Advanced Shortest Paths: Contraction Hierarchies

Michael Levin

Higher School of Economics

Graph Algorithms
Data Structures and Algorithms

Outline

Learning Objectives

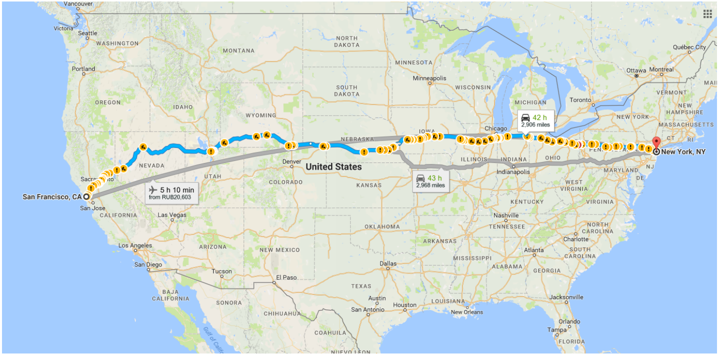
- Bidirectional Dijkstra can be 1000s of times faster than Dijkstra for social networks
- But just 2x speedup for road networks
- This lecture — great speedup for road networks

Highway Hierarchies

- Long-distance trips go through highways

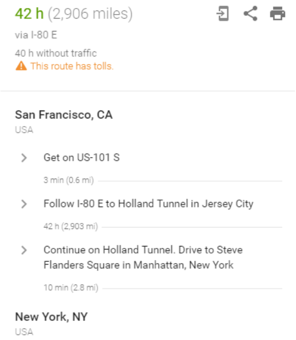
Highway Hierarchies

- Long-distance trips go through highways



Highway Hierarchies

- Long-distance trips go through highways



Highway Hierarchies

- Long-distance trips go through highways
- To get from A to B, first merge into a highway, then into a bigger highway, etc., then exit to a highway, then exit to a street, then go to B

Highway Hierarchies

- Long-distance trips go through highways
- To get from A to B, first merge into a highway, then into a bigger highway, etc., then exit to a highway, then exit to a street, then go to B
- Less important roads merge into more important roads

Highway Hierarchies

- Long-distance trips go through highways
- To get from A to B, first merge into a highway, then into a bigger highway, etc., then exit to a highway, then exit to a street, then go to B
- Less important roads merge into more important roads
- Hierarchy of roads

Highway Hierarchies

- There are algorithms based on this idea
- “Highway Hierarchies” and “Transit Node Routing” by Sanders and Schultes
- Millions of times faster than Dijkstra
- Pretty complex
- This lecture — “Contraction Hierarchies”, thousands of times faster than Dijkstra

Node Ordering

Idea behind Contraction Hierarchies - Node Ordering

- Nodes can be ordered by some "importance"
- Importance first increases, then decreases back along any shortest path
- E.g., points where a highway merges into another highway
- Can use bidirectional search

Importance Ideas

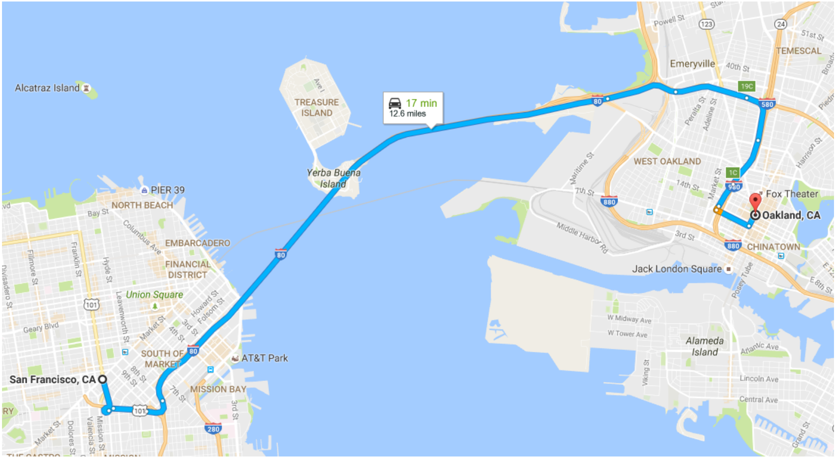
Many shortest paths involve important nodes

Importance Ideas

Important nodes are spread around

Importance Ideas

Important nodes are sometimes unavoidable



Shortest Paths with Preprocessing

- Preprocess the graph
- Find distance and shortest path in the preprocessed graph
- Reconstruct the shortest path in the initial graph

Question

If we are allowed to do preprocessing, why not just compute all the pairwise distances between the nodes of the initial graph and store them in one table? Then it will only cost O(1) to retrieve the distance between any two nodes.

- ○ Such preprocessing cannot be done in polynomial time
- ○ This is impossible at all
- ◉ Such preprocessing can be done, however, for practical applications on the real road networks, such as Europe, USA, or even some big city, it would take too long to compute, and the resulting table would be too big to store

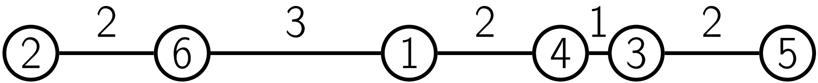
✓ **Correct**
Indeed, for a graph of USA with 20M nodes and 50M edges, running Dijkstra's algorithm with priority queue from each node would take time proportional to $|V| \cdot |E| \log |V| \sim 20\,000\,000 \cdot 50\,000\,000 \cdot 25 = 25 \cdot 10^{15}$. If we assume a small constant here and roughly 1 billion operations per second, it will still take us about a year to compute the table. Also, storing it would require at least $20\,000\,000 \cdot 20\,000\,000 \cdot 4 = 1.6 \cdot 10^{15}$ bytes or 1.6 Petabytes.

