 2.4: Forward Search from **start** to goal

GA* (32) (GPU A*) algorithm is a parallel implementation of A* algorithm on GPU by Zhou and Zeng. The authors have proposed to compute A* search with multiple priority queues to exploit the computation power of the GPU. They have also shown that GPU A* outperforms parallel CPU A* algorithm and sequential A* algorithm.

A* algorithm is used for path planning in robotics but robots also require to move in an unknown terrain for such applications. D* (28) algorithm was proposed by Koenig S et al. In D* algorithm authors proposed to search backwards from sink to source for the graphs where edge weigthts can change.

Building from D* and A* algorithm A Stentz et el. proposed focussed D* (29) algorithm which adds heuristics in the D* search to improve its performance.

Later on A Stentz et al. also proposed D* Lite (15) which is current state-of-the-art algorithm for path planning in robotics. The authors proposed to combine LPA* (16) and focussed D* algorithm into D* Lite algorithm.

In the paper "Sparsification—a technique for speeding up dynamic graph algorithms provide data strutures" (6) by Italiano, G. F. et al. the authors have maintained the graph where edges are inserted and deleted, and keep track of the following properties: minimum spanning forests, graph connectivity, graph 2-edge connectivity, and bipartiteness in time $O(n^{\frac{1}{2}})$ per change.

2.3.1 D* Algorithm and Variants

The family of D* algorithms solves the problem of path planning with free-space assumption (17). When navigating in unknown terrain, the robot knows some part of the map and the goal coordinates in the map, and the robot makes an assumption about the unknown terrain. As it moves towards the goal, it gathers more information about the map and then re-plans the route. The D* family of algorithms have the following variants:

- **Original D*** (28) is an incremental search algorithm developed by Anthony Stentz to solve the problem of dynamic path planning, In the D* algorithm arc-cost(weights) of the graph can change during navigating through the terrain.
- In **focussed D*** (29) Anthony Stentz combined the original D* algorithm with A* algorithm, it uses heuristics for improved performance and to direct the search towards the goal coordinates.
- **D* Lite** (15) by Sven Koenig and Maxim Likhache is an extension of focussed D* algorithm it builds on the concepts from LPA* (16) and dynamic SWSF-FP (25).

Here we will discuss the original D* algorithm (28), shown in Algorithm 2.3. In the D* algorithm instead of planning from the start(or *source*) node as in Dijkstra's algorithm and A* algorithm, the author had proposed to do a backward search from the end(or *destination*). Backward search can also be implemented for Dijkstra's algorithm and A* algorithm, as shown in figure 2.5.

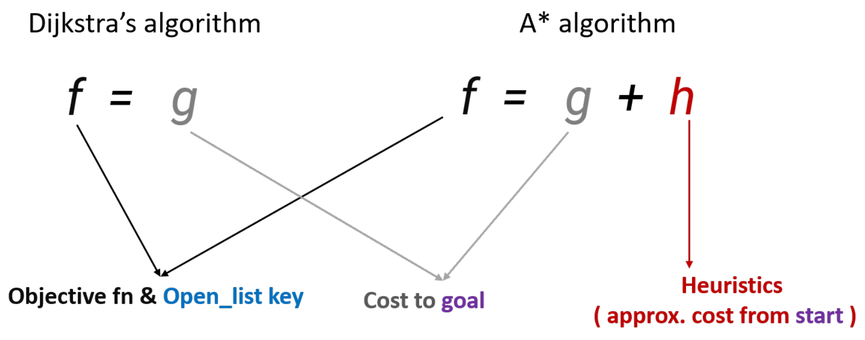


Figure 2.5: Backward Search from **goal** to start (4)

In Algorithm 2.3 $t(X)$ represents the tag of a state X which can be:

- NEW: node which is never placed in open_list.
- OPEN: nodes which are in open_list.
- CLOSED: nodes that are already explored.
- RAISE: nodes whose cost has become higher than before.
- LOWER: nodes whose cost has become lower than before.

$h(X)$ is the path cost and $k(X)$ is the smallest value of $h(X)$ in P(priority queue or open_list). For nodes X which now is in LOWER state, propagate the information about path cost reduction to its neighbours and insert the neighbour if it is in NEW state or it's new cost from node X is lower than the earlier computed cost of the neighbour.

For nodes X, which are at RAISE state the information about the increase in path cost is propagated to its neighbours. If the neighbour has state NEW or the optimal_parent of the neighbour is the node X, and the cost of the neighbour has changed, insert the neighbour in open_list. Else if it is not the optimal_parent and cost of neighbour is lower from X then insert X in the open_list. Else if the optimal_parent of the neighbour is not X and the cost of X is lower if reached from the neighbour than insert the neighbour in open_list.

Algorithm 2.3: The original D* algorithm (28) (4)

```
Input: List of all states $L$
Output: Least cost path from source to destination
1  for each $X \in L$ do
2      $t(X) = \texttt{NEW}$
3  end
4  $h(G) = 0$
5  INSERT$(O, G, h(G))$
6  $X_c = S$
7  $P$ =INIT_PLAN$(O, L, X_c, G)$
8  if $P = NULL$ then
9      return NULL
10 end
11 while $X_c \neq G$ do
12     PREPARE_REPAIR$(O, L, X_c)$
13     P = REPAIR_REPLAN$(O, L, X_c, G)$
14     if $P = NULL$ then
15         return NULL
16     end
17     $X_c$ = the second element of $P$ {Move to next element in $P$}
18 end
19 return $X_c$
```

Focussed D* (29) algorithm adds heuristics in the incremental search algorithm D* to guide its search towards the *destination* as in A* algorithm while retaining property of the arc-cost changes in the graph while navigating. Figure 2.6 shows the additional information in focussed D* algorithm with respect to A* algorithm. The name D* algorithm comes from its similarity with A* algorithm and it is also

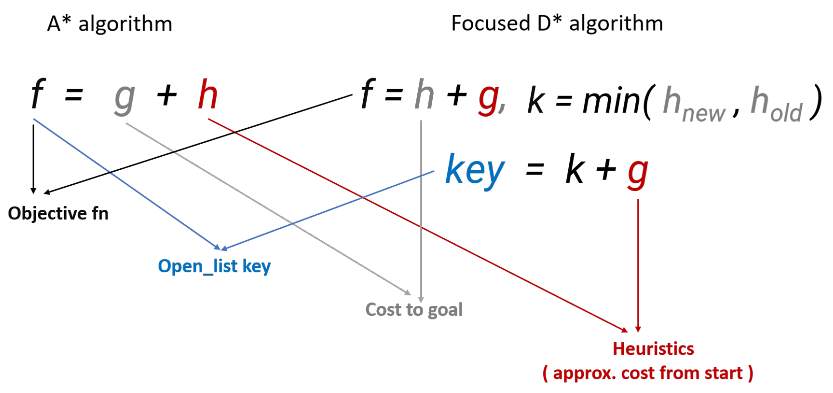


Figure 2.6: **Dynamic** Backward Search from **goal** to start (4)

known as dynamic A*. The D* family of algorithms only handles the change in the arc-cost while in most of the cases there can be addition and deletion of edges which is currently not handled by D* algorithm.

Current systems mostly use D* Lite (15) algorithm as it is faster and memory efficient than its counterparts. The D* algorithm is further extended to Field D* (8) and Theta* (24) algorithms for motion planning. D* family of algorithms have been tested on Mars rovers Opportunity and Spirit and is the winner of the DARPA Urban Challenge for the navigation system.

CHAPTER 3

Insertion of Edges

In this chapter, we will discuss a partially dynamic problem where updates consist of only the insertion of new edges in the graph. Consider a Railway network of a country or state, where new and faster rail lines are continuously been added between cities. As the new rail lines are available, the shortest route between any two cities will change accordingly.

At first we discuss how the insertion of a single edge can affect the optimal path between given two nodes. In the later section we will consider a set of updates that occur at the same time.

3.1 Single Edge Insertion

In the incremental setting, there are only edge-insertions, i.e., edge $u \rightarrow v$ is added in G where $u \in V$ and $v \in V$.

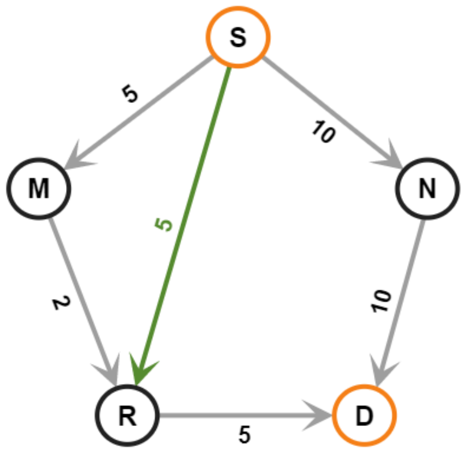


Figure 3.1: Insertion of an edge

In the graph in Figure 3.1 the optimal path for S to D before insertion was $S \rightarrow M \rightarrow R \rightarrow D$, when we insert edge $S \rightarrow R$ of cost 5, the cost f of node R changes and the optimal path changes to $S \rightarrow R \rightarrow D$.

Lemma 3.1.1. *Insertion of an edge can not increase the cost of the source to destination optimal path.*

Proof. Suppose we add and an edge $u \to v$ with weight w_{uv} where $u \in V$ and $v \in V$. The cost of reaching v from source using edge $u \to v$ is $g(v)_{new} = g(u)+w_{uv}$. There can be two cases:

> *Case 1:* If $g(v)_{new} < g(v)$, if optimal path consist of node v then the decrease of cost of v will decrease the cost of optimal path, as if there is path from v to destination then source $\to v \to$ destination path's cost decreases due to decrease in optimal cost of v. Thus it becomes the new optimal path with lesser cost. If it doesn't then the optimal path remains the same.
>
> *Case 2:* If $g(v)_{new} \geq g(v)$, suppose node v is our destination, from definition optimal path is a path with least cost associated with it, so the optimal cost to reach v is $g(v)$, so the addition of this edge does not affect the cost of any node of the graph which implies the cost of optimal path remains same.

In both cases, cost of the optimal path doesn't increase, hence proved □

Lemma 3.1.2. *If we add an edge $u \to v$ and $f(u) > f(destination)$ then addition of this edge will not affect the optimal path.*

Proof. Given $f(destination) < f(u)$, As weights are positive $w_{uv} > 0$. So cost of any such path from source $\to u \to v \to$ destination will be $g(u)+$ cost of reaching destination from u but as $g(u) > g(destination)$ the cost of such paths will always be larger and hence it can't be the optimal path. So the addition of such edges doesn't affect the optimal path □

Corollary 3.1.2.1. *If we add an edge $u \to v$ and $f(u) = \infty$ then addition of this edge will not affect the optimal path.*

Proof. If $f(u) = \infty$ then $f(u) > f(destination)$. So it follows from Lemma 3.1.2 that addition of such edges doesn't affect the optimal path. □

We know from Lemma 3.1.1 that insertions can't increase the cost of the optimal path but they can decrease the cost, and there can be a new optimal path from the inserted edge. Also, if we insert $u \to v$ and $g(v)$ changes then all its descendants in the graph whose g are computed based on v become stale values. There can be two approaches to update the cost of the graph. Due to this insertions either *propagate* the new value as insertion took place or compute the new value on-demand when necessary i.e *lazy update*. We prefer the first one as it simplifies a lot of computation.

3.2 Batch Insertion

Instead of adding each edge one by one, we process a group of edges inserted at a particular instance of time to utilize the computation power of the GPU. First, we make an update_list of vertices from the inserts as if $u \to v$ is inserted and $f(v)$ decreases then we add v to update_list. Then until update_list is empty we expand each child of nodes in update_list and if the new cost of a child is lesser than the old cost, we add the child in update_list.

At the end of propagation if there is a change in optimal cost then it would also be propagated to the destination node we prove the same below.

Lemma 3.2.1. *At the end of propagation, all the nodes $\in V$ have the latest cost f which is the optimal cost in the new graph including inserts.*

Proof. Suppose we insert edge $u \to v$. Lets take a node $n \in$ Graph.

> *Case 1:* If $n \in$ sub-graph(v), since $v \in$ update list and at each iteration we add nodes which belong to sub-graph of v and whose $f(n)$ is decreased. According to Lemma 3.1.1 Insertion can only decrease the cost, so if $n \in$ update list at some iteration then its $f(n)$ is decreased and that is the new optimal cost $f(n)$, if $n \notin$ update list then $f(n)_{new}$ due to insertion is greater than $f(n)_{old}$ in which case $f(n)_{old}$ is the optimal cost as per the definition of optimal cost.
>
> *Case 2:* If $n \notin$ sub-graph(v) which implies that there can be no change in the optimal_path from source to n (as there is no structural change in this part of graph), thus its cost remains unchanged and it is the optimal cost of node n in new graph.

