Weighted Graphs and Algorithms

Advanced Programming and Algorithmic Design

Alberto Casagrande
Email: `acasagrande@units.it`

a.y. 2019/2020

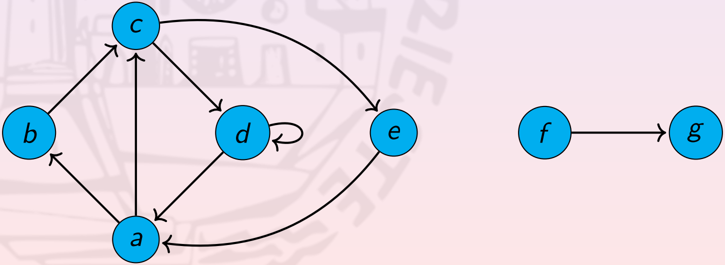
Weighted Graphs (Graph Theory)

Are tuples (V, E, W) where:

Weighted Graphs (Graph Theory)

Are tuples (V, E, W) where:

(V, E) is an (un)directed graph

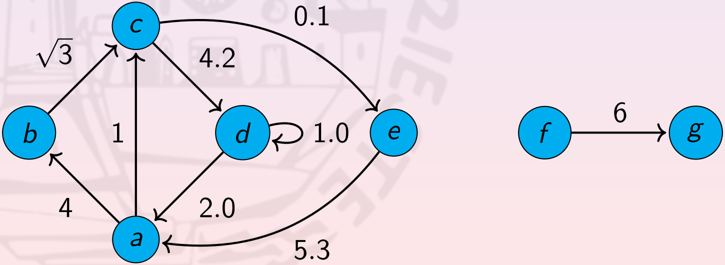


Weighted Graphs (Graph Theory)

Are tuples (V, E, W) where:

(V, E) is an (un)directed graph

W is a function mapping edges into weights



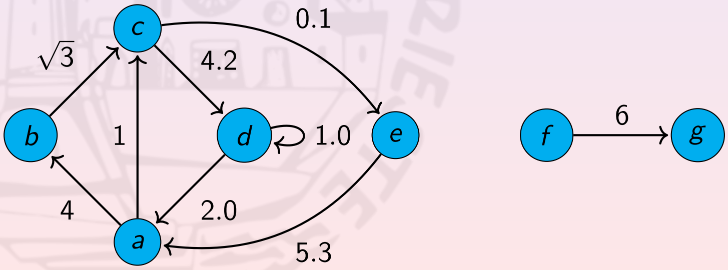
Weighted Graphs (Graph Theory)

Are tuples (V, E, W) where:

(V, E) is an (un)directed graph

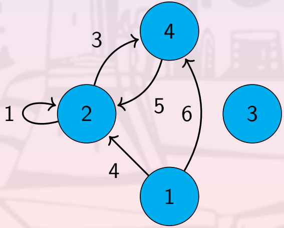
W is a function mapping edges into weights

The length of a path is the sum of all its edge labels



Representing Graphs

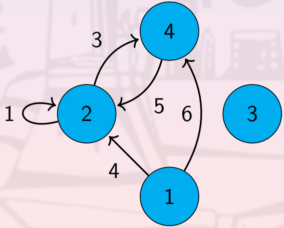
Two main ways:

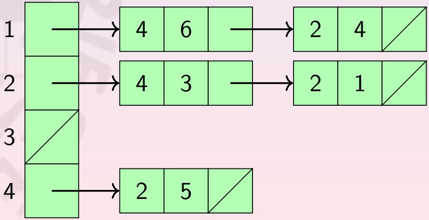


Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)

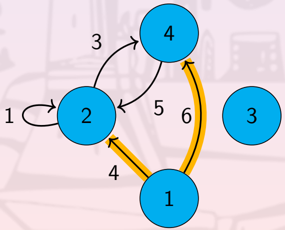


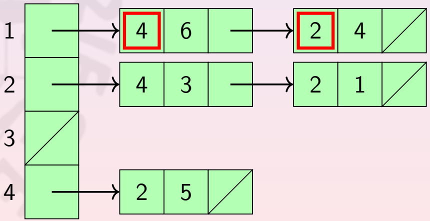


Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

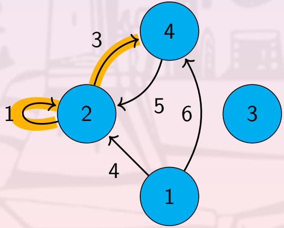


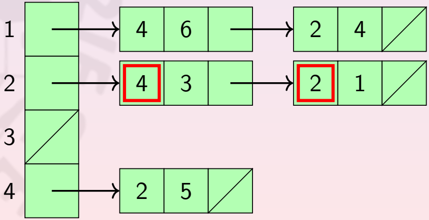


Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

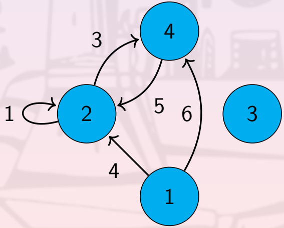


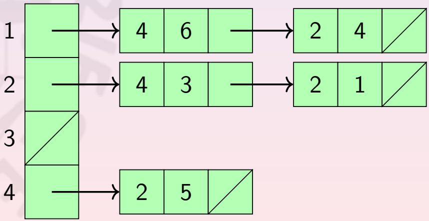


Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

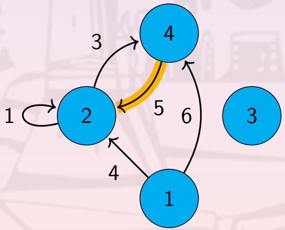


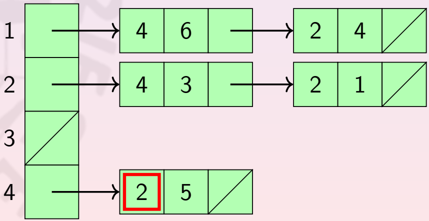


Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

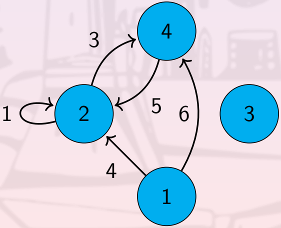




Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

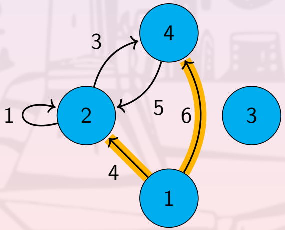


	1	2	3	4
1	N	4	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

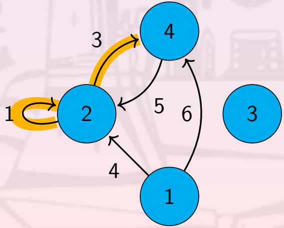


	1	2	3	4
1	N	4	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)

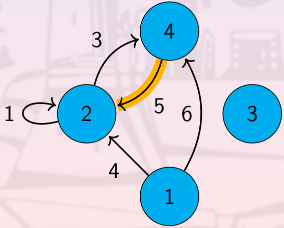


	1	2	3	4
1	N	4	N	6
2	N	$\boxed{1}$	N	$\boxed{3}$
3	N	N	N	N
4	N	5	N	N

Representing Graphs

Two main ways:

- adjancecy lists (usually, for sparse graphs)
- adjancecy matrix (usually, for dense graphs)



	1	2	3	4
1	N	4	N	6
2	N	1	N	3
3	N	N	N	N
4	N	$\boxed{5}$	N	N

Single Source Shortest Paths

Problem Definition and A Possible Strategy

We want to compute all the shortest paths from a single node s

How to solve this problem?