Outline

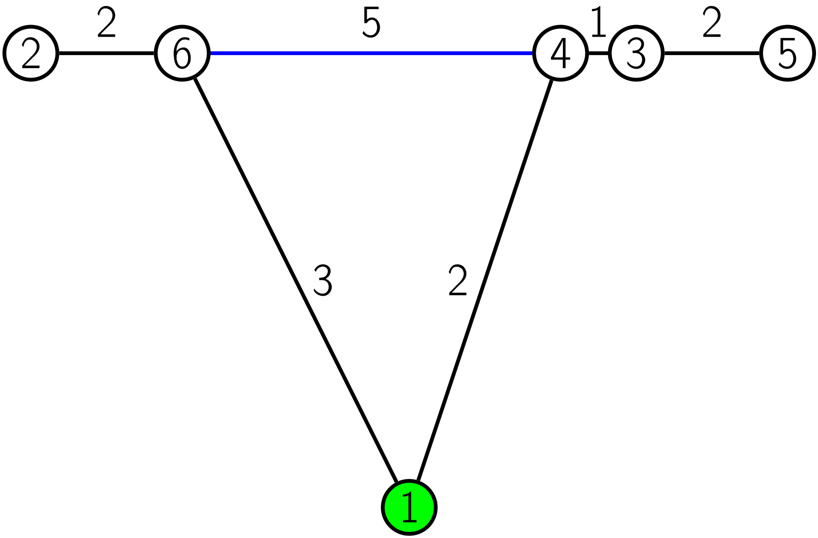
Preprocessing

- Eliminate nodes one by one in some order
- Add shortcuts to preserve distances
- Output: augmented graph + node order

