Shortest Paths: Algorithms for standard variants

Algorithms and Networks 2017/2018
Johan M. M. van Rooij
Hans L. Bodlaender
Shortest path problem(s)

**Undirected single-pair shortest path problem**
- Given a graph $G=(V,E)$ and a length function $\ell: E \rightarrow \mathbb{R}_{\geq 0}$ on the edges, a start vertex $s \in V$, and a target vertex $t \in V$, find the shortest path from $s$ to $t$ in $G$.
  - The length of the path is the sum of the lengths of the edges.

**Variants:**
- Directed graphs.
  - Travel on edges in one direction only, or with different lengths.
- More required than a single-pair shortest path.
  - Single-source, single-target, or all-pairs shortest path problem.
- Unit length edges vs a length function.
- Positive lengths vs negative lengths, but no negative cycles.
  - Why no negative cycles? Directed Acyclic Graphs?
Shortest paths and other courses

- Algorithms course (Bachelor, level 3)
  - Dijkstra.
  - Bellman-Ford.
  - Floyd-Warshall.

- Crowd simulation course (Master, period 2)
  - Previously known as the course ‘Path Planning’.
    - A* and bidirectional Dijkstra (maybe also other courses).
    - Many extensions to this.
Shortest paths in algorithms and networks

This lecture:
- Recap on what you should know.
  - Floyd-Warshall.
  - Bellman-Ford.
  - Dijkstra.
- Using height functions.
  - Optimisation to Dijkstra: A* and bidirectional search.
  - Johnson’s algorithm (all-pairs).

Next week:
- Gabow’s algorithm (using the numbers).
- Large scale shortest paths algorithms in practice.
  - Contraction hierarchies.
  - Partitioning using natural cuts.
Applications

- Route planning.
  - Shortest route from A to B.
  - Subproblem in vehicle routing problems.
  - Preprocessing for travelling salesman problem on graphs.

- Subroutine in other graph algorithms.
  - Preprocessing in facility location problems.
  - Subproblem in several flow algorithms.

- Many other problems can be modelled a shortest path problems.
  - State based problems where paths are sequences of state transitions.
  - Longest paths on directed acyclic graphs.
Notation and basic assumption

Notation
- $|V| = n$, $|E| = m$.
- For directed graphs we use $A$ instead of $E$.
- $d_l(s, t)$: distance from $s$ to $t$: length of shortest path from $s$ to $t$ when using edge length function $l$.
- $d(s, t)$: the same, but $l$ is clear from the context.
- In single-pair, single-source, and/or single-target variants, $s$ is the source vertex and $t$ is the target vertex.

Assumption
- $d(s, s) = 0$: we always assume there is a path with 0 edges from a vertex to itself.
Shortest Paths – Algorithms and Networks

RECAP ON ALGORITHMS YOU SHOULD KNOW
Algorithms on (directed) graphs with negative weights

- **Floyd-Warshall:**
  - In $O(n^3)$ time: all pairs shortest paths.
  - For instance with negative weights, no negative cycles.

- **Bellman-Ford algorithm:**
  - In $O(nm)$ time: single source shortest path problem
  - For instance with negative weights, no negative cycles reachable from $s$.
  - Also: detects whether a negative cycle exists.
Floyd-Warshall, 1962
All-Pairs Shortest Paths

**Algorithm:**
1. Initialise: $D(u,v) = \ell(u,v)$ for all $(u, v) \in A$
   
   $D(u,v) = \infty$ otherwise
2. For all $u \in V$
   - For all $v \in V$
     - For all $w \in V$
       - $D(v,w) = \min\{ D(v,u) + D(u,w), D(v,w) \}$

- Dynamic programming in $O(n^3)$ time.
- **Invariant:** (outer loop)
  - $D(v,w)$ is length of the shortest path from $v$ to $w$ only visiting
    vertices that have had the role of $u$ in the outer loop.
- Correctness follows from the invariant.
Bellman-Ford algorithm, 1956
Single source for graphs with negative lengths

1. Initialize: set $D[v] = \infty$ for all $v \in V \setminus \{s\}$. set $D[s] = 0$ (s is the source).

2. Repeat $|V|-1$ times:
   - For every edge $(u,v) \in A$:
     $D[v] = \min\{ D[v], D[u] + \ell(u,v) \}$.

3. For every edge $(u,v) \in A$
   - If $D[v] > D[u] + \ell(u,v)$ then there exists a negative cycle.

**Invariant:** If there is no negative cycle reachable from $s$, then after $i$ runs of main loop, we have:
   - If there is a shortest path from $s$ to $u$ with at most $i$ edges, then $D[u] = d(s,u)$, for all $u$.

