A* for Dynamic Graphs on GPU

Lokesh Koshale (CS15B049)

1 Introduction

A* [7, 16] is one of the widely used path planning and shortest path approximation algorithms. It is used in several applications due to its performance and accuracy. A* is applied in a diverse set of problems from path-finding in video games and robotics to codon optimization in genetics. In several real-life graph applications such as wireless networks, the underlying graph is not static but is evolving. D* [2, 17], focused D* [3], and D* lite [10] form a family of informed incremental search algorithms where edge cost can change while the optimal path is being explored. In this work, we focus on A* for graphs that are subjected to update operation, where an update operation refers to the insertion or deletion of an edge. Instead of performing A* again from start each time graph is subject to update, our algorithm process the sub-graphs which are affected by the update. Compared to repeated A* our algorithm performs better and we have observed an increase in speedup which is proportional to the number of updates the graph is subjected to. For temporal graph available at SNAP [9] for 100 updates we got 25×–40× of performance improvement over repeated A* search. A* has various applications and one of them is in energy-efficient routing for wireless sensor networks, where the graph is inherently dynamic in nature. One of such routing algorithm is EERP [6] we have done a comparison between EERP with native A* and EERP with our algorithm, We got 24×–55× performance improvement by using our algorithm.
In this document section 2 explains A* algorithm and how it is implemented for GPU. Section 3 discusses how insertions are processed in parallel to find the optimal path. Section 4 explains the implications deletion on the optimal path and also reasons why deletion in the cyclic graph requires additional checks. Section 5 presents methods for processing insertion and deletion combined together and contains the proof of correctness of our algorithm. Section 6 contains a detailed analysis of experiments performed and its results. Section 7 shows how our algorithm can be used to improve performance in various applications that uses A*.

2 Background

In this section we discuss A* and its GPU implementation in detail. Algorithm 1 A* takes a Graph with source node(start) and destination node(end) as input and returns the optimal path from source to the destination node. We initialize open_list which is a priority queue with start node(Line 5). As cost of reaching source from source is 0, $g(start)$ is set to 0 (Line 7),Then we compute the heuristics_function (Line 8) from start to end which is the approximate

cost of reaching destination from source and then use it compute $f(start)$(Line 9). While open_list is not empty we extract a node with the minimum cost from it(Line 12), Then we check if the extracted node is the destination node, if it is then we return the path(Line 14). Otherwise, we add this node to closed_list(Line 15). Then we compute the cost of reaching the child from the source through this parent i.e g value for each of its children(Line 18-19). If the newly computed cost is less than old cost i.e $g(child)$ and the node is present in either list we remove the node from the list(Line 21-25). Then if the child is not present at either list we compute the $f(child)$ with new cost and add it to the open_list (Line 27-31). If we exhaust all nodes and haven't found the destination node we return failure(Line 33) as there is no path from source to destination.

Algorithm 1: A* [7]

```
1  Input: A Graph(V, E) with source node start and goal node end.
2  Output: Least cost path from start to end.
3  Steps:
4  Initialize
5      open_list = {start}        /* list of nodes to be traversed */
6      closed_list = {}           /* list of already traversed node */
7      g(start) = 0               /* cost from source node to node */
8      h(start) = heuristic_function(start,end) /* estimated cost from
           source to goal node */
9      f(start) = g(start) + h(start)   /* total cost from goal to node */
10
11     while open_list is not empty:
12         m = node on top of open_list with least f
13         if m == end
14             return
15         remove m from open_list
16         add m to close list
17
18         for each n in child(m):
19             cost = g(m) + distance(m,n)
20
21             if n ∈ open_list and cost < g(n):
22                 remove n from open_list  /* new path is better */
23
24             if n ∈ closed_list and cost < g(n):
25                 remove n from closed_list
26
27             if n ∉ open_list and n ∉ closed_list:
28                 g(n) = cost
29                 h(n) = heuristic_function(n, end)
30                 f(n) = g(n) + h(n)
31                 add n to open_list
32
33     return failure
```

General-purpose computation on graphics processing units (GP-GPU) has been widely used to accelerate numerous computational tasks. In multi-core systems to improve the performance, one has to increase the number of cores

and adding more and more cores increases the cost thus GPU is cost-efficient alternative hardware to execute parallel algorithms which also provides better performance. Zhou and Zeng [15] proposed a parallel variant of A* for GPU. We borrow the parallelization of A* on GPU from them. The bottleneck for parallelization is the sequential nature of Priority Queue which is the primary data structure to implement A*, to utilize the many-core GPU architecture the authors proposed to have multiple priority queues thus expanding many nodes at once. In the paper, the authors claim that GA* achieves 30×–45× performance improvement from the CPU implementation.

Algorithm 2 describes the framework of GA* algorithm using parallel priority queues. For each state s in the open_list or closed_list, $s.node$ stands for the last node in the path represented by s, $s.f$ and $s.g$ store the values of $f(s)$ and $g(s)$, respectively, and $s.prev$ stores the pointer to the previous state that expanded s, which is used to regenerate a path from a given state. List S stores the expanded nodes and list T stores the nodes after the removal of duplicated nodes. Lines 24-29 detect the duplicated nodes, where $H[n]$ represents the state in the closed list in which the last node in its path is node n. Through synchronization operations, which are computationally cheap on GPUs, we can push nodes expanded from the same parent into different queues (Line 32), as nodes with the same parent tend to have similar priority values.