Problem Definition and A Possible Strategy

We want to compute all the shortest paths from a single node s

How to solve this problem? Any similar problem has been solved?

Problem Definition and A Possible Strategy

We want to compute all the shortest paths from a single node s

How to solve this problem? Any similar problem has been solved?

Computing the shortest paths from s in a non-weighted graph

Solved by using BFS in time $O(|V| + |E|)$

Problem Definition and A Possible Strategy

We want to compute all the shortest paths from a single node s

How to solve this problem? Any similar problem has been solved?

Computing the shortest paths from s in a non-weighted graph

Solved by using BFS in time $O(|V| + |E|)$

Let us have a look to BFS and try to adapt to SSSP

BFS Main Ingredients

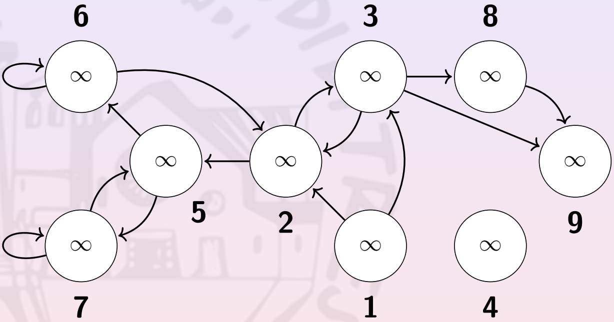
Nodes are WHITE, GRAY, or BLACK colored

- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors have not
- BLACK nodes have been discovered and all their neighbors have been discovered too

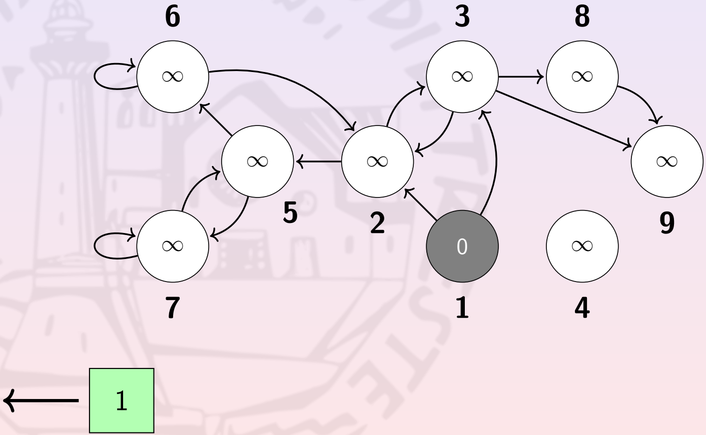
The search exclusively evolves from GRAY nodes