□

Lemma 3.2.2. *At the end of propagation $f(destination)$ is the optimal cost of reaching destination from source in the new graph G(V, E+Inserts).*

Algorithm 3.1: Propagation of Insertions

```
Input: List of newly inserted edges, N, E
Output: Updated optimal path from source to destination
propagate_insertions(inserted_edges, N, E)
begin
    update_list = create_update_list(inserted_edges)
    while update_list is not empty do
        s = size(update_list)
        flag = array(N,false)
        expand(update_list, flag, s)
        update_list = generate_update_list(flag)
    end
end

expand(update_list, s, flag)
begin
    id = global_thread_id
    node = update_list[id]
    for each child of node do
        cost = $g(\texttt{node}) + weight(\texttt{node}, child) + h(child)$
        lock(child)
        if f(child) > cost then
            f(child) = cost
            optimal_parent(child) = node
            flag[child]= true
        end
        unlock(child)
    end
end
```

Proof. Follows from Lemma 3.2.1, *destination* $\in V$, so at the end of propagation $f(destination)$ is the optimal cost i.e. optimal cost of reaching *destination* from *source* in new graph including inserts i.e. G(V, E+Inserts) □

3.3 Summary

In this chapter, we discussed how single edge insertion can affect the optimal path and stated and proved some important property of the graph when it is subjected to insertion . Later we generalize the ideas to multiple insertions of edges and propose the parallel algorithm which finds the new optimal path for any number of updates. This sums up how we process the edges which are inserted in the graph. In the next chapter 4, we will discuss how the deletion of edges of the graph affects the optimal path.

CHAPTER 4

Deletion of Edges

In this chapter, we will discuss a partially dynamic problem where updates only consist of the deletion of edges of the graph. Continuing from the example of Railway track from the previous chapter. In a rail network the rail lines can break or need maintenance. In such time the rail line is inoperable and thus becomes non-commutable for the trains, such cases can be easily modeled as deletion of an edge in the network graph of the Railway. Due to this the shortest available rail distance between affected stations will change accordingly.

4.1 Single Edge Deletion

In *decremental setting* there are only removal of edges $u \rightarrow v$ where $u \in V$ and $v \in V$. Addition of edges can only reduce the Optimal Cost (Lemma 3.1.1), whereas deletion of edges can increase the Optimal Cost (Lemma 4.1.1) and can also make the graph disconnected, which poses a new problem as the new optimal path may contain many nodes which are not even explored yet. Further, if multiple deletions are happening at the same time one can read stale values for the path which doesn't even exist in the new graph. This extra set of problems requires some extra computation which makes deletion computationally costlier than the insertions. In the example graph in Figure 4.1 the optimal path from S to D before deletion was $S \rightarrow R \rightarrow D$. When we delete the edge $S \rightarrow R$, the optimal path changes to $S \rightarrow M \rightarrow R \rightarrow D$.
Below we prove some important properties that hold when we remove an edge from the graph. The following properties hold when we have the optimal path for the graph before deletions.

Lemma 4.1.1. *Deletion of an edge $u \rightarrow v$ where $u \in V$ and $v \in V$ can not decrease $f(v)$.*

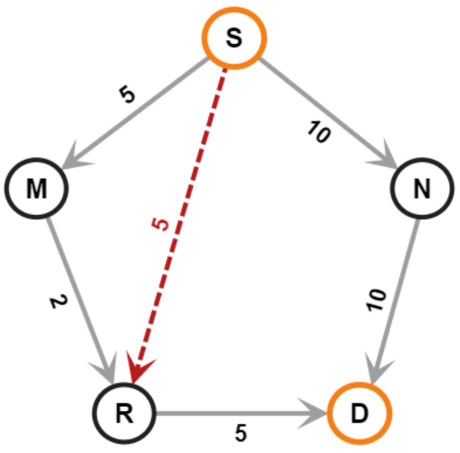


Figure 4.1: Deletion of an edge

Proof. Let edge $u \to v$ get deleted from graph then,

> *Case 1:* If $u =$ optimal_parent(v) then, source $\to u \to v$ was the optimal path of reaching v from source. So after deletion, $f(v)_{new} = g(p)+w_{sv}+h(v)$ where p is parent of v with least g. In A* we chose the path with least f as we chose u before p implies $f(v)_{new} \geq f(v)_{old}$.
>
> *Case 2:* If u is not the optimal_parent(v), then let p be the optimal_parent(v), deletion of edge $u \to v$ doesn't affect the $f(v)$ as the least cost of reaching v is from p.

From above we can say that deletion of edge $u \to v$ can only increase $f(v)$. Hence proved. □

Lemma 4.1.2. *If edge$(u, v) \notin$ optimal path, then deletion of such edges doesn't affect the optimal path.*

Proof. Suppose we delete edge$(u, v) \notin$ optimal path then, as from Lemma 4.1.1 deletion of such edges can only increase $f(v)$ so if there is a path from source $\to v \to$ destination then its cost will be increased, as optimal path is least-cost path such paths cannot be an optimal path, Hence deletion of such edges doesn't affect the optimal path. □

Lemma 4.1.3. *If edge(u,v) is deleted and optimal_parent(v)≠u, then deletion of such edges doesn't affect the optimal path.*

Proof. If u is not the optimal_parent(v), then let $s =$ optimal_parent(v), deletion of edge $u \to v$ doesn't affect the $f(v)$ as the least cost of reaching v is from s, as there is no change in cost of any nodes so optimal path remains same. □

4.2 Batch Deletion

Instead of deleting edges one by one we process deletion in batches, where a batch contains all the deleted edges before a query. From Lemma 4.1.1 we know that deletion of edges can increase the optimal cost, to compute the new optimal cost, we first propagate the change in cost due to deletion of edges to all affected nodes and then we perform a check that we are not violating the Lemma 2.1.1 after propagation, which is an essential property to always hold so that later we can process the next incoming updates if it violates Lemma 2.1.1 then we start A* from where we left before i.e using the open_list we have computed before. In the end, we will have the new optimal path with the optimal cost for new graph encompassing all the deletions.

Propagation for Deletion:

1. For each deleted edge $u \rightarrow v$, if $f(v) \neq \infty$ and optimal_parent(v)= u then for such edges we compute the cost of reaching v from its parent and choose the least one as optimal_parent(v) and update $f(v)$. If v doesn't have any parent then $f(v) = \infty$, and we add v to update_list.
2. For each child of nodes in update_list, we compute the cost f of the child with respect to the node and if new cost f is more than $f(child)$ and optimal_parent(child) is node then, we compute the cost of reaching child from all of its parents and update f to the least cost of the parents and set that parent as optimal_parent. And we add the child to the next update_list.
3. We repeat steps 1 and 2 until update_list is empty.

Optimal Cost is the least cost of reaching the node but we increase the cost of the child at step 2 in the above procedure. Why we do that is because from Lemma 4.1.1 we know that f might increase due to deletion of edges and we need to propagate the updated higher cost. Also since there is no special ordering of nodes to execute the propagation it may happen that while we are propagating for a node its sub-graph might already have been updated so to make sure we have latest cost value among those, we choose the least cost parent and then propagate that cost downwards if applicable.

In the example graph in Figure 4.2, the optimal path before deletion is $S \rightarrow M \rightarrow O \rightarrow P \rightarrow R \rightarrow D$. The edges $M \rightarrow O$ and $P \rightarrow R$ are deleted from graph.

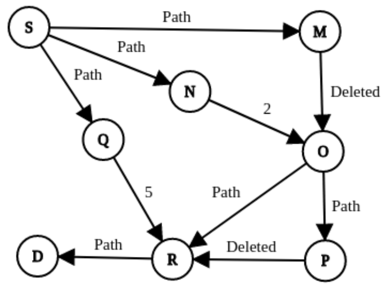


Figure 4.2: Batched deletion of multiple edges

So $R, O \in$ update_list. While propagating for O we might find R again from path $O \rightarrow R$. But R has already been updated, if in that update R has chosen path $S \rightarrow M \rightarrow O \rightarrow R$ as its optimal path then we have stale value for $f(R)$ as $M \rightarrow O$ is deleted but since R's update happened first it read the stale value before propagation. In such cases, we need to find the parent with the least cost. Thus we compute the cost of reaching node from all its parents and choose the parent with the least cost as optimal_parent.

4.3 Deletion and Cycles

We propagate the deletions so that each node whose cost is already computed will have the latest value. There is a special case of deletions with a cycle where even after propagation we can get stale value if we don't perform certain checks while propagating the latest value to nodes.

In the example graph in Figure 4.3 the optimal path is from $S \rightarrow ABCD \rightarrow E$,

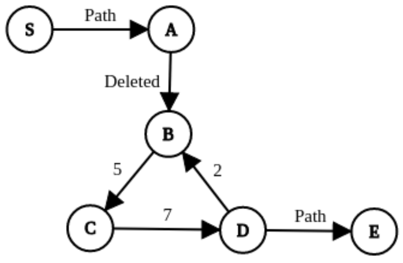


Figure 4.3: Deletion in Cyclic Graphs

where S is source and E is destination. When we remove the edge $A \rightarrow B$ and recompute the $f(B)$ there exist only one parent of B which is node D. So the $f(B)_{new}$ is computed as $f(B)_{new} = g(D) + 2 + h(B)$. Which uses value of $g(D)$. As we remove $A \rightarrow B$ there is no path from $S \rightarrow D$. So $g(D)_{new}$ is ∞, but we never arrive at that proposition because in propagation of deletion we will propagate from B and $f(D)$ will be updated based on the stale value of $f(B)$. Thus we have wrong cost value after deletion of such edges. The main reason this happens is due to the fact that cost $f(D)$ is computed with respect to B as $B \in OptimalPath(S, D)$. To avoid such cases while choosing the optimal_parent we have to eliminate the parents whose optimal_ancestor is current node as shown in Algorithm 4.1.

Algorithm 4.1: Check Cycles

```
check_cycle(node, parent)
begin
    current_node = node
    while there exists optimal_parent(current_node) do
        if current_node == parent then
            return true
        end
        current_node = optimal_parent(current_node)
    end
    return false
end
```

4.4 Parallel Deletion and Cycle

When there are multiple deletions happening at the same time, there can be a cycle of optimal_parent. Thus when we do the above check using Algorithm 4.1, we get into an infinite loop. We have defined optimal_parent as a non-cyclic path to start node. So to eliminate such a cycle when we process deletion we have to do additional checks for removing such cycles.

In the example graph in Figure 4.4, the optimal path from $S \rightarrow D$ is $SABCD$. When we delete edges $A \rightarrow B$ and $G \rightarrow F$, for node B the new parent is I as $B \notin optimal\_parents(I)$. Also the new parent of F becomes E as $E \notin optimal\_parents(F)$. This is because we are checking for optimal_parent for both nodes in parallel thus both the check happen at the same time and both read the optimal_parent values before it got changed by either one of them. Thus forming a cycle of optimal_parents which violates our definition of optimal_parent. To

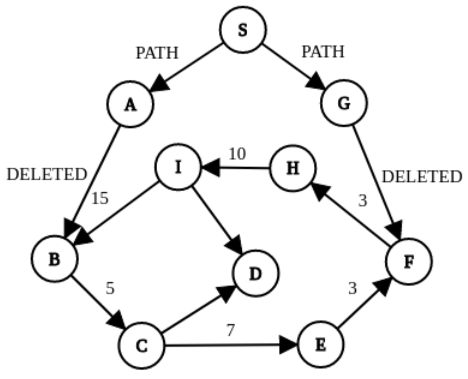


Figure 4.4: Parallel Deletion in Cyclic Graphs

solve this we perform additional cycle check after each iteration of propagation for deletion and if such cycle found we remove them.
Now the question arises to remove which newly formed edge of the cycle i.e. remove I =optimal_parent(B) or remove E = optimal_parent(F). which one belongs to the optimal path? the answer is we can just remove any of cycle edges and our propagation algorithm(next section) will take care of which edge actually belongs to the cycle.

4.5 Propagation for Deletions

Parallel propagation for deletion of edges that handles all the corner cases discussed in the above sections is shown in the algorithm 4.2. In Insertion we have proved lemma 3.2.2 which identifies that at the end of the propagation we will have the optimal_path. The same is not true for propagation for deletion, we get the optimal_path at the end of the propagation of deletion only if *destination* node satisfies the condition given in Lemma 4.5.1. We have also given a counterexample to prove the same in 4.5.2 and we prove the correctness of the propagation algorithm in Lemma 4.5.3.

Lemma 4.5.1. *At the end of propagation for deletion all node $n \in V$ such that $f(n)_{new} \neq \infty$ has the latest $f(n)$ which is optimal cost in the new graph including deletions.*

Proof. Suppose we delete edge $u \to v$. Let node $n \in V$:

Algorithm 4.2: Propagation of Deletions

```
Input: List of newly inserted edges, N, E
Output: Updated optimal path from source to destination
propagate_deletions(deleted_edges, N, E)
begin
    update_list = create_update_list(deleted_edges)
    check_optimal_parent_cycle(update_list)
    while update_list is not empty do
        s = size(update_list)
        flag = array(N,0)
        expand(update_list, flag, s)
        update_list = generate_update_list(flag)
        check_optimal_parent_cycle(update_list)
    end
end

expand(update_list, s, flag)
begin
    id = global_thread_id
    node = update_list[id]
    for each child of node do
        lock(child)
        $g(child) = g(\texttt{node}) + weight(\texttt{node}, child)$
        cost = $g(child) + h(child)$
        if f(child) > cost then
            $f(child)$ = cost
            optimal_parent(child) = node
            flag[child]= 1
        else if f(child) < cost and optimal_parent(child)== node then
            for each parent of child do
                if check_ cycle(child,parent) then
                    continue
                end
                if f(child) > g(parent)+weight(parent,child) +h(child)
                 then
                    f(child) = g(parent)+weight(parent,child) +h(child)
                    optimal_parent[child] = parent
                end
            end
            flag[child]= 1
        end
        unlock(child)
    end
end
```

Case 1: If $n \in$ sub-graph(v) then there can be two sub-cases:

1. If $v \in$ Optimal_Path(n) which implies $f(n) \neq \infty$, when we add v in up-

date_list we compute the least cost of reaching v from all of its remaining parents. We also make sure that the parent we choose is such that its cost is not computed with respect to v which implies its the optimal_cost of reaching v. Since $v \in$ Optimal_Path(n) , $\exists$ parent p of n such that $p \in$ Optimal_Path(n) and $p \in$ sub-graph(v), which implies $p \in$ update_list at some iteration of propagation, when p tries to update n, n will choose the best available parent. Thus cost of $f(n)$ is optimal_cost of reaching n in new graph.