   - If there is no negative cycle reachable from $s$, then every vertex has a shortest path with at most $n - 1$ edges.

   - If there is a negative cycle reachable from $s$, then there will always be an edge where an update step is possible.

Clearly: $O(nm)$ time
Finding a negative cycle in a graph

- Reachable from s:
  - Apply Bellman-Ford, and look back with pointers.
  - Or: add a vertex s with edges to each vertex in G.
Basics of single source algorithms (including Bellman-Ford)

- Each vertex \( v \) has a variable \( D[v] \).
  - Invariant: \( d(s,v) \leq D[v] \) for all \( v \)
  - Initially: \( D[s] = 0; \, v \neq s: \, D[v] = \infty \)
  - \( D[v] \) is the shortest distance from \( s \) to \( v \) found thus far.

- Update step over edge \((u,v)\):
  - \( D[v] = \min\{ D[v], D[u] + \ell(u,v) \} \).

- To compute a shortest path (not only the length), one could maintain pointers to the `previous vertex on the current shortest path' (sometimes NULL): \( p(v) \).
  - Initially: \( p(v) = \text{NULL} \) for each \( v \).
  - Update step becomes:
    - If \( D[v] > D[u] + \ell(u,v) \) then
      - \( D[v] = D[u] + \ell(u,v) \);
      - \( p(v) = u \);
    - End if;

\( p \)-values build paths of length \( D[v] \). Shortest paths tree!
Almost all roads lead to Rome.
Visualising a shortest paths tree...
What Rome?
Dijkstra’s algorithm, 1956

Dijkstra’s Algorithm
1. Initialize: set $D[v] = \infty$ for all $v \in V \setminus \{s\}$, $D[s] = 0$.
2. Take priority queue $Q$, initially containing all vertices.
3. While $Q$ is not empty,
   - Select vertex $v$ from $Q$ with minimum value $D[v]$.
   - Update $D[u]$ across all outgoing edges $(v,u)$.
     - $D[v] = \min\{ D[v], D[u] + \ell(u,v) \}$.
   - Update the priority queue $Q$ for all such $u$.

- Assumes all lengths are non-negative.
- Correctness proof (done in `Algoritmiek` course).

- Note: if all edges have unit-length, then this becomes **Breath-First Search**.
  - Priority queue only has priorities 1 and $\infty$. 
On Dijkstra’s algorithm

Running time depends on the data structure chosen for the priority queue.
- Selection happens at most n times.
- Updates happen at most m times.

Depending on the data structure used, the running time is:
- O(n^2): array.
  - Selection in O(n) time, updates in O(1) time.
- O((m + n) log n): Red-black tree (or other), heap.
  - Selection and updates in O(log n).
- O(m + n log n): Fibonacci heaps.
  - Selection (delete min) in amortised O(log n), update in amortised O(1) time.
OPTIMISATIONS FOR DIJKSTRA’S ALGORITHM
Optimisation: bidirectional search

- For a single pair shortest path problem:
  - Start a Dijkstra-search from both sides simultaneously.
  - Analysis needed for more complicated stopping criterion.
  - Faster in practice.

- Combines nicely with another optimisation that we will see next (A*).
Consider shortest paths in geometric setting.
- For example: route planning for car system.

Standard Dijkstra would explore many paths that are clearly in the wrong direction.
- Utrecht to Groningen would look at roads near Den Bosch or even Maastricht.

A* modifies the edge length function used to direct Dijkstra’s algorithm into the right direction.
- To do so, we will use a heuristic as a height function.
Modifying distances with height functions

Let \( h: V \rightarrow \mathbb{R}_{\geq 0} \) be any function to the positive reals.

Define new lengths \( \ell_h \): \( \ell_h(u,v) = \ell(u,v) - h(u) + h(v) \).

- Modify distances according to the height \( h(v) \) of a vertex \( v \).

**Lemmas:**

1. For any path \( P \) from \( u \) to \( v \): \( \ell_h(P) = \ell(P) - h(u) + h(v) \).
2. For any two vertices \( u, v \): \( d_h(u,v) = d(u,v) - h(u) + h(v) \).
3. \( P \) is a shortest path from \( u \) to \( v \) with lengths \( \ell \), if and only if, it is so with lengths \( \ell_h \).