Algorithm 2: GA* [15]

```
1  procedure GA*(s, t, k) !find the shortest path from s to t with k queues
2  Let {Q_i}_{i=1}^k be the priority queues of open_list
3  Let H be the closed_list
4  PUSH(Q_1, s)
5  m ← nil        !m stores the best target state
6  while Q is not empty do:
7      Let S be an empty list
8      for i ← 1 to k in parallel do:
9          if Q_i is empty then
10             continue
11         end if
12         q_i ← Extract(Q_i)
13         if q_i.node = t then:
14             if m = nil or f(q_i) < f(m) then:
15                 m ← q_i
16             end if
17             continue
18         end if
19         S ← S + EXPAND(q_i)
20     end for
21     if m ≠ nil and f(m) ≤ min_{q∈Q} f(q) then:
22         return the path generated from m
23     end if
24     T ← S
25     for s' ∈ S in parallel do:
26         if s'.node ∈ H  and H[s'.node].g < s'.g then:
27             remove s' from T
28         end if
29     end for
30     for t' ∈ T in parallel do:
```

```
31          t'.f ← f(t')
32          PUSH t' to one of priority queues
33          H[t'.node] ← t'
34      end for
35  end while
36  end procedure
```

Below are some terminology we will be using in this document.

2.1 Terminology

- $g(n)$: cost of reaching node n from source
- $h(n)$: approximate cost of reaching destination from node n
- $f(n)$: approximate cost of reaching destination from source via node n $f(n) = g(n) + h(n)$
- w_{uv}: weight associated with the edge $u \to v$
- open_list : list of nodes to be traversed.
- update_list: list of nodes to be traversed while performing propagation for insertion/deletion of edges.
- optimal_cost: same as $f(n)$
- optimal_parent: back edge to the parent from which optimal cost is calculated
- Optimal_Path: source to destination optimal path, which can be traced by following optimal_parent from destination
- $f(n)_{old}$: approximate cost of reaching destination from source via node n before updates (insertion/deletion)
- $f(n)_{new}$: approximate cost of reaching destination from source via node n after updates (insertion/deletion)

We make certain claims regarding the state of the graph and prove them, which will be later useful in deriving some major algorithmic choices for processing addition and deletion of edges.

Lemma 2.1. *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 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 no 15 of Algorithm [**algo:a˙star**]:A*).So n should be chosen before d and which implies that n is already visited(expanded). which 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. *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.*

3 Incremental Processing

In the incremental setting, there are only edge-insertions, i.e., edge $u \to v$ is added in G where $u \in V$ and $v \in V$. In the graph in Figure 1 the Optimal Path

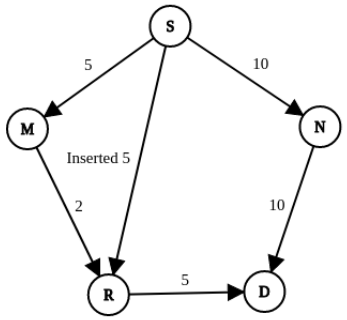


Figure 1: Insertion of an edge

before insertion was $S \to M \to R \to D$, when we insert edge $S \to D$, the cost f of node R changes and Optimal Path changes to $S \to R \to D$.

Lemma 3.1. *Insertion of an edge can not increase the cost of 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 has no effect on cost of any node of graph which implies the cost of Optimal Path remains same.

In both cases, cost of the Optimal Path doesn't increase, Hence Proved □

Lemma 3.2. *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 3.2.1. *If 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 3.2 that addition of such edges doesn't affect the optimal path. □

We know from Lemma 3.1 that insertions can't increase the cost of optimal path but they can decrease the cost, and there can be a new optimal path from the inserted edge. Also if we insert the $u \to v$ and $g(v)$ changes then all its descendants in the graph whose g are computed based on v are stale values. There can be two approaches to update the cost of 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.

3.1 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 parallelism and computation power of 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.

Algorithm 3: Propagation of Insertions

```
procedure: propagate_insertions( list< pair<u,v> > Inserts, N, E):
    update_list = make_update_list(Inserts);

    while update_list not empty:
        s = size(update_list)
        #array of N elements initialized to 0
        flag = array(N,0)
        #s parallel
        expand(update_list,s,flag)
        #create update list from flag
        update_list = generate_update_list(flag)

#GPU kernel
procedure: expand(update_list,s,flag):
    id = global_thread_id;
    node = update_list[id];

    for each child ch of node:
        lock(ch)
```

```
20          if f(ch) > g(node) + weight(node,ch) + h(ch):
21              f(ch) = g(node) + weight(node,ch) + h(ch)
22              optimal_parent[ch] = node
23              flag[ch] = 1
24          unlock(ch)
```

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 3.3. *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 belongs to sub-graph of v and whose $f(n)$ is decreased. According to Lemma 3.1 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 it's cost remains unchanged and which is the optimal cost of node n in new graph.

□

Lemma 3.4. *At the end of propagation $f(destination)$ is the optimal cost of reaching destination from source in the new graph G(V, E+Inserts).*

Proof. Follows from Lemma 3.3, *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) □

4 Decremental Processing

In Decremental setting there are only removal of edges $u \to v$ where $u \in V$ and $v \in V$. Addition of edges can only reduce the Optimal Cost (Lemma 3.1), whereas deletion of edges can increase the Optimal Cost (Lemma 4.1) 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.

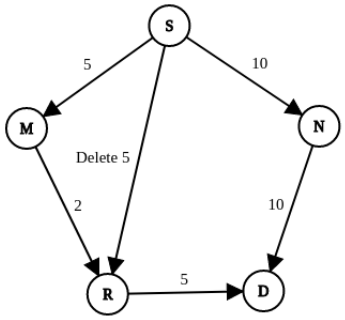


Figure 2: Deletion of an edge

In the example graph in Figure 2 the Optimal Path before deletion was $S \to R \to D$, when we delete the edge $S \to R$, the Optimal Path changes to $S \to M \to R \to D$.
Below we prove some important properties that holds when we remove edges from graph. The following properties holds when we have the Optimal Path for the graph before deletions.

Lemma 4.1. *Deletion of edge $u \to v$ where $u \in V$ and $v \in V$ can not decrease $f(v)$.*

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

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

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

Lemma 4.2. *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 deletion of such edges can only increase $f(v)$ so if there is a path from source $\to v \to$ destination then it's 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 4.3. *If edge(u,v) is deleted and optimal_parent(v)$\neq$u, then deletion of such edges doesn't affect the Optimal Path.*

Proof. If u is not the $\text{optimal\_parent}(v)$, then let $s = \text{optimal\_parent}(v)$, 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. □

4.1 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 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 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 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 $u \to v$, if $f(v) \neq \infty$ and optimal_parent(v)= u then for such edges we compute the cost of reaching v from its parent's and chose 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 child with respect to 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 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 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.