2. If $v \notin$ Optimal_Path(n),then $\exists$ optimal_parent p of n such that $v \notin$ Optimal_Path(p), thus $n \notin$ update_list at any iteration of propagation so $f(n)$ remains same which is the optimal_cost from Lemma 4.1.2.

Case 2: If $n \notin$ sub-graph(v) then $v \notin$ Optimal_Path(n) also $v \notin$ update_list at any iteration of propagation so $f(n)$ remains same which is the optimal_cost from Lemma 4.1.2 □

Lemma 4.5.2. *At the end of propagation $f(destination)$ might not be the optimal cost of reaching destination from source in new graph.*

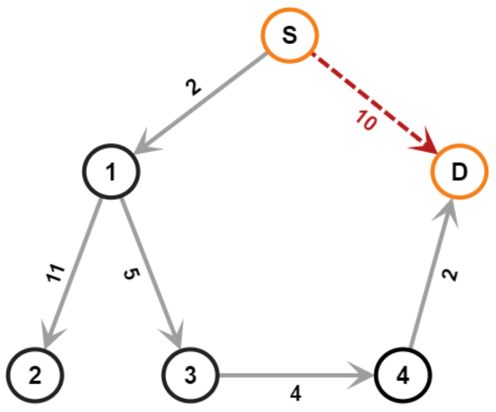


Figure 4.5: Example graph where edge $S \to D$ is deleted

Proof. To prove, it is sufficient to give an example where $f(destination)$ is not the optimal cost after propagation for deletion.
In the graph in Figure 4.5 at the end of A* we will have:
open_list : {2,3}
closed_list : {S,1,D}
Thus node 4 is not visited thus having $f(4) = \infty$, after propagation for deletion as D has only one parent 4 its cost is computed with respect to 4 so $f(D) = \infty$, but in the new graph the optimal cost is $f(D) = 13$ with Optimal_Path $S \to 1 \to 3 \to 4 \to D$.

Thus in the above graph, $f(D)$ after the propagation of deletion is not the optimal cost in the new graph. $\square$

Lemma 4.5.3. *After propagation of deletions if $f(destination) < f(n)\ \forall\ n \in open\_list$ then $f(destination)$ is the optimal cost of reaching destination from source.*

Proof. After propagation of deletion $\forall$ nodes $n \in$ open_list either $f(n)_{new}$ is the optimal_cost or $f(n)_{new} = \infty$, from Lemma 4.5.1. Also given $f(destination) < f(n)\ \forall\ n \in$open_list. As weights are non-negative so for all descendants of n $f(\text{descendants}) > f(n)$. So reaching *destination* from any node in update_list will have larger cost than current value, thus $f(destination)$ is the optimal cost of reaching destination from source. $\square$

From Lemma 4.5.2 we know that after propagation for deletion $f(destination)$ is not the optimal cost, also from Lemma 4.5.3 we know that if $f(destination) < f(n)\ \forall\ n \in$ open_list then $f(destination)$ is the optimal cost. To satisfy the condition we need to perform A* after propagation for deletion as after A* search $f(destination) < f(n)\ \forall\ n \in$open_list and thus $f(destination)$ is the optimal cost.

Algorithm 4.3: Procedure for Deletions

```
Deletions(deleted_edges, N, E)
begin
    propagate_deletions(deleted_edges, N, E)
    destination_cost = $f$(destination)
    min_cost = $\infty$
    for $k \leftarrow 0$ to $K$ do
        min_node = top($open\_list_k$)
        min_cost = min( min_cost, $f(min\_node)$ )
    end
    if destination_cost > min_cost then
        GPU A*(open_list, source, destination, K)
    end
end
```

4.6 Summary

In this chapter, we discussed how to process the deletion of single and multiple edges in the graph and find the new optimal path. We also proved some important properties related to the deletion of edges like when edges are deleted in the graph the cost of the nodes will only increase. Then we proposed a propagation method for the deletion of edges in the graph and stated the cases where it will fail, and updated the propagation to handle such cases also. We have shown the updated propagation method and discussed the completeness of the algorithm. We may need to perform additional A* after the propagation of deletion to find the new optimal path which is unlike the propagation of insertion discussed in the previous chapter 3 where we found the optimal path right after the propagation. But we don't start the A* again from the start, instead, we start from the last saved state form the open_list(priority queue) of A* algorithm. Till now we have discussed the insertion and deletion of edges separately in the next chapter 5 we will show how to find the optimal path when addition and deletion of edges happen at the same time.

CHAPTER 5

Fully Dynamic

Real-world graphs are dynamic and edges are changing continuously. So there are the insertions of edges and deletion of edges at the same instance. In such cases if a query is raised to find the optimal path from source to the destination we can retrieve the optimal path without re-executing A* from start again.

5.1 Separate Propagation

As we have already discussed how to find the optimal path when edges are either added only or deleted only we can infer the dynamic change in edges at a single instance as addition happening first then deletion or vice versa. So we process addition first as insertion only then we process deletion as deletion only. It may happen that some nodes will get updated multiple times in two different propagations. But as insertion and deletion require a different set of instructions it is a good choice to separate the propagation.

Algorithm 5.1: Separate Propagation

```
separate_propagation($inserted\_edges, deleted\_edges, N, E$)
begin
    propagate_insertions($inserted\_edges, N, E$)
    propagate_deletions($deleted\_edges, N, E$)
end
```

Lemma 5.1.1. *After separate propagation $f(destination)$ is the optimal cost of reaching destination from source in new graph including insertions and deletions.*

Proof. In separate propagation first, we propagate for insertions, and as from Lemma 3.2.2 at end of it we have $f(destination)$ as the optimal cost in the new graph which includes all inserted edges. In the new graph, we propagate for deletions and then perform GA*. So from Lemma 4.5.3 we have $f(destination)$

as the optimal cost in the graph which includes both insertion and deletion. Thus $f(destination)$ is the optimal cost of reaching the destination from the source in the new graph including insertion and deletions. □

5.2 Simultaneous Propagation

We have designed the propagation of insertions(Section 3.2) and propagation of deletions (Section 4.2) such as some parts of them can be overlapped. We can process both insertion and deletion at the same time by considering them as an update. But since both propagations are happening at the same time there are few additional cases we need to take care of thus adding some more checks as given below.

5.2.1 Insert-Delete Cycle

Since we have insertions and deletions propagating at the same time, If we don't perform enough checks we can get into cycles for certain cases.

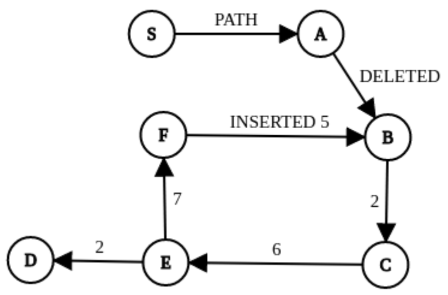


Figure 5.1: Cyclic graph with parallel insertion and deletion

In the graph in Figure 5.1 we remove $A \to B$ and insert $F \to B$. Due to deletion $f(B) = \infty$, when we propagate for edge $F \to B$, the cost of reaching B from F via newly added edge is less than infinity. Thus we compute the new cost and make optimal_parent(B) $= F$. But when we try to retrace the path from F to S we get $F \to E \to C \to B \to F$ thus we have a optimal_parent cycle. It is because the cost of F itself is a stale value due to the fact that its cost is computed with respect to B and deleted edge $A \to B$. To remove such cases we have to extensively check for insertion of edges $u \to v$, if $v \in$optimal_parents(u) for all insertions.

5.2.2 Propagation Cycle

Since Insertions and Deletions are propagating at the same if there is a cycle and each reaches the cycle at different iterations. It might happen due to the structure of the graph that they propagate indefinitely on such cycles if the check mentioned in the above section is not implemented for each insertion.

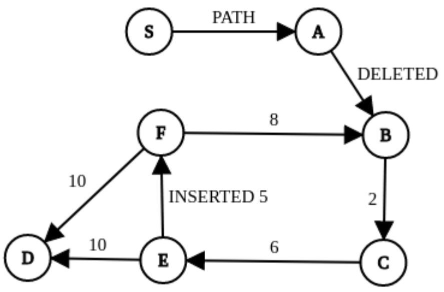


Figure 5.2: Infinite propagation due to formation of cycles

In the graph in Figure 5.2 $A \rightarrow B$ is deleted and $E \rightarrow F$ is added. Due to deletion cost of $B = \infty$ and due to insertion cost of F is changed in next iteration cost of $C = \infty$ as there is no path from $S \rightarrow C$ in new graph, also due to edge $f \rightarrow B$ the cost of B is changed as the new cost is less than ∞. In the next iteration cost of $E = \infty$ and cost of C is changed. Similarly in the next iteration cost of $F = \infty$, and cost of E is changed, and similarly the propagation goes on the loop indefinitely. The problem here is we computed the cost of B from F when F has the stale value and its cost is computed with respect to B itself. This can be removed by doing the check mentioned in the above Section 4.3.

5.3 Summary

In this chapter, we discussed how to process both the insertion and deletion of edges occurring at the same time. We proposed two different methods for it, first to split the update in insertion and deletion of edges and then propagate each of them separately i.e. separate propagation and consider both of them as a single update operation in the graph and propagate them simultaneously i.e.

Algorithm 5.2: Propagation of Updates

```
propagate_updates(inserted_ edges, delted_ edges, N, E)
begin
    update_list = create_update_list( inserted_edges )
    update_list = append( update_list,
      create_update_list(deleted_edges) )
    while update_ list is not empty do
        s = size(update_list)
        flag = array(N,0)
        expand(update_list, flag, s)
        update_list = generate_update_list(flag)
        check_optimal_parent_cycle(update_list)
    end
end

expand(update_ list, flag, s)
begin
    id = global_thread_id
    node = update_list[id]
    for each child of node do
        lock(child)
        cost = g(node) + weight(node,child) + h(child)
        if f(child)> cost then
            f(child) = cost
            optimal_parent(child) node
            flag[child] = 1
        else if f(child) < cost and optimal_parent(child) is node then
            for each parent of child do
                if check_ cycle(child,parent) then
                    continue
                end
                new_cost = g(parent) + weight(parent,child) +
                  h(child)
                if f(child) > new_ cost then
                    f(child) = new_cost
                    optimal_parent(child) = parent
                end
            end
            flag[child] = 1
        end
        unlock(child)
    end
end
```

simultaneous propagation. We also show various cases that arise when simultaneous propagation is performed with the graph which consists of cycles and how to update the algorithm to encapsulate those cases too. In the next chapter 7 we will show how to apply the methods discussed in the above chapters to various applications of A* algorithm.

CHAPTER 6

Experimental Results

In this chapter, we will compare and analyze dynamic algorithms proposed in the previous chapters. In the previous chapters we have proposed propagation for insertion of edges, deletion of edges and fully dynamic updates to find the new optimal path, instead of this we can perform A* search from start for each update. We have considered multiple A* search as base line to compare with the proposed algorithm.

For all the experiments the dynamic graphs we have picked are from SNAP (20) dataset. Since we are concerned about path finding in large graphs on GPU, the temporal graphs we have chosen are:

- **Live Journal** $(N = 3.9M, E = 34.6M)$:It is a free on-line community with almost 10 million members. Live Journal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong.
- **Youtube** $(N = 1.1M, E = 2.9M)$: Youtube is an video sharing and streaming platform by Google Inc. In the Youtube social network, users form friendship each other and users can create groups which other users can join.
- **Wiki Talk** $(N = 1.1M, E = 5.1M)$:Wikipedia is a free encyclopedia written collaboratively by volunteers around the world. Each registered user has a talk page, that she and other users can edit in order to communicate and discuss updates to various articles on Wikipedia.
- **Ask Ubuntu** $(N =, 0.1M, E = 0.9M)$: The graph contains the temporal network of interactions on the stack exchange web site Ask Ubuntu. The infractions can be asking questions, answering questions, commenting on the question, and commenting on the answer etc.
- **Math Overflow** $(N = 24K, E = 506K)$: This is a temporal network of interactions on the stack exchange web site Math Overflow.
- **SuperUser** $(N = 194K, E = 1.4M)$: This is a temporal network of interactions on the stack exchange web site Super User.