- Height function is often called a **potential function**.
  - We will use height functions more often in this lecture!
A heuristic for the distance to the target

Definitions:

- Let $h: V \rightarrow \mathbb{R}_{\geq 0}$ be a heuristic that approximates the distance to target $t$.
  - For example, Euclidean distance to target on the plane.
- Use $h$ as a height function: $\ell_h(u, v) = \ell(u, v) - h(u) + h(v)$.
  - The new distance function measures the deviation from the Euclidean distance to the target.
- We call $h$ admissible if:
  - for each vertex $v$: $h(v) \leq d(v, t)$.
- We call $h$ consistent if:
  - For each $(u, v)$ in $E$: $h(u) \leq \ell(u, v) + h(v)$.
- Euclidean distance is admissible and consistent.
A* algorithm uses an admissible heuristic

- A* using an admissible and consistent heuristic $h$ as height function is an optimisation to Dijkstra.
- We call $h$ **admissible** if:
  - For each vertex $v$: $h(v) \leq d(v, t)$.
  - **Consequence**: never stop too early while running A*.
- We call $h$ **consistent** if:
  - For each $(u,v)$ in $E$: $h(u) \leq \ell(u,v) + h(v)$.
  - **Consequence**: all new lengths $\ell_h(u,v)$ are non-negative.
  - If $h(t) = 0$, then consistent implies admissible.

- A* without a consistent heuristic can take exponential time.
  - Then it is not an optimisation of Dijkstra, but allows vertices to be reinserted into the priority queue.
  - When the heuristic is admissible, this guarantees that A* is correct (stops when the solution is found).
A* and consistent heuristics

In A* with a consistent heuristic arcs/edges in the wrong direction are less frequently used than in standard Dijkstra.

- Faster algorithm, but still correct.

Well, the quality of the heuristic matters.

- $h(v) = 0$ for all vertices $v$ is consistent and admissible but useless.
- Euclidian distance can be a good heuristic for shortest paths in road networks.
Advanced Shortest Path Algorithms

Algorithms and Networks 2017/2018
Johan M. M. van Rooij
Hans L. Bodlaender
New Planning for Shortest Paths Subjects

- Last week:
  - Recap on what you should know.
    - Floyd-Warshall.
    - Bellman-Ford.
    - Dijkstra.
  - Using height functions.
    - Optimisation to Dijkstra: A* and bidirectional search.

- This lecture:
  - Johnson’s algorithm (all-pairs).
  - Gabow’s algorithm (using the numbers).

- Probably in the T.B.A. slot in week 50 (before first exam).
  - Large scale shortest paths algorithms in practice.
    - Contraction hierarchies
    - Partitioning using natural cuts.
JOHNSON’S ALGORITHM
All Pairs Shortest Paths: Johnson’s Algorithm

Observation

- If all weights are non-negative we can run Dijkstra with each vertex as starting vertex.
  - This gives $O(n^2 \log n + nm)$ time using a Fibonacci heap.
  - On sparse graphs, this is faster than the $O(n^3)$ of Floyd-Warshall.

- **Johnson**: all-pairs shortest paths improvement for sparse graphs with reweighting technique:
  - $O(n^2 \log n + nm)$ time.
  - Works with negative lengths, but no negative cycles.
  - Reweighting using height functions.
A recap on height functions

Let \( h: V \rightarrow \mathbb{R} \) be any function to the reals.

Define new lengths \( \ell_h: \ell_h(u,v) = \ell(u,v) - h(u) + h(v) \).

Modify distances according to the height \( h(v) \) of a vertex \( v \).

**Lemmas:**
1. For any two vertices \( u, v \): \( d_h(u,v) = d(u,v) - h(u) + h(v) \).
2. For any path \( P \) from \( u \) to \( v \): \( \ell_h(P) = \ell(P) - h(u) + h(v) \).
3. \( P \) is a shortest path from \( u \) to \( v \) with lengths \( \ell \), if and only if, it is so with lengths \( \ell_h \).

**New lemma:**
4. \( G \) has a negative-length circuit with lengths \( \ell \), if and only if, it has a negative-length circuit with lengths \( \ell_h \).
What height function $h$ is good?

- Look for height function $h$ such that:
  - $\ell_h(u, v) \geq 0$, for all edges $(u, v)$.

