

Advanced Shortest Paths: Contraction Hierarchies

Michael Levin

Higher School of Economics

Graph Algorithms
Data Structures and Algorithms

Outline

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

Learning Objectives

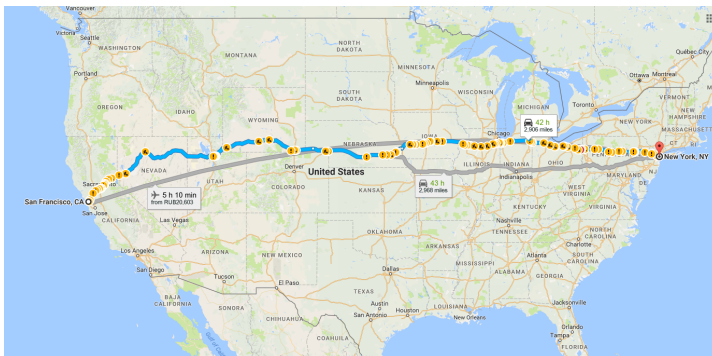
- Bidirectional Dijkstra can be 1000s of times faster than Dijkstra for social networks
- But just 2x speedup for road networks
- This lecture — great speedup for road networks

Highway Hierarchies

- Long-distance trips go through highways

Highway Hierarchies

- Long-distance trips go through highways



Highway Hierarchies

- Long-distance trips go through highways

42 h (2,906 miles)



via I-80 E

40 h without traffic

⚠ This route has tolls

San Francisco, CA

USA

> Get on US-101 S

3 min (0.6 mi)

> Follow I-80 E to Holland Tunnel in Jersey City

42 h (2,903 mi)

> Continue on Holland Tunnel. Drive to Steve Flanders Square in Manhattan, New York

10 min (2.8 mi)