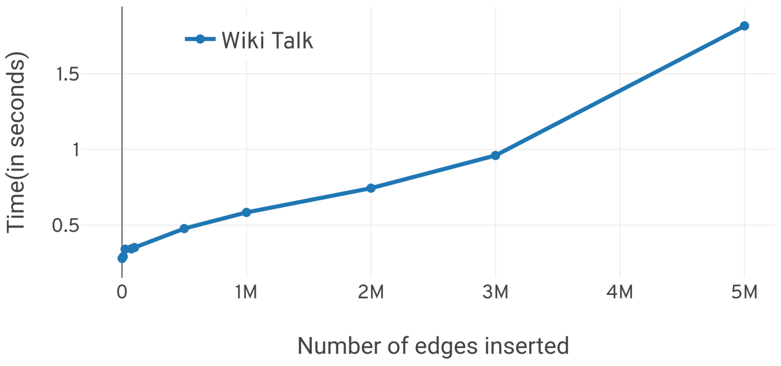
6.1 Experimental Setup

The server we used to collect the data is running on RHEL with an Nvidia Tesla K40c GPU, which is based on the *Kepler* architecture and has 15 SM each having 192 cores(total 2880 cores), it has 12 GB of GDDR5 Global Memory with 384-bit memory bus. The GPU operates at a frequency of 745 MHz, which can be boosted up to 876 MHz, the clock speed for memory is 1502 MHz. Mostly all data are collected using this server unless stated otherwise. The server has an Intel(R) Xeon(R) E5-2640v4 CPU, which operates at 2.4GHz and has 40 cores and 62GB RAM. We have used **cuda-toolkit** version 10.0 to compile and run our GPU programs.

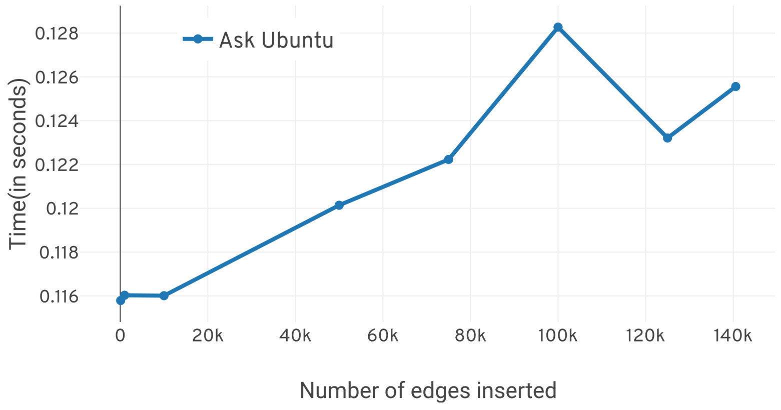
6.2 Insertion of Edges

When the graph is only subjected to the insertion of edges, the amount of processing required to find the new optimal path is based on how many nodes are being affected (cost change) due to the insertion of edges, which generally increases as we increase the number of edges being inserted. Figure 6.1 shows the increase in execution time. We can also infer that the increase is linear for a certain part of the plot in each sub-figure. In Figure 6.1c there is a dip then rise in execution time, it is because initially, the destination node was unreachable so it has to process all nodes and as we keep inserting edges the destination becomes reachable from source and it then became shorter as more edges got inserted in the graph. After a while due to increase in the number of edges the size of the graph grew and also the number of nodes which are updated simultaneously grew so contention for lock increases thus increasing the execution time.

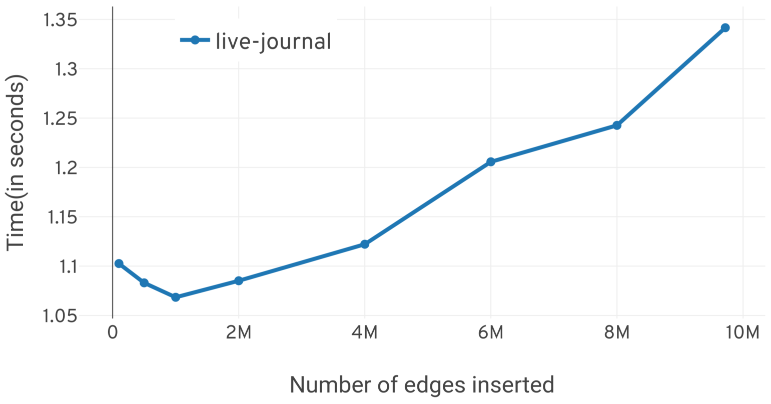
For computing optimal path in a graph with incoming updates there are two ways to process the insertion of edges, merge the edges inserted with original graph and re-compute A* algorithm, or process only the inserted edges and get the path as described in Algorithm 3.1. The plot 6.2a compares the execution time between both the approaches, propagation takes lesser time than re-executing the



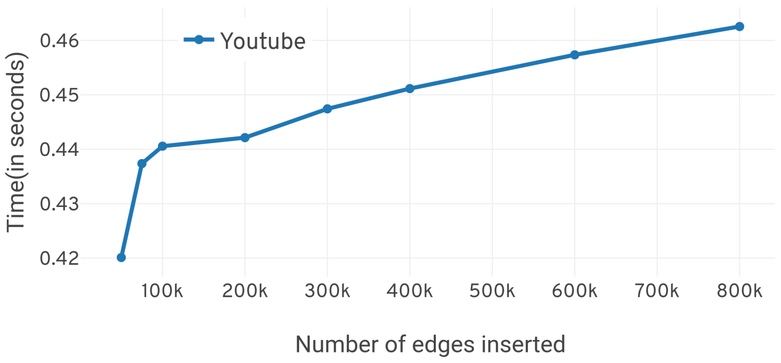
(a)



(b)



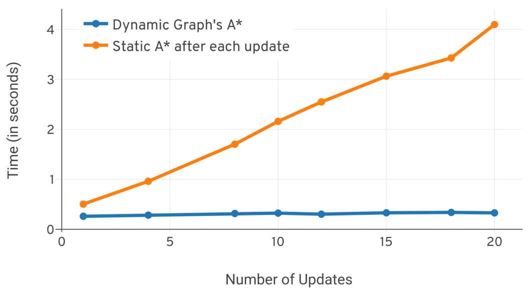
(c)



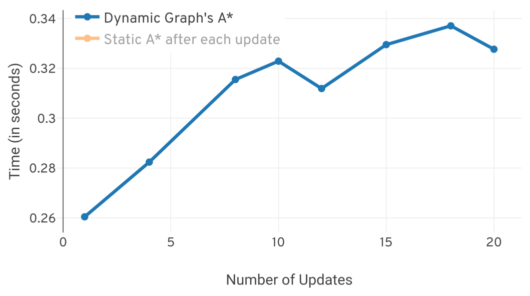
(d)

Figure 6.1: Execution time when graph is subjected to only Insertions

A* algorithm because instead of beginning from scratch again we propagate the change in the cost to the affected nodes and while doing so get the optimal path as stated in lemma 3.2.2. The plot 6.2a shows the run time trends for the propagation of inserted edges as the number of updates is increased, as the number of updates increases more and more edges need to be processed so the overall time increases but the growth is linear. Table 6.1 enlists the propagation timings for graph with 1M nodes and 5M edges.



(a) Multiple static A* vs Dynamic Graph's A*(propagation) 3.1



(b) Execution time of propagation vs Number of updates

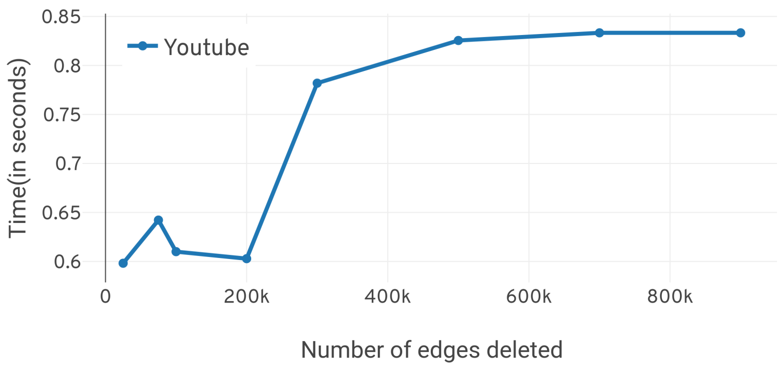
Figure 6.2: Execution Time vs Number of Queries, G(N=1M,E=5M)

No. of Queries	Algorithm 3.1	GPU A*	Source Node	Sink Node
1	0.260	0.503	22025	883802
4	0.282	0.960	22025	883802
8	0.315	1.702	22025	883802
10	0.322	2.158	22025	883802
12	0.301	2.548	22025	883802
15	0.329	3.063	22025	883802
18	0.337	3.428	22025	883802
20	0.327	4.097	22025	883802

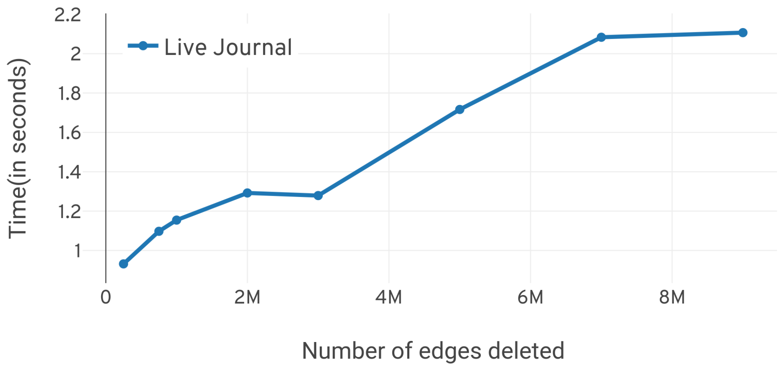
Table 6.1: Run-time of propagation vs number of updates in the graph(N=1M,E=5M), time in seconds

6.3 Deletion of Edges

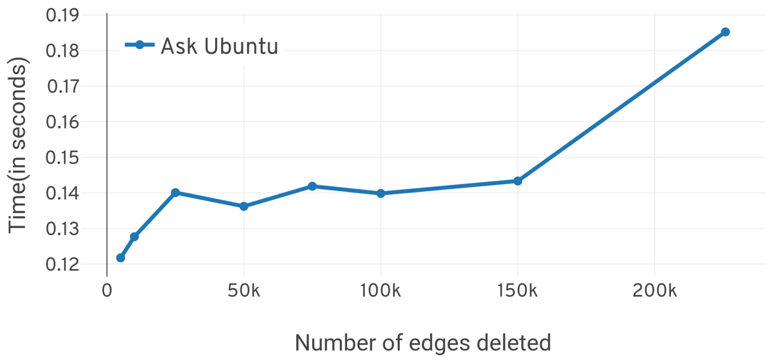
When edges are deleted in a graph as the number of deleted edges increases there is need to process more nodes in parallel and thus contention at lock also increases which in turns increases the execution time, as shown in Figure 6.3. We also observe that there are certain points where the execution time increases rapidly in each Figure 6.3a, 6.3b, 6.3c, 6.3d It might be due to the sudden increase in number of edges i.e $(u \to v)$ deleted which are part of optimal path to the node v, which



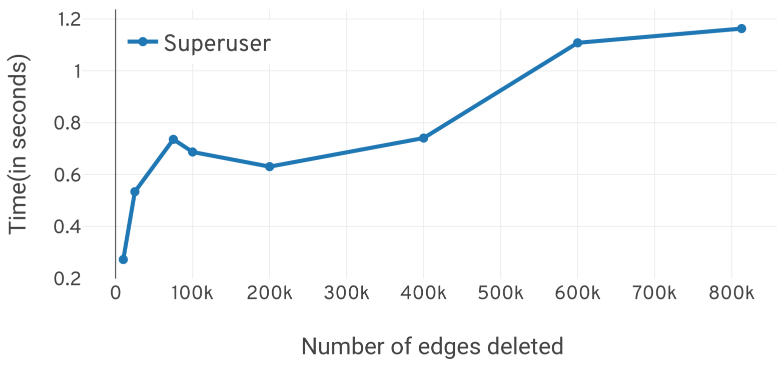
(a)



(b)



(c)



(d)

Figure 6.3: Execution time when graph is subjected to only deletions

increases the amount of processing required.

Table 6.2 shows the trend in execution time of Algorithm 4.2 as the number of updates the graph is subjected to increases and updates consists of only deletion of the edges in the graph. As the number of updates increases the run time also increases as the number of nodes which need to be processed also increases and a node can be updated multiple times in different waves of the propagation.

