



A* for Dynamic Graphs on GPU

By: Lokesh Koshale (CS15B049)

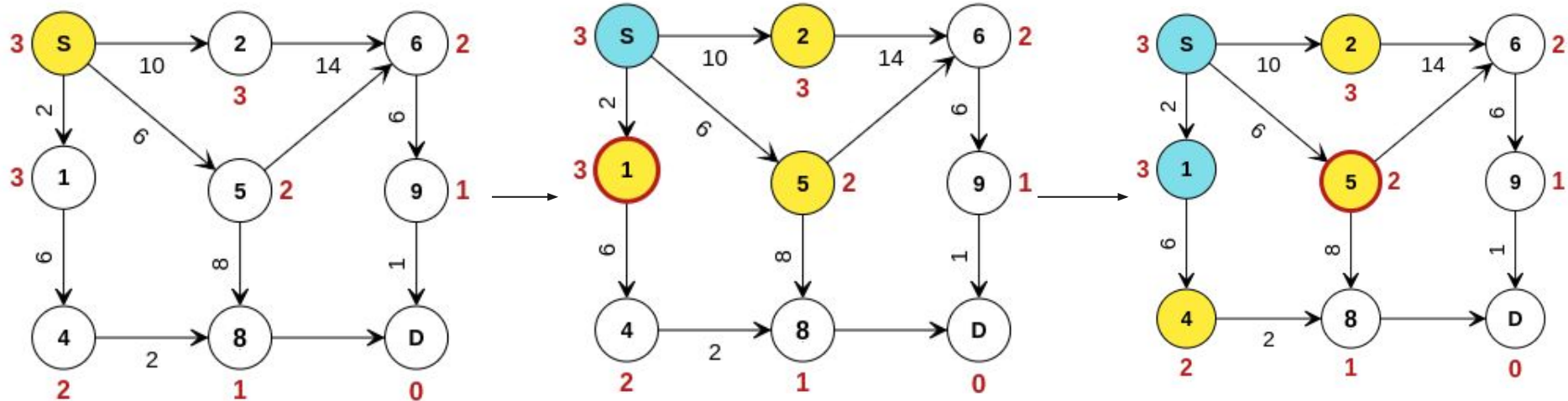
A* Algorithm





- A* is one of the widely used path-finding algorithms, developed by Peter Hart, Nils Nilsson and Bertram Raphael in 1968.
- It uses heuristic to guide search while ensuring that it will compute a path with minimum cost.
- $f(n) = g(n) + h(n)$.
- A* is used in wide variety of applications from path finding in robotics, solving puzzle and mazes to designing novel proteins and optimizing codons in DNA sequences.

1. **Initialize:** set $OPEN=[s]$, $CLOSED=[]$, $g(s)=0$, $f(s)=h(s)$
2. **Fail:** If $OPEN=[]$, then terminate and fail
3. **Select:** Select a state with minimum cost n , from $OPEN$ and save in $CLOSED$
4. **Terminate:** If $n \in G$ then terminate with success and return $f(s)$
5. **Expand:** For each successors m of n
For each successor, m , insert m in $OPEN$ only if
 if $m \notin [OPEN \cup CLOSED]$
 set $g(m) = g(n) + C[n, m]$
 Set $f(m) = g(m) + h(m)$
 if $m \in [OPEN \cup CLOSED]$
 set $g(m) = \min\{g[m], g(n) + C[n, m]\}$
 Set $f(m) = g(m) + h(m)$
 If $f[m]$ has decreased and $m \in CLOSED$ move m to $OPEN$
6. **Loop:** Goto step 2