New York, NY

USA

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

- 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



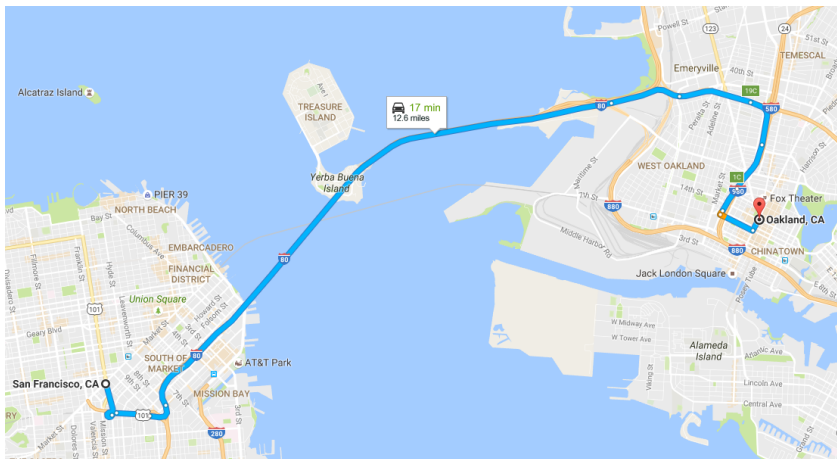
Importance Ideas

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

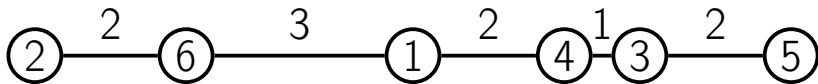
Outline

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

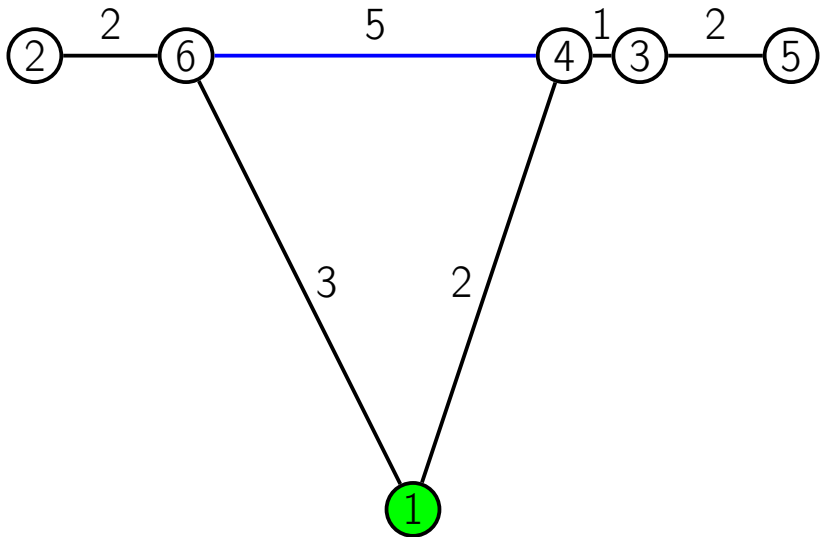
Preprocessing

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

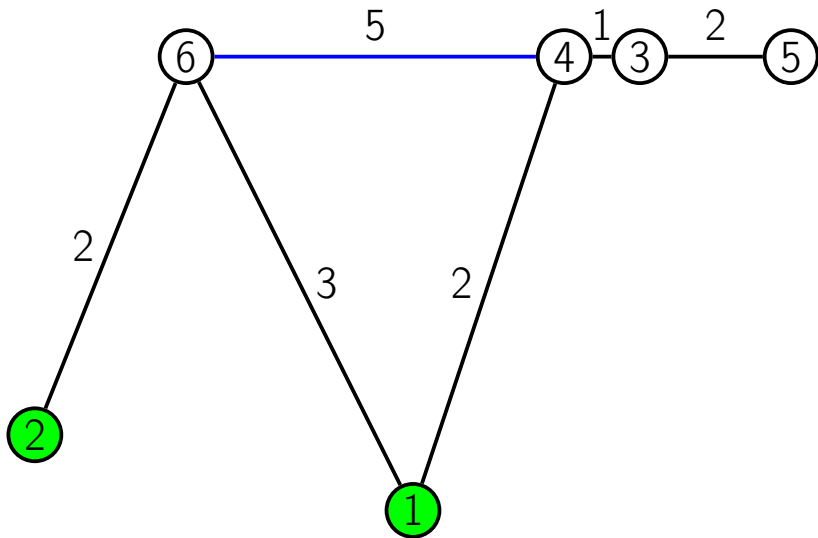
Node Contraction



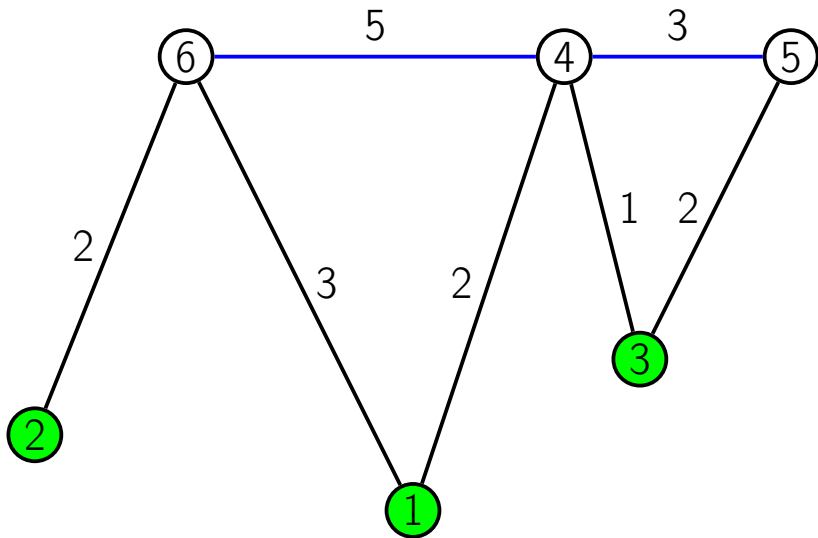
Node Contraction



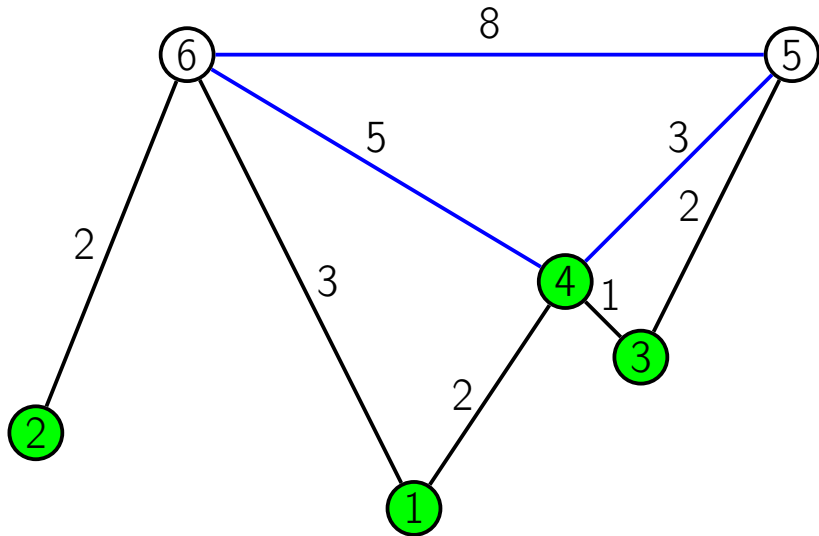
Node Contraction



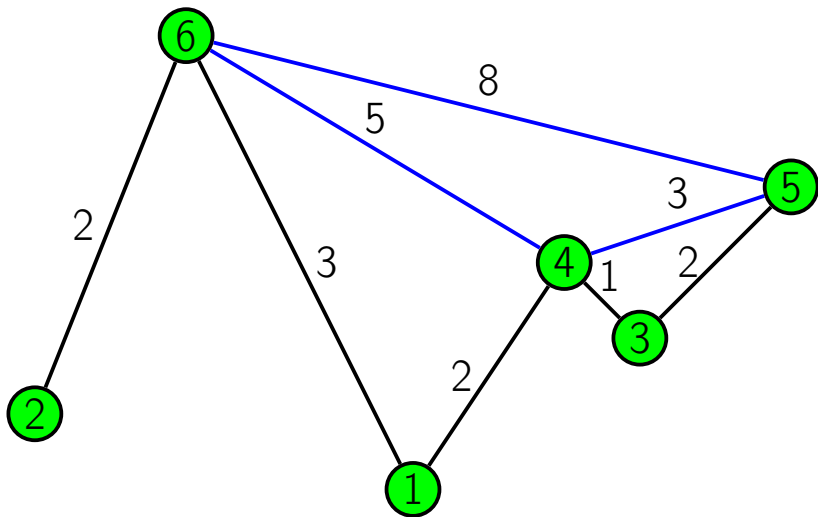
Node Contraction