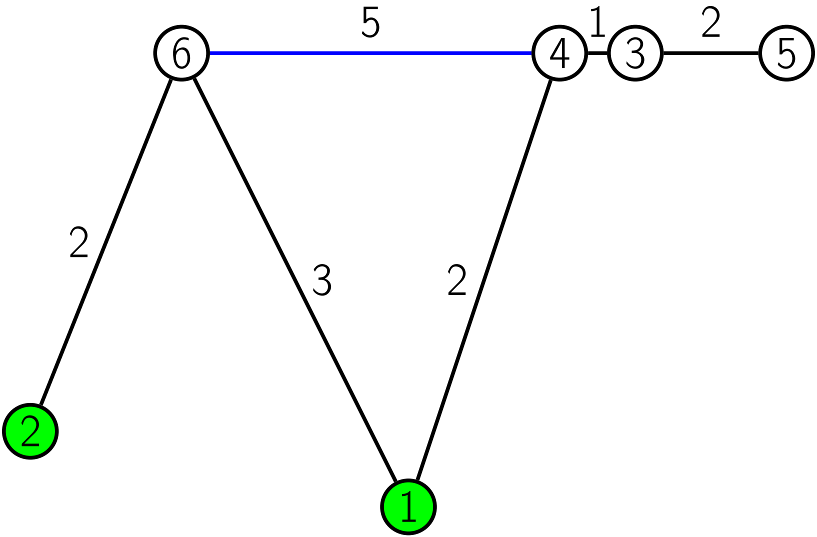
Node Contraction



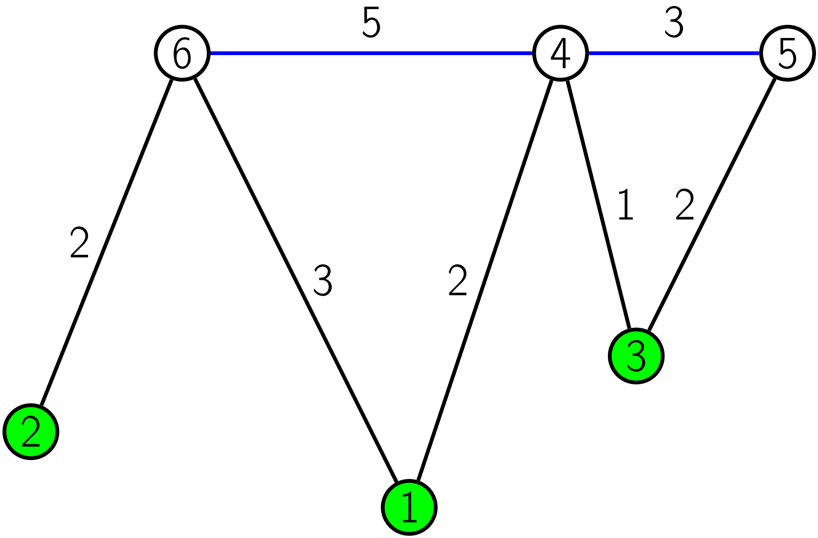
Node Contraction



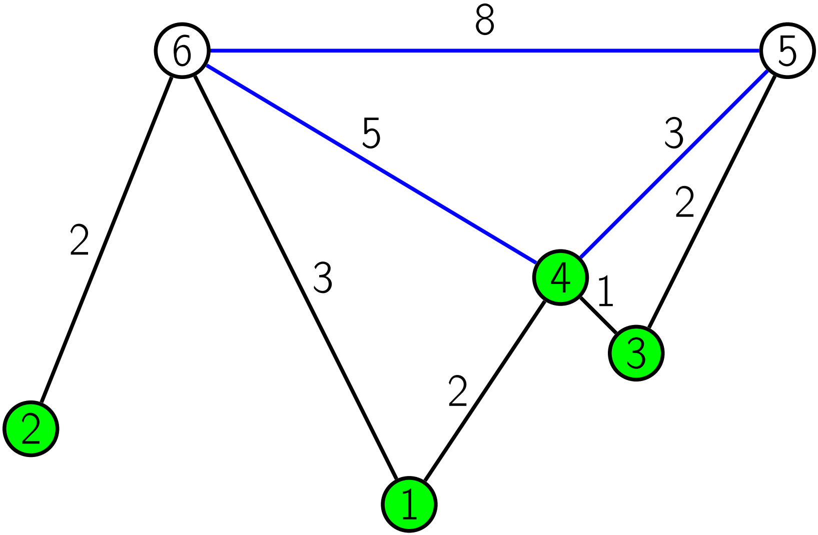
Node Contraction



Node Contraction

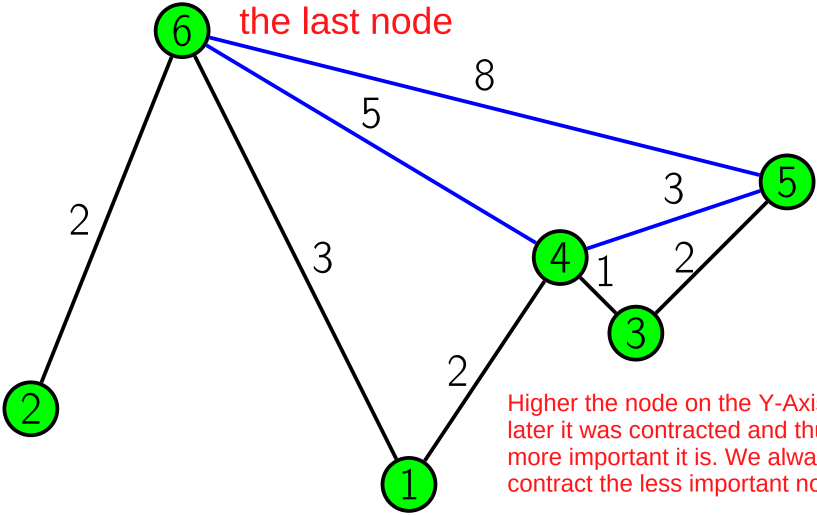


Node Contraction



Node Contraction

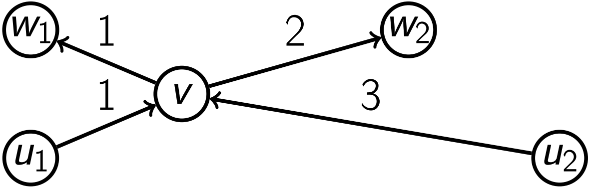
NO Need to Contract Node 6 as it is the last node



Higher the node on the Y-Axis the later it was contracted and thus the more important it is. We always first contract the less important node

Witness Paths

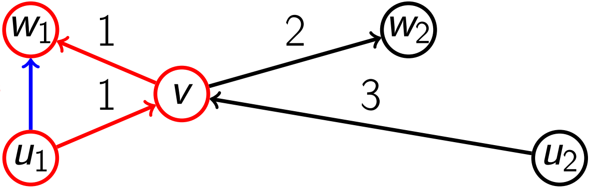
- Contraction of node v



Witness Paths

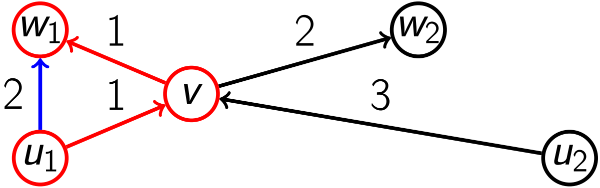
- Contraction of node v
- For every pair of edges (u, v), (v, w) add a new edge (u, w)

The new path blue is added if we assume that u1 to v and v to w1 is the shortest path



Witness Paths

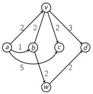
- Contraction of node v
- For every pair of edges (u, v), (v, w) add a new edge (u, w)
- $\ell(u, w) \leftarrow \ell(u, v) + \ell(v, w)$



Witness Path

Question

How many shortcuts will we have to add to the augmented graph if we decide to contract node v in the graph below? Note that there is an undirected edge between nodes b and w, and in this case we assume that this edge goes both from b to w and from w to b.



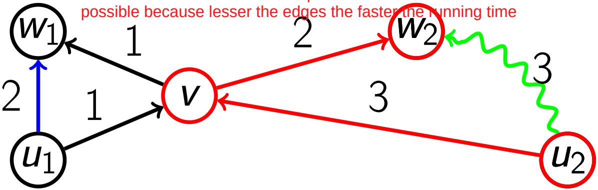
- 0
- 2
- 5
- 3
- 1
- 6
- 4

Correct

Correct! Node v has incoming edges from nodes a and b and outgoing edges to b, c and d (the undirected edge (b, v) goes in both directions). This gives us potential shortcuts (a, b), (a, c), (a, d), (b, c) and (b, d). For (a, b), there is a direct edge (a, b) of length 1 that is shorter than the path through v of length $2 + 2 = 4$, so we don't need to add a shortcut. For (a, c), the path through v has length $2 + 2 = 4$, and the only path not going through v is the edge (a, c) of length 5, so we need to add a shortcut. For (a, d), the path through v has length $2 + 3 = 5$, but there is also a path $a \to b \to w \to d$ not going through v and having length $1 + 2 + 2 = 5$, so there is no need to add a shortcut. For (b, c), there is no path which bypasses v, so we need to add a shortcut. For (b, d), the path through v has length $2 + 3 = 5$, but there is a path $b \to w \to d$ of length $2 + 2 = 4$, so there is no need for a shortcut. In total, we need to add 2 shortcuts - between a and c, and between b and c.

- Contraction of node v
- For every pair of edges $(u, v), (v, w)$ add a new edge (u, w)
- $\ell(u, w) \leftarrow \ell(u, v) + \ell(v, w)$
- But only if there is no witness path P_{uw} shorter than $\ell(u, v) + \ell(v, w)$ and bypassing v

If there is a shorter path without including v then we should completely bypass v. These paths are called witness path because these are witnesses that we don't need to add the shortcut and In practice we want to avoid shortcuts as much as possible because lesser the edges the faster the running time



Outline

1. Contraction Hierarchies
2. Preprocessing
3. **Witness Search**
4. Query
5. Query Correctness
6. Node Ordering

Witness Search

When contracting node v, for any pair of edges (u, v) and (v, w) we want to check whether there is a witness path from u to w bypassing v with length at most $\ell(u, v) + \ell(v, w)$ — then there is no need to add a shortcut from u to w.

Definition

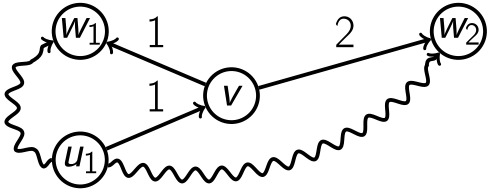
Witness search is the search for a witness path.

Definition

If there is an edge (u, v), call u a predecessor of v. If there is an edge (v, w), call w a successor of v.

Witness Search

- For each predecessor u_i of v, run Dijkstra from u_i ignoring v
- Essential for good query performance
- Otherwise the augmented graph will be very dense



Witness Search Optimizations

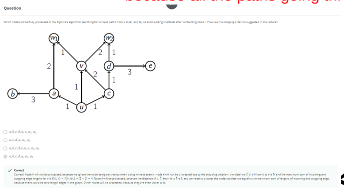
- Stop Dijkstra when distance from the source becomes too big
- Limit the number of hops

Of course we will not find all the witness paths but the speed of pre-processing will increase

Stop Dijkstra

- If $d(u_i, x) > \max\limits_{u,w}(\ell(u,v) + \ell(v,w))$, there is no witness path going through x

because all the paths going through x will be larger then the d(u,x)



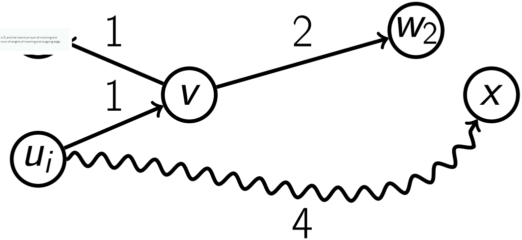
Question

Which nodes will be fully processed in the Dijkstra's algorithm searching for witness paths from u to w_1 and w_2 to avoid adding shortcuts after contracting node v if we use the stopping criterion suggested in the lecture?