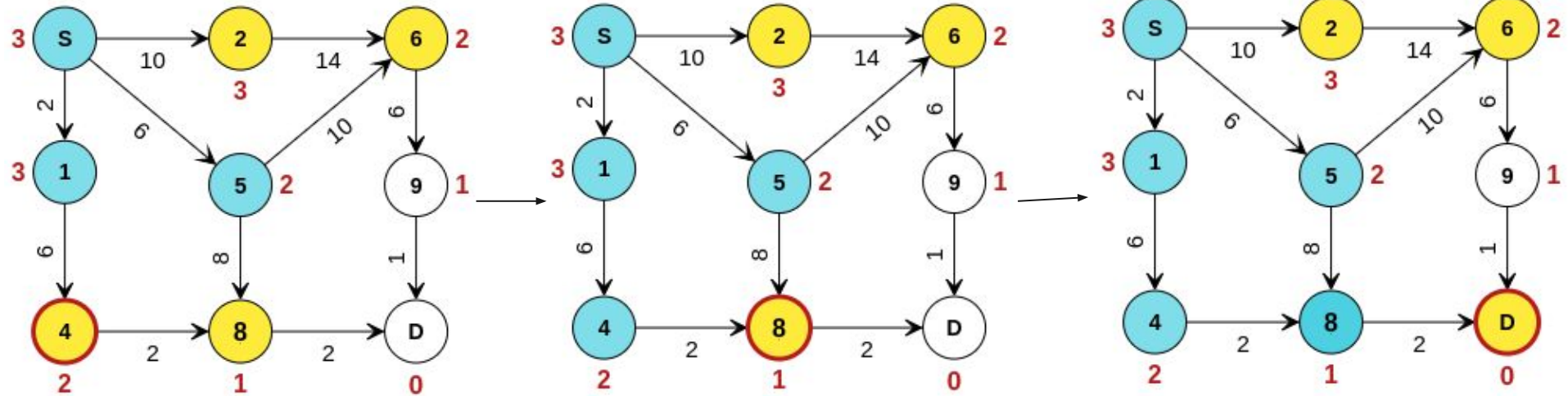
A* Algorithm: Example



 Nodes in Open List

 Nodes in Closed List

A* Algorithm: Example



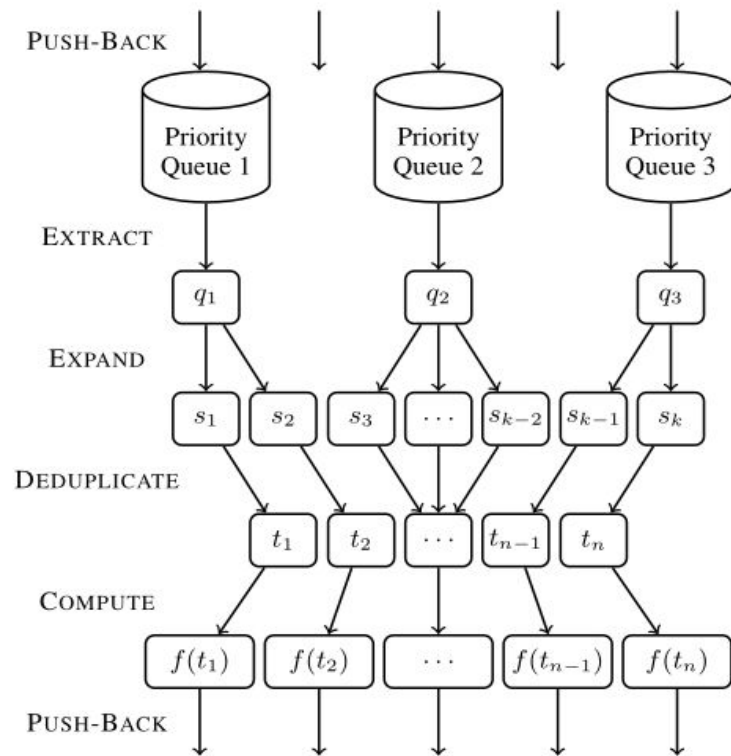
Nodes in Open List



Nodes in Closed List

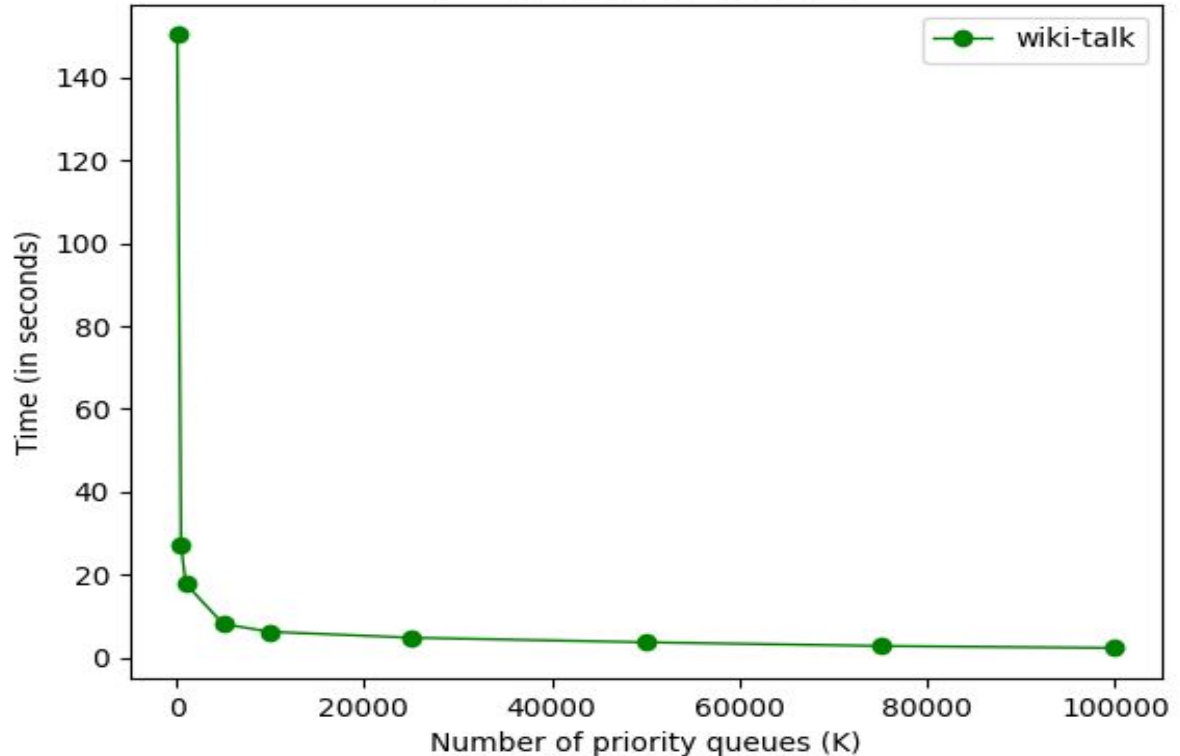
A* algorithm on GPU

- Proposed in paper “Massively Parallel A* Search on a GPU” by Yichao Zhou and Jianyang Zeng, 2015.
- Priority queue operations are the most time-consuming parts in A* search, because sequential operations are inefficient for a GPU processor.
- Instead of just using one single priority queue for the open list, the authors allocate a large number of priority queues during A* search.
- Each time the authors extract multiple states from individual priority queues, which thus parallelizes the sequential part of the A* algorithm.



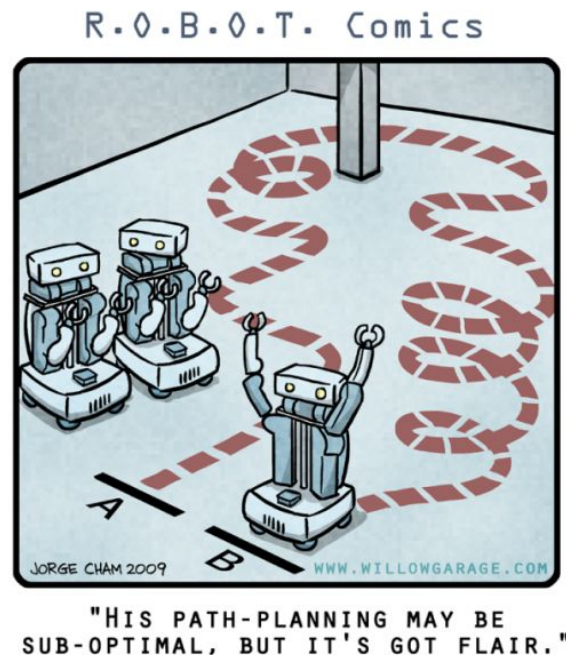
GPU A* : Execution Time vs Number of Priority Queues

As number of parallel priority queue increases, execution time decreases, because more number of nodes can be expanded simultaneously.



Related work

- D^* , D^* Lite and focused D^* is a family of incremental search algorithms.
- All three search algorithms solve the same assumption-based path planning problems, including planning with the free space where a robot has to navigate to given goal coordinates in unknown terrain.
- The graph is continuously changing while we try to find the shortest path to the destination, i.e. cost of edges is changed and also the number of edges remains fixed.
- The major difference between our algorithm and D^* is we allow addition/deletion of edges, and we find the optimal path from source to destination after every update..



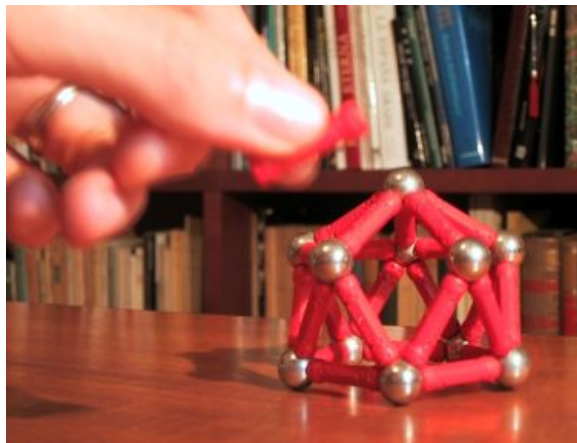
Dynamic Graphs

Graphs that are subject to an Update operation.

Typical Updates: $\text{Insert}(u,v)$
 $\text{Delete}(u,v)$
 $\text{SetWeight}(u,v,w)$

Partially dynamic problems: Graphs subject to insertions only, or deletions only, but not both.

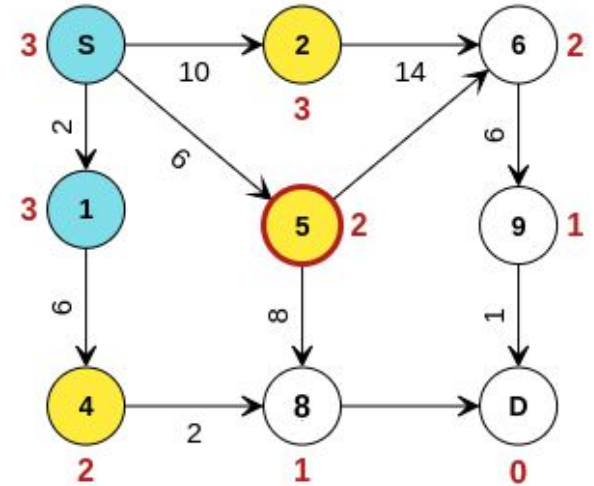
Fully dynamic problems : Graphs subject to intermixed sequences of insertions and deletions



Property of A* search

Lemma 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).

Since everytime we extract the node with lowest cost from open_list, if node n is extracted at step t , then all nodes in open_list should have higher cost than n , which implies all nodes with less cost are already expanded.



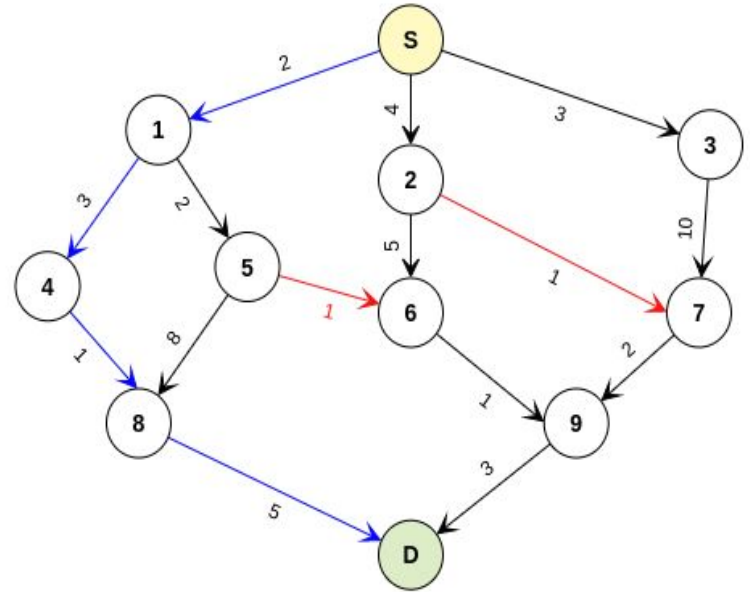
(Proof)

Partially Dynamic: Insertions only

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

Lemma 2: Insertion of an edge can not increase the cost of source to destination Optimal Path. [\(Proof\)](#)

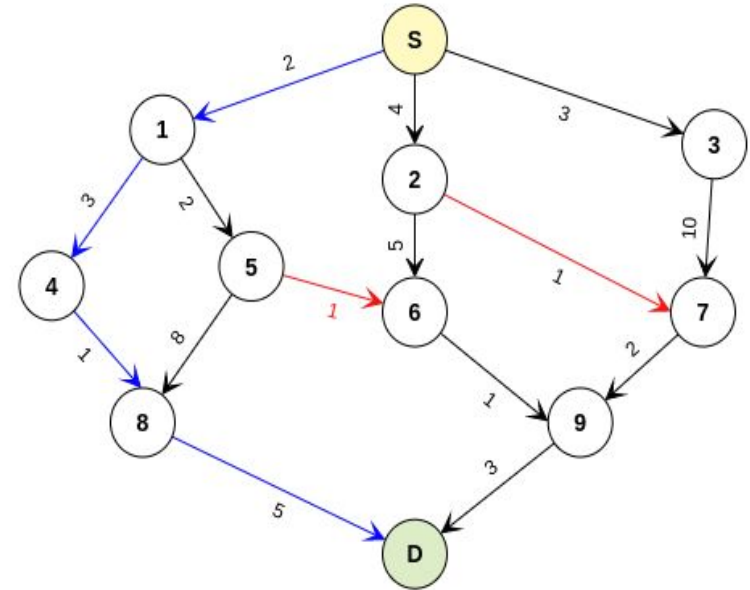
Lemma 3: If we add an edge $u \rightarrow v$ and $f(u) > f(\text{destination})$ then addition of this edge will not affect the Optimal Path. [\(Proof\)](#)



In graph above, optimal path from S to D is colored blue, and edges $5 \rightarrow 6$ and $2 \rightarrow 7$ is inserted.