Node Contraction

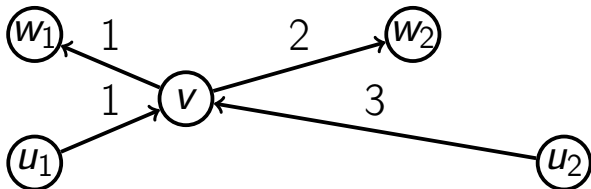


Node Contraction



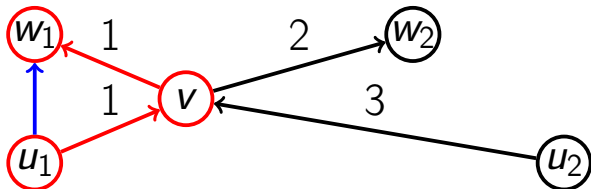
Witness Paths

- Contraction of node v



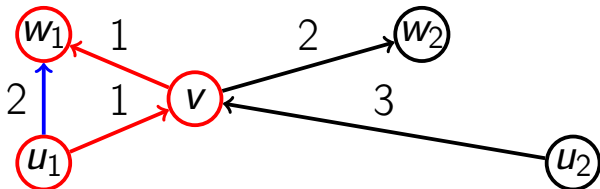
Witness Paths

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



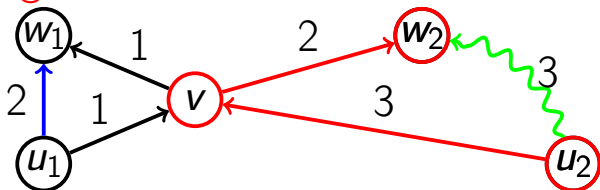
Witness Paths

- 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 Paths

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



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 $\ell(u, v) + \ell(v, w) - 1$ — then there is no need to add a shortcut from u to w .

Definition

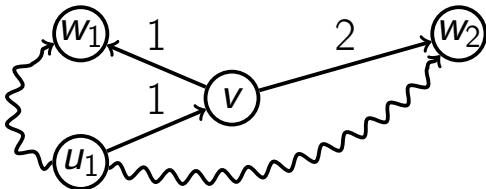
Witness search is the search for a **witness path**.