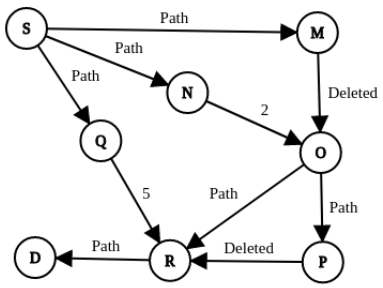


Figure 3: Batched deletion of multiple edges

In the example graph in Figure 3, the Optimal Path before deletion is $S \to M \to O \to P \to R \to D$. The edges $M \to O$ and $P \to R$ are deleted from

graph. So $R, O \in$ update_list, While propagating for O we might find R again from path $O \to R$ but R has already updated, if in that update R has chosen path $S \to M \to O \to R$ as its Optimal Path then we have stale value for $f(R)$ as $M \to O$ 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.

4.2 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. as given in the example below:

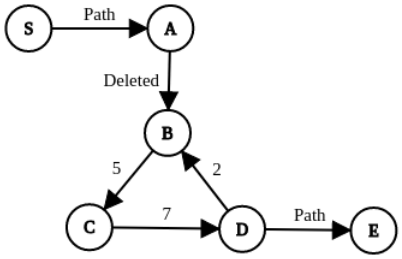


Figure 4: Deletion in Cyclic Graphs

In the example graph in Figure 4 the Optimal Path is from $S \to ABCD \to E$, where S is source and E is destination. When we remove the edge $A \to B$, and recompute the $f(B)$, there exist only one parent D so the $f(B)_{new}$ is computed by $f(B)_{new} = g(D) + 2 + h(B)$, which is old value of $g(D)$, as we remove $A \to B$, there is no path from $S \to D$ 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 stale value of $f(B)$ and 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 wrt 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.

Algorithm 4: Check Cycles

```
procedure: check_cycle(node,parent):
    current_node = node
    while there exists optimal_parent(current_node):
        if current_node == parent:
            return true
        current_node = optimal_parent(current_node)

    return false
```

4.3 Parallel Deletion and Cycle

When there are multiple deletions happening at the same time, there can be a cycle of optimal_parent, and thus when we do the above check, we get into an infinite loop also we defined optimal_parent as a non-cyclic path to start node. To eliminate such a cycle when we process deletion we have to do additional processing of removing such cycles.

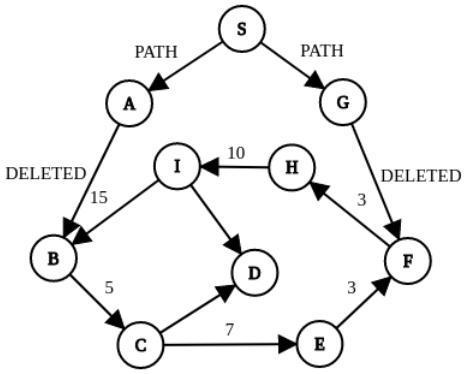


Figure 5: Parallel Deletion in Cyclic Graphs

In the example graph in Figure 5, the Optimal Path from $S \to D$ is $SABCD$,when we delete edges $A \to B$ and $G \to F$,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 the check happens at the same time and both reads the optimal_parent values before it gets changed by either one of them. Thus forming a cycle of optimal_parents which violates our definition of optima_parent, to 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 will take care of which edge actually belong to the cycle.

The propagation for deletion handling all the above cases is given below:

Algorithm 5: Propagation of Deletions

```
procedure: propagate_deletions( list< pair<u,v> > Deletions, E, N):
     update_list = make_update_list(Deletions);
     check_optimal_parent_cycle(update_list);

     while update_list not empty:
        s = size(update_list)
        #array of N elements initialized to 0
        flag = array(N,0)
        #s parallel
        expand(update_list,s,flag)
```

```
11
12          #create update list from flag
13          update_list = generate_update_list(flag)
14
15          #remove optimal_parent cycle
16          check_optimal_parent_cycle(update_list);
17
18  #GPU kernel
19  procedure: expand(update_list,s,flag):
20      id = global_thread_id;
21      node = update_list[id];
22
23      for each child ch of node:
24          lock(ch)
25
26          if f(ch) > g(node) + weight(node,ch) + h(ch):
27              f(ch) = g(node) + weight(node,ch) + h(ch)
28              optimal_parent[ch] = node
29              flag[ch] = 1
30
31          else if f(ch) < g(node) + weight(node,ch) + h(ch) and
                 optimal_parent[ch] == node:
32              for all parents p of ch:
33
34                  if check_cycle(ch,p) == true:
35                      continue
36
37                  if f(ch) > g(p) + weight(p,ch) + h(ch):
38                      f(ch) = g(p) + weight(p,ch) + h(ch)
39                      optimal_parent[ch] = p
40
41              flag[ch] = 1
42
43          unlock(ch)
```

Lemma 4.4. *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 new graph including deletions.*

Proof. Suppose we delete edge $u \to v$. Let node $n \in V$:
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 update_list we compute the least cost of reaching v from all of its remaining parent we also make sure that the parent we chose is such that its cost in not computed wrt 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 chose 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 4.2.