- $a, b, c, d, u, v, w_1, w_2$
- a, c, d, u, w_1, w_2
- $a, b, c, d, e, u, v, w_1, w_2$
- a, b, c, d, u, w_1, w_2

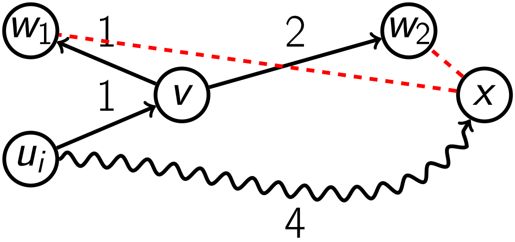
Correct

Correct! Node v will not be processed, because we ignore the node being contracted when doing witness search. Node e will not be processed due to the stopping criterion: the distance $d(u, e)$ from u to e is 5, and the maximum sum of incoming and outgoing edge lengths for v is $\ell(c, v) + \ell(v, w_2) = 2 + 2 = 4$. Node b will be processed, because the distance $d(u, b)$ from u to b is 4, and we need to process the nodes at distance equal to the maximum sum of lengths of incoming and outgoing edge, because there could be zero-length edges in the graph. Other nodes will be processed, because they are even closer to u.



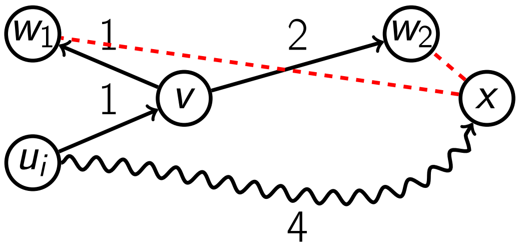
Stop Dijkstra

- If $d(u_i, x) > \max_{u,w}(\ell(u,v) + \ell(v,w))$, there is no witness path going through x



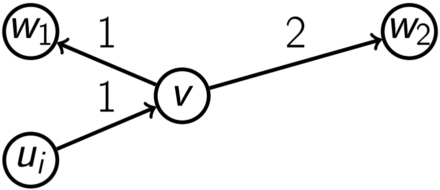
Stop Dijkstra

- If $d(u_i, x) > \max\limits_{u,w}(\ell(u,v) + \ell(v,w))$, there is no witness path going through x
- Limit the distance by $\max\limits_{u,w}(\ell(u,v) + \ell(v,w))$



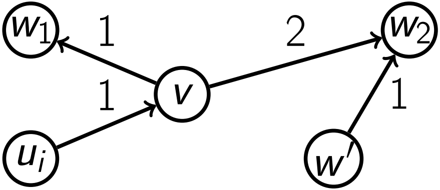
Stop Dijkstra

- Consider any predecessor w' of any successor w of v



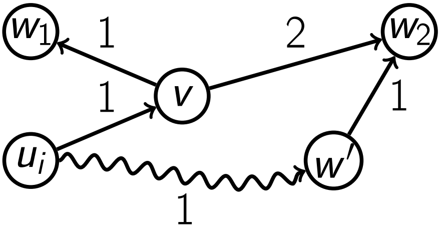
Stop Dijkstra

- Consider any predecessor w' of any successor w of v



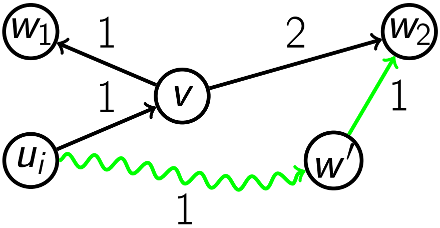
Stop Dijkstra

- Consider any predecessor w' of any successor w of v
- If $d(u, w') + \ell(w', w) \leq \ell(u, v) + \ell(v, w)$, there's a witness path



Stop Dijkstra

- Consider any predecessor w' of any successor w of v
- If $d(u, w') + \ell(w', w) \leq \ell(u, v) + \ell(v, w)$, there's a witness path



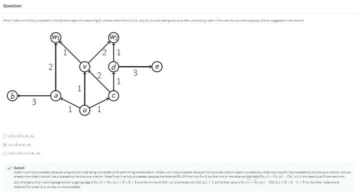
Stop Dijkstra

- If $d(u, w') + \ell(w', w) \leq \ell(u, v) + \ell(v, w)$, there's a witness path

Stop Dijkstra

Very Important Slide. Study Conditions Carefully

- If
  $$d(u, w') + \ell(w', w) \leq \ell(u, v) + \ell(v, w),$$
  there's a witness path
- Limit the distance by
  $$\max_{u,w} \max_{(w',w)} (\ell(u, v) + \ell(v, w) - \ell(w', w))$$



Question

Which nodes will be fully processed in the Dijkstra's algorithm searching for witness paths from u to w_1 and w_2 to avoid adding shortcuts after contracting node v if we use the improved stopping criterion suggested in the lecture?

- a, b, c, d, u, w_1, w_2
- a, c, d, u, w_1, w_2
- $a, b, c, d, e, u, v, w_1, w_2$

Correct

Node v won't be processed, because we ignore the node being contracted while performing witness search. Node e won't be processed, because the improved criterion doesn't process any nodes that wouldn't be processed by the previous criterion, and we already know that e wouldn't be processed by the previous criterion. Node b won't be fully processed, because the distance $d(u, b)$ from u to b is 4, but the limit on the distance $\max_{u,w} \max_{w'} \ell(u, v) + \ell(v, w) - \ell(w', w)$ in this case is just 3: the maximum sum of lengths of an incoming edge and an outgoing edge is $\ell(c, v) + \ell(v, w_2) = 2 + 2 = 4$, and the minimum $\ell(w', w)$ is achieved with $\ell(d, w_2) = 1$, so the final value is $\ell(c, v) + \ell(v, w_2) - \ell(d, w_2) = 2 + 2 - 1 = 3$. All the other nodes are at distance 3 or closer to u, so they will be processed.