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

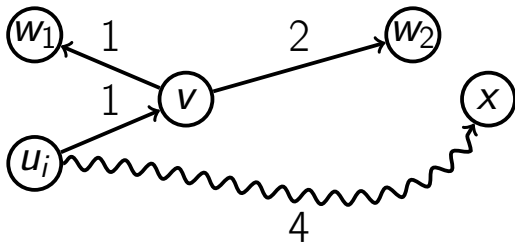


Witness Search Optimizations

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

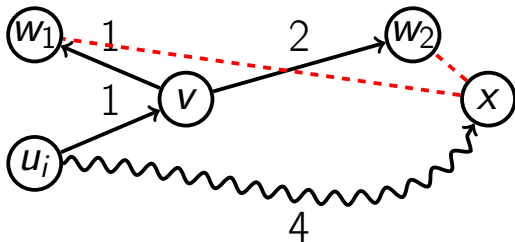
Stop Dijkstra

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



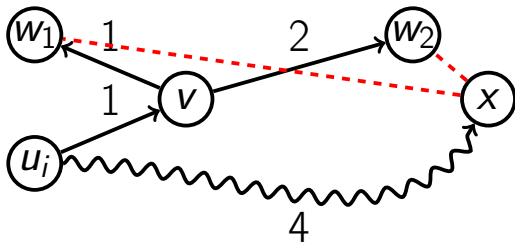
Stop Dijkstra

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



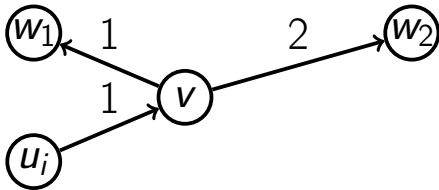
Stop Dijkstra

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



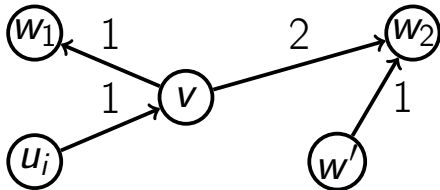
Stop Dijkstra

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



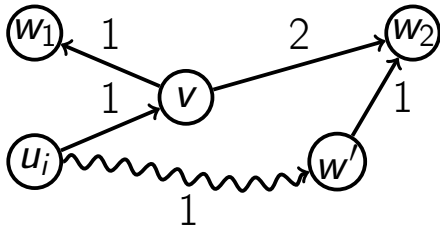
Stop Dijkstra

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



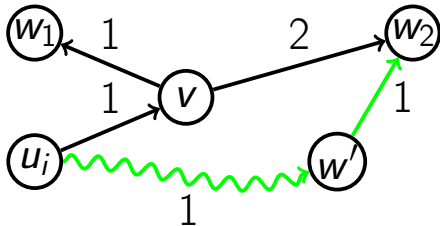
Stop Dijkstra

- 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

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

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

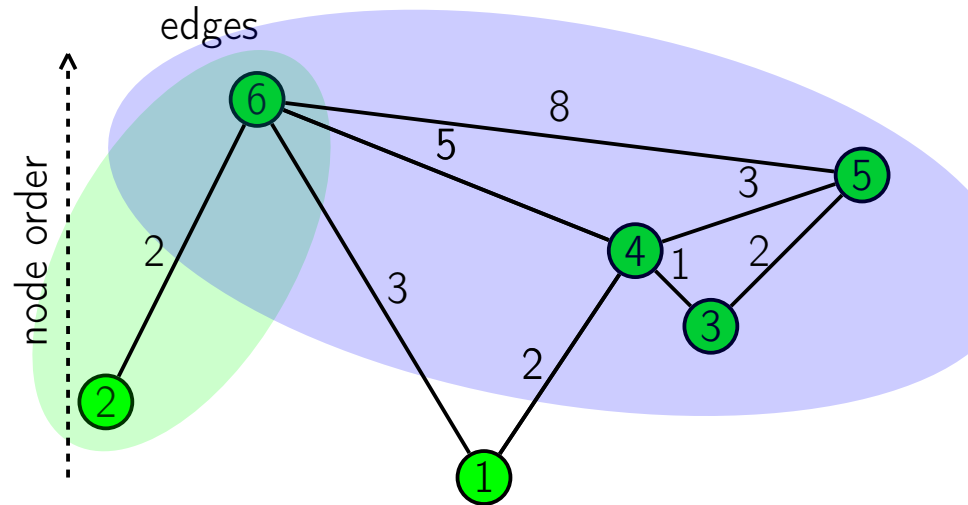
- Limit the number of “hops” in Dijkstra
- Consider only shortest paths from source with at most k edges
- If witness path not found, add a shortcut
- Tradeoff between preprocessing time and augmented graph size
- E.g., start with $k = 1$, increase gradually to $k = 5$ in the end