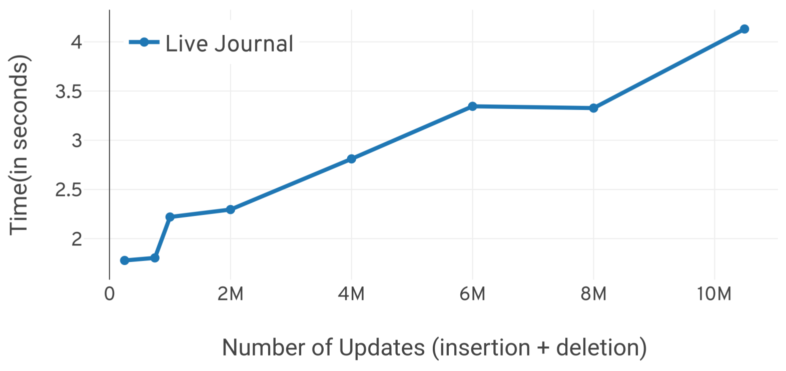
No. of Updates	Algorithm 4.2	GPU A*	Source Node	Destination Node
1	0.376	0.282	7813	960378
2	0.381	0.457	7813	960378
5	0.422	0.905	7813	960378
8	0.447	1.409	7813	960378
10	0.477	1.656	7813	960378
14	0.526	2.147	7813	960378
18	0.579	2.491	7813	960378
20	0.642	2.541	7813	960378

Table 6.2: Execution time of propagation vs Number of updates for deletion of edges in the graph(N=1M,E=5M), time in seconds

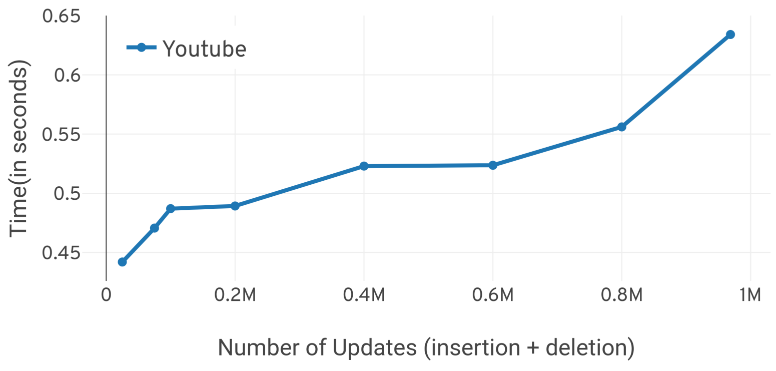
6.4 Fully Dynamic

In dynamic setting we process both insertion and deletion at the same time. With an increase in the number of updates, the execution time increases as in Figure 6.4. We can also observe that the sub-figures are similar to Figure 6.3 that is because propagation for deletion takes more computation power than for insertion.

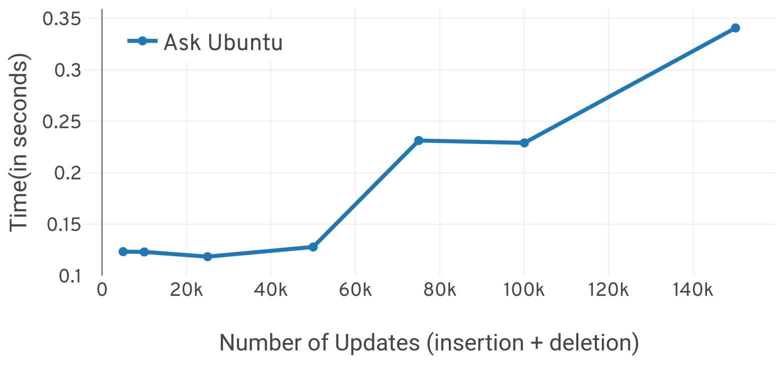
The amount of execution time needed to process deletion is much larger than inserts. As shown in Figure 6.5a where the total number of updates is kept constant but the number of updates which are insertion or deletion is changed we observe that there is a drastic drop in execution time as the percentage of deletion becomes 0. It shows that even for a small amount of deletion the execution time is high compared to insertions. It is because we are performing multiple checks for the cycle when we process deletion which takes a major amount of execution time, while in insertion there is no need to check for the cycle when we are doing separate propagation. In Figure 6.5b we compare the two variants of the proposed algorithm (Dynamic Graphs' GPU A* algorithm), simultaneous propagation and separate propagation, separate propagation outperforms simultaneous propaga-



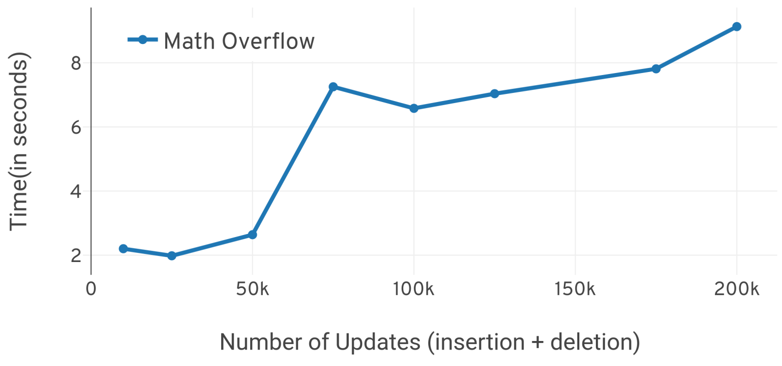
(a)



(b)



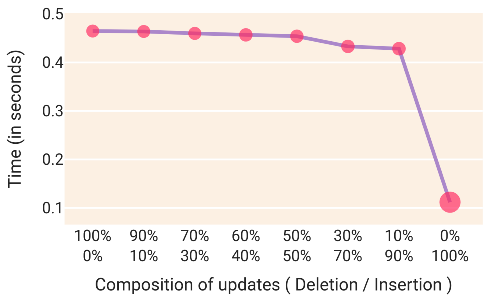
(c)



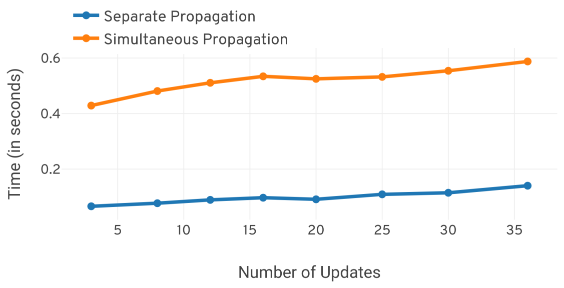
(d)

Figure 6.4: Execution time vs Number of Updates

tion as the latter require more computation for the additional checks as discussed in Section 5.2.1 and Section 5.2.2.



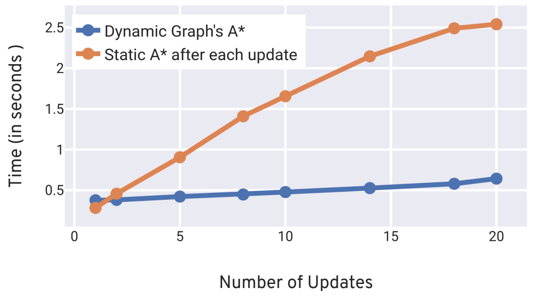
(a)



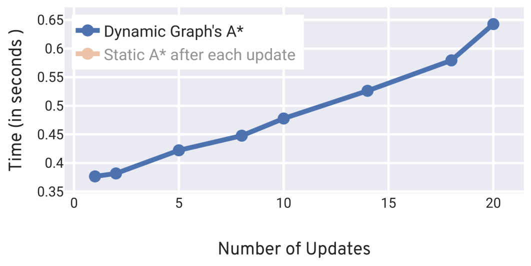
(b)

Figure 6.5: (a)Run Time vs Composition of updates, (b)Different versions of DGA* propagation

We compare the performance of our algorithm than by executing GPU A* repeatedly in Figure 6.6 which shows how execution time varies when we increase the number of queries (finding the path from source to destination after some update) increases. Performing GPU A* after each set of updates is almost linear but propagating the updates as described in the above sections takes considerably less amount of execution time. The speed-up we get is proportional to the number of queries we preform, as the number of queries increases we get better speed up. In Figure 6.6a updates contain of both insertion and deletion of edges, we can infer that when the number of queries is one the multiple GPU A* performs better than proposed separate propagation Algorithm 5.1(DGA*) but as we increase the number of queries DGA*(Dynamic Graphs' GPU A* algorithm) perform better than executing GPU A* multiple times from *source*. The speedup we get by applying the proposed Algorithm 5.1 is proportional to the number of queries (updates) graph is subjected to, we also have seen an increase in speedup as the size (nodes, edges) of graph increases. To test the capability of our algorithm we used real-world temporal graphs and networks taken from SNAP (20) results shown in table 6.3. We achieved 37× speedup for 100 queries.



(a) DGA* vs static GPU A*



(b) Algorithm 5.1(DGA*)

Figure 6.6: Execution Time vs Number of Queries

No.	Graph	N	E	Queries	DGA*	GPU A*	speedup
1	live-journal	3,997,962	34,681,189	10	6.01	33.93	5×
2	wiki-talk	1,140,149	5,133,140	10	12.24	24.84	2×
3	ask-Ubuntu	159,316	964,437	10	0.25	1.31	5×
4	YouTube	1,157,828	2,987,624	10	0.81	5.78	7×
5	math-overflow	24,818	506,550	10	0.09	0.67	7×
6	superuser	194,085	1,443,339	10	0.41	2.89	7×
7	live-journal	3,997,962	34,681,189	100	11.41	424.06	37×

Table 6.3: Comparison of execution time of DGA* with repeated GPU A* algorithm (Time in seconds)

6.5 Summary

In this chapter we have discussed the run time trends for the proposed dynamic algorithms with respect to the amount of updates and number of updates the graphs is subjected to. We have also compare the algorithms with multiple static A* search algorithm and achieved $2\times - 37\times$ speedup. We have also compared the two different method of propagation of dynamic updates and found that separate propagation outperforms simultaneous propagation. In next Chapter 7, we will discuss about the various applications of the A* algorithm and how to apply the proposed algorithms in such applications.

CHAPTER 7

Applications

A* has various applications. In this chapter, we will discuss three applications of A* algorithm that are energy efficient routing in WSN, dynamic puzzles and path planning for AGV. We have first briefly discussed the problem statement and then have presented the modified algorithm for each application. We also analyze the performance improvement or degradation compared to the repeated A* algorithm.

7.1 Wireless Sensor Networks

Wireless sensor network (WSN) is a set of sensor nodes, which are scattered in a large area to monitor the physical and environmental conditions. Sensor nodes monitor physical and environmental conditions like pressure, temperature, humidity, vibration, and pollution levels and send the data to a *sink* or*base station* for further analysis and storage. The collected data from a group of the nearby sensor are collected at relay nodes, which are responsible to transmit the collected data to the *sink*. The sensor nodes are lightweight and have limited processing power and memory capacity, small communication range and are operated by a small battery. The energy loss of the sensor is proportionate to the distance of communication and in most cases, there is no way to change the batteries. Thus energy is one of the major constraints responsible for the longevity of the network.

In most of the routing algorithms the best path is chosen for transmission of the data from the source sensor node to the *sink*(*base station*). As more and more data is collected the same path is chosen again and again for the transmission of the data, thus quickly depleting the energy of the node which is in the best path. This considerably decreases the overall life-time of the network. The life-time of the WSN is described as the total rounds of the communications between the sensor nodes and the *sink* till one of the sensor nodes dies, i.e. energy of the sensor node becomes 0.

There are many routing algorithms which try to increase the lifetime of the network like SPIN-EC (18), LNDIR (27), S. AlShawi et al. (1) proposed a fuzzy A* algorithm to increase the lifetime of the network, Keyur Rana and Mukesh Zaveri (26) proposed another algorithm which uses A* to find the best route while taking account of the residual energy at each node. Ali Ghaffari (11) proposed Energy Efficient Routing Protocol (EERP) which uses A* with the cost of node dependent on energy, packets transmitted or received, and buffer available at the node. The energy associated with sensor nodes is constantly changing as packets

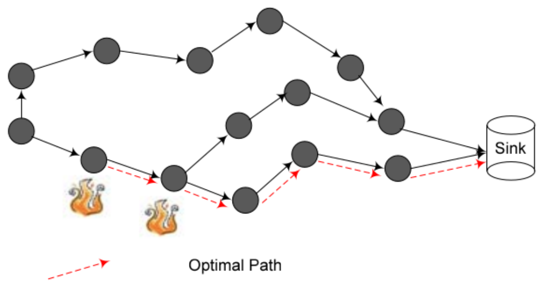


Figure 7.1: Energy-efficient data forwarding in wireless sensor networks(11)

are received by that node, thus modifying the edge weights of the network. So after each round, A* is performed to find the energy-efficient path to the Base station. Our algorithm can eliminate applying A* again and again, as we can simulate the weight change as deletion of that edge and then insertion of the same edge with the new weight. To show this and evaluate the performance gain we implemented EERP (11) and instead of doing repeated A* we used the DGA* algorithm to find the optimal path.

To increase the network lifetime and to make sure that sensor nodes are not drained faster in EERP (11) the author has proposed that weights of edges will depend on the residual energy, free buffer and packet reception rate. The weight of the next neighbour is defined as an aggregation of the above parameters, given as g 7.1.

$$g(n) = \text{Min}(\alpha(\frac{E_{ini}(n)}{E_{res}(n}) + \beta(\frac{N_t(n)}{N_r(n)}) + \gamma(\frac{B_{ini}(n)}{B_f(n)}) \tag{7.1}$$

In Equation 7.1, $E_{ini}(n)$ and $E_{res}(n)$ are the initial and residual energies of the node n. $N_r(n)$ and $N_t(n)$ are the transmitted and received packets,$B_f(n)$ and $B_{ini}(n)$ referred to the amount of free and initial buffers of node n respectively. α, β and γ are weight parameters such that $\alpha + \beta + \gamma = 1$.
In EERP the author has defined the heuristic($h(x)$) for a node as the average number of hops needed from current node to reach the *destination*. It is calculated by dividing the Euclidean distance from current node to the *destination*, to the average distance between current node and its neighbours. The proposed h is given in Equation 7.2.

$$h(n) = \frac{d(n, s)}{\text{avg}(d(n, j))} \tag{7.2}$$

To build the routing table, the authors (26) have proposed algorithm where A* is performed for each node to find the best path to sink, after each round.