Processing Insertions

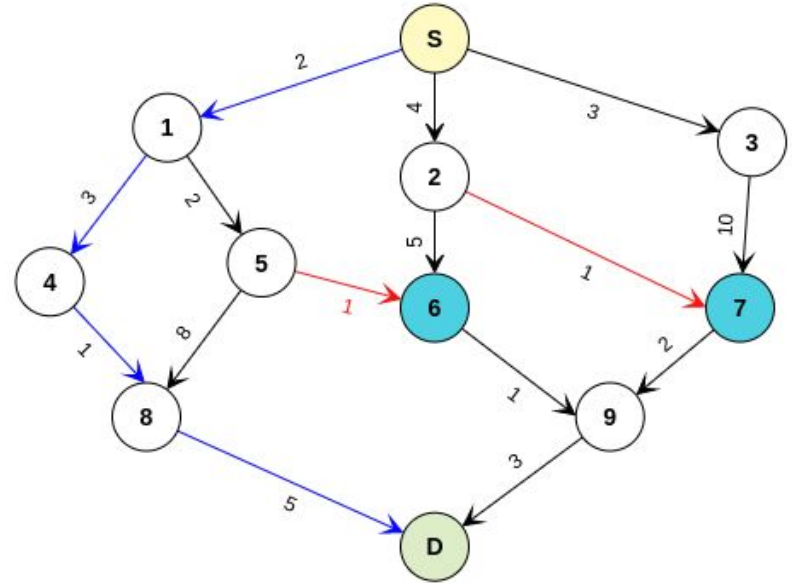
1. For edges(u,v) inserted, add node v to update_list, if $f(v)_{\text{new}} < f(v)_{\text{old}}$.
2. While update_list not empty:
 - a. Extract node n from update_list.
 - b. For each child of n:
 - i. lock(child)
 - ii. if $f(\text{child})_{\text{new}} < f(\text{child})_{\text{old}}$, add child to update_list.
 - iii. unlock(child)



In graph above, optimal path from S to D is colored blue, and edges $5 \rightarrow 6$ and $2 \rightarrow 7$ is inserted.

Propagating Insertions

1. For edges (u,v) inserted, add node v to `update_list`, if $f(v)_{\text{new}} < f(v)_{\text{old}}$.
2. While `update_list` not empty:
 - a. Extract node n from `update_list`.
 - b. For each child of n :
 - i. `lock(child)`
 - ii. if $f(\text{child})_{\text{new}} < f(\text{child})_{\text{old}}$, add child to `update_list`.
 - iii. `unlock(child)`

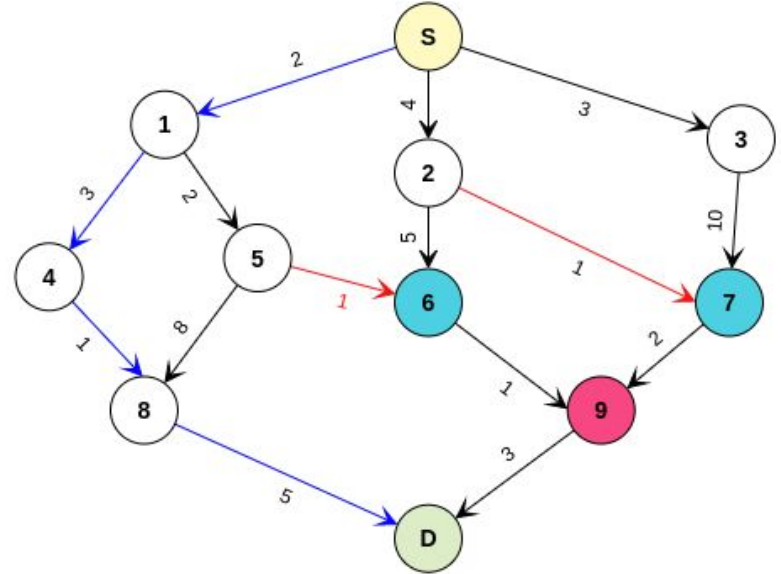


Add {6,7} to update list.

Propagating Insertions

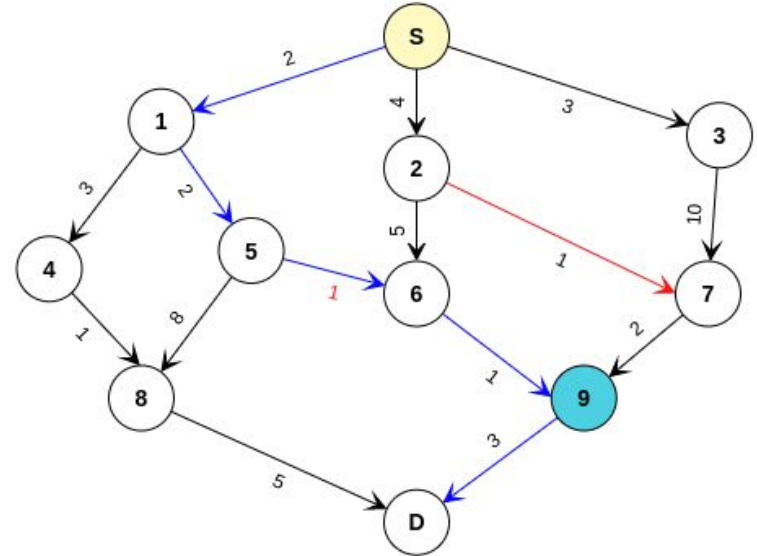
1. For edges(u,v) inserted, add node v to update_list, if $f(v)_{\text{new}} < f(v)_{\text{old}}$.
2. While update_list not empty:
 - a. Extract node n from update_list.
 - b. For each child of n:
 - i. lock(child)
 - ii. if $f(\text{child})_{\text{new}} < f(\text{child})_{\text{old}}$, add child to update_list.
 - iii. unlock(child)

Array f is declared as volatile, so that if node 6 changes f(9) then node 7 will read the updated value and not the cached value of f(9).



Node 6 and 7 both tried to update node 9, therefore we require a lock.

- iii. `unlock(child)`



Node 9 is added to update_list, in next iteration $f(D)$ is updated by 9, and thus Optimal Path is changed.

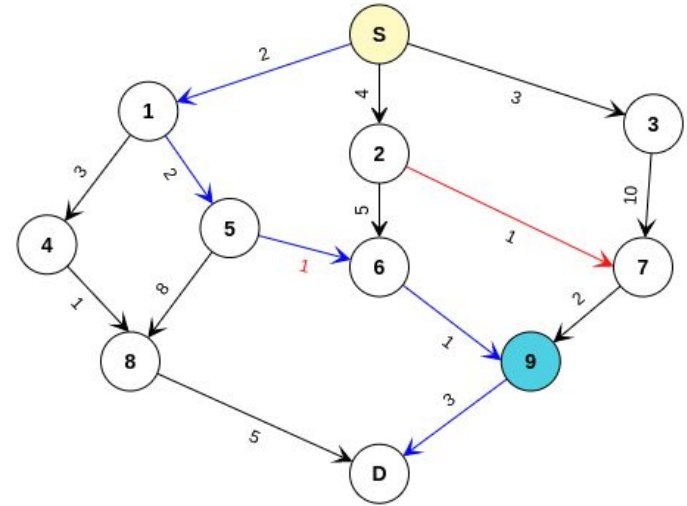
Propagating Insertions

Lemma 4: At the end of propagation, all the nodes $\in V$ will have the latest cost $f()$ which is the optimal cost in the new graph including inserts.