Outline

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

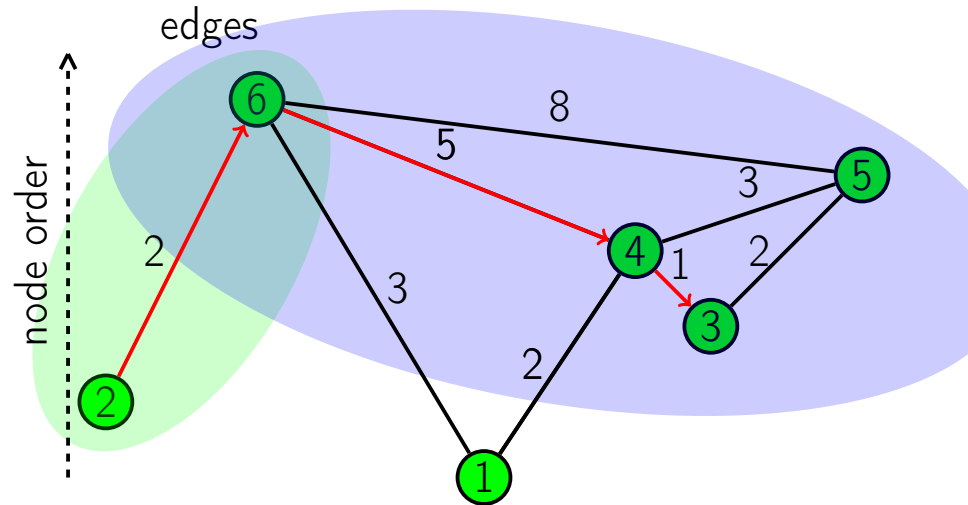
Bidirectional Dijkstra

- Bidirectional Dijkstra using only **upwards** edges



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, \dots)

```
estimate  $\leftarrow +\infty$ 
Fill dist,  $\text{dist}^R$  with  $+\infty$  for each node
dist[s]  $\leftarrow 0$ ,  $\text{dist}^R[t] \leftarrow 0$ 
proc  $\leftarrow$  empty,  $\text{proc}^R \leftarrow$  empty
while there are nodes to process:
    v  $\leftarrow$  ExtractMin(dist)
    if dist[v]  $\leq$  estimate:
        Process(v, ...)
    if v in  $\text{proc}^R$  and dist[v] +  $\text{dist}^R[v] <$  estimate:
        estimate  $\leftarrow$  dist[v] +  $\text{dist}^R[v]$ 
     $v^R \leftarrow$  ExtractMin( $\text{dist}^R$ )
    Repeat symmetrically for  $v^R$ 
return estimate
```

ComputeDistance(s, t, \dots)

$estimate \leftarrow +\infty$

Fill $dist$, $dist^R$ with $+\infty$ for each node

$dist[s] \leftarrow 0$, $dist^R[t] \leftarrow 0$

$proc \leftarrow \text{empty}$, $proc^R \leftarrow \text{empty}$

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(dist)$

 if $dist[v] \leq estimate$:

 Process(v, \dots)

 if v in $proc^R$ and $dist[v] + dist^R[v] < estimate$:

$estimate \leftarrow dist[v] + dist^R[v]$

$v^R \leftarrow \text{ExtractMin}(dist^R)$

 Repeat symmetrically for v^R

return *estimate*

ComputeDistance(s, t, \dots)

$\text{estimate} \leftarrow +\infty$

Fill dist , dist^R with $+\infty$ for each node

$\text{dist}[s] \leftarrow 0$, $\text{dist}^R[t] \leftarrow 0$

$\text{proc} \leftarrow \text{empty}$, $\text{proc}^R \leftarrow \text{empty}$

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(\text{dist})$

 if $\text{dist}[v] \leq \text{estimate}$:

 Process(v, \dots)

 if v in proc^R and $\text{dist}[v] + \text{dist}^R[v] < \text{estimate}$:

$\text{estimate} \leftarrow \text{dist}[v] + \text{dist}^R[v]$

$v^R \leftarrow \text{ExtractMin}(\text{dist}^R)$

 Repeat symmetrically for v^R

return *estimate*

ComputeDistance(s, t, \dots)

estimate $\leftarrow +\infty$

Fill dist, dist^R with $+\infty$ for each node

dist[s] $\leftarrow 0$, $\text{dist}^R[t] \leftarrow 0$

proc \leftarrow empty, $\text{proc}^R \leftarrow$ empty

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(\text{dist})$

 if dist[v] \leq estimate:

 Process(v, \dots)

 if v in proc^R and $\text{dist}[v] + \text{dist}^R[v] < \text{estimate}$:

 estimate $\leftarrow \text{dist}[v] + \text{dist}^R[v]$

$v^R \leftarrow \text{ExtractMin}(\text{dist}^R)$

 Repeat symmetrically for v^R

return *estimate*

ComputeDistance(s, t, \dots)

$\text{estimate} \leftarrow +\infty$

Fill dist , dist^R with $+\infty$ for each node

$\text{dist}[s] \leftarrow 0$, $\text{dist}^R[t] \leftarrow 0$

$\text{proc} \leftarrow \text{empty}$, $\text{proc}^R \leftarrow \text{empty}$

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(\text{dist})$

 if $\text{dist}[v] \leq \text{estimate}$:

 Process(v, \dots)

 if v in proc^R and $\text{dist}[v] + \text{dist}^R[v] < \text{estimate}$:

$\text{estimate} \leftarrow \text{dist}[v] + \text{dist}^R[v]$

$v^R \leftarrow \text{ExtractMin}(\text{dist}^R)$

 Repeat symmetrically for v^R

return *estimate*

ComputeDistance(s, t, \dots)

$estimate \leftarrow +\infty$

Fill $dist$, $dist^R$ with $+\infty$ for each node

$dist[s] \leftarrow 0$, $dist^R[t] \leftarrow 0$

$proc \leftarrow \text{empty}$, $proc^R \leftarrow \text{empty}$

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(dist)$

 if $dist[v] \leq estimate$:

 Process(v, \dots)

 if v in $proc^R$ and $dist[v] + dist^R[v] < estimate$:

$estimate \leftarrow dist[v] + dist^R[v]$

$v^R \leftarrow \text{ExtractMin}(dist^R)$

 Repeat symmetrically for v^R

return *estimate*

ComputeDistance(s, t, \dots)

$estimate \leftarrow +\infty$

Fill $dist$, $dist^R$ with $+\infty$ for each node

$dist[s] \leftarrow 0$, $dist^R[t] \leftarrow 0$

$proc \leftarrow \text{empty}$, $proc^R \leftarrow \text{empty}$

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(dist)$

 if $dist[v] \leq estimate$:

 Process(v, \dots)

 if v in $proc^R$ and $dist[v] + dist^R[v] < estimate$:

$estimate \leftarrow dist[v] + dist^R[v]$

$v^R \leftarrow \text{ExtractMin}(dist^R)$

 Repeat symmetrically for v^R

return *estimate*

ComputeDistance(s, t, \dots)

estimate $\leftarrow +\infty$

Fill dist, dist^R with $+\infty$ for each node

dist[s] $\leftarrow 0$, $\text{dist}^R[t] \leftarrow 0$

proc \leftarrow empty, $\text{proc}^R \leftarrow$ empty

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(\text{dist})$

 if dist[v] \leq estimate:

 Process(v, \dots)

 if v in proc^R and $\text{dist}[v] + \text{dist}^R[v] < \text{estimate}$:

 estimate $\leftarrow \text{dist}[v] + \text{dist}^R[v]$

$v^R \leftarrow \text{ExtractMin}(\text{dist}^R)$

 Repeat symmetrically for v^R

return estimate

ComputeDistance(s, t, \dots)

$\text{estimate} \leftarrow +\infty$

Fill dist , dist^R with $+\infty$ for each node

$\text{dist}[s] \leftarrow 0$, $\text{dist}^R[t] \leftarrow 0$

$\text{proc} \leftarrow \text{empty}$, $\text{proc}^R \leftarrow \text{empty}$

while there are nodes to process:

$v \leftarrow \text{ExtractMin}(\text{dist})$

 if $\text{dist}[v] \leq \text{estimate}$:

 Process(v, \dots)

 if v in proc^R and $\text{dist}[v] + \text{dist}^R[v] < \text{estimate}$:

$\text{estimate} \leftarrow \text{dist}[v] + \text{dist}^R[v]$

$v^R \leftarrow \text{ExtractMin}(\text{dist}^R)$

 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?

Outline

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

Augmented Graph

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 $u \rightarrow v \rightarrow w$ of length $\ell(u, v) + \ell(v, w) = \ell(u, w)$ before adding this shortcut, so $d^+(s, t)$ can't be less than $d(s, t)$
- Thus $d^+(s, t) = d(s, t)$



Definition

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

Definition

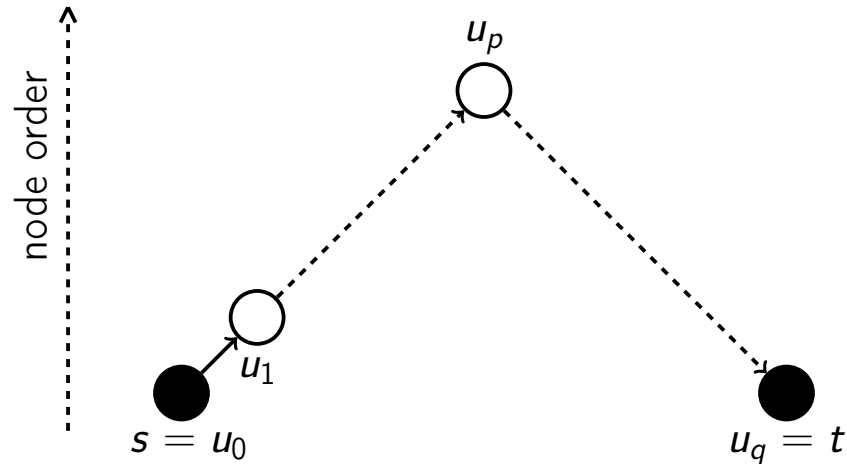
A path $P: v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ in the augmented graph G^+ is called **increasing** if $r(v_1) < r(v_2) < \dots < r(v_k)$. Similarly, P is called **decreasing** if $r(v_1) > r(v_2) > \dots > r(v_k)$.