Their processing order is handled by a FIFO queue

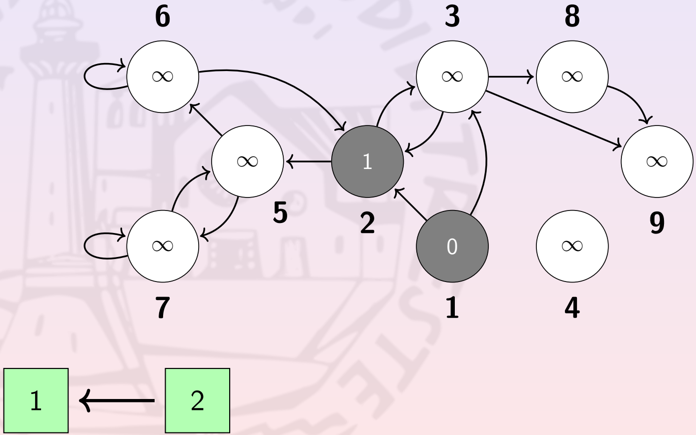
BFS: A Working Example



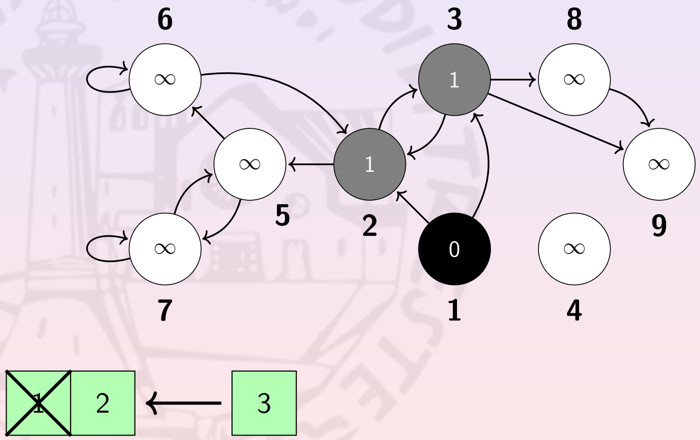
BFS: A Working Example



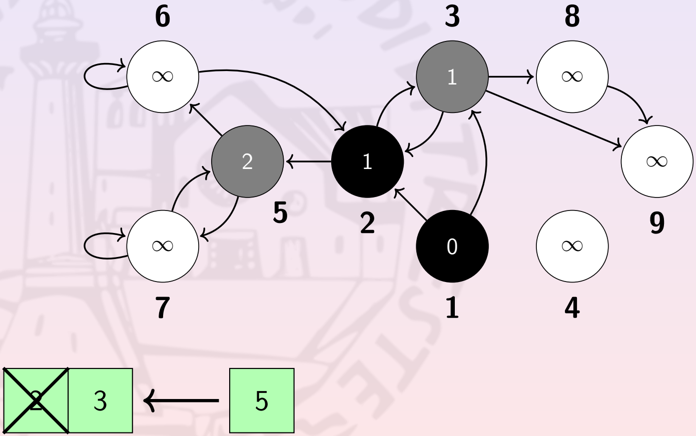
BFS: A Working Example



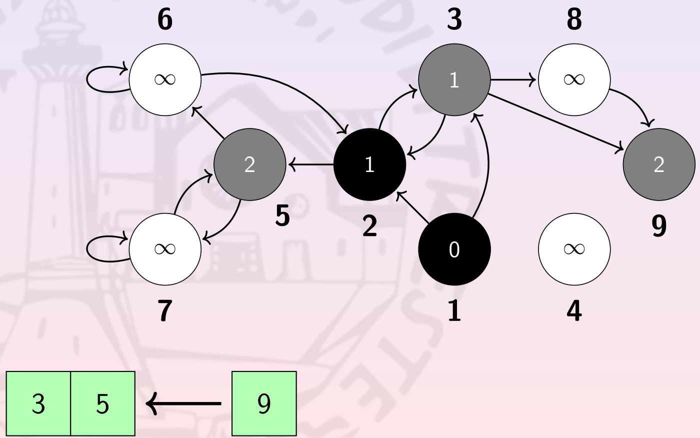
BFS: A Working Example



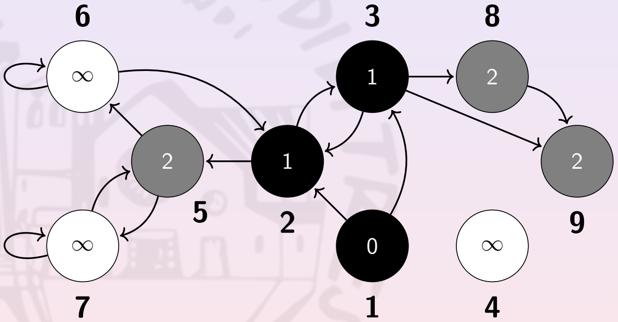
BFS: A Working Example

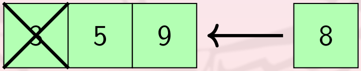


BFS: A Working Example

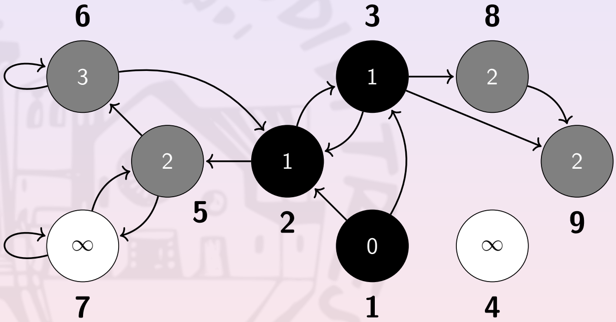


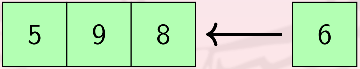
BFS: A Working Example



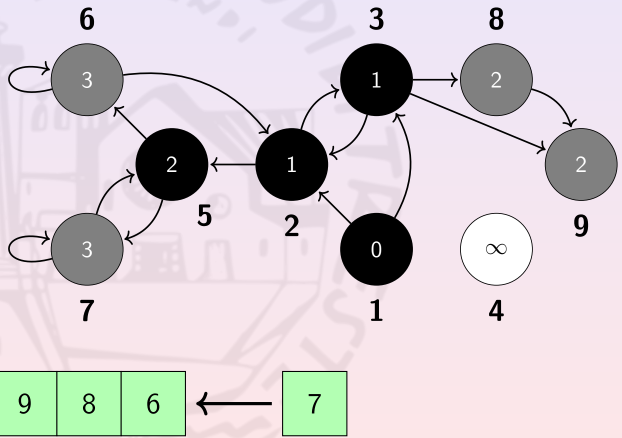


BFS: A Working Example

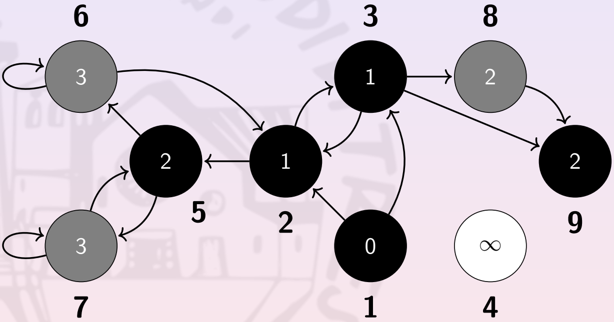




BFS: A Working Example

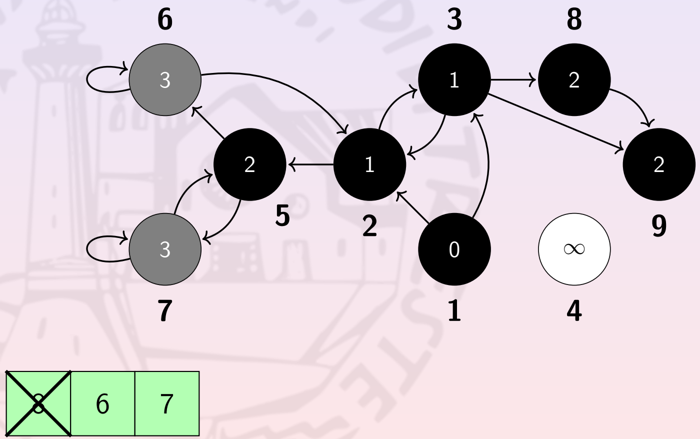


BFS: A Working Example

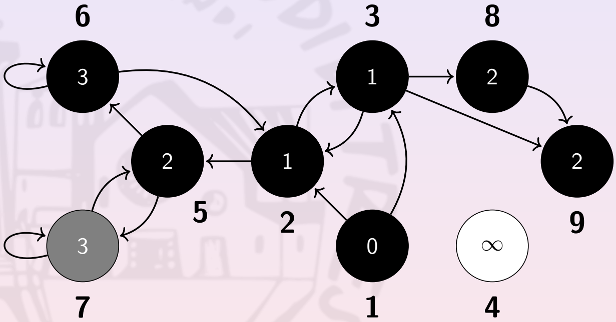


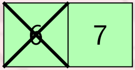
~~9~~	8	6	7

BFS: A Working Example

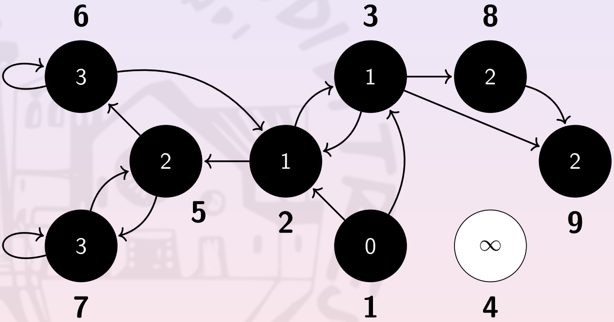


BFS: A Working Example





BFS: A Working Example



BFS: Pseudo-Code

```
def BFS_SET(v, color, d, pred):
    v.color ← color
    v.d ← d
    v.pred ← pred


def BFS_INIT(G,s):
    for v in G.V:
        BFS_SET(v, WHITE, ∞, NIL)
    endfor
    BFS_SET(s, GRAY, 0, s)

    return BUILD_QUEUE([s])
```

BFS: Pseudo-Code (Cont'd)

```
def BFS(G, s):
  Q ← BFS_INIT(G, s)
  while Q ≠ ∅:
    u ← DEQUEUE(Q)

    for v in G.Adj[u]:
      if v.color = WHITE:
        BFS_SET(v, GRAY, u.d+1, u)
        ENQUEUE(Q, v)
      endif
    endfor
    u.color ← BLACK
  endwhile
enddef
```

Upgrading BFS to Deal With Weights

BFS sets v's distance to $u.d + 1$ where u is queue head

Is it possible to upgrade this assignment to deal with weights?

Upgrading BFS to Deal With Weights

BFS sets v's distance to $u.d + 1$ where u is queue head

Is it possible to upgrade this assignment to deal with weights?

What if v's distance is set to $u.d + W[(u, v)]$?

Upgrading BFS to Deal With Weights

BFS sets v's distance to $u.d + 1$ where u is queue head

Is it possible to upgrade this assignment to deal with weights?

What if v's distance is set to $u.d + W[(u, v)]$?

It does NOT work!

Why Does BFS Work Properly?

BFS correctly sets v's distance because...

Lemma

Let $Q = [u_1, \ldots, u_n]$ be the queue during BFS. Then $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$ and $u_n.d \leq u_1.d + 1$.

So, $u_1.d + 1 \leq u_i.d + 1$ for all $i \in [2, n]$

Why Does BFS Work Properly?

BFS correctly sets v's distance because...

Lemma

Let $Q = [u_1, \ldots, u_n]$ be the queue during BFS. Then $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$ and $u_n.d \leq u_1.d + 1$.

So, $u_1.d + 1 \leq u_i.d + 1$ for all $i \in [2, n]$

If v is a successor of u_1, any other path reaching v through a node in Q is longer than $u_1.d + 1$

Why Does BFS Not Work on Weighted Graphs?

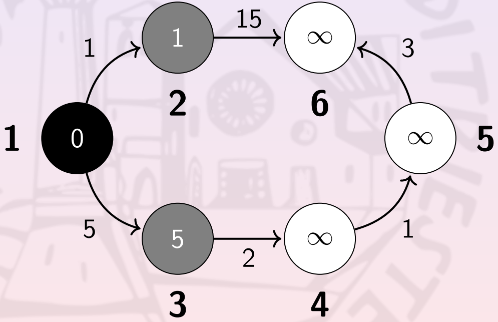
Even if $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$, there may be $(u_k, \bar{v})$ s.t.

$$u_1.d + W[(u_1, v)] > u_k.d + W[(u_k, \bar{v})]$$

Why Does BFS Not Work on Weighted Graphs?