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 4.2 □

Lemma 4.5. *At the end of propagation $f(destination)$ might not be the optimal cost of reaching destination from source in new graph.*

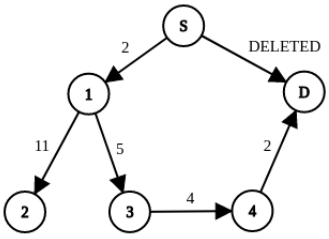


Figure 6

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 6 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 wrt 4 so $f(D) = \infty$, but int the new graph the optimal cost is $f(D) = 13$ with Optimal_Path $S \to 1 \to 3 \to 4 \to D$.
Thus in above graph $f(D)$ after the propagation of deletion is not the optimal cost in new graph. □

Lemma 4.6. *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 $\forall$ node $n \in$ open_list either $f(n)_{new}$ is the optimal_cost or $f(n)_{new} = \infty$, from Lemma 4.4. Also given $f(destination) < f(n)\ \forall\ n \in$open_list, As weights are non-negative so for all descendants of n $f($ 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. □

From Lemma 4.5 we know that after propagation for deletion $f(destination)$ is not the optimal cost, also from Lemma 4.6 we know that if $f(destination) < f(n)\ \forall\ n \in$ open_list then $f(destination)$ is the optimal cost, to satisfy the this condition we need to perform A* after propagation for deletion as after A* $f(destination) < f(n)\ \forall\ n \in$open_list and thus $f(destination)$ is the optimal cost.

Algorithm 6: Procedure for Deletions

```
1 procedure: Deletions(list< pair<u,v> > deleted_edges,N,E)
2     propagate_deletions(deleted_edges,N,E);
3     # A* from old saved state
```

```
4     GA*(open_list,closed_list,source,destination,K)
```

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 propagation. But as insertion and deletion require a different set of instruction it is a good choice to separate the propagation.

Algorithm 7: Separate Propagation

```
1 procedure: separate_propagation(list< pair<u,v> > inserted_edges,list<
      pair<u,v> > deleted_edges,N,E)
2     propagate_insertions(inserted_edges,N,E)
3     propagate_deletions(deleted_edges,N,E)
4     # A* from old saved state
5     GA*(open_list,closed_list,source,destination,K)
```

Lemma 5.1. *After separate propagation $f(destination)$ is the optimal cost of reaching destination from source in new graph including insertion and deletions.*

Proof. In separate propagation first we propagate for insertions and as from Lemma 3.4 at end of it we have $f(destination)$ as the optimal cost in new graph which includes all inserted edges. In the new graph, we propagate for deletions and then perform GA* so from Lemma 4.6 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 Insertion(section 3.1) and propagation of deletions (section 4.1) such some parts of them can be overlapped and 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.

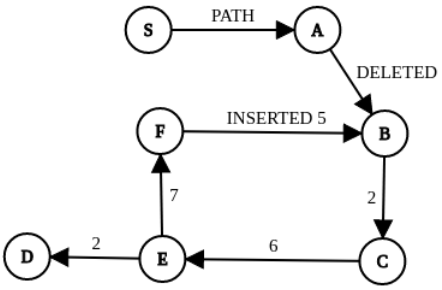


Figure 7: Cyclic graph with parallel insertion and deletion

In the graph in Figure 7 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 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 wrt 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.

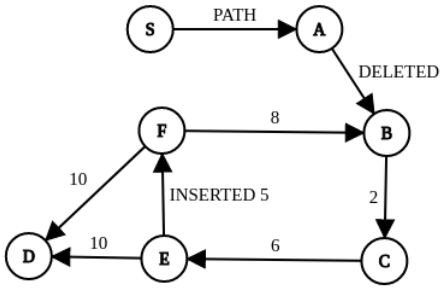


Figure 8: Infinite propagation due to formation of cycles

In the graph in Figure 8 $A \to B$ is deleted and $E \to F$ 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 $S \to C$ in new graph, also due to edge $f \to B$ 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 wrt B itself. This can be removed by doing the check mentioned in the above section 4.2.

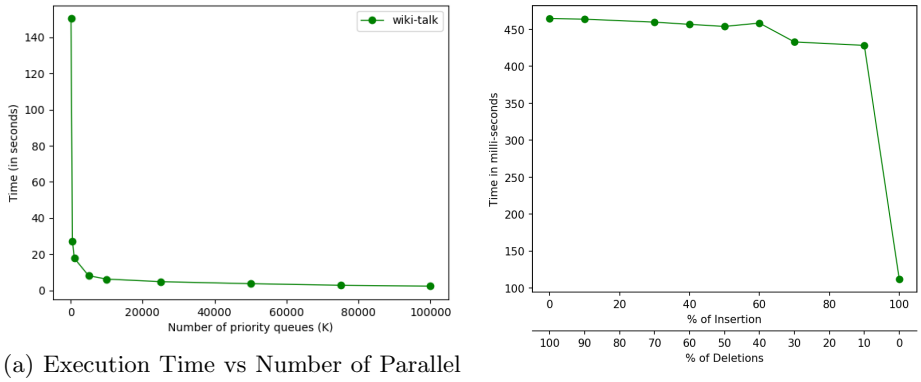
Algorithm 8: Propagation of Updates

```
procedure: propagate_updates( list< pair<u,v> > Deletions,list<
    pair<u,v> > Inserts, E, N):
    update_list = make_update_list_insertion(Inserts);
    update_list = append( update_list,
        make_update_list_deletion(Deletions) )

    while update_list not empty:
       s = size(update_list)
       #array of N elements initialized to 0
       flag = array(N,0)
       #s parallel
       expand(update_list,s,flag)
       #create update list from flag
       update_list = generate_update_list(flag)

#GPU kernel
procedure: expand(update_list,s,flag):
    id = global_thread_id;
    node = update_list[id];

    for each child ch of node:
        lock(ch)

        if f(ch) > g(node) + weight(node,ch) + h(ch):
            f(ch) = g(node) + weight(node,ch) + h(ch)
            optimal_parent[ch] = node
            flag[ch] = 1

        elif f(ch) < g(node) + weight(node,ch) + h(ch) and
            optimal_parent[ch] == node:
            for all parents p of ch:

                if check_cycle(ch,p) == true:
                    continue

                if f(ch) > g(p) + weight(p,ch) + h(ch):
                    f(ch) = g(p) + weight(p,ch) + h(ch)
                    optimal_parent[ch] = p

            flag[ch] = 1

        unlock(ch)
```

6 Experimental Results

In this section, we have compared the performance of our algorithm with multiple GPU A* algorithms. We also discuss how various parameters affect the performance of our algorithm(DA*). The GPU we used the experiments is a Tesla K40c which has 15 SM each having 192 cores(total 2880 cores), it has 12 GB of Global Memory and has a clock rate of 740 MHz.