(Proof)

Let's take a node n , from the graph after insertion of $\text{edge}(u,v)$ and its propagation:

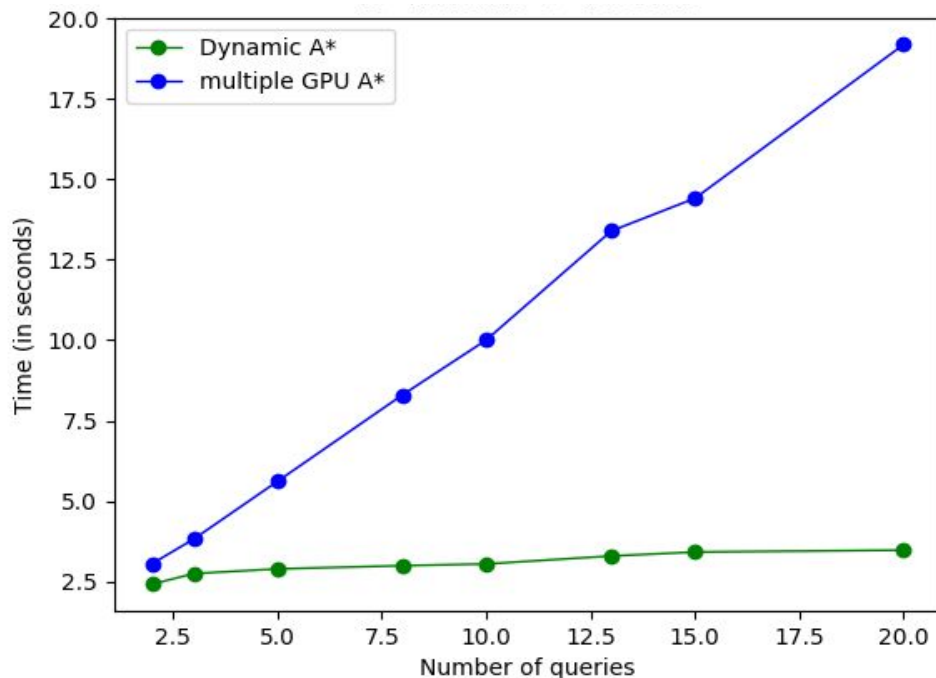
- If n is not in subgraph of v , then it is not affected by addition of this edge.
- If n is in subgraph of v , and it is added to update_list at some iteration then its cost is decreased which is the new optimal cost.
- If n is in subgraph of v but its not added in update_list then there exist a path with lower cost from source to n which doesn't include $\text{edge}(u,v)$.



Experimental Result: Insertions

The plot compares re-executing A* after each update with performing propagation for insertion after each update.

We observe that as the number of times a graph is subjected to update (an query) increases our algorithm performs better.



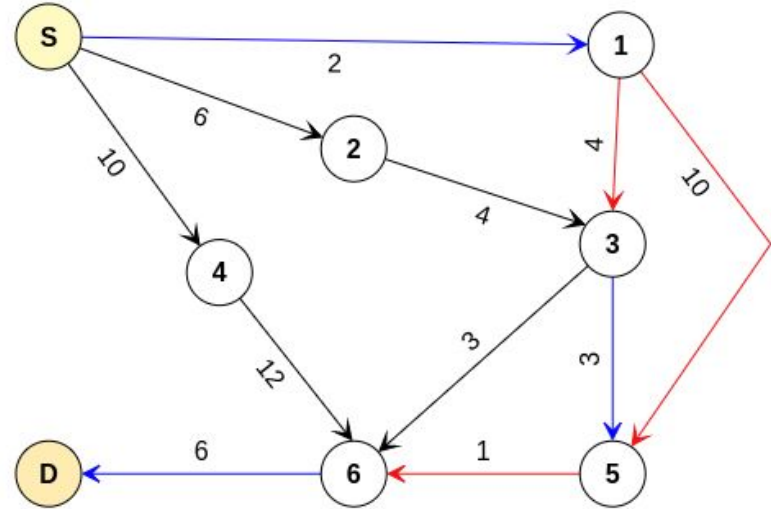
Partially Dynamic: Deletions only

In Decremental setting there are only removal of edges $u \rightarrow v$ where $u \in V$ and $v \in V$.

Lemma 5: Deletion of edge $u \rightarrow v$ where $u \in V$ and $v \in V$ can not decrease $f(v)$. [\(Proof\)](#)

Lemma 6: If edge (u,v) is deleted and $\text{optimal_parent}(v) \neq u$, then deletion of such edges doesn't affect the Optimal Path. [\(Proof\)](#)

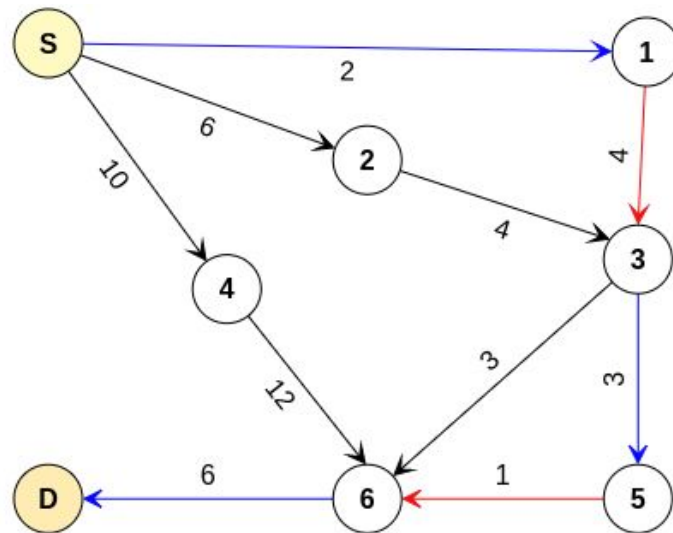
Lemma 7: If edge $(u,v) \notin \text{Optimal Path}$, then deletion of such edges doesn't affect the Optimal Path



In graph above, optimal path from S to D is colored blue, and edges $1 \rightarrow 3$, $1 \rightarrow 5$ and $5 \rightarrow 6$ are deleted. Deletion of $1 \rightarrow 5$ doesn't affect optimal path.

Processing Deletions

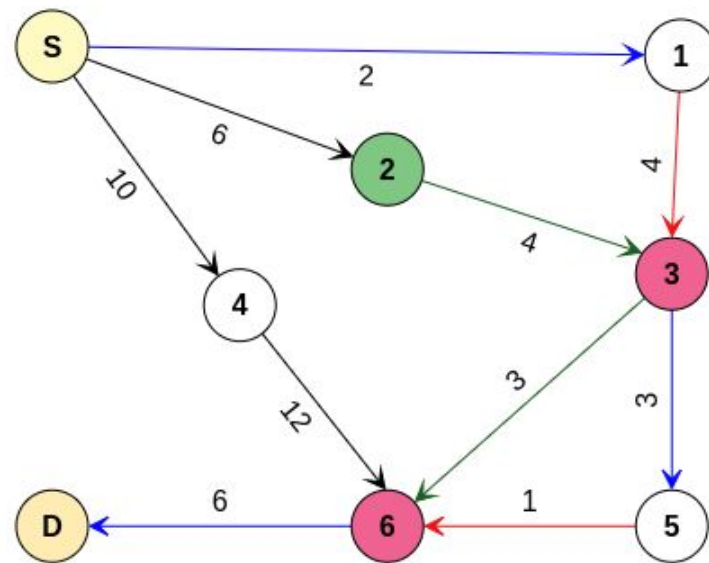
1. For each deleted edge $u \rightarrow v$:
 - a. if $f(v) \neq \infty$ and $\text{optimal_parent}(v) = u$, compute $f(v)$ from all of its parents and chose the least one as $\text{optimal_parent}(v)$ and update $f(v)$,
 - b. if v doesn't have any parent then $f(v) = \infty$
 - c. add v to update_list .
2. While update_list is not empty:
 - a. Extract node n from update_list .
 - b. For each child of n such that $\text{optimal_parent}(\text{child}) = n$:
 - i. If $f(\text{child}) > f(\text{child})_{\text{new}}$ then compute $f(\text{child})$ from all parents of child node and chose the least one as optimal_parent and add child to update_list .



In graph above, optimal path from S to D is colored blue, and edges $1 \rightarrow 3$, $1 \rightarrow 5$ and $5 \rightarrow 6$ are deleted.

Processing Deletions

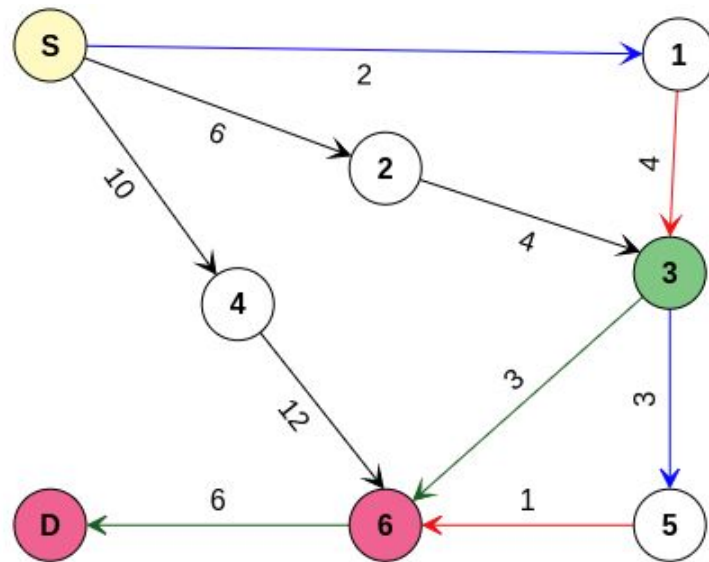
1. For each deleted edge $u \rightarrow v$:
 - a. if $f(v) \neq \infty$ and $\text{optimal_parent}(v) = u$, compute $f(v)$ from all of its parents and chose the least one as $\text{optimal_parent}(v)$ and update $f(v)$,
 - b. if v doesn't have any parent then $f(v) = \infty$
 - c. add v to update_list .
2. While update_list is not empty:
 - a. Extract node n from update_list .
 - b. For each child of n such that $\text{optimal_parent}(\text{child}) = n$:
 - i. If $f(\text{child}) > f(\text{child})_{\text{new}}$ then compute $f(\text{child})$ from all parents of child node and chose the least one as optimal_parent and add child to update_list .