Even if $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$, there may be $(u_k, \bar{v})$ s.t.

$$u_1.d + W[(u_1, v)] > u_k.d + W[(u_k, \bar{v})]$$

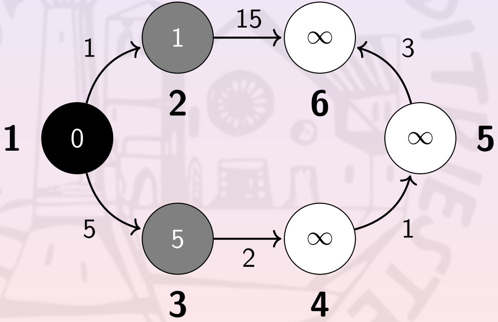


2	3

Why Does BFS Not Work on Weighted Graphs?

Even if $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$, there may be $(u_k, \bar{v})$ s.t.

$$u_1.d + W[(u_1, v)] > u_k.d + W[(u_k, \bar{v})]$$



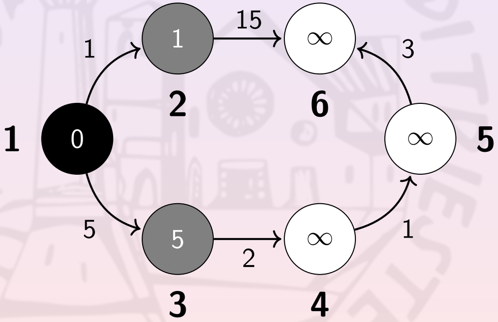
2	3

$2.d + W[(2, 6)]$ $3.d + W[(3, 4)]$

Why Does BFS Not Work on Weighted Graphs?

Even if $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$, there may be $(u_k, \bar{v})$ s.t.

$$u_1.d + W[(u_1, v)] > u_k.d + W[(u_k, \bar{v})]$$



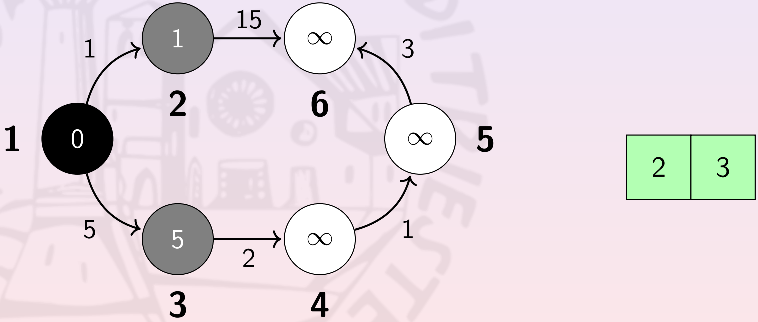
2	3

$$2.d + W[(2,6)] = 1 + 15 \qquad 5 + 2 = 3.d + W[(3,4)]$$

Why Does BFS Not Work on Weighted Graphs?

Even if $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$, there may be $(u_k, \bar{v})$ s.t.

$$u_1.d + W[(u_1, v)] > u_k.d + W[(u_k, \bar{v})]$$



$$2.d + W[(2, 6)] = 1 + 15 > 5 + 2 = 3.d + W[(3, 4)]$$

Any Idea to Solve This Problem?

Any Idea to Solve This Problem?

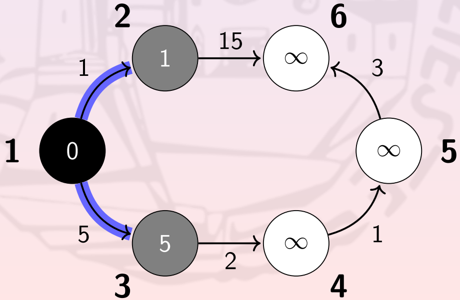
Enqueuing not-discovered nodes in place of the just discovered

These nodes are “pre-labeled” with a candidate distance

Any Idea to Solve This Problem?

Enqueuing not-discovered nodes in place of the just discovered

These nodes are “pre-labeled” with a candidate distance

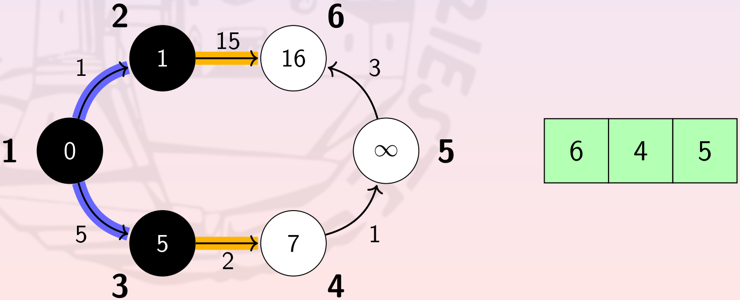


2	3

Any Idea to Solve This Problem?

Enqueuing not-discovered nodes in place of the just discovered

These nodes are "pre-labeled" with a candidate distance

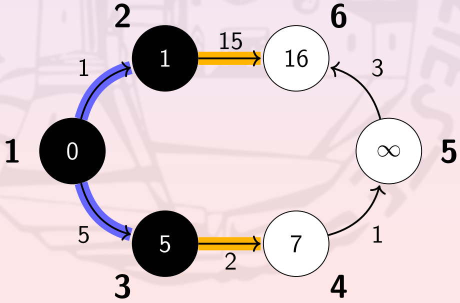


Any Idea to Solve This Problem? Dijkstra's Algorithm

Enqueuing not-discovered nodes in place of the just discovered

These nodes are "pre-labeled" with a candidate distance

At each step a node having minimal candidate distance is extracted and "finalized". Its outgoing neighbors are updated



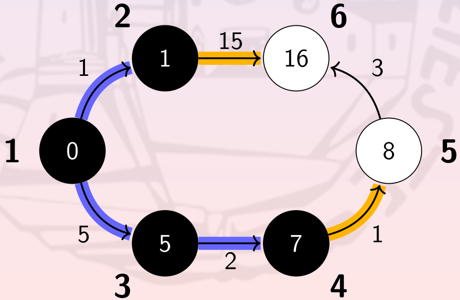
6	4	5

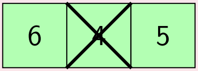
Any Idea to Solve This Problem? Dijkstra's Algorithm

Enqueuing not-discovered nodes in place of the just discovered

These nodes are "pre-labeled" with a candidate distance

At each step a node having minimal candidate distance is extracted and "finalized". Its outgoing neighbors are updated



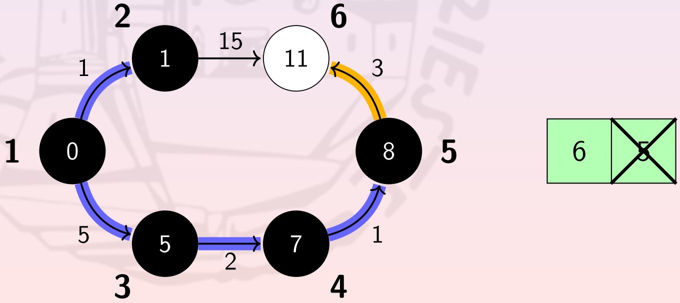


Any Idea to Solve This Problem? Dijkstra's Algorithm

Enqueuing not-discovered nodes in place of the just discovered

These nodes are "pre-labeled" with a candidate distance

At each step a node having minimal candidate distance is extracted and "finalized". Its outgoing neighbors are updated