The performance of GPU A*(GA*)[15] is dictated by the expansion of nodes simultaneously which is restricted by how many parallel priority queues are created. As we increase the number of priority queues the execution time decreases because we process a large number of nodes in parallel. The trend follows as shown in Figure 9a.



(a) Execution Time vs Number of Parallel Priority Queues

(b) Execution Time vs Edge density

Figure 9

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 insertions, which generally increases as we increase the number of edges being inserted. Figure 10 shows the increase in execution time, we can also infer that the increase is linear for certain part of in each sub-figure. In Figure 10c there is a dip then rise, it is because initially the destination node was unreachable so it has to process all nodes and as we keep inserting edges the source-destination path is created and it then became shorter as more edges got inserted in graph, after a while due to increase in 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.

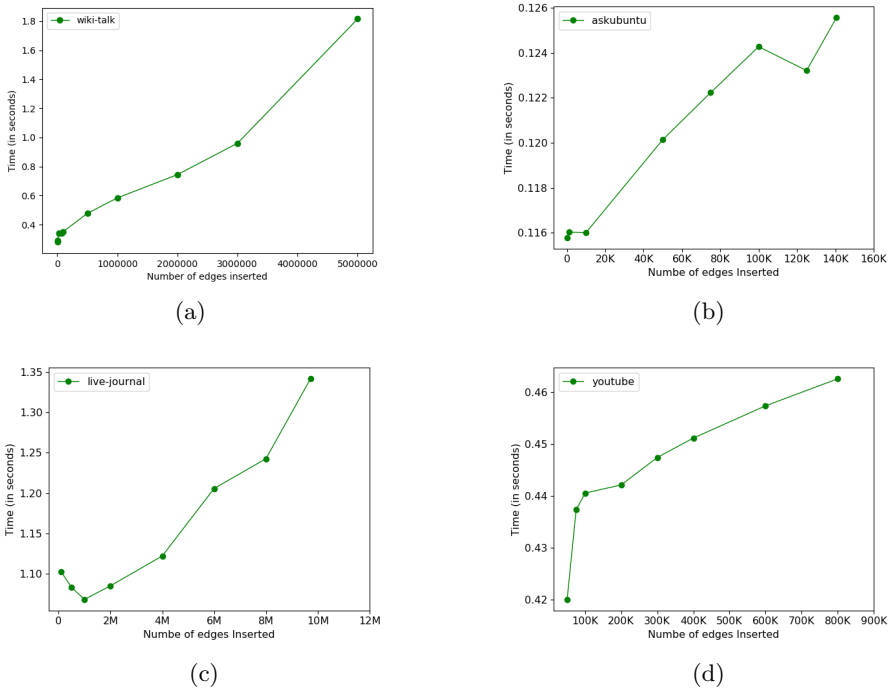


Figure 10: Execution time when graph is only subjected to Insertions

Similarly for deletion as deletion size increases we need to process more nodes,and contention at lock also increases thus increases the execution time, as shown in Figure 11. We also observe that there are certain points where the execution time increase drastically in each Figure 11a, 11b, 11c, 11d It might be due to the sudden increase in number of edges i.e $(u \rightarrow v)$ deleted which are part of optimal path to the node v, which increase the amount of processing required.

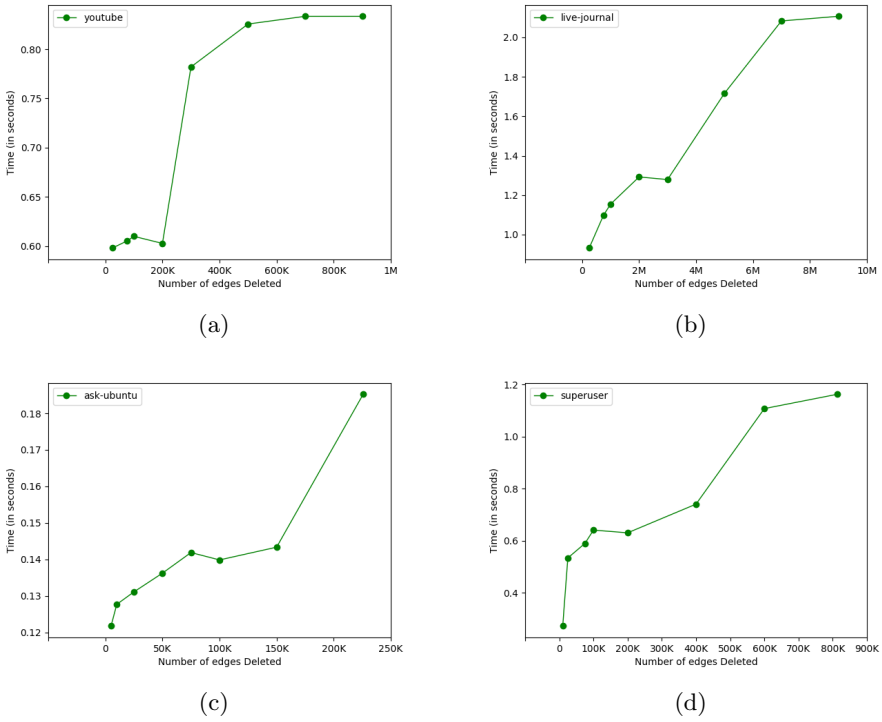


Figure 11: Execution time when graph is only subjected to Deletions—

In a dynamic setting when 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 12. We can also observe that the sub-figures looks more similar to Figure 11 that is because propagation for deletion takes more computation power than for insertion.