Add 3,6 to update list, with $\text{optimal_parent}(3) = 2$, and $\text{optimal_parent}(6) = 3$. (reads stale value)

Propagating Deletions

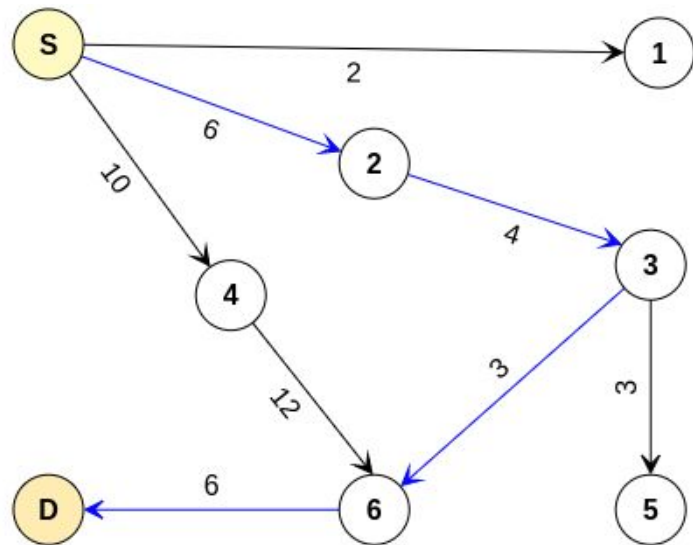
1. For each deleted edge $u \rightarrow v$:
 - a. if $f(v) \neq \infty$ and $\text{optimal_parent}(v) = u$, compute $f(v)$ from all of its parents and chose the least one as $\text{optimal_parent}(v)$ and update $f(v)$,
 - b. if v doesn't have any parent then $f(v) = \infty$
 - c. add v to update_list .
2. While update_list is not empty:
 - a. Extract node n from update_list .
 - b. For each child of n such that $\text{optimal_parent}(\text{child}) = n$:
 - i. If $f(\text{child}) > f(\text{child})_{\text{new}}$ then compute $f(\text{child})$ from all parents of child node and chose the least one as optimal_parent and add child to update_list .



Node D and node 6 are added to update list. (now node 6 reads latest value of node 3 and chooses its optimal parent from $\{4,3\}$), node D has stale value

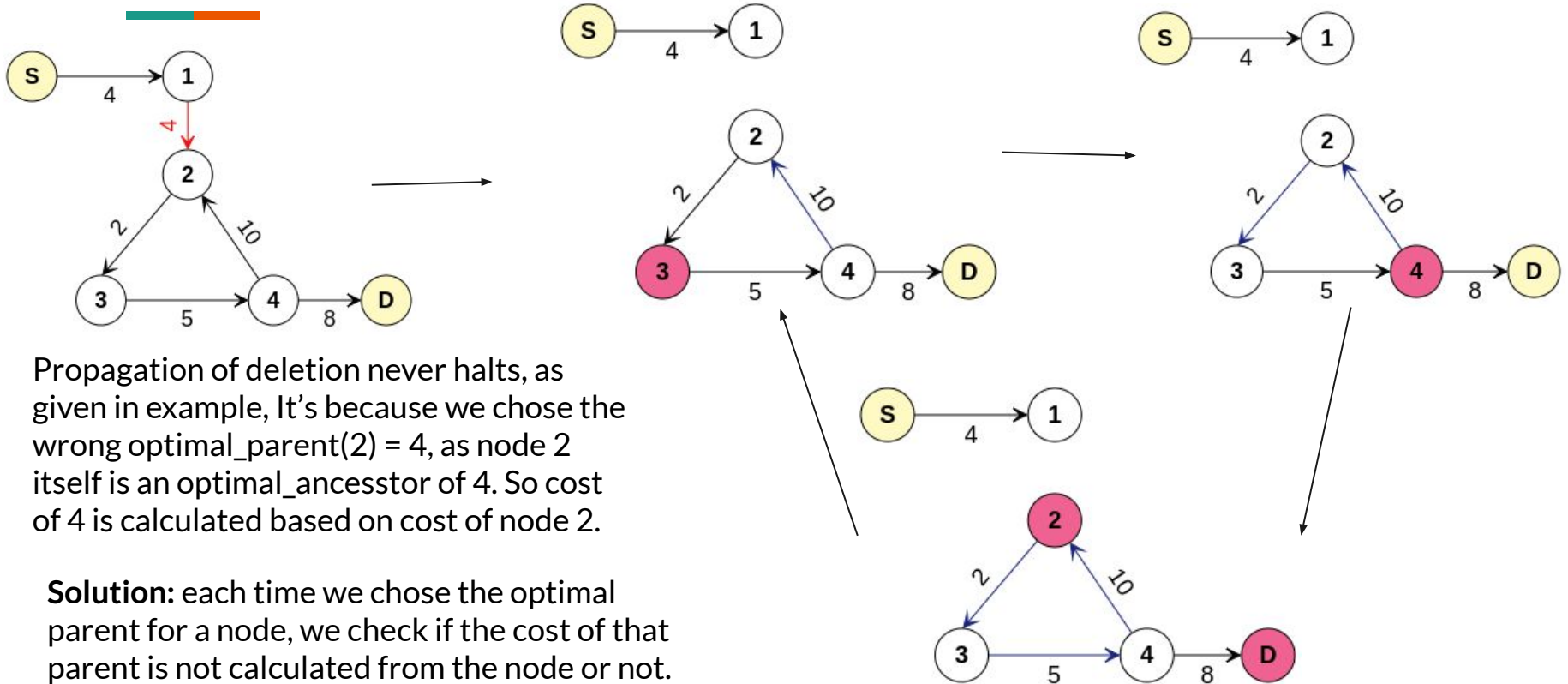
Propagating Deletions

1. For each deleted edge $u \rightarrow v$:
 - a. if $f(v) \neq \infty$ and $\text{optimal_parent}(v) = u$, compute $f(v)$ from all of its parents and chose the least one as $\text{optimal_parent}(v)$ and update $f(v)$,
 - b. if v doesn't have any parent then $f(v) = \infty$
 - c. add v to update_list .
2. While update_list is not empty:
 - a. Extract node n from update_list .
 - b. For each child of n such that $\text{optimal_parent}(\text{child}) = n$:
 - i. If $f(\text{child}) > f(\text{child})_{\text{new}}$ then compute $f(\text{child})$ from all parents of child node and chose the least one as optimal_parent and add child to update_list .



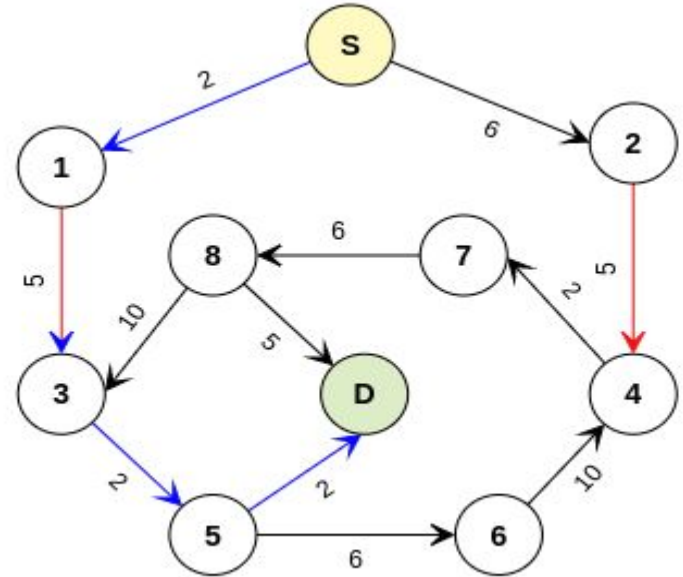
Node 6 propagates the latest value to D, thus changing the optimal path.