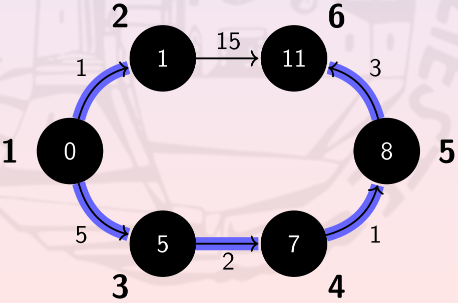


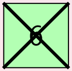
Any Idea to Solve This Problem? Dijkstra's Algorithm

Enqueuing not-discovered nodes in place of the just discovered

These nodes are "pre-labeled" with a candidate distance

At each step a node having minimal candidate distance is extracted and "finalized". Its outgoing neighbors are updated





Some Observations on Dijkstra's Algorithm

BFS queue has became a priority queue w.r.t. candidate distance

Some Observations on Dijkstra's Algorithm

BFS queue has became a priority queue w.r.t. candidate distance

It does not need coloring:

- BFS WHITE nodes correspond to nodes in Q
- nodes are finalized as soon as extracted from Q

Some Observations on Dijkstra's Algorithm

BFS queue has became a priority queue w.r.t. candidate distance

It does not need coloring:

- BFS WHITE nodes correspond to nodes in Q
- nodes are finalized as soon as extracted from Q

Paths are treated as distances: the predecessor of a node is updated every time a new possible minimum distance arises

Some Observations on Dijkstra's Algorithm

BFS queue has became a priority queue w.r.t. candidate distance

It does not need coloring:

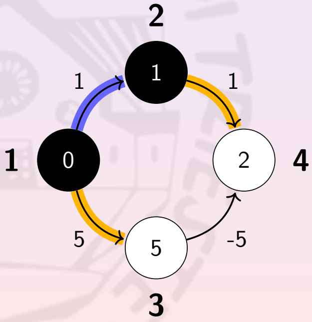
- BFS WHITE nodes correspond to nodes in Q
- nodes are finalized as soon as extracted from Q

Paths are treated as distances: the predecessor of a node is updated every time a new possible minimum distance arises

It is a kind of meta-algorithm: it does not specify how to handle the queue

Some Observations on Dijkstra's Algorithm (Cont'd)

No negative weights: if they were allowed, the minimal path to the node extracted from Q could be not discovered



Dijkstra's Algorithm: Pseudo-Code

```
def INIT_SSSP(G):
  for v in G.V:
    v.d ← ∞
    v.pred ← NIL
  endfor
enddef

def RELAX(Q, u, v, w):
  if u.d + w < v.d:
    UPDATE_DISTANCE(Q, v, u.d + w)
    v.pred ← u
  endif
enddef
```

Dijkstra's Algorithm: Pseudo-Code (Cont'd)

```
def DIJKSTRA(G, s):
  INIT_SSSP(G, s)
  s.d ← 0

  Q ← BUILD_QUEUE(G.V)
  while not IS_EMPTY(Q):
    u ← EXCTRACT_MIN(Q)

    for (v, w) in G.Adj[u]:
      RELAX(Q, u, v, w)
    endfor
  endwhile
enddef
```

Dijkstra's Algorithm: Complexity

All the nodes are in Q at the beginning of the computation

Dijkstra's Algorithm: Complexity

All the nodes are in Q at the beginning of the computation

One node u is extracted at each while-loop iteration

The for-loop iterates on the adjacency list of u

Dijkstra's Algorithm: Complexity

All the nodes are in Q at the beginning of the computation

One node u is extracted at each while-loop iteration

The for-loop iterates on the adjacency list of u

Globally, the for-loop performs $|E|$ iterations

Dijkstra's Algorithm: Complexity

All the nodes are in Q at the beginning of the computation

One node u is extracted at each while-loop iteration

The for-loop iterates on the adjacency list of u

Globally, the for-loop performs $|E|$ iterations

The overall complexity of Dijkstra's algorithm is

$$T_D(G) = \Theta(|V|) + T_B(|V|) + |V| * T_E(|V|) + |E| * T_U(|V|)$$

where T_B, T_E, and T_U are the complexities of `BUILD_QUEUE`, `EXCTRACT_MIN`, `UPDATE_DISTANCE`

Dijkstra's Algorithm: Complexity (Cont'd)

Queue Data Structure	$\mathbf{T_B(n)}$	$\mathbf{T_E(n)}$	$\mathbf{T_U(n)}$	$\mathbf{T_D(G)}$
Arrays	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\|V\|^2 + \|E\|)$

Dijkstra's Algorithm: Complexity (Cont'd)

Queue Data Structure	$T_B(n)$	$T_E(n)$	$T_U(n)$	$T_D(G)$
Arrays	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\lvert V\rvert^2 + \lvert E\rvert)$
Binary Heaps	$\Theta(n)$	$O(\log n)$	$O(\log n)$	$O((\lvert V\rvert + \lvert E\rvert) * \log \lvert V\rvert)$

Dijkstra's Algorithm: Complexity (Cont'd)

Queue Data Structure	$T_B(n)$	$T_E(n)$	$T_U(n)$	$T_D(G)$
Arrays	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\lvert V\rvert^2 + \lvert E\rvert)$
Binary Heaps	$\Theta(n)$	$O(\log n)$	$O(\log n)$	$O((\lvert V\rvert + \lvert E\rvert) * \log \lvert V\rvert)$
Fibonacci Heaps[1]	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	$O(\lvert E\rvert + \lvert V\rvert * \log \lvert V\rvert)$

[1] Amortized time

All Pairs Shortest Paths

Problem Definition and Possible Strategies

We want to compute the shortest paths between all the pairs of nodes

How to solve this problem?

Problem Definition and Possible Strategies

We want to compute the shortest paths between all the pairs of nodes

How to solve this problem?

Run Dijkstra's algorithm and use each node as source

Problem Definition and Possible Strategies

We want to compute the shortest paths between all the pairs of nodes

How to solve this problem?

Run Dijkstra's algorithm and use each node as source

No negative edges

Some Revealing Insights

Let us consider graphs whose nodes are natural numbers

Let $p = e_1, \ldots, e_h$ be the shortest path from i to j

Let k be the "greatest" internal node in the path

Some Revealing Insights

Let us consider graphs whose nodes are natural numbers

Let $p = e_1, \ldots, e_h$ be the shortest path from i to j

Let k be the "greatest" internal node in the path

There exists $\bar{h}$ s.t. $e_{\bar{h}-1}$ and $e_{\bar{h}}$ have k as destination and source

Some Revealing Insights

Let us consider graphs whose nodes are natural numbers

Let $p = e_1, \ldots, e_h$ be the shortest path from i to j

Let k be the “greatest” internal node in the path

There exists $\bar{h}$ s.t. $e_{\bar{h}-1}$ and $e_{\bar{h}}$ have k as destination and source

So, $e_1, \ldots, e_{\bar{h}-1}$ and $e_{\bar{h}}, \ldots, e_h$ are shortest paths between i and k and between k and j

Some Revealing Insights

Let us consider graphs whose nodes are natural numbers

Let $p = e_1, \ldots, e_h$ be the shortest path from i to j

Let k be the "greatest" internal node in the path

There exists $\bar{h}$ s.t. $e_{\bar{h}-1}$ and $e_{\bar{h}}$ have k as destination and source

So, $e_1, \ldots, e_{\bar{h}-1}$ and $e_{\bar{h}}, \ldots, e_h$ are shortest paths between i and k and between k and j