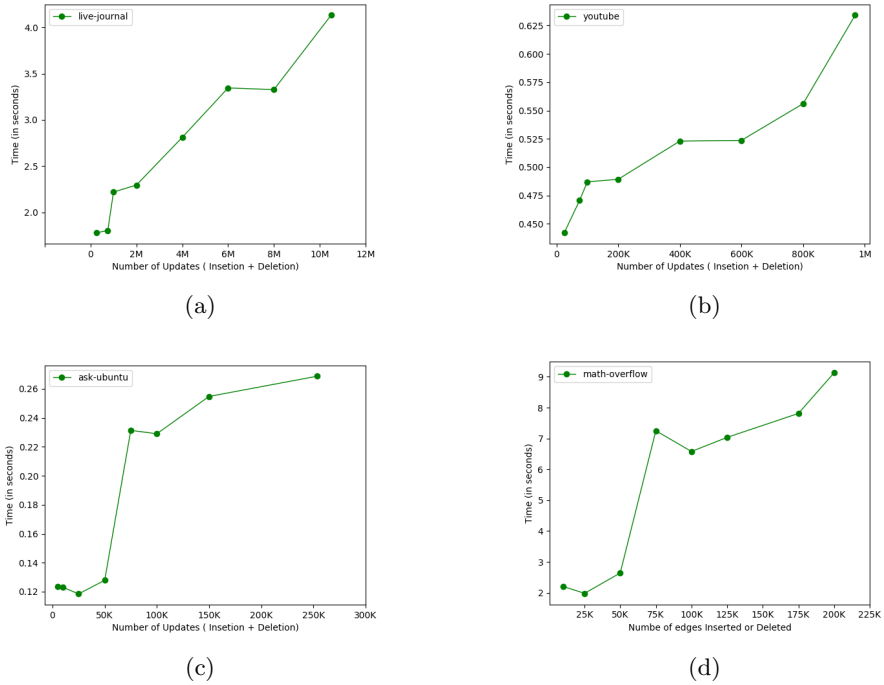
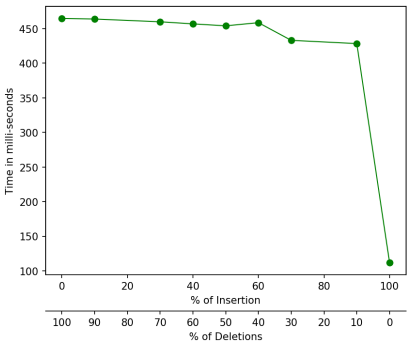
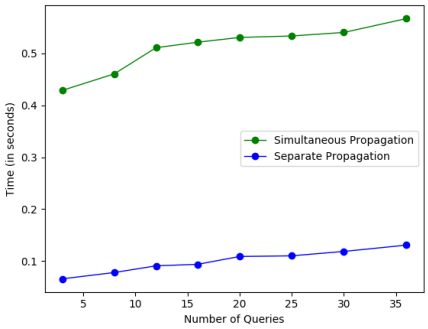


Figure 12: Execution time vs Number of Updates

The amount of execution time needed to process deletion is much larger than inserts. As shown in Figure 13a where the total number of updates is kept constant but the amount 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 13b we compare the two variants of DA*, simultaneous propagation and separate propagation, separate propagation outperforms simultaneous propagation as the later require more computation for the additional checks as discussed in section 5.2.1 and section 5.2.2.



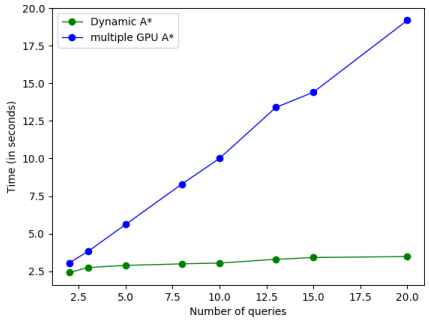
(a) Run Time vs % Insert/Delete



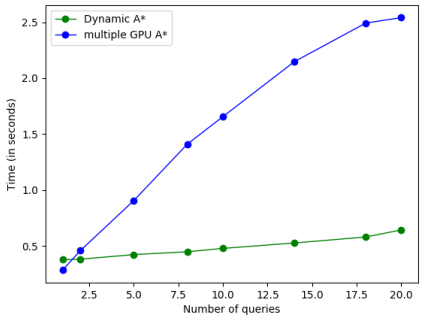
(b) Different version of DA* propagation

Figure 13

Here we compare the performance of our algorithm than by doing GPU A* repeatedly. The below Figure 14 shows how execution time varies when we increase the number of queries (finding the path from source to destination after some update) increases. Performing GA* 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 can get is proportional to the number of queries we preform, as the number of queries increases we get the better and better speed up. In the Figure 14a updates only consist of *inserts* while the Figure 14b consisting of both insertion and deletion. In Figure 14b we can infer that when the number of queries is 1 multiple GA* perform better than Dynamic A* but as we increase the number of queries Dynamic A* perform better than multiple GA*. The speedup we get by applying our algorithm 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 [9] results shown in table 1. We got 37x speedup for 100 queries.



(a) Only Insertions



(b) Both Insertions and Deletions

Figure 14: Execution Time vs Number of Queries

Table 1: Comparison of DA* with repeated GA* (Time in seconds)

No.	Graph	N	E	Queries	DA*	repeated GA*	speedup
1	live-journal	3,997,962	34,681,189	10	6.01	33.93	5×
2	wiki-talk	1140149	7833140	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×

7 Applications

In this section, we will discuss how our algorithm can be applied to problems with different characteristics which uses A* to solve the problem.