- If so, we can:
  - Compute $\ell_h(u, v)$ for all edges.
  - Run Dijkstra but now with $\ell_h(u, v)$.

- We will construct a good height function $h$ by solving a single-source shortest path problem using Bellman-Ford.
Choosing $h$

1. Add a new vertex $s$ to the graph.
   - If negative cycle detected: stop.
3. Set $h(v) = -d(s,v)$

Note: for all edges $(u,v)$:

- $l_h(u,v) = l(u,v) - h(u) + h(v)$
  $= l(u,v) + d(s,u) - d(s,v) \geq 0$
  because: $d(s,u) + l(u,v) \geq d(s,v)$
Johnson’s algorithm

1. Build graph $G'$ (as shown).
2. Compute $d(s,v)$ for all $v$ using Bellman-Ford.
3. Set $l_h(u,v) = l(u,v) + d_{G'}(s,u) - d_{G'}(s,v)$ for all $(u,v) \in A$.
4. For all $u$ do:
   - Use Dijkstra’s algorithm to compute $d_h(u,v)$ for all $v$.
   - Set $d(u,v) = d_h(u,v) - d_{G'}(s,u) + d_{G'}(s,v)$.

Running time:
- $O(nm)$ for the single call to Bellman-Ford.
- $n$ times a call to Dijkstra in $O(m + n \log n)$.

$O(n^2 \log n + nm)$ time
GABOW’S ALGORITHM: USING THE NUMBERS
Using the numbers

Consider the single source shortest paths problem with non-negative integer distances.
- Suppose $\Delta$ is an upper bound on the maximum distance from $s$ to a vertex $v$.
- Let $L$ be the largest length of an edge.
- Single source shortest path problem is solvable in $O(m + \Delta)$ time.

Compare to Dijkstra with Fibonacci heap: $O(m + n \log n)$.
- If $\Delta$ of $O(m)$, then this algorithm is linear.
In $O(m + \Delta)$ time

- We use Dijkstra, using $D[v]$ for length of the shortest path found thus far, using the following as a priority queue.
     - **Invariant:** for all $v$ with $D[v] \leq \Delta$: $v$ is in $L[D[v]]$.
  2. Keep a current minimum $\mu$.
     - **Invariant:** all $L[k]$ with $k < \mu$ are empty.

Run ‘Dijkstra’ while:
  - Update $D[v]$ from $x$ to $y$: take $v$ from $L[x]$, and add it to $L[y]$. This takes $O(1)$ time each.
  - Extract min: while $L[\mu]$ empty, $\mu ++$; then take the first element from list $L[\mu]$.

- Total time: $O(m + \Delta)$
Corollary and extension

- We can solve single-source shortest paths (without negative arc-lengths) in $O(m+\Delta)$ time.

- **Corollary:** Single-source shortest path in $O(m+nL)$ time.
  - Take $\Delta=nL$.

- **Extension:** Gabow (1985): Single-source shortest path problem can be solved in $O(m \log_R L)$ time, where:
  - $R = \max\{2, \frac{m}{n}\}$.
  - $L$: maximum length of edge.
- Gabow’s algorithm uses a scaling technique!
Gabow’s Algorithm: Main Idea

**Sketch of the algorithm:**

- First, build a scaled instance:
  - For each edge $e$ set $\ell'(e) = \lfloor \ell(e) / R \rfloor$.

- Recursively, solve the scaled instance and switch to using Dijkstra ‘using the numbers’ if weights are small enough.

- $R \times d_{\ell'}(s,v)$ is when we scale back our scaled instance.
  - We want $d(s,v)$.
  - What error did we make while rounding?

- Another shortest paths instance can be used to compute the error correction terms on the shortest paths!
  - How does this work? See next slides.
Computing the correction terms through another shortest path problem

Set for each arc \((x,y) \in A:\)
- \(Z(x,y) = \ell(x,y) + R \cdot d_{l'}(s,x) - R \cdot d_{l'}(s,y)\)
- Works like a height function, so the same shortest paths!
- Height function \(h(x) = -R \cdot d_{l'}(s,x)\)
- \(Z\) compares the differences of the shortest paths (with rounding error) from \(s\) to \(x\) and \(y\) to the edge length \(\ell(x,y)\).