Justification of Bidirectional Search

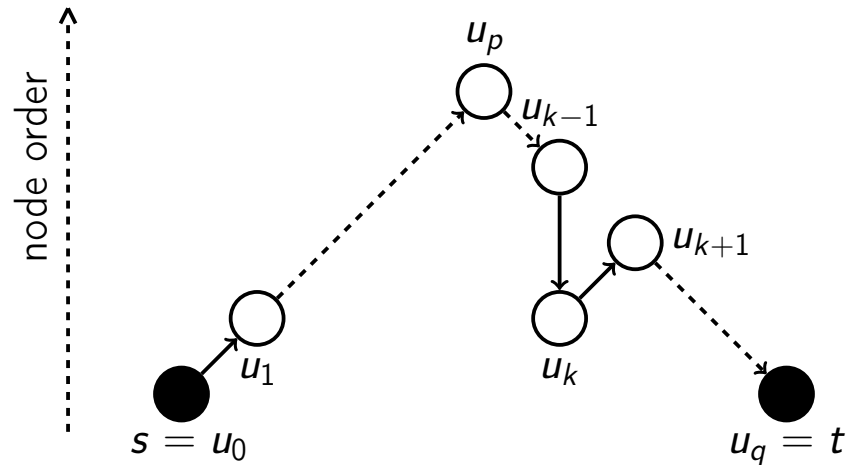
Lemma

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.