Deletion and Cycle



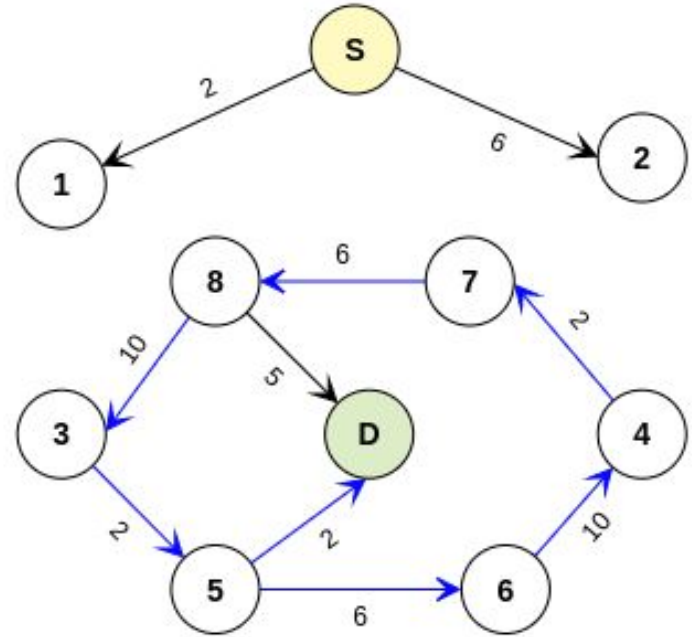
Parallel deletion and cycle

- Edges $1 \rightarrow 3$ and $2 \rightarrow 4$ are deleted.
- Node 3,4 are added in update list.
- When node 3 performs cycle check for its only parent node 5, it reads $5 \rightarrow 6 \rightarrow 4 \rightarrow 2 \rightarrow S$, as node 4 deletion is process is executing in parallel, so it reads stale value and sets node 5 as its optimal_parent.
- Similarly node 4 sets node 6 as its optimal_parent.
- When try to get path from node $D \rightarrow S$, we get stuck at optimal_parents cycle, and node D has wrong cost value.



Parallel deletion and cycle

- This problem arises due to dependency cycle and because both nodes are processed simultaneously.
- We can eliminate such cases if we process them in sequential manner.
- Since the number of such cases are low, we first process deletion in parallel allowing formation of such cycles.
- Then for each node in `update_list`, sequentially using single thread we check if there is optimal_parent cycle, if so we eliminate such cycle by selecting next optimal parent of that node and then propagating for it.

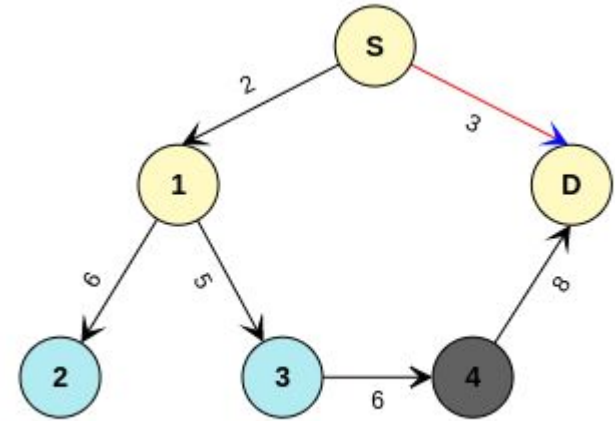


Propagation of deletions

Lemma 8: At the end of propagation for deletion all node $n \in V$ such that $f(n)_{\text{new}} \neq \infty$ has the latest $f(n)$ which is optimal cost in new graph including deletions. [\(proof\)](#)

Lemma 9: At the end of propagation $f(\text{destination})$ might not be the optimal cost of reaching destination from source in new graph

Lemma 10: After propagation of deletions if $f(\text{destination}) < f(n) \forall n \in \text{open list}$ then $f(\text{destination})$ is the optimal cost of reaching destination from source. [\(proof\)](#)



Open List = {2,3}

Closed List = {S,1,D}

$f(4) = \infty$.

Deleted edge $S \rightarrow D$.

After propagation $f(D) = \infty$.

Which proves lemma 9, as new optimal path is $S \rightarrow 1 \rightarrow 3 \rightarrow 4$.

Fully Dynamic : Separate Propagation



- Real-world graphs are dynamic and edges are changing continuously so there are insertions of edges and deletions of edges at the same time.
- 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.
- **Separate Propagation:**
 - a. Propagate for Insertion of edges
 - b. Propagate for deletion of edges.
 - c. Do A* if necessary

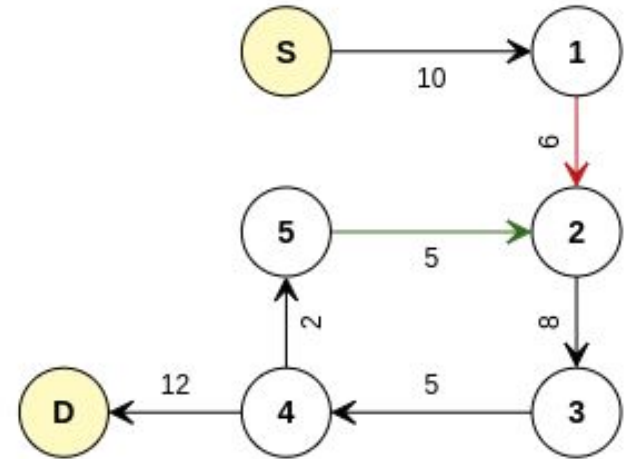
Fully Dynamic : Simultaneous Propagation



- Consider insertion and deletion of an edge as an update.
- Create `update_list` which has all nodes v , such that $u \rightarrow v$ is either inserted or deleted and $f(v)$ is changed (decreased for insertion and increased for deletion).
- While `update_list` not empty:
 - a. Extract node n from update list.
 - b. For each child of n :
 - i. If $f(\text{child})$ decreases, update $f(\text{child})$ and make n `optimal_parent` and add child to update list.
 - ii. If $f(\text{child})$ increases and n is `optimal_parent(child)`, choose the best `optimal_parent` from all parents of child and add child to `update_list`.
- Add necessary cycle checks for b(ii).

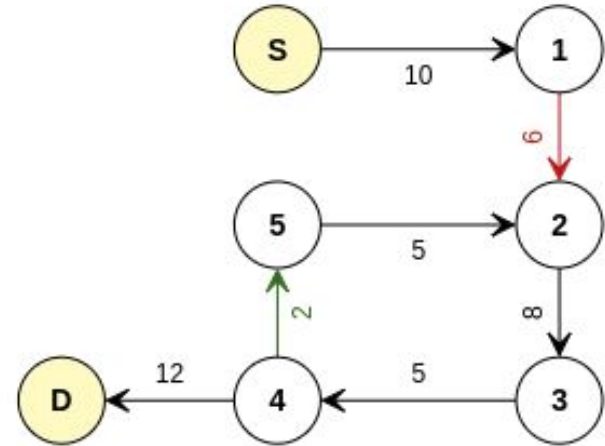
Fully Dynamic : Simultaneous Propagation

- Edge $1 \rightarrow 2$ is deleted.
- Edge $5 \rightarrow 2$ is added.
- While building update_list, suppose deletion happens first, it will set $f(2) = \infty$.
- And then insertion will make node 5 as optimal_parent of 2, as deletion of edge $1 \rightarrow 2$ is not yet propagated.
- But $f(2)$ is optimal_ancestor(5) thus It creates an optimal_parent cycle.
- Cycle Check is required even for insertions in simultaneous propagation.



Fully Dynamic : Simultaneous Propagation

1. Edge $1 \rightarrow 2$ is deleted
2. Edge $4 \rightarrow 5$ is inserted
3. Deletion changes $f(2) = \infty$ and insertion sets $f(5)$.
4. At propagation, $f(3) = \infty$ and node 5 sets $f(2)$.
5. Next step $f(4) = \infty$ and node 2 sets $f(3)$.
6. Next step $f(5) = \infty$ and node 3 sets $f(4)$.
7. Next step $f(2) = \infty$ and node 4 sets $f(5)$.
8. Repeats forever.



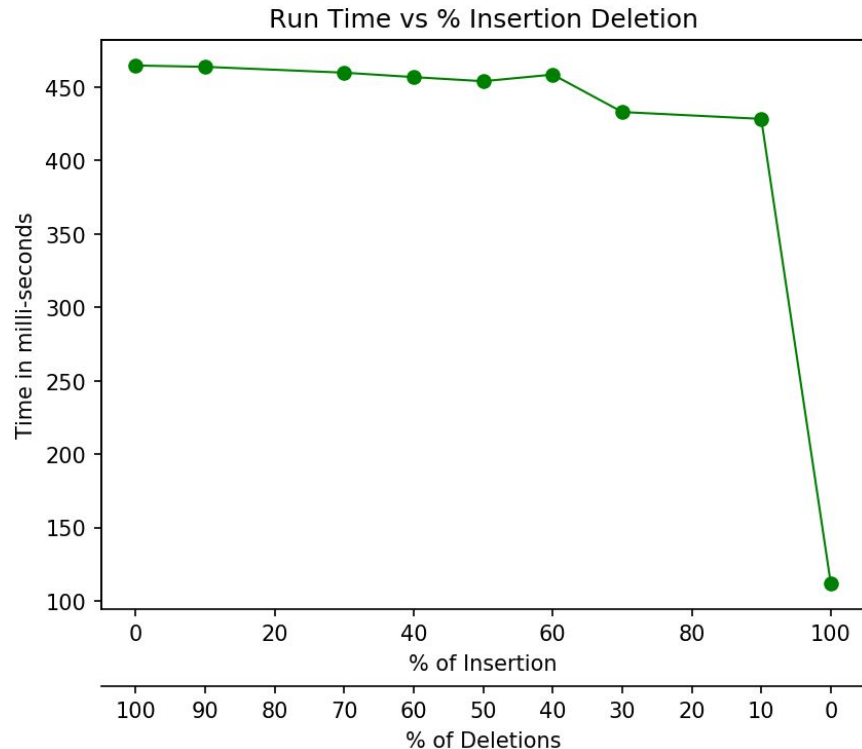
Cycle Check is required at every step of propagation.