Limit the Hops

- Limit the number of “hops” in Dijkstra

Limit the Hops

- Limit the number of “hops” in Dijkstra
- Consider only shortest paths from source with at most k edges

Limit the Hops

- Limit the number of "hops" in Dijkstra
- Consider only shortest paths from source with at most k edges
- If witness path not found, add a shortcut

Limit the Hops

- Limit the number of "hops" in Dijkstra
- Consider only shortest paths from source with at most k edges
- If witness path not found, add a shortcut
- Tradeoff between preprocessing time and augmented graph size

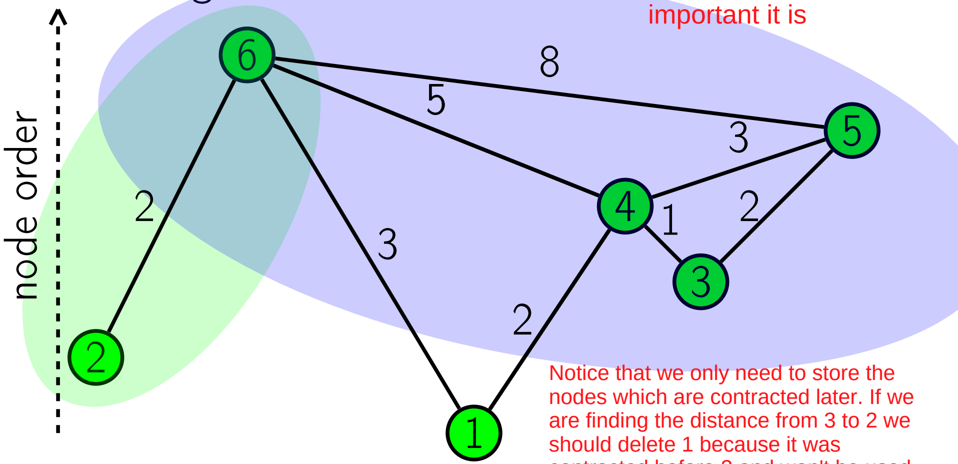
Limit the Hops

- Limit the number of “hops” in Dijkstra
- Consider only shortest paths from source with at most k edges
- If witness path not found, add a shortcut
- Tradeoff between preprocessing time and augmented graph size
- E.g., start with $k = 1$, increase gradually to $k = 5$ in the end

Outline

Bidirectional Dijkstra

- Bidirectional Dijkstra using only upwards edges

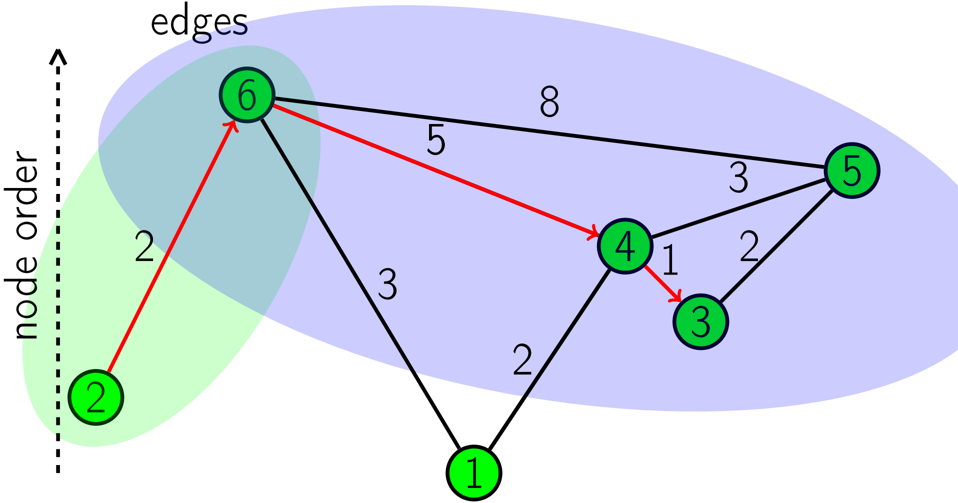


Higher the node the more important it is

Notice that we only need to store the nodes which are contracted later. If we are finding the distance from 3 to 2 we should delete 1 because it was contracted before 3 and won't be used

Bidirectional Dijkstra

- Bidirectional Dijkstra using only upwards edges



Bidirectional Dijkstra

- Bidirectional Dijkstra using only upwards edges
- Don't stop when some node was processed both by forward and backward searches
- Stop Dijkstra when the extracted node is already farther than the target

ComputeDistance$(s, t, \ldots)$

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance($s, t, \ldots$)

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance($s, t, \ldots$)

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance($s, t, \ldots$)

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance($s, t, \ldots$)

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance($s, t, \ldots$)

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance$(s, t, \ldots)$

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance$(s, t, \ldots)$

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

ComputeDistance($s, t, \ldots$)

```
estimate ← +∞
Fill dist, dist^R with +∞ for each node
dist[s] ← 0, dist^R[t] ← 0
proc ← empty, proc^R ← empty
while there are nodes to process:
  v ← ExtractMin(dist)
  if dist[v] ≤ estimate:
    Process(v, ...)
  if v in proc^R and dist[v] + dist^R[v] < estimate:
    estimate ← dist[v] + dist^R[v]
  v^R ← ExtractMin(dist^R)
  Repeat symmetrically for v^R
return estimate
```

Conclusion

- Preprocessing via nodes contraction

Conclusion

- Preprocessing via nodes contraction
- Query via Bidirectional Dijkstra

Conclusion

- Preprocessing via nodes contraction
- Query via Bidirectional Dijkstra
- Are we done?

Conclusion

- Preprocessing via nodes contraction
- Query via Bidirectional Dijkstra
- Are we done?
- Why is algorithm for query correct?

Because we don't know yet whether this algorithm works correctly or not. We don't know if this estimate is correct. Also it only returns the estimate of the distance from source to target but it doesn't return the actual shortest path between source and target in the initial graph.

- Using the standard bidirectional Djikstra algorithm we can reconstruct the path in the augmented graph between source and target using the special array in the previous node in the shortest path from source to this node and then same for the backward path

Outline

1. Contraction Hierarchies
2. Preprocessing
3. Witness Search
4. Query
5. Query Correctness   SKIP QUERY CORRECTNESS
6. Node Ordering

Augmented Graph

SKIP QUERY CORRECTNESS

Definition

The augmented graph $G^+ = (V, E^+)$ is the graph on the same set of vertices V as the initial graph G and an augmented set of edges E^+ that contains all the initial edges E of the graph G along with the shortcuts added at the preprocessing stage.