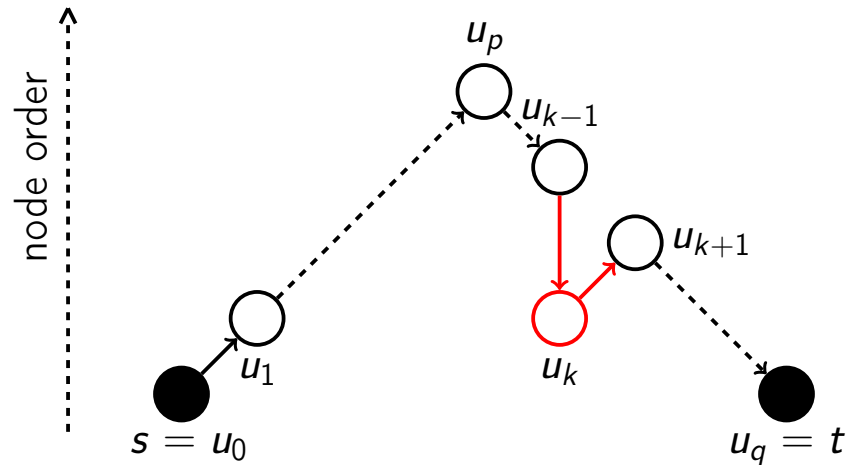
Proof Idea



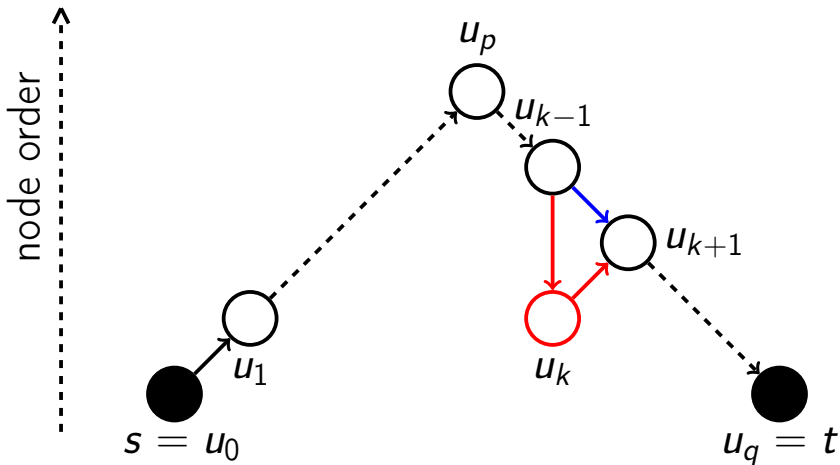
Proof Idea



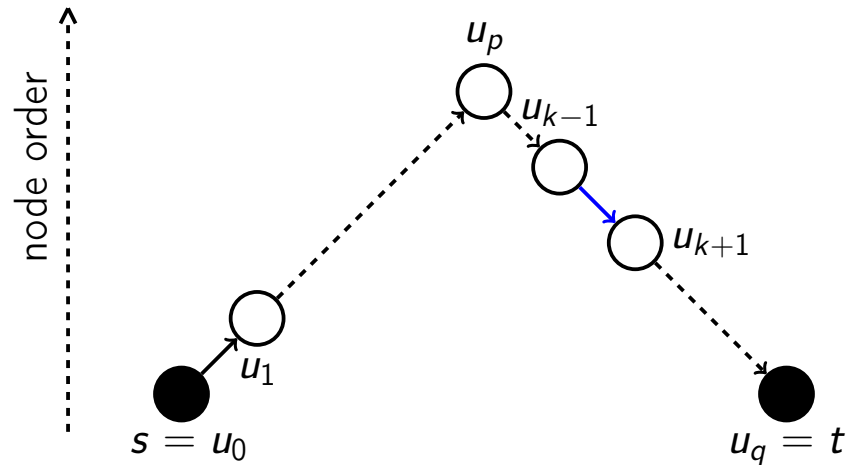
Proof Idea



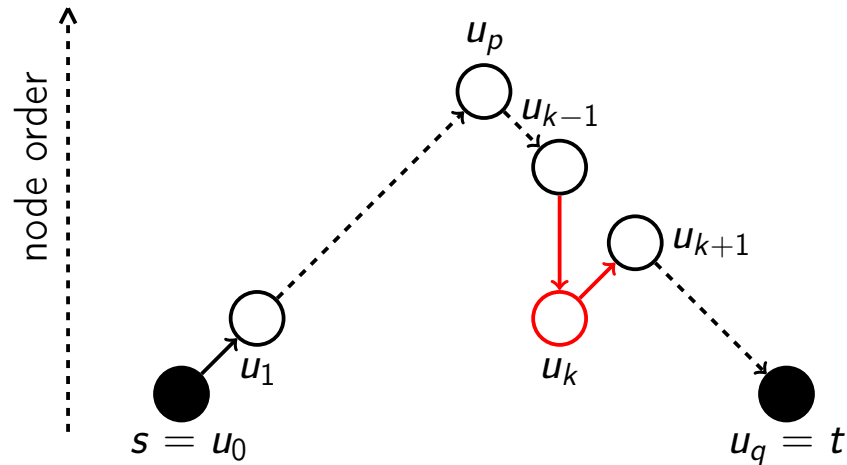
Proof Idea



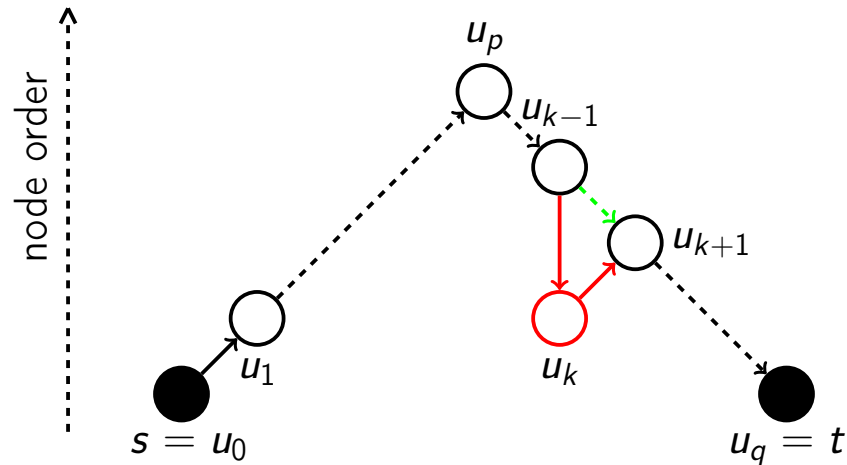
Proof Idea



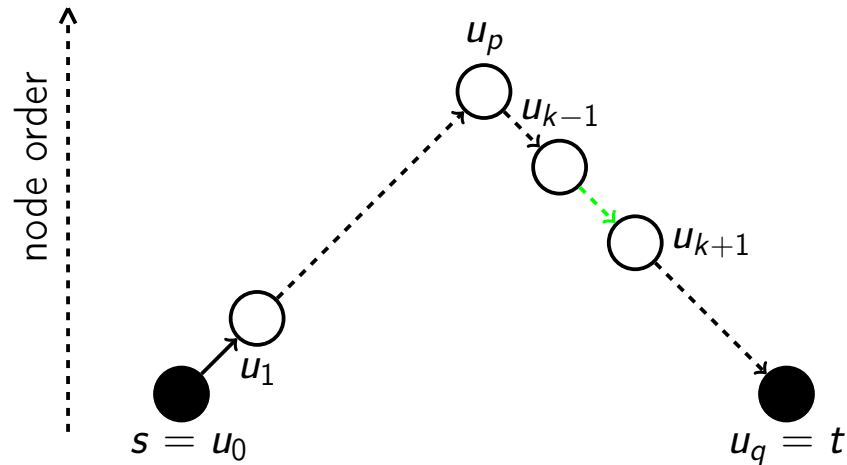
Proof Idea



Proof Idea



Proof Idea



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 $P: s = u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_k = t$ 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 $u_{k-1} \rightarrow u_k \rightarrow u_{k+1}$, 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 $u_{k-1} \rightarrow u_k \rightarrow u_{k+1}$, and $m(P'') > m(P^*)$ — contradiction □

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

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

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
- Number of contracted neighbors
- Shortcut cover
- 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!