7.1 Wireless Sensor Networks

Wireless sensor networks (WSN) are a group of spatially dispersed and dedicated sensors(sensor nodes) for monitoring and recording the physical conditions of the environment and organizing the collected data at a central location(base station). WSNs measure environmental conditions like temperature, sound, pollution levels, humidity, wind, etc. Sensor nodes are deemed to be resource-constrained in terms of energy, communication range, memory capacity, and processing capability. Sensor nodes are also light-weight, low power and are operated by a small battery. The energy they lose is proportionate to the distance of communication. There is no way to recharge these batteries in most of the cases. Generally in the routing algorithm, the best path is chosen for transmission of data from source to destination. Over the period of time, if the same path is chosen for all communications in order to achieve better performance in terms of quick transmission time, then those nodes which are on this path will get drained faster.
There are many routing algorithms which try to increase the lifetime of the network like SPIN-EC [8], LNDIR [11], S. AlShawi et.al [1] proposed a fuzzy A* algorithm to increase the lifetime of network, Keyur Rana and Mukesh Zaveri [13] proposed another algorithm which uses A* to find the best route while taking account of the residual energy at each node. Ali Ghaffari [6] 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.

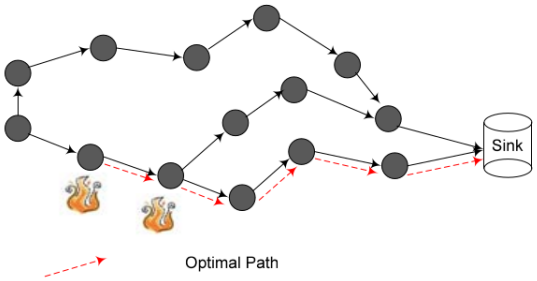


Figure 15: Energy-efficient data forwarding in wireless sensor networks[6]

The Energy associated with sensor nodes is constantly changing as packets 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 [6] and instead of doing repeated A* we used our DA* to find the optimal path.

In EERP [6] author proposed a scheme where the value of function $f(x)$ is equal to the node cost of node n, and the intention is to forward data packets to the next neighbor node which has higher residual energy, higher free buffer, and higher packet reception rate. To achieve this, they made use of the aggregated weight of the above-mentioned routing parameters. They define the aggregated weight of a next neighbor node as the sum of normalized weights of its routing metrics as follows:

$$g(n) = \text{Min}(\alpha(\frac{E_{ini}(n)}{E_{res}(n} + \beta(\frac{N_t(n)}{N_r(n)}) + \gamma(\frac{B_{ini}(n)}{B_f(n)})) \tag{1}$$

Where $E_{res}(n)$ and $E_{ini}(n)$ are residual and initial energy of node n respectively, $N_r(n)$ and $N_t(n)$ are the number of transmitted and received packets respectively, $B_f(n)$ and $B_{ini}(n)$ referred to the number of free and initial buffer of node n respectively, α, β and γ are weight parameters and $\alpha + \beta + \gamma = 1$. And the value of $h(n)$ function can be calculated as:

$$h(n) = \frac{d(n,s)}{\text{avg}(d(n,j))} \tag{2}$$

Where $d(n,s)$ is the distance between the node and sink node and $avg(d(n,j))$ is the average distance between the node n and its one hop neighbouring nodes(j). To build the routing table authors[13] proposed algorithm where A* is performed for each node to find the best path to sink, after each round.

Algorithm 9: Pseudo Code for Proposed Efficient Routing Algorithm[13]

```
1 Input: Sensor Network
2 Output : Life of Sensor Network in terms of rounds
3 BEGIN
```

```
    InitilizeNetwork()
    EstimateDistance()           // finds distance to the sink
    WHILE NOT END_ASTAR()        // N-of-N metric [11]
        InitializeSolArray()     // initialize solution
                                 //array to store routing schedule
        FOR each node i in the Network DO
            CreateTree (i)       //using A-Star algorithm
            PrepareSolArray()    //prepare routing schedule
        END FOR
        BroadcastSolution()     // BS broadcasts routing schedule
        UpdateEnergy()           //Energy update for relay nodes
        CountRound = CountRound + 1 //count n/w life
    END WHILE
    PRINT CountRound             //print n/w lifetime in
END
```

Instead of performing A* for each node to find the path, we take the idea from D* [2] where to find the path the search starts from the destination node towards the source node. So to find paths to all nodes our we start A* from sink node and end when we have visited all nodes. After BroadCastSolution() we UpdateEnergy() and then add all those edges to update_list whose weights are modified and then on it we apply our propagation methods described in the above sections to find the optimal_path and thus forming new routing table at the base station.

Algorithm 10: EERP with DA*

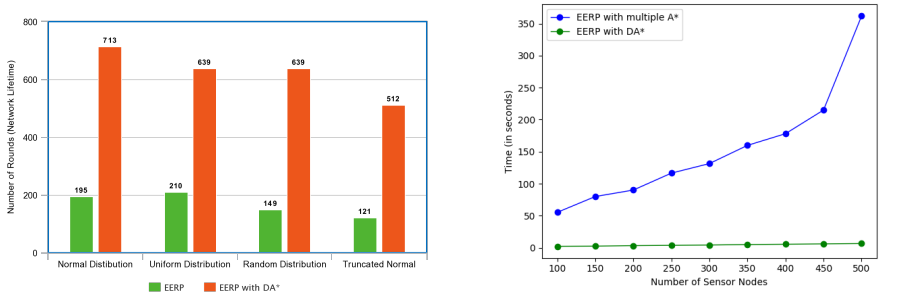
```
Input: Sensor Network
Output: Life of Sensor Network in terms of rounds
BEGIN
    InitilizeNetwork()
    EstimateDistance()       // finds distance to the sink
    InitializeSolArray()      //array to store routing schedule

    ASTAR_From_Sink()       //finds path to all nodes from sink
    BroadcastSolution()     // BS broadcasts routing schedule
    UpdateEnergy()            //Energy update for relay nodes
    CountRound = CountRound + 1

    WHILE NOT END_ASTAR()
        Create_UpdateList() //list of edges whose weight is changed
        Propagation()       //Propagate change using methods of DA*
        Update_SolArray()   //Update routing table with new path
        BroadcastSolution()      // BS broadcasts routing schedule
        UpdateEnergy()          //Energy update for relay nodes
        CountRound = CountRound + 1
    END WHILE
    PRINT CountRound             //print n/w lifetime
END
```

The energy consumption model of the EERP [6], the authors used the first order radio model which is the typical model in the area of routing protocol evaluation in WSN [14]. According to this model, the energy consumed for transmitting

and receiving k-bit data can be calculated as follows [14]:

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

Here we compare the life expectancy of network and performance of algorithms EERP [6], A *[7], DA* for routing the packets in WSN. All algorithms are implemented in CUDA and ran on Tesla K40c a keepler based GPU with 15 SMs and 192 SP per SM for a total of 2880 SPs.