Distance Preservation

Lemma

The distance $d^+(s, t)$ between any two nodes s and t in the augmented graph $G^+ = (V, E^+)$ is equal to the distance $d(s, t)$ between these nodes in the initial graph $G = (V, E)$.

Proof

- Edges are only added to G, so $d^{+}(s, t) \leq d(s, t)$

Proof

- Edges are only added to G, so $d^+(s, t) \leq d(s, t)$
- For any added shortcut (u, w), there was a path $u \rightarrow v \rightarrow w$ of length $\ell(u, v) + \ell(v, w) = \ell(u, w)$ before adding this shortcut, so $d^+(s, t)$ can't be less than $d(s, t)$

Proof

- Edges are only added to G, so $d^+(s, t) \leq d(s, t)$
- For any added shortcut (u, w), there was a path $u \to v \to w$ of length $\ell(u, v) + \ell(v, w) = \ell(u, w)$ before adding this shortcut, so $d^+(s, t)$ can't be less than $d(s, t)$
- Thus $d^+(s, t) = d(s, t)$ $\square$

Definition

The rank $r(v)$ of vertex v is the position of v in the node order returned by the preprocessing stage.

Definition

A path $P\colon v_1 \to v_2 \to \ldots \to v_k$ in the augmented graph G^+ is called increasing if $r(v_1) < r(v_2) < \ldots < r(v_k)$. Similarly, P is called decreasing if $r(v_1) > r(v_2) > \ldots > r(v_k)$.

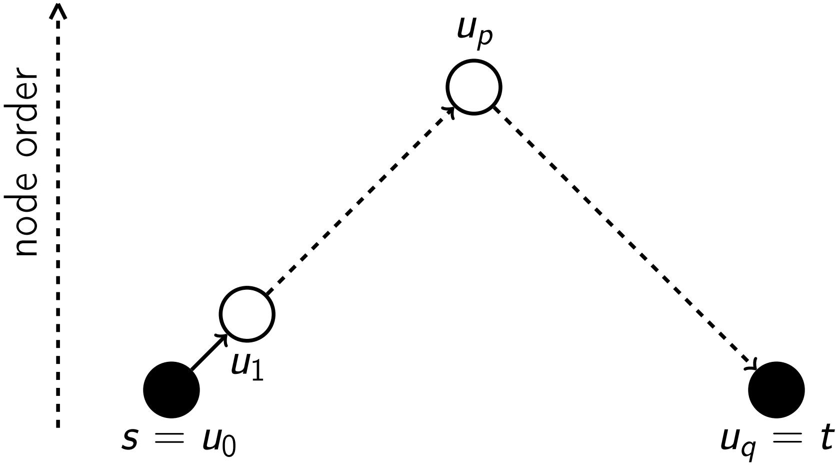
The higher the rank more important it is and later it was contracted

Justification of Bidirectional Search

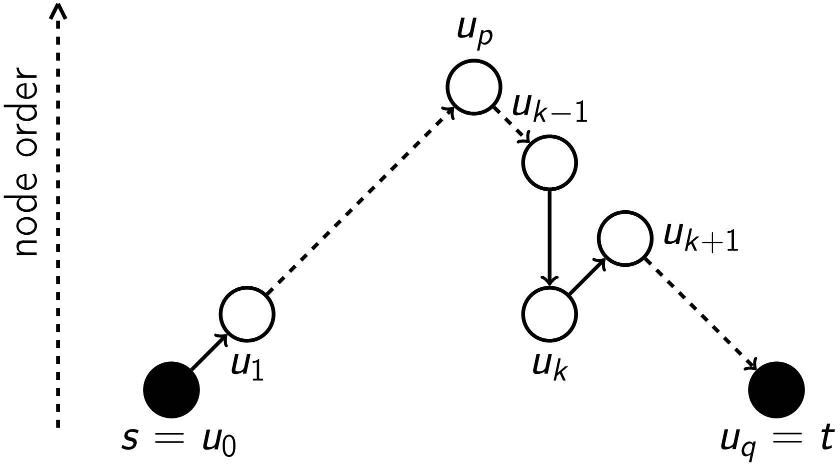
Lemma

For any s and t, the augmented graph $G^+ = (V, E^+)$ contains a shortest path P_{st} such that the subpath P_{sv} is increasing and P_{vt} is decreasing.