Claim: For all vertices \(v\) in \(V:\)
- \(d(s,v) = d_Z(s,v) + R \cdot d_{l'}(s,v)\)

Proof by property of the height function:
- \(d_Z(s,v) = d(s,v) - h(s) + h(v)\)
  - \(= d(s,v) + R \cdot d_{l'}(s,s) - R \cdot d_{l'}(s,v)\) \((d_{l'}(s,s) = 0)\)
  - \(= d(s,v) - R \cdot d_{l'}(s,v)\) \((\text{next reorder})\)

Thus, we can compute distances for \(\ell\) by computing distances for \(Z\) and for \(l'\).
Gabow’s algorithm

Algorithm
1. If $L \leq R$, then:
   - Solve the problem using the $O(m+nL)$ algorithm (Base case)
2. Else:
   - For each edge $e$: set $l'(e) = \lceil l(e) / R \rceil$.
   - Recursively, compute the distances but with the new length function $l'$.
   - Set for each edge $(u,v)$:
     $$Z(u,v) = l(u,v) + R \cdot d'(s,u) - R \cdot d'(s,v).$$
   - Compute $d_Z(s,v)$ for all $v$ (how? After the example!)
   - Compute $d(s,v)$ using:
     $$d(s,v) = d_Z(s,v) + R \cdot d'(s,v).$$
Example
A property of Z

For each arc \((u,v) \in A\) we have:
\[
Z(u,v) = l(u,v) + R \cdot d\ell'(s,u) - R \cdot d\ell'(s,v) \geq 0
\]

Proof:
\[
\begin{align*}
&d\ell'(s,u) + l'(u,v) \geq d\ell'(s,v) & \text{(triangle inequality)} \\
l'(u,v) \geq d\ell'(s,v) - d\ell'(s,u) & \text{(rearrange).} \\
R \cdot l'(u,v) \geq R \cdot (d\ell'(s,v) - d\ell'(s,u)) & \text{(times R).} \\
l(u,v) \geq R \cdot l'(u,v) \geq R \cdot (d\ell'(s,v) - d\ell'(s,u)) & \text{(definition of } l(u,v)) \\
l(u,v) + R \cdot d\ell'(s,u) - R \cdot d\ell'(s,v) \geq 0
\end{align*}
\]

Therefore, a variant of Dijkstra can be used to compute distances for Z.
Computing distances for Z

For each vertex v we have:
\[ d_Z(s,v) \leq nR \] for all v reachable from s

Proof:
- Consider a shortest path P for distance function \( t' \) from s to v.
- For each of the less than n edges e on P, \( t(e) \leq R + R*t'(e) \).
- So, \( d(s,v) \leq t(P) \leq nR + R*t'(P) = nR + R * d_{t'}(s,v) \).
- Use that \( d(s,v) = d_Z(s,v) + R * d_{t'}(s,v) \).

So, we can use the O(m+nR) algorithm (Dijkstra with doubly-linked lists) to compute all values \( d_Z(v) \).
Running time of Gabow’s algorithm?

**Algorithm**

1. If \( L \leq R \), then
   - solve the problem using the \( O(m+nR) \) algorithm (Base case)

2. Else
   - For each edge \( e \): set \( l'(e) = \lfloor l(e) / R \rfloor \).
   - Recursively, compute the distances but with the new length function \( l' \).
   - Set for each edge \((u,v)\):
     \[ Z(u,v) = l(u,v) + R \cdot d_{l'}(s,u) - R \cdot d_{l'}(s,v) \]
   - Compute \( d_Z(s,v) \) for all \( v \) *(how? After the example!)*
   - Compute \( d(s,v) \) using:
     \[ d(s,v) = d_Z(s,v) + R \cdot d_{l'}(s,v) \]

**Gabow’s algorithm uses** \( O(m \log_R L) \) time.
Large Scale Practical Shortest Path Algorithms

Algorithms and Networks 2017/2018
Johan M. M. van Rooij
Hans L. Bodlaender
Large scale shortest paths algorithms

- The world’s road network is huge.
  - Open street map has: 2,750,000,000 nodes.
  - Standard (bidirectional) Dijkstra takes too long.
  - Many-to-many computations are very challenging.

- We briefly consider two algorithms.
  - Contraction Hierarchies.
  - Partitioning through natural cuts.

- Both algorithms have two phases.
  - Time consuming preprocessing phase.
  - Very fast shortest paths queries.