Algorithm 7.1: Efficient Routing Algorithm (26)

```
Input: Sensor Network
Output: Life of Sensor Network in term of rounds
begin
    init_network()
    calculate_distance()                    // Finds distance to the sink
    flag_end_algo = false                   // End if energy of a node becomes 0
    round = 0
    while not flag_end_algo do
        init_route()                        // Initialize the routing table
        for each node in network do
            create_tree(node, sink)         // GPU A* to find optimal path
            prepare_route()
        end
        broadcast_solution()
        update_energy(flag_end_algo)        // Update the energy of sensors
        round = round + 1
    end
    print(round)
end

create_tree(source, sink)
begin
    GPU A*(source, sink, K)                 // K Parallel
end
```

Instead of performing A* for each node to find the path, we take the idea from D* (28) where to find the path, the search starts from the destination node towards the source node. So to find paths to all nodes we start A* search from sink node and end when we have visited all the nodes. As shown in Algorithm 7.2, first we build the network and calculate heuristic values which is the euclidean distance from *sink* node and set the number of rounds as zero (line 2-6). Then we create the A* search tree from *sink* (as root node) and broadcast the optimal path to each sensor node. At each round some portion of sensor node (here approx. 60%) sends the packets to *sink* node using their optimal paths. As a sensor node send and receives a packet its residual energy changes (line 10). While none of the node is fully exhausted we continue the above process of creating the A* search tree and sending the solution (line 13-20). At the end the number of round is the lifetime of the network.

Algorithm 7.2: EERP with creation and propagation of updates

```
    Input: Sensor Network
    Output: Life of Sensor Network in term of rounds
 1  begin
 2      init_network()
 3      calculate_distance()                      // finds the distance to sink
 4      init_sol_array()
 5      round = 0
 6      flag_end_algo = false             // End if energy of a node becomes 0
 7
 8      create_tree(sink)
 9      broadcast_solution()               // Broadcasts the routing schedule
10      update_energy(flag_end_algo)         // Energy update for relay nodes
11      round = round + 1
12
13      while not flag_end_algo do
14          create_update_list()
15          separate_propagation()         // Propagate the updates, algo 5.1
16          update_sol_array()        // Update routing table with new paths
17          broadcast_solution()
18          update_energy(flag_end_algo)
19          round = round + 1
20      end
21      print(round)
22  end
23
24  create_tree( sink)
25  begin
        /* A* from sink, finds the optimal path to all the nodes from the
           sink                                                          */
26      GPU A*(sink, K)                                      // K Parallel
27  end
```

The energy consumption model of the EERP (11), the authors used the first order radio model which is the typical model in the area of routing protocol evaluation in WSN (13). According to this model, the energy consumed for transmitting and receiving k-bit data can be calculated as follows (13):

$$E_{Tx}(k) = k(E_{elec} + \epsilon_{amp}.d^2)$$
$$E_{Rx}(k) = k.E_{elec}$$
$$E_T(k) = E_{Tx}(k) + E_{Rx}(k) = k(2E_{elec} + \epsilon_{amp}.d^2)$$

7.1.1 Experimental Evaluation

All algorithms are implemented in CUDA and ran on Tesla K40c a Kepler based GPU with 15 SMs and 192 SP per SM for a total of 2880 SPs. In this section, we will compare the life expectancy of network and performance of algorithms EERP (11), A *(12), DGA* for routing the packets in a Wireless Sensor Network.

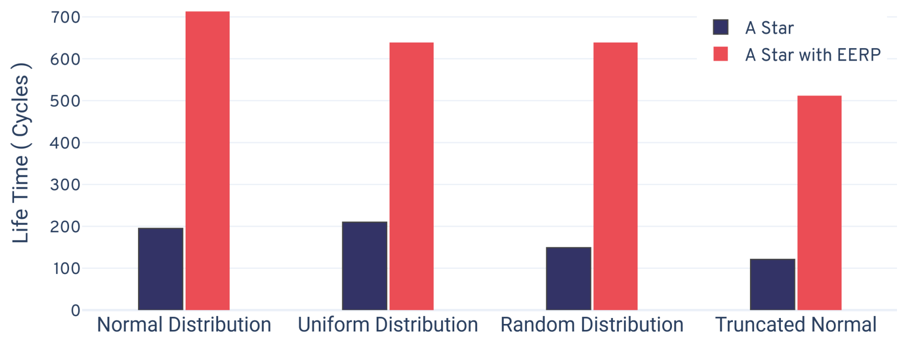


Figure 7.2: Network Lifetime Comparison

In Figure 7.2 we compare A* routing algorithm with EERP (11) with different node (sensor node) distribution. For all the distributions we have taken that sensor nodes area dispersed in area of 100×100 m^2. The base station is located at coordinate (50,50) and the number of nodes is 150 with each having range of 20m. EERP increases the network lifetime by almost 3× in each distribution. Most of the real-world networks follow a normal distribution as more sensor nodes are concentrated near the base station (sink). Figure 7.3 shows how the execution time of EERP is boosted by using our algorithm, for 100 nodes in the network we achieved $24 times$ speedup. As the number of nodes in the network increases the execution time of EERP with static A* search increases much faster than EERP with DGA* search. With 500 nodes we achieved 55*times* speedup. It is because our algorithm scales well for the number of updates (here number of rounds) and the size of the graph compared to repeated A* from scratch, as we update only the parts of the graph which are affected by the change.

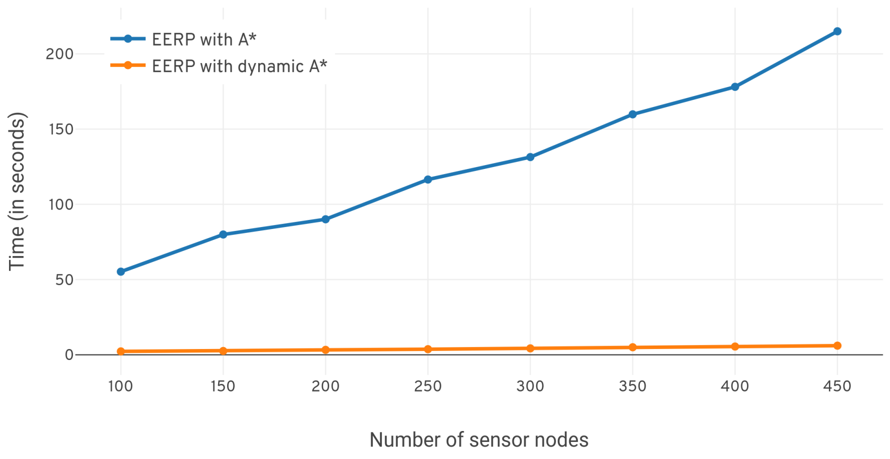


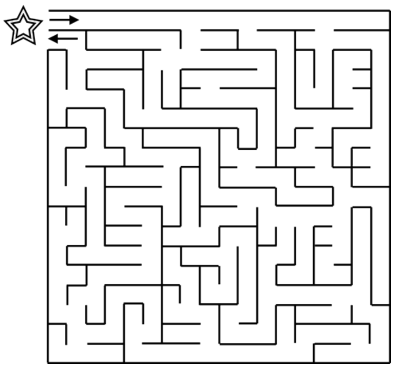
Figure 7.3: Execution Time vs Number of Nodes

7.2 Solving Puzzles

The puzzle is a game or problem that tests cleverness, imagination, and knowledge of the person (or solver). Puzzles are mainly seen as a form of amusement but they can also arise from serious mathematical and logical problems. Puzzles can be categorized as logical puzzles, mazes, mathematical puzzles, etc. Also in most of the video games, the base problem is a puzzle. Many algorithms have been devised to generate and solve different kinds of puzzles. A* search algorithm is one such algorithm which solves a wide variety of puzzles such as mazes, sliding puzzle, etc.

Here we will focus on solving mazes, a **maze** can be defined as a network of the path in which one has to find a way out of the maze. A standard maze is shown in Figure 7.4a. The maze problem can be traced back to the greek myth about *Theseus* who was tasked to kill the giant *minotaur*, he solved the maze by using a ball of thread that helped him to find his way back. Maze solving is one of the most common problems in mobile robotics and AI.

Tremaux's algorithm (2), Maze-routing algorithm (7), BFS, DFS, and A* algorithm (12) are well-known algorithms to solve a maze. A* is also used to find the optimal path when there is more than one path in a maze, shown in example Figure 7.4b.


(a) Standard Maze (33)

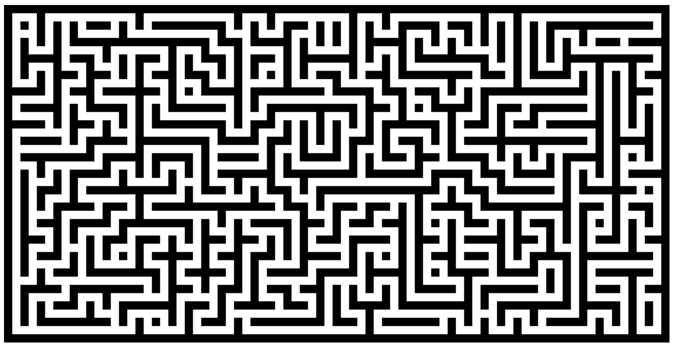

(b) A maze with no dead ends, i.e. multiple paths (34)

Figure 7.4: Example Mazes

7.2.1 Dynamic Mazes

Mazes can also be described as a set of paths with many obstacles located in a path. Most of the mazes are static, i.e. the path and obstacles are fixed but in video games where the frame changes in less than 10th of a second the obstacles can change their locations, and also the path can change. In a video game, screen size is fixed we can represent it as a 2D matrix. Let's also fix a point in the matrix as the starting point and another one as the endpoint. Now when the frame changes it can be described as insertion and deletion of edges in the matrix. To solve such mazes or pathfinding in the grid where obstacles and edges can change, instead of performing A* from start we can apply the proposed Algorithm 5.1 to find the next optimal path.

In our experiment, we fixed the grid size and for a fixed grid we then insert and delete edges in the grid. Our method is presented in Algorithm 7.3, first we create the $n \times m$ maze with the obstacles (line 2). While maze is subjected to an update, we first create update_list from the updates and append new obstacles in it (line 4-6). Then we call the solver algorithm which uses separate propagation on GPU to find the new optimal path for the maze (line 14-20). If the update list generated was empty, implies there were no edge updates nor obstacle movement then we set the flag to end the algorithm (line 9).

Algorithm 7.3: Maze solver for dynamic updates

```
   Input: Grid dimensions(n,m)
   Output: optimal path
 1 begin
 2     maze = create_maze(n,m)
 3     end = false
 4     while not end do
 5         update_list = maze.update()
 6         update_list.append(maze.create_obstacles())
 7         solver(maze,update_list)
 8         if update_list is empty then
 9             end = true
10         end
11     end
12 end
13
14 solver(maze, update_list)
15 begin
16     if update_list not empty then
17         create_diff_csr(update_list)
18         inserted_edges = create_insertion_list(update_list)
19         deleted_edges = create_deletion_list(update_list)
20         separate_propagation(inserted_edges,deleted_edges,N,E)
           /* optimal path can be traced by following otpimal_parent of
              destination                                                 */
21     end
22 end
```

7.2.2 Experimental Evaluation

We have used the GPU nvidia Tesla-T4 to gather all the execution time of the Algorithm 7.3. Tesla-T4 is based on the nvidia's *Turing* architecture and has 2560 CUDA cores and 12GB of global memory.

For evaluation, we have used grid-based square mazes with multiple optimal paths, the obstacle is dynamic and can move around the grid randomly. For an $n \times n$ grid the *source* node is fixed at $(0, 0)$ and *destination* is at $(n-1, n-1)$. Figure 7.5 shows the execution time of Algorithm 7.3 as we increase the grid dimensions. As we increase the grid dimensions the number of nodes and edges grows polynomially and thus more time is required to find the new optimal path. The number of

N	Updates	Update size	Start	End	DGA*	GPU A*	PQ
500	10	49800	0	249999	0.51	5.60	10k
1000	10	199935	0	999999	2.58	28.22	10k
1500	10	449188	0	2249999	6.76	98.87	100k
2000	10	798408	0	3999999	19.57	141.19	100k
2500	10	1249576	0	6249999	37.18	405.71	100k
3000	10	1798182	0	3999999	65.80	710.76	1M
3500	10	2449763	0	12249999	126.59	1363.66	1M

Table 7.1: Execution Timings for N×N grid puzzles, time in seconds

obstacles in the grid is roughly 20% of the nodes in the grid. The number of the updates the grid(or graph) is subjected to is kept constant at 10 for the Figure 7.5. The static A* takes more time to compute the path as it has to start from scratch every-time and the difference between the two algorithms widens as the grid size and update size grows in millions, that is because A* algorithm grows much faster compared to the Algorithm 7.3. The execution time is listed in the Table 7.1 in detail.