Experimental Results

The plot shows variation in execution time as we change the composition of the updates.

We observed that even for 10% of deletion and 90% insertion the execution time is quite high compared to 100% of insertion.

It's because deletion is computationally costlier than insertions due to the cycle checks.



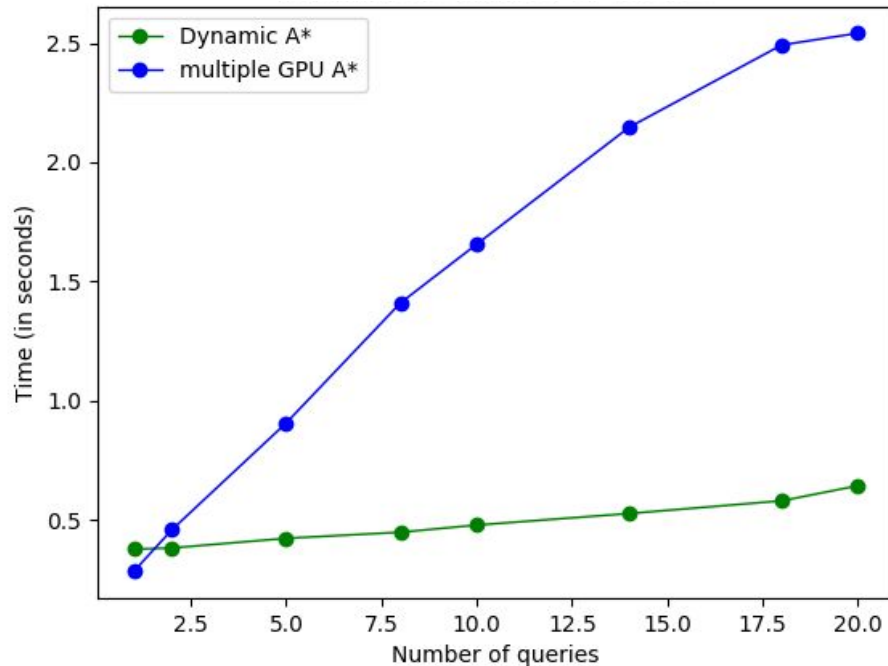
Experimental Results



The plot compares re-executing A* from scratch with our algorithm.

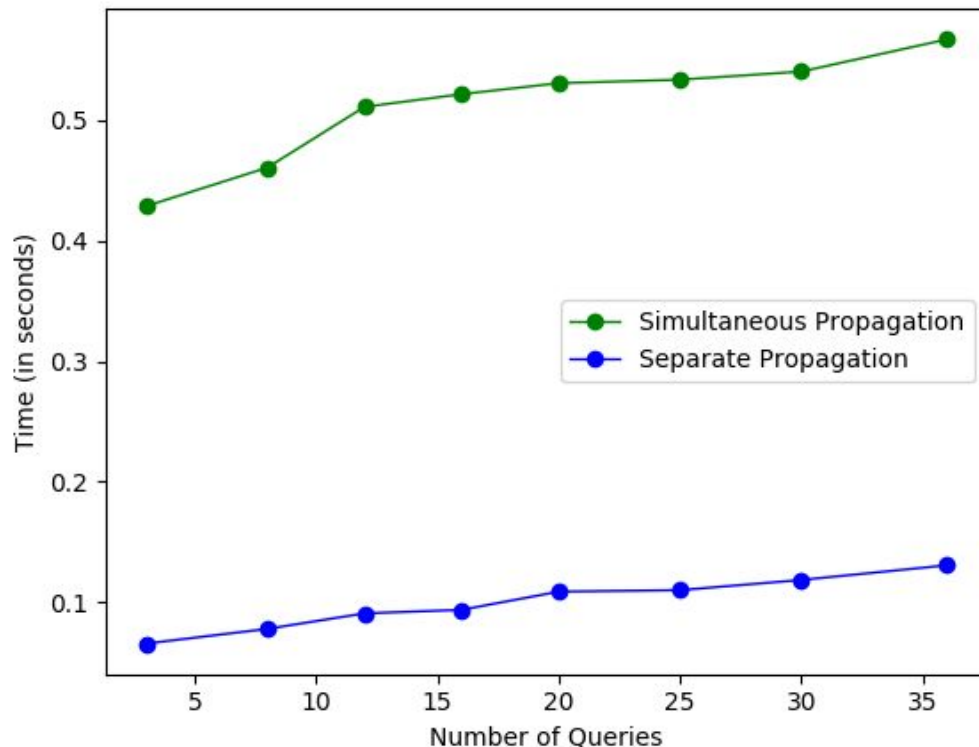
Initially multiple A* performs better but as the number of time the graph is subjected to updates increases our algorithm outperforms it.

The speedup we get is directly proportional to the number of queries.



Experimental Results

Separate Propagation performs better than simultaneous propagation, as simultaneous propagation requires more cycle checks compared to separate propagation.



Experimental results



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×

Table: Compares Execution Time of multiple A* with our algorithm for various temporal graph
(Time in seconds)

Applications: Wireless Sensor Networks

- In Wireless Sensor Network, light-weight, low power and small size sensor nodes are used which are operated by a small battery. Sensors nodes are used for monitoring physical phenomena like temperature, humidity, vibrations and so on..
- There are many routing algorithms which try to increase the lifetime of the network by minimising the total energy consumption or by avoiding lower energy level nodes for routing of packets.
- A^* is used in many such energy efficient algorithm one of such is Energy efficient routing protocol (EERP).

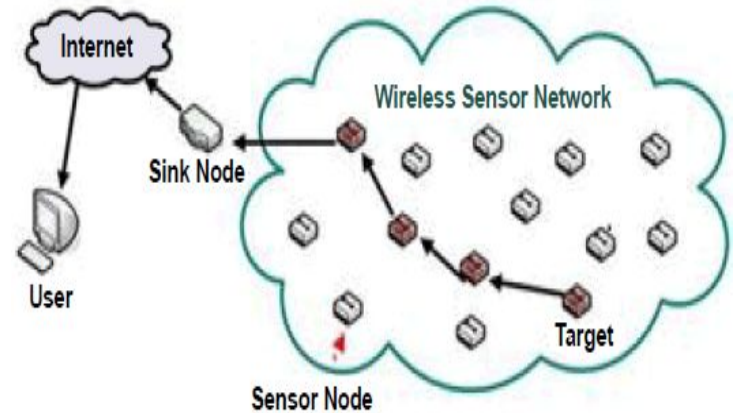


Figure 1: Wireless sensor network.

Energy Efficient Routing Protocol

In EERP the edge weight is a function of the residual Energy of the node. In each round of packet transmission Energy is drained which changes the edge weights in the network, thus the network graph is dynamic in nature.

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

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

```
1 Input: Sensor Network
2 Output : Life of Sensor Network in terms of rounds
3 BEGIN
4     InitilizeNetwork()
5     EstimateDistance()           // finds distance to the sink
6     WHILE NOT END_ASTAR()       // N-of-N metric [11]
7         InitializeSolArray()    // initialize solution
8                                 //array to store routing schedule
9     FOR each node i in the Network DO
10         CreateTree (i)         //using A-Star algorithm
11         PrepareSolArray()      //prepare routing schedule
12     END FOR
13     BroadcastSolution()        // BS broadcasts routing schedule
14     UpdateEnergy()             //Energy update for relay nodes
15     CountRound = CountRound + 1 //count n/w life
16 END WHILE
17 PRINT CountRound              //print n/w lifetime in
18 END
```