Their interal nodes are "smaller" than k

Floyd-Warshall's Algorithm: Intuition

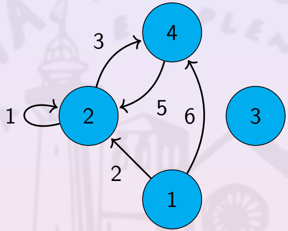
We can incrementally build shortest paths by admitting new untouched internal nodes at each step

If $D^{(k-1)}$ contains the lengths of the shortest paths whose internal nodes are smaller than k, we can compute $D^{(k)}$ as

$$D^{(k)}[i,j] = \min(D^{(k-1)}[i,k] + D^{(k-1)}[k,j], D^{(k-1)}[i,j])$$

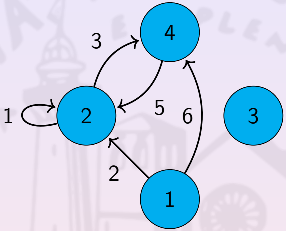
The same criterium applies to $\Pi^{(k)}$ where $\Pi^{(k)}[i,j]$ is the predecessor of j in the smallest path between i and j

Floyd-Warshall's Algorithm: A Working Example



	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

Floyd-Warshall's Algorithm: A Working Example

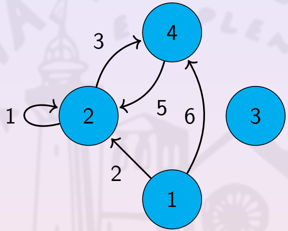


$$D^{(0)} = \begin{pmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & \infty & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 \end{pmatrix}$$

	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

$$\Pi^{(0)} = \begin{pmatrix} \mathrm{NIL} & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \end{pmatrix}$$

Floyd-Warshall's Algorithm: A Working Example

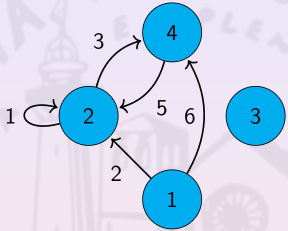


$$D^{(0)} = \begin{pmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & \infty & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 \end{pmatrix}$$

	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

$$\Pi^{(0)} = \begin{pmatrix} \mathrm{NIL} & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \end{pmatrix}$$

Floyd-Warshall's Algorithm: A Working Example

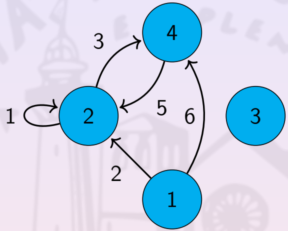


$$D^{(1)} = \begin{pmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & \infty & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 \end{pmatrix}$$

	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

$$\Pi^{(1)} = \begin{pmatrix} \mathrm{NIL} & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \end{pmatrix}$$

Floyd-Warshall's Algorithm: A Working Example

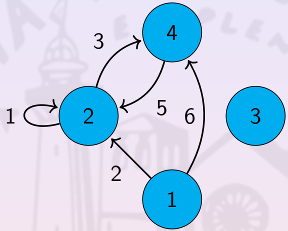


$$D^{(2)} = \begin{pmatrix} 0 & 2 & \infty & \color{red}{5} \\ \infty & 0 & \infty & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 \end{pmatrix}$$

	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

$$\Pi^{(2)} = \begin{pmatrix} \mathrm{NIL} & 1 & \mathrm{NIL} & \color{red}{2} \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \end{pmatrix}$$

Floyd-Warshall's Algorithm: A Working Example

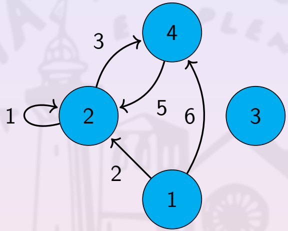


$$D^{(3)} = \begin{pmatrix} 0 & 2 & \infty & 5 \\ \infty & 0 & \infty & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 \end{pmatrix}$$

	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

$$\Pi^{(3)} = \begin{pmatrix} \mathrm{NIL} & 1 & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \end{pmatrix}$$

Floyd-Warshall's Algorithm: A Working Example



$$D^{(4)} = \begin{pmatrix} 0 & 2 & \infty & 5 \\ \infty & 0 & \infty & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & \infty & 0 \end{pmatrix}$$

	1	2	3	4
1	N	2	N	6
2	N	1	N	3
3	N	N	N	N
4	N	5	N	N

$$\Pi^{(4)} = \begin{pmatrix} \mathrm{NIL} & 1 & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \end{pmatrix}$$

Floyd-Warshall's Algorithm: Pseudo-Code

```
def FLOYD_WARSHALL_STEP(old_D, old_P):
  D ← COPY_MATRIX(old_D)
  P ← COPY_MATRIX(old_P)

  for i ← 1 upto |G.V|:
    for j ← 1 upto |G.V|:
      if old_D[i][j] > old_D[i][k] + old_D[k][j]:
        D[i][j] ← old_D[i][k] + old_D[k][j]
        P[i][j] ← old_P[k][j]
      endif
    endfor
  endfor

  return (D, P)
enddef
```

Floyd-Warshall's Algorithm: Pseudo-Code

```
def FLOYD_WARSHALL(G):
  D[0] ← INIT_MATRIX_D0(G.W)
  P[0] ← INIT_MATRIX_P0(G.W)

  for k ← 1 upto |G.V|:
    D[k], P[k] ← FLOYD_WARSHALL_STEP(D[k−1],P[k−1])
  endfor

  return (D[|G.V|], P[|G.V|])
enddef
```

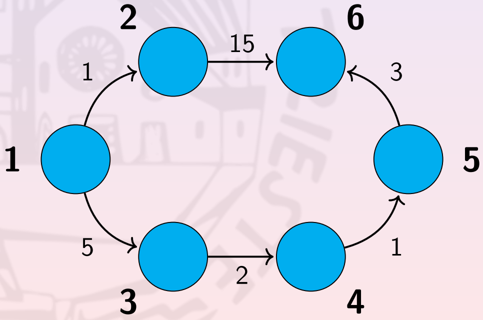
Routing

The Routing Problem

Given a weighted graph G,

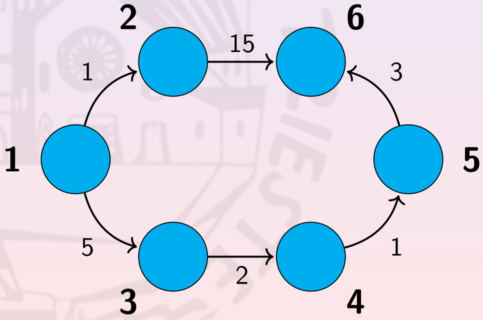
The Routing Problem

Given a weighted graph G,



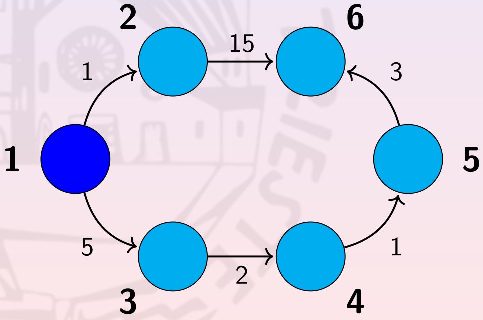
The Routing Problem

Given a weighted graph G, source s,



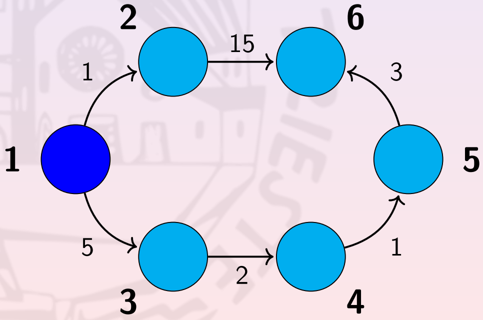
The Routing Problem

Given a weighted graph G, source s,



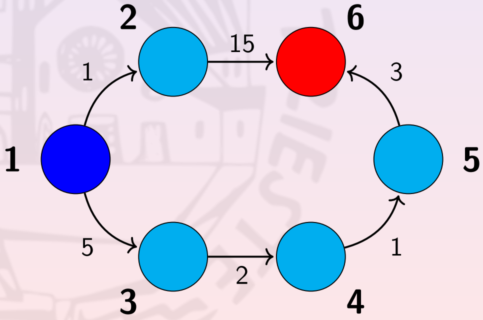
The Routing Problem

Given a weighted graph G, source s, and a destination d...