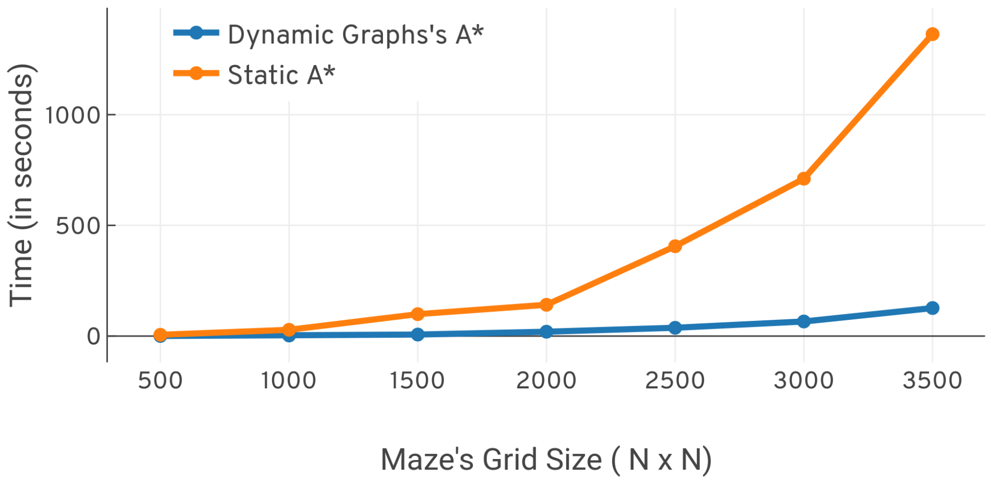


Figure 7.5: Execution Time vs Grid Size for N×N dynamic maze

7.3 Path Planning for AGV

In this section we will discuss path planning for AGV (Automated Guided Vehicle) using the A* algorithm. AGV is a robot that follows previously defined paths using wire or magnetic strips they use sensors to navigate in the known environment. Figure 7.6 shows a map of a warehouse where AGVs are used to load and unload the cargo. Nowadays AGV has become an integral part of the global supply chain and logistics due to the rise of automated logistics systems. AGV reduces the labour cost, can work in conditions that are harsh for humans, and improve the overall working environment for the workers and it plays important role in unifying the information flow in the logistics system. One of the major challenges in AGV control systems is to find the optimal path for multiple AGVs with collision avoidance and zone control. Chunbao Wang and Lin Wang et al. (30) have proposed shortest time planing algorithm which utilizes the K-shortest path obtained from running the *Improved A-Star algorithm* (30) which also takes into account the number of turns in a path, they have also proposed a dynamic algorithm for multi-AGV path planning with collision avoidance.

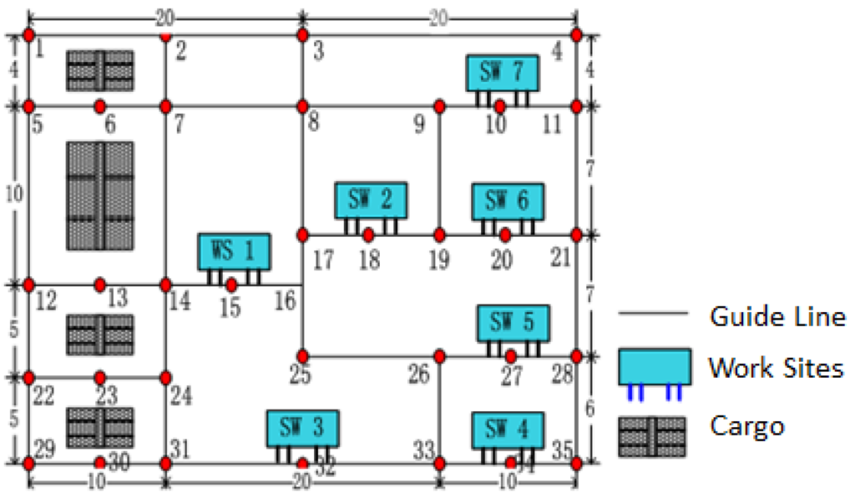


Figure 7.6: Environment Map (30)

7.3.1 K-Shortest Path Problem

In the Chapter 2, we have extensively discussed the optimal path or the shortest path problem and what are the various algorithms available and their variants for such a problem. In a later chapter, we have proposed methods to take into account the dynamic insertion and deletion of edges. There are cases where we are not just concerned about the shortest path but also but the next K-1 shortest path. This additional information is the basis of various routing algorithms. Many algorithms discussed for finding the shortest path can be extended to find then K-shortest path in a graph i.e. Dijkstra's algorithm, A* algorithm. Gang Liu and K.G. Ramakrishnan et al. have proposed *A* Prune* (21) algorithm which finds the K-shortest path in a graph subjected to multiple constraints.

For path planning of an AGV, one has to address what is a reasonable (admissible) path?. Many researchers equal it to the *shortest path* while other scholars consider it to be the *smoothest path*, such is the influence of turning in path planning for AGV. In the Algorithm 7.4 we have shown A* algorithm on GPU which finds the K-shortest time path and store it in the AGV's library for the navigation. To find the K-shortest path, first A* search from *source* to *destination*(sink) is completed. Suppose the shortest path is of length n is $v_0, v_1, \ldots, v_n$, then edge $v_{n-1} \rightarrow v_n$ is deleted from graph and then again A* search is started from *source* to *sink*. It is repeated for k times till k-shortest paths have been found.

Algorithm 7.4: A* algorithm to find K-shortest path

```
   Input: Graph, source, sink, K
   Output: Array of K shortest paths
1  begin
2      path_library ={}
3      path = {}
4      number_pq = set_number_of_threads()
5      for $i \leftarrow 0$ to $K$ do
           /* Get the shortest path in path array                  */
6          GPU A*(source, sink, number_pq, path)        // in Parallel
           /* for path of size n, delete edge(path[n-1],path[n])    */
7          remove_edge(Graph, path)
8          path_library.insert(path)
9      end
10 end
```

In Algorithm 7.4 every time we start the search from *source* to *destination*, instead of it in Algorithm 7.5 we use the propagation for deletion as discussed in Section 4.5. Instead of performing A* search K times, we propagate the update to the affected edges.

7.3.2 Experimental Evaluation

We have used the GPU nvidia tesla-T4 to gather all the execution time of the Algorithm 7.5. Tesla-T4 is based on the nvidia's *Turing* architecture and has 2560 CUDA cores and 12GB of global memory.

In Figure 7.7, we compare the execution time for Algorithm 7.4 and Algorithm 7.5

Algorithm 7.5: A* algorithm with dynamic updates to find K-shortest path

```
   Input: Graph, source, sink, K
   Output: Array of K shortest paths
1  begin
2      path_library ={}
3      path = {}
4      number_pq = set_number_of_threads()
5      GPU A*(source, sink, number_pq, path)                // in Parallel
       /* for path of size n, delete edge(path[n-1],path[n])           */
6      remove_edge(Graph, path)
7      path_library.insert(path)
8      for $i \leftarrow 1$ to $K$ do
           /* Get the shortest path in path array                      */
9          propagate_deletions(deleted_edges,N,E)           // in Parallel
           /* for path of size n, delete edge(path[n-1],path[n])       */
10         remove_edge(Graph, path)
11         path_library.insert(path)
12     end
13 end
```

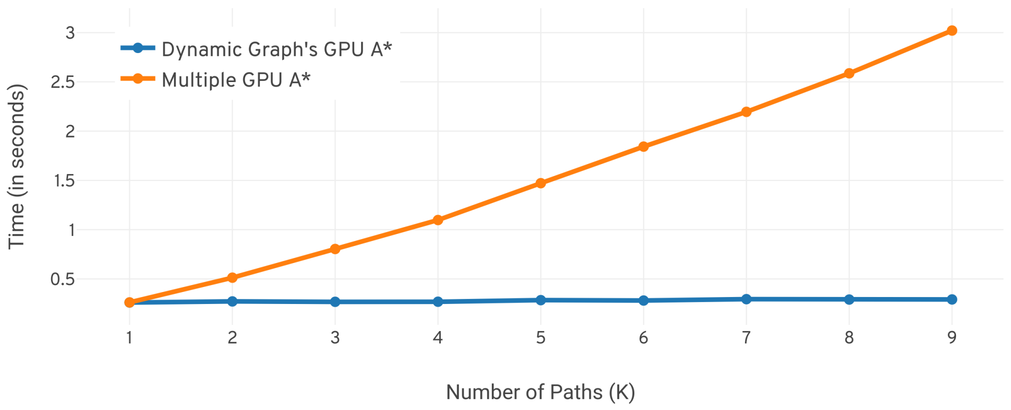


Figure 7.7: K-Shortest Path algorithm with GPU A* vs with DGA*, G(N=1M,E=8M)

as the number of paths to be found(K) increases. Execution time for Algorithm 7.4 grows linearly in multiples of time taken to find the first path from *source* to *destination* as we start A* algorithm every time the edge closest to *destination* in the optimal path is deleted. While for Algorithm 7.5 DGA* is used as there is no insertion only propagation for deletion is required. As only a single edge is deleted at the time the cases of parallel deletion and cycles(Section 4.4) doesn't arise in this case and thus reduces the run time, also as we only propagate the changes to affected nodes whose cost is already computed, we have noticed that in most of the cases the A* algorithm is invoked from the saved state after the propagation of deletion to find the optimal path. It is because when we delete the edge $u \rightarrow destination$ in the optimal path which is closest to the *destination*, the next path from $source \rightarrow destination$ doesn't pass from node u again but there is some other path from *source* to *destination* which contains many edges from old optimal path thus reducing the search space further as we have already explored such edges.

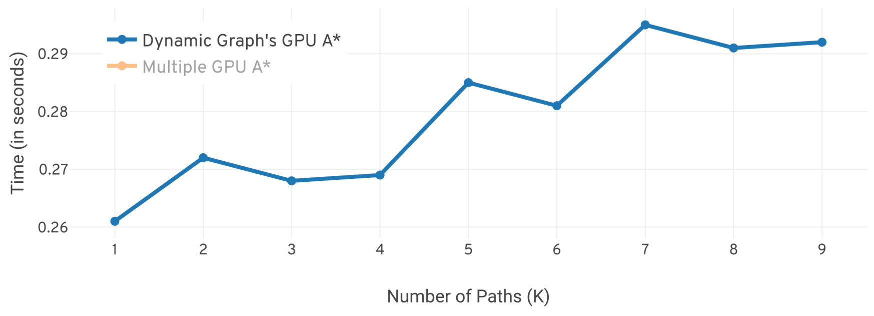


Figure 7.8: Execution time for K-shortest path algorithm with DGA* for varying K

Figure 7.8 shows the run time of Algorithm 7.5 (K-shortest path algorithm with DGA*) with varying K (number of path to be found). While the overall execution time increases with increase in the K, but there are few points where the run time surges. It is because the random nature of propagation for deletion when we add nodes in the update list and parallel priority queues, which in turns changes the contention at the lock. The K values for which the run time increases rapidly belong to such cases. Table 7.2 lists the execution time for various K value for both algorithm.

K	K-path with DGA*	K-path with GPU A*	parallel PQ
1	0.261	0.262	10k
2	0.272	0.513	10k
3	0.268	0.804	10k
4	0.269	1.098	10k
5	0.285	1.472	10k
6	0.281	1.843	10k
7	0.295	2.196	10k
8	0.291	2.587	10k
9	0.292	3.021	10k

Table 7.2: Comparison between K-shortest path algorithm with GPU A* and Dynamic Graph's GPU A* algorithms, time in seconds

CHAPTER 8

Heuristics

The core of the A* algorithm is that it uses heuristics to speed up the search. The heuristic function also plays a key role in determining the correctness and admissibility of the algorithm. Heuristic function $h(x)$ is an approximation of the cost needed to reach *destination* from any node $x \in \mathbb{V}$. There can be five possible scenarios with heuristics function h: $\forall x \in V$

1. If $h(x) = 0$, then A* turns into Dijkstra's Algorithm.
2. If $h(x) \leq$ cost of reaching destination from x, we are guaranteed to find the shortest path but it will explore more paths.
3. If $h(x) =$ cost of reaching destination from x, we will follow only the best path.
4. If $h(x)$ is sometimes greater than cost of reaching destination from x. A* is not guaranteed to find the shortest path.
5. If $h(x) >> g(x)$, then only $h(x)$ plays a role and A* turns into greedy best first search.

A* search is called **admissible** when it is guaranteed to return the optimal path, for which $h(x) \leq$ cost of reaching destination from x, or $h(x) = 0$. Also in some situations like in games, one might be interested in finding the *'good'* solution rather than the *'best'* solution, in such cases h can be set accordingly. There is a **trade-off** between *speed* and *accuracy* for A* algorithm which depends on the heuristic function. Most commonly used heuristic function for navigation and pathfinding in grids are grid Euclidean distance, Diagonal distance, and Manhattan distance. In many graph problems, the average number of hops and BFS distance is also taken as a viable heuristic function. The heuristic function to use is dependent on the problem space and should be formulated as per the need considering the trade-off between accuracy and speed.

8.1 Heuristic Types

We have discussed Dynamic Graphs in Section 1.2. In the dynamic graph, the graph structure itself changes with respect to the time, thus there is a possibility that heuristic values computed earlier might also become obsolete. For Dynamic Graphs, we can divide the heuristic function into the following two categories:

1. h is independent of the addition and deletion of edges and may depend on some other attribute of the nodes. e.g., latitude and longitude coordinates for computing the distances.
2. h is dependent on the edges of the graph and thus adding and deleting edges will also change the h. e.g., h is taken as BFS distance from the *destination*.

Till now all the cases and algorithms we have shown in this document we have assumed option 1, h is static and doesn't depend on the addition and removal of the edges in the graph.

8.2 Updating Heuristics

In this section, we will discuss how insertion and deletion of edges can affect h, if h is dependent on the structure of the graph e.g., BFS distance from *destination*.