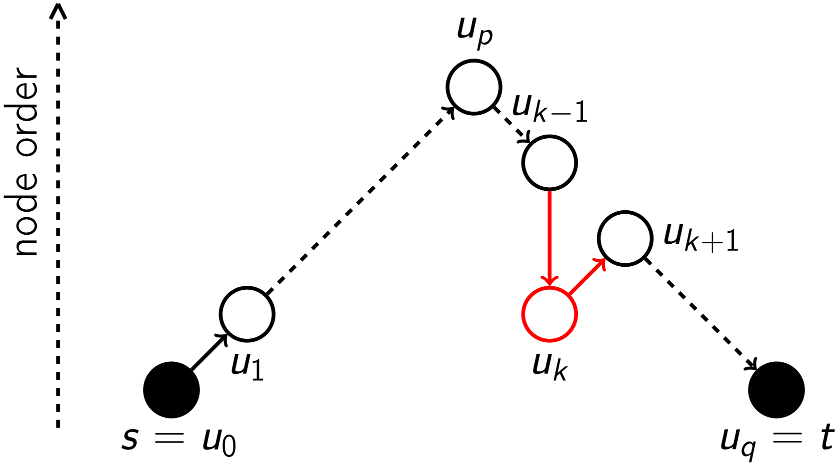
Proof Idea



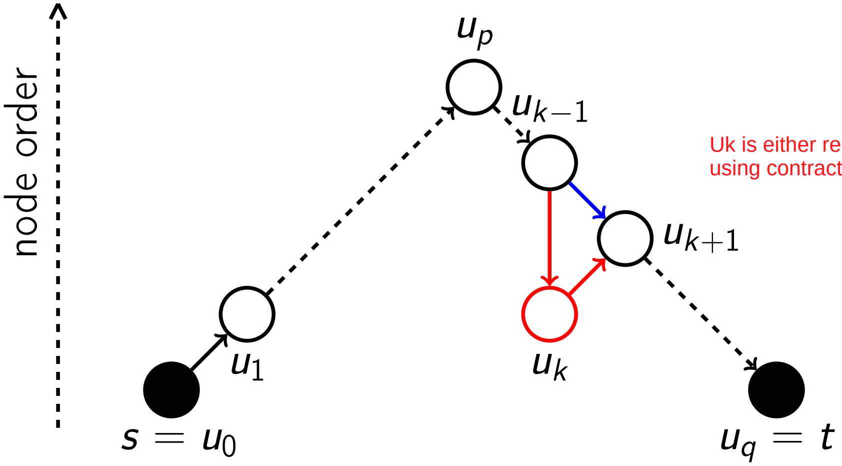
Proof Idea



Proof Idea

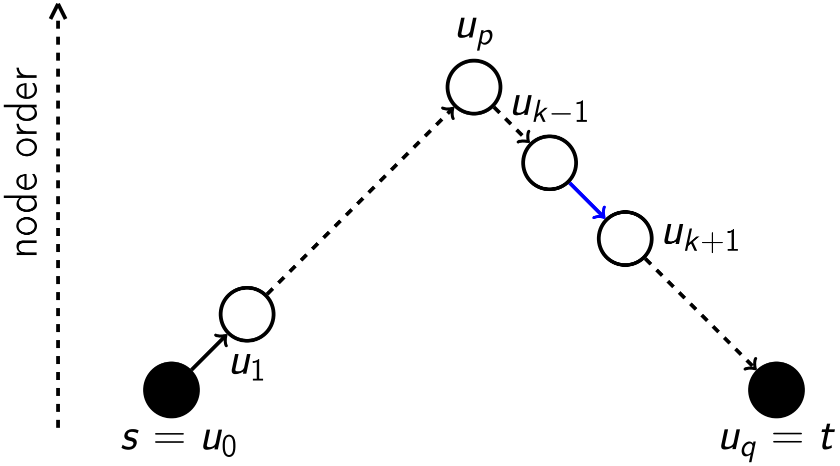


Proof Idea

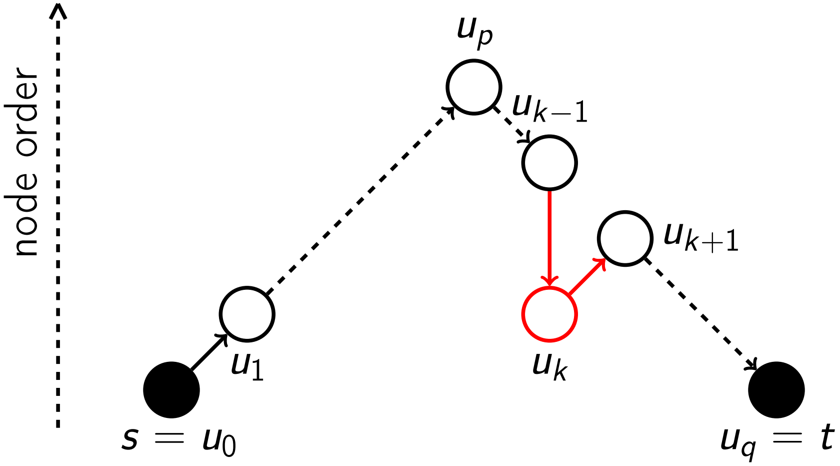


Uk is either removed
using contraction

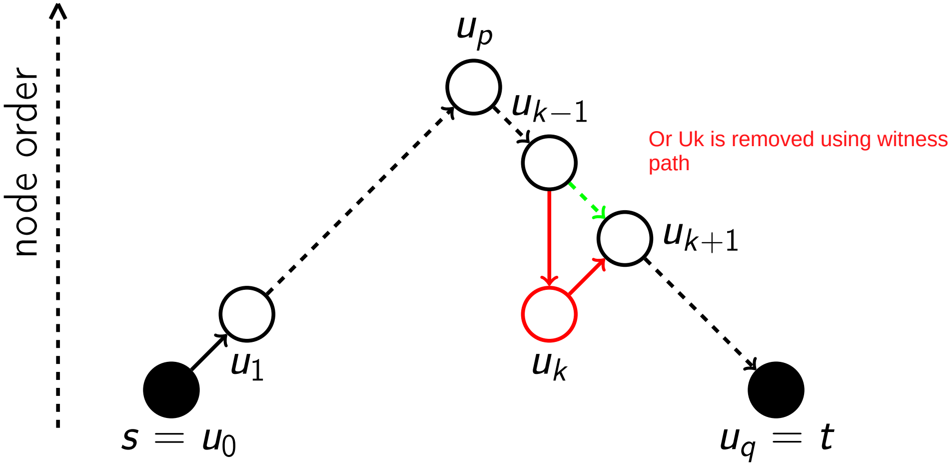
Proof Idea



Proof Idea



Proof Idea



node order
u_p
u_{k-1}
Or Uk is removed using witness path
u_{k+1}
u_1
u_k
$s = u_0$
$u_q = t$

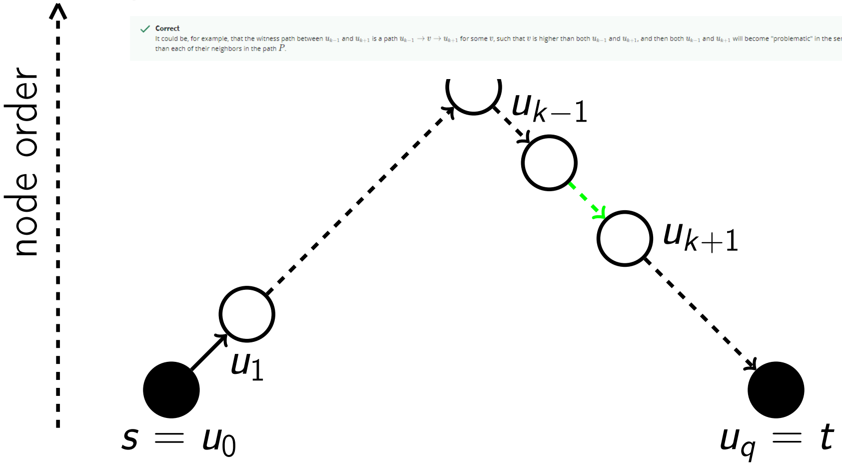
Proof Idea

Question

Is it possible that after removing u_k and edges $(u_{k-1}, u_k), (u_k, u_{k+1})$ from the path P and replacing them with the witness path $P_{u_{k-1}u_{k+1}}$ between u_{k-1} and u_{k+1} found while contracting u_k there will be even more nodes u_i in P such that both neighbors of u_i in P are higher than u_i itself?