The Routing Problem

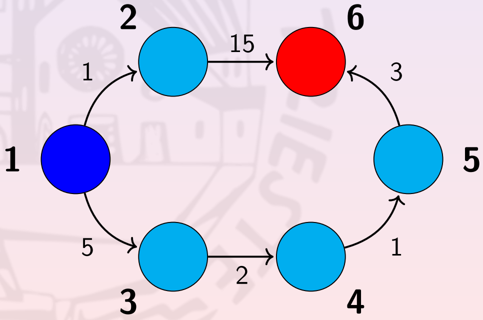
Given a weighted graph G, source s, and a destination d...



The Routing Problem

Given a weighted graph G, source s, and a destination d...

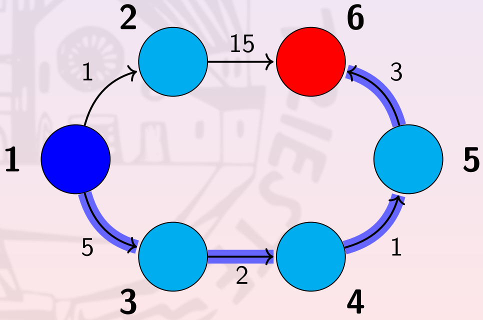
we aim for the shortest path in G from s to d



The Routing Problem

Given a weighted graph G, source s, and a destination d...

we aim for the shortest path in G from s to d



Routing By Using Dijkstra

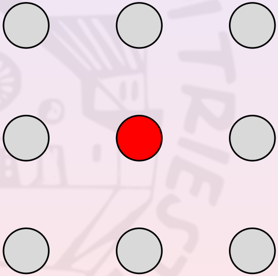
The routing problem is similar to SSSP

Let us try to use a “light” version of Dijkstra’s algorithm

The algorithm ends as soon as d has been finalized

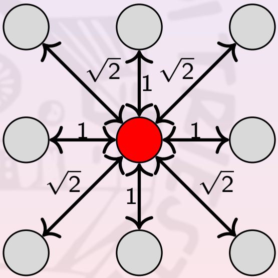
Routing By Using Dijkstra: What Is Going On?

Let us consider a strongly structured graph. . .



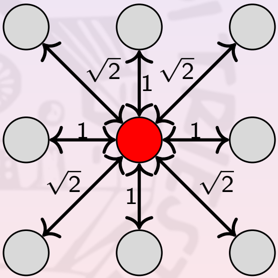
Routing By Using Dijkstra: What Is Going On?

Let us consider a strongly structured graph. . .



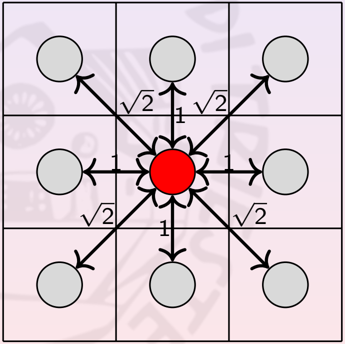
Routing By Using Dijkstra: What Is Going On?

Let us consider a strongly structured graph. . .



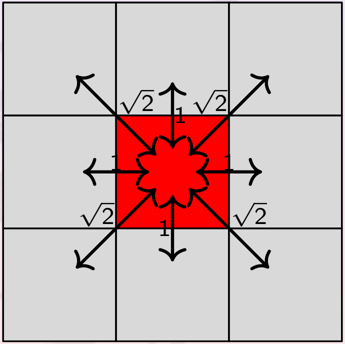
Routing By Using Dijkstra: What Is Going On?

Let us consider a strongly structured graph. . .

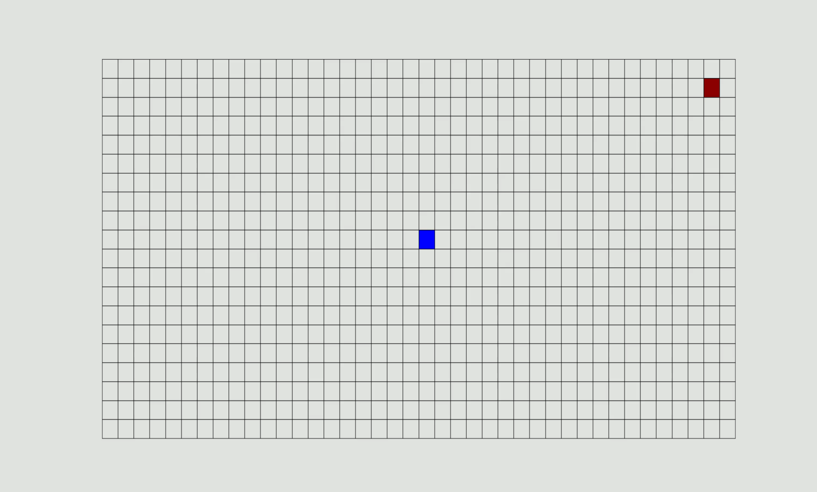


Routing By Using Dijkstra: What Is Going On?

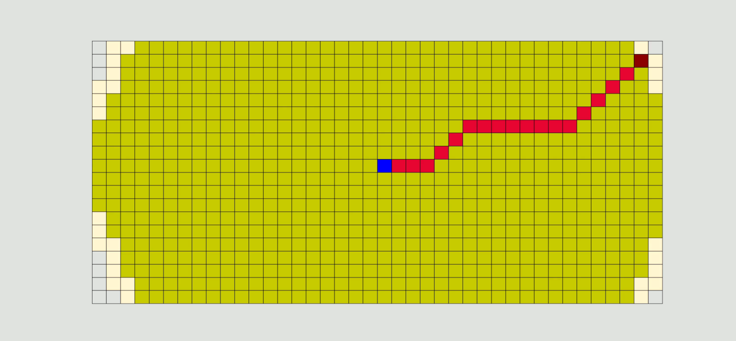
Let us consider a strongly structured graph...



Routing By Using Dijkstra: A “Large” Example



Routing By Using Dijkstra: A "Large" Example



- $\|s - d\|_2 \approx 19.70$
- Queue extractions: 763
- Route length: ≈ 21.31

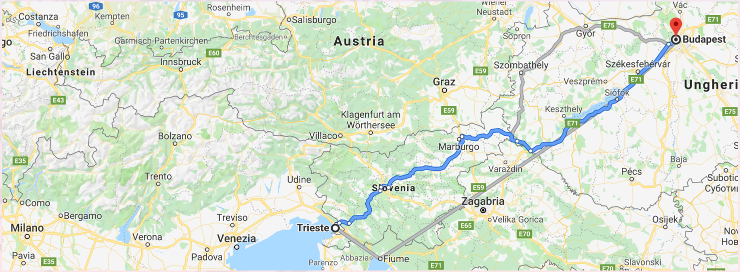
Anything Wrong With This Solution?