Let us first discuss how **deletion** of an edge affects the heuristic value. Suppose a single edge $u \to v$ is deleted then it will only impact $h(u)$ and predecessors of node u, $h(v)$ will remain same because path from v to *destination* is unchanged. If $h(u)$ is computed with respect to edge $u \to v$ then due to deletion of this edge $h(u)_{old} \leq h(u)_{new}$, where $h(u)_{old}$ is heuristic value before deletion of the edge and $h(u)_{new}$ is heuristic value after deletion of the edge. As we already discussed underestimating h doesn't affect the correctness and admissibility of the A* algorithm (property 2). If $h(u)$ is not computed with respect to $u \to v$ then deletion of such edges will not affect $h(u)$.

For **insertion** of an edge $u \to v$, similarly as above only h of u and its predecessor are affected. Addition of this edge might create a new path to *destination*, which can decrease the h value. So $h(u)_{old} \geq h(u)_{new}$, if h recalculated is lesser than before. So in the case of insertion we can overestimate the heuristic values.

Figure 8.1 shows an example graph where edge $S \rightarrow D$ is inserted. Without updating heuristic value (BFS distance from node D) of node S (old value is 3), A* algorithm returns the path $S \rightarrow A \rightarrow C \rightarrow D$, which is not the optimal path, as the new optimal path after insertion of the edge is $S \rightarrow D$, which can be returned if the h values of node S is updates to 1.

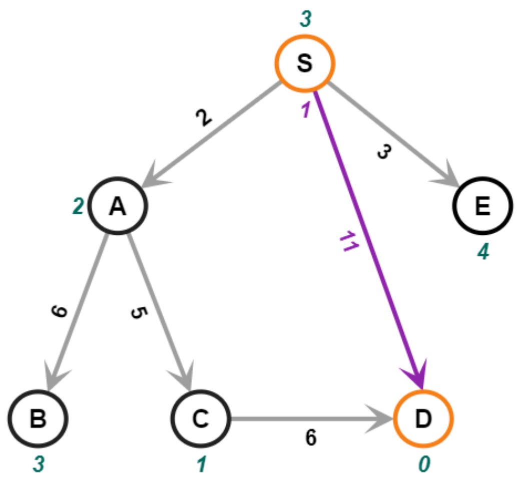


Figure 8.1: Example graph with inserted edge $S \rightarrow D$, heuristic values are shown green and updated h value in violet for S.

So to make sure that we have latest h values we propose Algorithm 8.1 for propagation of updated h values till the *source*. As discussed earlier only insertion of edges can affect the correctness of the A* algorithm where heuristics depends on the graph structure so if there is no insertion we simply return back (line 2-4). Then for each inserted edges $u \rightarrow v$ we recompute the heuristic value of node u and check if it is decreased, if decreased then we set the update_flag and update the heuristic value to latest. We require synchronization here as many threads can try to update the h val of node u (line 6-14). Then we create the update list from the update_flag and while the list is not empty we calculate the new h values and if it is decreased the we propagate it towards the *source* (line 17-30).

8.3 Experimental Results

We have used the GPU nvidia tesla-T4 to gather all the execution time of the Algorithm 8.1. Tesla-T4 is based on the nvidia's *Turing* architecture and has 2560 CUDA cores and 12GB of global memory. The figure 8.2 depicts the run-

Algorithm 8.1: Propagating Heuristics

```
Input: Graph, inserted_edges, N, E
begin
    if size(inserted_edges)==0 then
        return
    end
    update_flag = array(N)
    /* for each edges $u \to v$ in parallel */
    for each $(u, v) \in$ inserted_edges do
        lock(u)
        val =heuristic_function(u)
        if $val < h(u)$ then
            $h$(u) = val
            update_flag[u] = true
        end
        unlock(u)
    end
    update_list = create_list(update_flag)
    reset(update_flag)
    while update_list not empty do
        /* in Parallel */
        for each node in update_list do
            for each parent of node do
                lock(parent)
                val =heuristic_function(parent)
                if $val < h(parent)$ then
                    $h$(parent) = val
                    update_flag[parent] = true
                end
                unlock(parent)
            end
        end
        update_list = create_list(update_flag)
        reset(update_flag)
    end
end
```

time variation for finding the optimal path when the heuristic is updated and propagated for each query and when we don't update the heuristics. There is performance improvement when we don't update the heuristic values but it comes with the cost that the path might not be the shortest path. So there is a trade-off between performance and accuracy. This would have not been the case if the heuristic used was not depending on the graph structure as discussed in section 8.1. As the number of updates a graph is subjected to is increased the execution time for both cases increases in the almost same fashion.

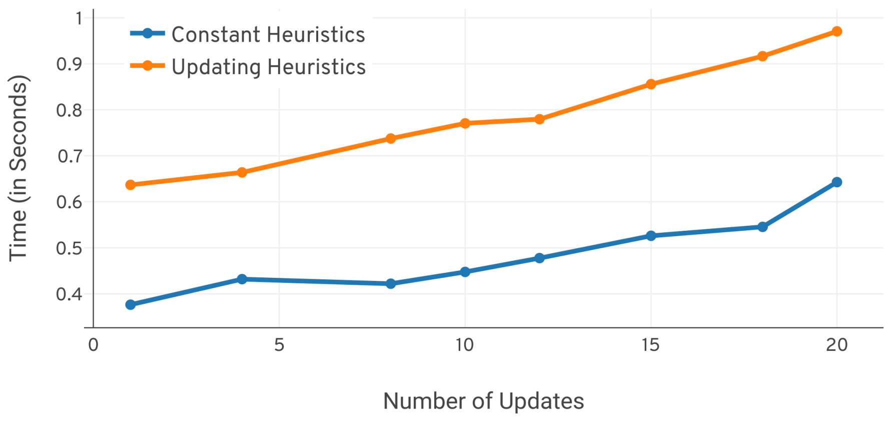


Figure 8.2: Execution Time for both policies of updating heuristics

8.4 Summary

In this chapter, we have discussed what role heuristic plays in the A* algorithm. We have discussed both types of heuristics based on their dependence on the graph's structure and how they change when we add or delete edge in the graph. For the heuristics that depends on the structure of the graph and will change on updates to the graph and might give a sub-optimal path if A* algorithm is executed. For those we have proposed Algorithm 8.1 which computes the new heuristic values and propagates to only such nodes whose heuristic values are not the latest one.

CHAPTER 9

Conclusion and Future Work

9.1 Conclusion

We have proposed an algorithm that can efficiently find the optimal path with the help of heuristics on GPU while taking account of updates in the form of insertion and deletions for the graph. Our algorithm is faster than doing repeated A* on GPU from scratch. We also found that insertions take less time to process than deletion and since both insertion and deletion require a different set of instructions to process, separate propagation outperforms simultaneous propagation.

9.2 Future Work

In the future, the Dynamic Graph's A* algorithm can be extended to run on a multi-GPU environment. We have not covered any algorithmic or GPU based optimizations and thus there is a large scope for optimizations for proposed algorithms on GPU i.e using streams, coalesced access, shared memory. We have published the C++/CUDA library on GitHub which can be used to boost the performance of many applications that uses the A* algorithm and are dynamic. Currently, we are updating the heuristics after each update for the ones which are dependent on the graph structure, this can further be studied and analyzed to find if there is a need to update it after every update or heuristics can be updated after an interval and give required accuracy. The parallel algorithms presented in the graph can be easily ported to OpenACC thus will not be restricted to GPUs which support the CUDA architecture.

REFERENCES

[1] **AlShawi, I. S.**, **L. Yan**, **W. Pan**, and **B. Luo** (2012). Lifetime enhancement in wireless sensor networks using fuzzy approach and a-star algorithm. *Sensors Journal*.

[2] **Anagnostou, L.** (2009). Maze solving algorithms: Tremaux's algorithm visual example. URL `https://www.youtube.com/watch?v=6OzpKm4te-E`.

[3] **Bellman, R.** (1958). On a routing problem. *Quarterly of applied mathematics*, **16**(1), 87–90.

[4] **Choset, H.** (2010). Robotic motion planning:a* and d* search. URL `https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf`.

[5] **Dijkstra, E. W.** (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, **1**(1), 269–271.

[6] **Eppstein, D.**, **Z. Galil**, **G. F. Italiano**, and **A. Nissenzweig** (1997). Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, **44**(5), 669–696.

[7] **Fattah, M.**, **A. Airola**, **R. Ausavarungnirun**, **N. Mirzaei**, **P. Liljeberg**, **J. Plosila**, **S. Mohammadi**, **T. Pahikkala**, **O. Mutlu**, and **H. Tenhunen**, A low-overhead, fully-distributed, guaranteed-delivery routing algorithm for faulty network-on-chips. *In Proceedings of the 9th International Symposium on Networks-on-Chip*. 2015.

[8] **Ferguson, D.** and **A. Stentz**, Field d*: An interpolation-based path planner and replanner. *In Robotics research*. Springer, 2007, 239–253.

[9] **Floyd, R. W.** (1962). Algorithm 97: shortest path. *Communications of the ACM*, **5**(6), 345.

[10] **Ford Jr, L. R.** and **D. R. Fulkerson**, *Flows in networks*, volume 54. Princeton university press, 2015.

[11] **Ghaffari, A.** (2014). An energy efficient routing protocol for wireless sensor networks using a-star algorithm. *Journal of applied research and technology*, **12**(4), 815–822.

[12] **Hart, P. E.**, **N. J. Nilsson**, and **B. Raphael** (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, **4**(2), 100–107.

[13] **Heinzelman, W. R.**, **A. Chandrakasan**, and **H. Balakrishnan**, Energy-efficient communication protocol for wireless microsensor networks. *In Proceedings of the 33rd annual Hawaii international conference on system sciences*. IEEE, 2000.

[14] **Italiano, G. F.** (2012). Dynamic graphs. URL `http://cs.ioc.ee/ewscs/2012/italiano/dynamic1.pdf`.

[15] **Koenig, S.** and **M. Likhachev** (2005). Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, **21**(3), 354–363.

[16] **Koenig, S.**, **M. Likhachev**, and **D. Furcy** (2004). Lifelong planning a*. *Artificial Intelligence*, **155**(1-2), 93–146.

[17] **Koenig, S.**, **C. Tovey**, and **Y. Smirnov** (2003). Performance bounds for planning in unknown terrain. *Artificial Intelligence*, **147**(1-2), 253–279.

[18] **Kulik, J.**, **W. Heinzelman**, and **H. Balakrishnan** (2002). Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless networks*, **8**(2-3), 169–185.

[19] **Leach, A. R.** and **A. P. Lemon** (1998). Exploring the conformational space of protein side chains using dead-end elimination and the a* algorithm. *Proteins: Structure, Function, and Bioinformatics*, **33**(2), 227–239.

[20] **Leskovec, J.** and **A. Krevl** (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[21] **Liu, G.** and **K. Ramakrishnan**, A* prune: an algorithm for finding k shortest paths subject to multiple constraints. *In Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2. IEEE, 2001.

[22] **Liu, X.**, **R. Deng**, **J. Wang**, and **X. Wang** (2014). Costar: A d-star lite-based dynamic search algorithm for codon optimization. *Journal of theoretical biology*, **344**, 19–30.

[23] **Malhotra, G.**, **H. Chappidi**, and **R. Nasre**, Fast dynamic graph algorithms. *In* **L. Rauchwerger** (ed.), *Languages and Compilers for Parallel Computing*. Springer International Publishing, Cham, 2019. ISBN 978-3-030-35225-7.

[24] **Nash, A.**, **K. Daniel**, **S. Koenig**, and **A. Felner**, Theta^*: Any-angle path planning on grids. *In AAAI*, volume 7. 2007.

[25] **Ramalingam, G.** and **T. Reps** (1996). An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, **21**(2), 267–305.

[26] **Rana, K.** and **M. Zaveri**, A-star algorithm for energy efficient routing in wireless sensor network. *In Trends in Network and Communications*. Springer, 2011, 232–241.

[27] **Shahzad, M. K.**, **D. T. Nguyen**, **V. Zalyubovskiy**, and **H. Choo** (2018). Lndir: A lightweight non-increasing delivery-latency interval-based routing for duty-cycled sensor networks. *International Journal of Distributed Sensor Networks*, **14**(4), 1550147718767605.

[28] **Stentz, A.**, Optimal and efficient path planning for partially known environments. *In Intelligent Unmanned Ground Vehicles*. Springer, 1997, 203–220.

[29] **Stentz, A.** *et al.*, The focussed d* algorithm for real-time replanning. *In IJCAI*, volume 95. 1995.

[30] **Wang, C.**, **L. Wang**, **J. Qin**, **Z. Wu**, **L. Duan**, **Z. Li**, **M. Cao**, **X. Ou**, **X. Su**, **W. Li**, *et al.*, Path planning of automated guided vehicles based on improved a-star algorithm. *In 2015 IEEE International Conference on Information and Automation*. IEEE, 2015.

[31] **Wheatman, B.** and **H. Xu**, Packed compressed sparse row: A dynamic graph representation. *In 2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018.

[32] **Zhou, Y.** and **J. Zeng**, Massively parallel a* search on a gpu. *In Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[33] **Efbrazil - Own work, CC BY-SA 3.0** (). `https://commons.wikimedia.org/w/index.php?curid=14479008`.

[34] **Purpy Pupple - Own work, CC BY-SA 3.0** (). `https://commons.wikimedia.org/w/index.php?curid=13326157`.

[35] **Wikipedia A* Algorithm** (). `https://en.wikipedia.org/wiki/A*_search_algorithm`.