(a) Network Lifetime Comparison

(b) Execution Time vs Number of Nodes

Figure 16

In Figure 16a we compare A* routing algorithm with EERP [6] with different node (sensor node) distribution. For all the distributions we have taken that sensor nodes area dispersed in are of 100x100 m^2, and 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 3x in each distribution. Most of the real-world networks follow a normal distribution as more sensor nodes are concentrated near the base station (sink). The Figure 16b shows how the execution time of EERP is boosted by using our algorithm, For 100 nodes in network, we get 24x speedup, as the number of nodes in network increases the execution time of EERP increases much faster than EERP with DA*, with 500 nodes we get 55x speedup. Its because our algorithm scales well wrt the number of updates (here number of rounds) and size of graph compared to repeating A*, as we only update the parts of the graph which are affected by the change.

Error in bar plot naming

8 Related Work

A* [7] is one of the widely used path-finding algorithms, Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968. It can be seen as an extension of Dijkstra's algorithm where $h(x) = 0\ \forall x, x \in V$. A* achieves better performance by using heuristics to guide its search. What makes it different from the greedy best-first algorithm is it takes already traveled distance $g(x)$ into account.
D* [2], focused D* [3] and D* lite [10] are a family of incremental search algorithms. The original D* [2] by Anthony Stentz is an informed incremental

search algorithm. Focused D*[3] is an informed incremental heuristic search algorithm by Anthony Stentz that combines ideas of A* [7] and the original D*[2]. Focused D* resulted from further development of the original D*. D* Lite [10] is an incremental heuristic search algorithm by Sven Koenig and Maxim Likhachev that builds on LPA* [5], an incremental heuristic search algorithm that combines ideas of A* and Dynamic SWSF-FP [12]. All three search algorithms solve the same assumption-based path planning problems, including planning with the free space assumption [4] where a robot has to navigate to given goal coordinates in unknown terrain. So in all the three algorithms, the graph is changing while we try to find the shortest path to the destination, this is a major difference between our algorithm and D* as we try to find the shortest path to destination after some changes(insertion + deletion) has occurred since our last search. Another major change is in D* only the cost of edges is changed while the number of edges remains fixed But in our case edges can be removed as well as added and the cost change can be emulated as deleting that edge and later adding the same edge with the new cost. The name D* comes from the term "Dynamic A*" because the algorithm behaves like A* except that the arc costs can change as the algorithm runs.

9 Conclusion and Future Work

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 with time. 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 due to the fact that both insertion and deletion requires a different set of instruction to process, separate propagation outperforms simultaneous propagation.
In the future, the possibility of Dynamic A* in a multi-GPU environment can also be explored. There can be algorithmic development in dynamic bi-directional A* on GPU. A* is used in a wide variety of applications and many such applications are based on graphs that are dynamic in nature. In the future, many such applications like maze solver, SCPR (structural based computational protein design) and codon optimization can be modified to use DA* to boost their performance.

References

[1] I. S. AlShawi et al. "Lifetime enhancement in wireless sensor networks using fuzzy approach and A-star algorithm". In: *Sensors Journal, IEEE* 12 (2012).

[2] Stentz Anthony. "Optimal and Efficient Path Planning for Partially-Known Environments". In: *Proceedings of the International Conference on Robotics and Automation* (1994).

[3] Stentz Anthony. "The Focussed D* Algorithm for Real-Time Replanning". In: *Proceedings of the International Joint Conference on Artificial Intelligence* (1995).

[4] Koenig S. Smirnov Y. Tovey C. “Performance Bounds for Planning in Unknown Terrain”. In: *Artificial Intelligence* (2003).

[5] Koenig S. Likhachev M. Furcy D. “Lifelong Planning A*”. In: *Artificial Intelligence* (2004).

[6] Ali Ghaffari. “An Energy Efficient Routing Protocol for Wireless Sensor Networks using A-star Algorithm”. In: *Journal of applied research and technology* (2014).

[7] Raphael B Hart P Nilsson N. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Trans. Syst. Science and Cybernetics* (1968).

[8] H. Balakrishnan Kulik W.R. Heinzelman. “Negotiation–Based Protocols for Disseminating Information in Wireless Sensor Networks”. In: *Wireless Networks* 8 (2002).

[9] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[10] Koenig S. Likhachev M. “Fast Replanning for Navigation in Unknown Terrain”. In: *Transactions on Robotics* (2005).

[11] Vyacheslav Zalyubovskiy Muhammad K Shahzad Dang Tu Nguyen. “LNDIR: A lightweight non-increasing delivery-latency interval-based routing for duty-cycled sensor networks”. In: *International Journal of Distributed Sensor Networks* 14(4) (2018).

[12] Reps T. Ramalingam G. “An incremental algorithm for a generalization of the shortest-path problem”. In: *Journal of Algorithms* (1996).

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

[14] A. Chandrakasan W. R. Heinzelman and H.Balakrishnan. “Energy-efficient communication protocol for wireless micro sensor networks”. In: *System Sciences, Proceedings of the 33rd Annual Haw International Conference* (2000).

[15] Yichao Zhou and Jianyang Zeng. “Massively Parallel A* Search on a GPU”. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015).

[16] Wikipedia A* Algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm.

[17] Wikipedia D* Algorithm. https://en.wikipedia.org/wiki/D*.