EERP with DA*

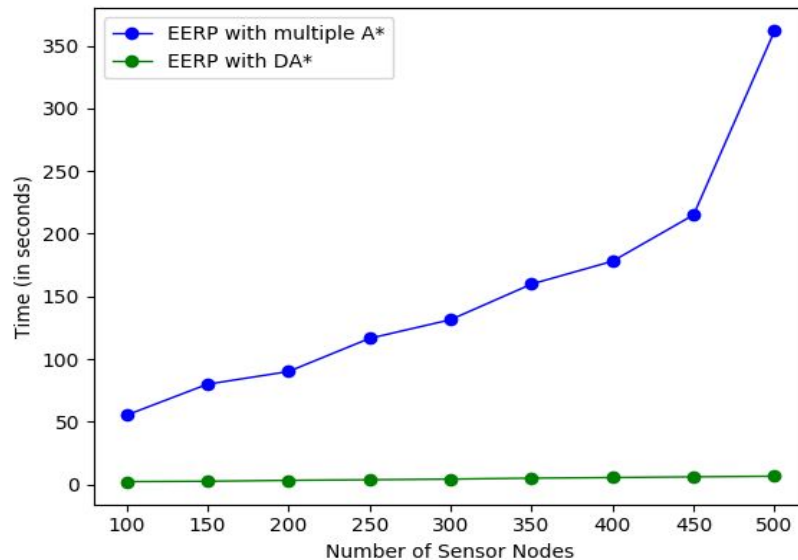
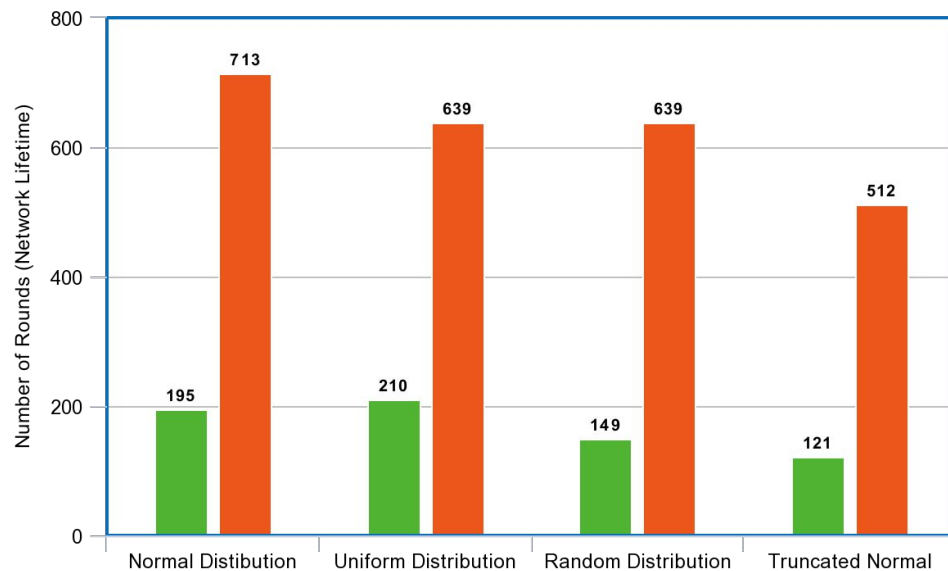


Instead of doing A* from each node to sink, we take idea from D* algorithm and do A* search backwards from sink till nodes are exhausted.

After each round of packet transmission instead of re-executing A* from start again we execute our propagation methods for the graph.

```
1 Input: Sensor Network
2 Output: Life of Sensor Network in terms of rounds
3 BEGIN
4     InitilizeNetwork()
5     EstimateDistance()           // finds distance to the sink
6     InitializeSolArray()        //array to store routing schedule
7
8     ASTAR_From_Sink()           //finds path to all nodes from sink
9     BroadcastSolution()         // BS broadcasts routing schedule
10    UpdateEnergy()              //Energy update for relay nodes
11    CountRound = CountRound + 1
12
13    WHILE NOT END_ASTAR()
14        Create_UpdateList()      //list of edges whose weight is changed
15        Propagation()            //Propagate change using methods of DA*
16        Update_SolArray()        //Update routing table with new path
17        BroadcastSolution()       // BS broadcasts routing schedule
18        UpdateEnergy()           //Energy update for relay nodes
19        CountRound = CountRound + 1
20    END WHILE
21    PRINT CountRound             //print n/w lifetime
22 END
```

WSN: Results



Thank You

And special thanks to prof. Rupesh Nasre for his invaluable guidance throughout this project.

Lemma 1



Lemma 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 $n \notin \text{closed_list}$ and $f_n < f_d$. If $n \notin \text{closed_list}$ then either node $n \in \text{open_list}$ or n is not explored yet which means $f_n = \infty$.

If $n \in \text{open_list}$ and $f_n < f_d$, In A^* to expand a node we always chose the node with least f (as step 3 of [Algorithm 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.

Lemma 2



Lemma 2: Insertion of an edge can not increase the cost of source to destination Optimal Path.

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

Case 1: If $g(v)_{\text{new}} < g(v)$, if optimal path consists of node v then the decrease in the cost of v will decrease the cost of Optimal Path, as if there is a path from v to destination then $\text{source} \rightarrow v \rightarrow \text{destination}$ path 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 $g(v)_{\text{new}} \geq g(v)$, 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 the same.

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

[\(back\)](#)

Lemma 3



Lemma 3: If we add an edge $u \rightarrow v$ and $f(u) > f(\text{destination})$ then addition of this edge will not affect the Optimal Path.

Proof: Given $f(\text{destination}) < f(u)$, As weights are positive $w_{uv} > 0$ so cost of any such path from source $\rightarrow u \rightarrow v \rightarrow \text{destination}$ will be $g(u) + \text{cost of reaching destination from } u \equiv f(u)$, but as $f(u) > f(\text{destination})$ 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

[\(back\)](#)

Lemma 4



Lemma 4: At the end of propagation, all the nodes $\in V$ will have the latest cost $f()$ which is the optimal cost in the new graph including inserts.

Proof: Suppose we insert edge (u,v) . Let's take a node $n \in \text{Graph}$.

Case 1: If $n \in \text{sub-graph}(v)$, since $v \in \text{update_list}$ and at each iteration we add nodes which belongs to sub-graph of v and whose $f(n)$ is decreased. so if $n \in \text{update list}$ at some iteration then its $f(n)$ is decreased and that is the new optimal cost, If $n \notin \text{update list}$ then $f(n)_{\text{new}}$ due to insertion is greater than $f(n)_{\text{old}}$ in which case $f(n)_{\text{old}}$ is the optimal cost, as it is the lowest cost associated with node n .

Case 2: If $n \notin \text{sub-graph}(v)$ 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 5



Lemma 5: Deletion of edge $u \rightarrow v$ where $u \in V$ and $v \in V$ can not decrease $f(v)$

Proof: Let edge $u \rightarrow v$ gets deleted from graph then:

Case 1: If $u = \text{optimal_parent}(v)$ then, $\text{source} \rightarrow u \rightarrow v$, was the optimal path of reaching v from source, So after deletion, $f(v)_{\text{new}} = g(s) + w_{sv} + h(v)$ 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 $f(v)_{\text{new}} \geq f(v)_{\text{old}}$.

Case 2: If u is not the $\text{optimal_parent}(v)$, then let s be the $\text{optimal_parent}(v)$, deletion of edge $u \rightarrow v$ 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 $u \rightarrow v$ can only increase $f(v)$. Hence Proved.

[\(back\)](#)

Lemma 6



Lemma 6: If edge(u,v) is deleted and $\text{optimal_parent}(v) \neq u$, 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 $u v$ doesn't affect the $f(v)$ as the least cost of reaching v is from s and as there is no change in cost of any nodes so Optimal Path remains same.

[\(back\)](#)

Lemma 8

Lemma 8: At the end of propagation for deletion all node $n \in V$ such that $f(n)_{\text{new}} \neq \infty$ has the latest $f(n)$ which is optimal cost in new graph including deletions.

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

[\(back\)](#)

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

- If $v \in \text{Optimal_Path}(n)$ 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 is not computed wrt v , which implies it's the optimal_cost of reaching v . Since $v \in \text{Optimal_Path}(n)$, \exists parent p of n such that $p \in \text{Optimal_Path}(n)$ and $p \in \text{sub-graph}(v)$, which implies $p \in \text{update_list}$ 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.
- If $v \notin \text{Optimal_Path}(n)$, then \exists optimal_parent p of n such that $v \notin \text{Optimal_Path}(p)$, thus $n \notin \text{update_list}$ at any iteration of propagation so $f(n)$ remains same which is the optimal_cost from Lemma 6.

Case 2: If $n \notin \text{sub-graph}(v)$ then $v \notin \text{Optimal_Path}(n)$ also $v \notin \text{update_list}$ at any iteration of propagation so $f(n)$ remains same which is the optimal_cost from Lemma 6.

Lemma 10



Lemma 10: After propagation of deletions if $f(\text{destination}) < f(n) \forall n \in \text{open_list}$ then $f(\text{destination})$ is the optimal cost of reaching destination from source

Proof: After propagation of deletion \forall node $n \in \text{open_list}$ either $f(n)_{\text{new}}$ is the optimal_cost or $f(n)_{\text{new}} \neq \infty$ from Lemma 8. Also given $f(\text{destination}) < f(n) \forall n \in \text{open_list}$, 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 $f(\text{destination})$ is the optimal cost of reaching destination from source.

[\(back\)](#)