- My goal: give you a rough idea on these algorithms.
  - Look up further details if you want to.
Contraction hierarchies

- Every node gets an extra number: its level of the hierarchy.
  - In theory: can be the numbers 1, 2, ..., |V|.
  - In practice: low numbers are crossings of local roads, high numbers are highway intersections.

- We want to run bidirectional Dijkstra (or a variant) such that:
  - The forward search only considers going ‘up’ in the hierarchy.
  - The backward search only considers going ‘down’ in the hierarchy.

If levels chosen wisely, a lot less options to explore.
Shortcuts in contraction hierarchies

- We want to run bidirectional Dijkstra (or a variant) such that:
  - The forward search only considers going ‘up’ in the hierarchy.
  - The backward search only considers going ‘down’ in the hierarchy.

- This will only be correct if we add shortcuts.
  - Additional edges from lower level nodes to higher level nodes are added as ‘shortcuts’ to preserve correctness.
  - Example: shortest path from u to v goes through lower level node w: add shortcut from u to v.
  - Shortcuts are labelled with the shortest paths they bypass.
Preprocessing for contraction hierarchies

Preprocessing phase:
- Assign a level to each node.
- For each node, starting from the lowest level nodes upwards:
  - Consider it’s adjacent nodes of higher level and check whether the shortest path between these two nodes goes through the current node.
  - If so, add a shortcut edge between the adjacent nodes.

- Great way of using preprocessing to accelerate lots of shortest paths computations.
- But:
  - This preprocessing is very expensive on huge graphs.
  - Resulting graph can have many more edges (space requirement).
  - If the graph changes locally, lots of recomputation required.
Partitioning using natural cuts

**Observation**
- Shortest paths from cities in Brabant to cities above the rivers have little options of crossing the rivers.
  - This is a natural small-cut in the graph.
- Many such cuts exist at various levels:
  - Sea’s, rivers, canals, ...
  - Mountains, hills, ...
  - Country-borders, village borders, ...
  - Etc...
Example of natural cuts

From: Delling, Goldberg, Razenshteyn, Werneck.
Graph Partitioning with Natural Cuts.
Using natural cuts

- (Large) parts of the graph having a small cut can be replaced by equivalent smaller subgraphs.
  - Mostly a clique on the boundary vertices.
  - Can be a different graph also (modelling highways).
Computing shortest paths using natural cuts

- When computing shortest paths:
  - For s and t, compute the distance to all partition-boundary vertices of the corresponding partition.
  - Replace all other partitions by their simplifications.
  - Graph is much smaller than the original graph.

- This idea can be used hierarchically.
  - First partition at the continent/country level.
  - Partition each part into smaller parts.
  - Etc.

- Finding good partitions with small natural cuts at the lowest level is easy.
- Finding a good hierarchy is difficult.
Finding natural cuts

Repeat the following process a lot of times:

- Pick a random vertex v.
- Use breadth first search to find a group of vertices around v.
  - Stop when we have x vertices in total → this is the core.
  - Continue until we have y vertices in total → this is the ring.
- Use max-flow min-cut to compute the min cut between the core and vertices outside the ring.
  - The resulting cut could be part of natural cut.

After enough repetitions:

- We find lots of cuts, creating a lot of islands.
- Recombine these islands to proper size partitions.
- Recombining the islands to a useful hierarchy is difficult.
**Conclusion on using natural cuts**

Natural cuts preprocessing has the advantage that:

- Space requirement less extreme than contraction hierarchies, because hierarchical partitioning of the graph into regions.
- Local changes to the graph can be processed with less recomputation time.
- Precomputation can also be used if travel times are departure time dependent.

Preprocessing is still very expensive on huge graphs.

**Last year experimentation project**: find a good algorithm for building a hierarchical natural cuts partitioning on real data.

  - Conclusion: hierarchical partitioning is best using 2-cuts.

**Possible new experimentation/thesis project**: Find effective 2-partitioning algorithms for this purpose.
CONCLUSION
Summary on Shortest Paths

- We have seen:
  - Recap on what you should know.
    - Floyd-Warshall.
    - Bellman-Ford.
    - Dijkstra.
  - Using height functions.
    - Optimisation to Dijkstra: A* and bidirectional search.
    - Johnson’s algorithm (all-pairs).
  - Gabow’s algorithm (using the numbers).
  - Large scale shortest paths algorithms in practice.
    - Contraction hierarchies.
    - Partitioning using natural cuts.

- Any questions?