- (selected) Yes
- No

Correct
It could be, for example, that the witness path between u_{k-1} and u_{k+1} is a path $u_{k-1} \to v \to u_{k+1}$ for some v, such that v is higher than both u_{k-1} and u_{k+1}, and then both u_{k-1} and u_{k+1} will become "problematic" in the sense that they are both lower than each of their neighbors in the path P.



Proof

- Assume for the sake of contradiction that no such path P_{st} exists

Proof

- Assume for the sake of contradiction that no such path P_{st} exists
- Then for any shortest path $P\colon s = u_1 \to u_2 \to \ldots \to u_k = t$ there is a node u_i, such that $r(u_{i-1}) > r(u_i) < r(u_{i+1})$ — call it a local minimum

Proof

- For any shortest path P between s and t, denote by $m(P)$ the minimum rank of a local minimum of this path

Proof

- For any shortest path P between s and t, denote by $m(P)$ the minimum rank of a local minimum of this path
- Consider the shortest path P^* with the maximum $m(P)$, consider the local minimum u_k with $r(u_{k-1}) > r(u_k) = m(P) < r(u_{k+1})$

Proof

- If a shortcut (u_{k-1}, u_{k+1}) was added when u_k was contracted, there is a shortest path P' with this shortcut instead of $u_{k-1} \to u_k \to u_{k+1}$, and P' doesn't contain u_k, so $m(P') > m(P^*) = r(u_k)$ — contradiction with the choice of P^* with the maximum $m(P)$

Proof

- Otherwise, there was a witness path from u_{k-1} to u_{k+1} comprised by nodes with rank higher than $r(u_k)$ (they were contracted after u_k) — there is a shortest path P'' with this path instead of $u_{k-1} \to u_k \to u_{k+1}$, and $m(P'') > m(P^*)$ — contradiction $\square$

Conclusion

- Preprocessing via node contraction

Conclusion

- Preprocessing via node contraction
- Query via Bidirectional Dijkstra in the augmented graph

Conclusion

- Preprocessing via node contraction
- Query via Bidirectional Dijkstra in the augmented graph
- Query correctness is proven

Conclusion

- Preprocessing via node contraction
- Query via Bidirectional Dijkstra in the augmented graph
- Query correctness is proven
- Are we done?

Conclusion

- Preprocessing via node contraction
- Query via Bidirectional Dijkstra in the augmented graph
- Query correctness is proven
- Are we done?
- How to select the node order?

Outline

Optimal Node Ordering

- So far our algorithm works for any node ordering

Optimal Node Ordering

- So far our algorithm works for any node ordering
- However, preprocessing and query time depend heavily on it

Optimal Node Ordering

- So far our algorithm works for any node ordering
- However, preprocessing and query time depend heavily on it
- Minimize the number of added shortcuts

Optimal Node Ordering

- So far our algorithm works for any node ordering
- However, preprocessing and query time depend heavily on it
- Minimize the number of added shortcuts
- Spread the important nodes across the graph

Optimal Node Ordering

- So far our algorithm works for any node ordering
- However, preprocessing and query time depend heavily on it
- Minimize the number of added shortcuts
- Spread the important nodes across the graph
- Minimize the number of edges in the shortest paths in the augmented graph

Order by Importance

- Introduce a measure of importance

Order by Importance

- Introduce a measure of importance
- Contract the least important node

Order by Importance

- Introduce a measure of importance
- Contract the least important node
- Importance can change after that

Algorithm

- Keep all nodes in a priority queue by decreasing importance

Algorithm

- Keep all nodes in a priority queue by decreasing importance
- On each iteration, extract the least important node

Algorithm

- Keep all nodes in a priority queue by decreasing importance
- On each iteration, extract the least important node
- Recompute its importance

Algorithm

- Keep all nodes in a priority queue by decreasing importance
- On each iteration, extract the least important node
- Recompute its importance
- If it's still minimal (compare with the top of the priority queue), contract the node

Algorithm

- Keep all nodes in a priority queue by decreasing importance
- On each iteration, extract the least important node
- Recompute its importance
- If it's still minimal (compare with the top of the priority queue), contract the node
- Otherwise, put it back into priority queue with new priority

Eventual Stopping

- If we don't contract a node, we update its importance
- After at most $|V|$ attempts all nodes have updated importance
- The node with the minimum updated importance will be contracted after that

Importance criteria

- Edge difference Related to the number of shortcuts added and minimizing the number of edges in the augmented graph
- Number of contracted neighbors Related to spreading of nodes across the graph
- Shortcut cover Related to how unavoidable the node is e.g. important node between SF and Oakland
- Node level

Edge Difference

- Want to minimize the number of edges in the augmented graph
- Number of added shortcuts $s(v)$, incoming degree $in(v)$, outgoing degree $out(v)$
- Edge difference
  $ed(v) = s(v) - in(v) - out(v)$
- Number of edges increases by $ed(v)$ after contracting v
- Contract node with small $ed(v)$

Contracted Neighbors

- Want to spread contracted nodes across the graph
- Contract a node with small number of already contracted neighbors $cn(v)$

Shortcut Cover

- Want to contract important nodes late
- Shortcut cover $sc(v)$ — the number of neighbors w of v such that we have to shortcut to or from w after contracting v
- If shortcut cover is big, many nodes "depend" on v
- Contract a node with small $sc(v)$

Node Level

- Node level $L(v)$ is an upper bound on the number of edges in the shortest path from any s to v in the augmented graph
- Initially, $L(v) \leftarrow 0$
- After contracting node v, for neighbors u of v do $L(u) \leftarrow \max(L(u), L(v) + 1)$
- Contract a node with small $L(v)$

Importance

- Use importance $I(v) = ed(v) + cn(v) + sc(v) + L(v)$
- You can play with weights of those 4 quantities in $I(v)$ and see how preprocessing time and query time change
- Each of the 4 quantities is necessary for fast preprocessing/queries
- Find a way to compute them efficiently at any stage of the preprocessing

Comparison with Dijkstra

- On a graph of Europe with 18M nodes, on random pairs of vertices Dijkstra works for 4.365s on average
- On the same graph and same random pairs, with the best set of heuristics Contraction Hierarchies work for 0.18ms on average — almost 25000 times faster!

Conclusion

- Preprocess by contracting nodes ordered approximately by importance
- Query by Bidirectional Dijkstra on the augmented graph
- Importance function is heuristic, but works well on road network graphs
- 1000s of times faster than Dijkstra
- Compete on the forums whose solution is the fastest!