The discovered route is for sure the shortest one, but. . .

Anything Wrong With This Solution?

The discovered route is for sure the shortest one, but. . .

the shortest path Trieste-Budapest probably avoids Milano and . . .

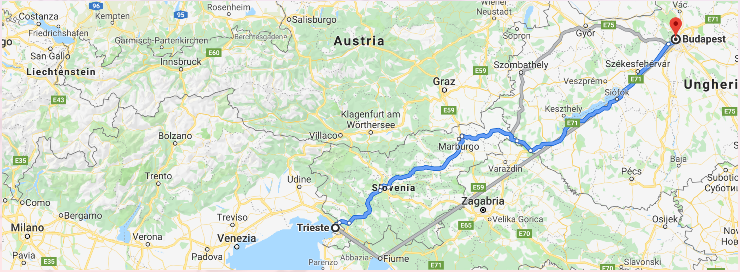


Anything Wrong With This Solution?

The discovered route is for sure the shortest one, but. . .

the shortest path Trieste-Budapest probably avoids Milano and . . .

we will never look in that direction for the solution!

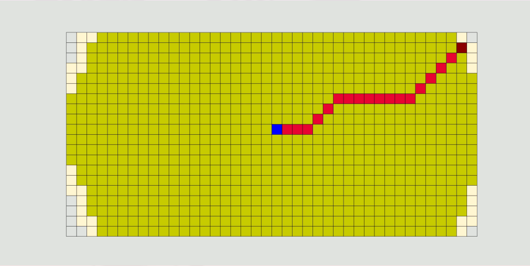


Anything Wrong With This Solution?

The discovered route is for sure the shortest one, but. . .

the shortest path Trieste-Budapest probably avoids Milano and . . .

we will never look in that direction for the solution!

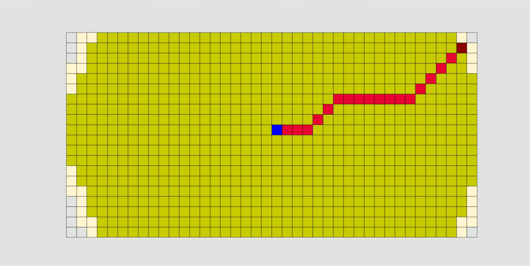


Anything Wrong With This Solution?

The discovered route is for sure the shortest one, but. . .

the shortest path Trieste-Budapest probably avoids Milano and . . .

we will never look in that direction for the solution! Why?



Heuristic Distance

We have in mind a distance h not embedded in the graph

Heuristic Distance

We have in mind a distance h not embedded in the graph

If the shortest path length from s to u is $u.d$, then

$$u.d + W(u, v) + h(v, d)$$

estimates the length of the path between s and d

Heuristic Distance

We have in mind a distance h not embedded in the graph

If the shortest path length from s to u is $u.d$, then

$$u.d + W(u, v) + h(v, d)$$

estimates the length of the path between s and d

h can be any distance e.g., Euclidean, Manhattan, etc.

Estimation accuracy depends on both h and G topology

A* Algorithm: Dijkstra + Heuristic Distance

The A* algorithm has the same structure of the Dijkstra's algorithm

A* Algorithm: Dijkstra + Heuristic Distance

The A* algorithm has the same structure of the Dijkstra's algorithm, but Q is sorted according

$$u.d + W(u, v) + h(v, d)$$

where $u.d + W(u, v)$ is the guessed shortest path length to v

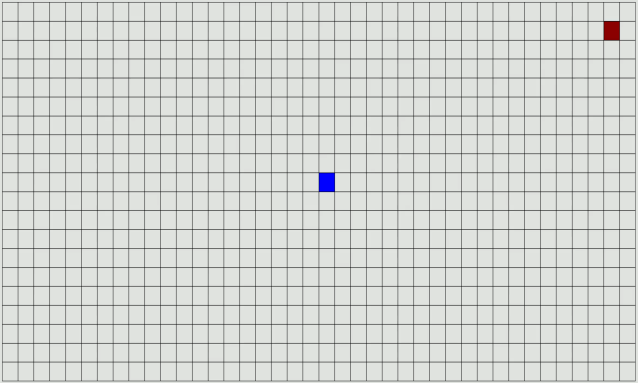
A* Algorithm: Pseudo-Code

```
def INIT_SSSP(G):
  for v in G.V:
    v.d ← ∞
    v.pred ← NIL
  endfor
enddef

def RELAX_ASTAR(Q, u, v, w, d, h):
  if u.d + w < v.d:
    UPDATE_DISTANCE(Q, v, u.d + w + h(v,d))
    v.pred ← u
  endif
enddef
```

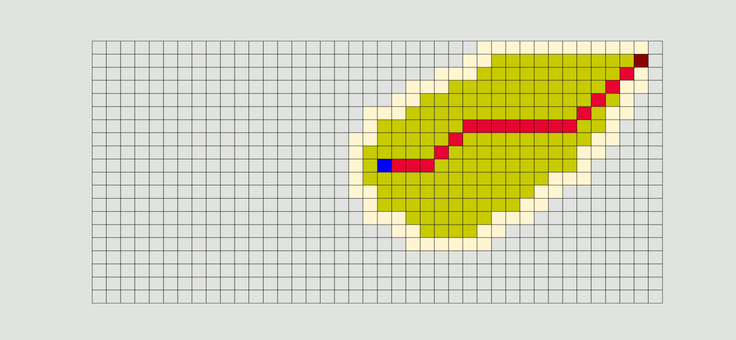
A* Algorithm: Pseudo-Code (Cont'd)

```
def ASTAR(G,s,d,h):
  INIT_SSSP(G,s)
  s.d ← h(s,d)

  Q ← BUILD_QUEUE(G.V)
  while not IS_EMPTY(Q):
    u ← EXCTRACT_MIN(Q)

    for (v, w) in G.Adj[u]:
      RELAX_ASTAR(Q,u,v,w,d,h)
    endfor
  endwhile
enddef
```

Using A* with Chebyshev Distance on Grid Example

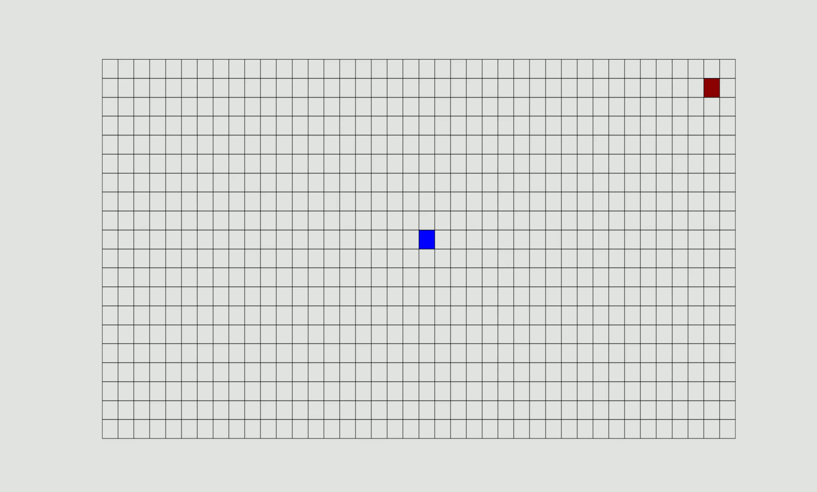


Using A* with Chebyshev Distance on Grid Example

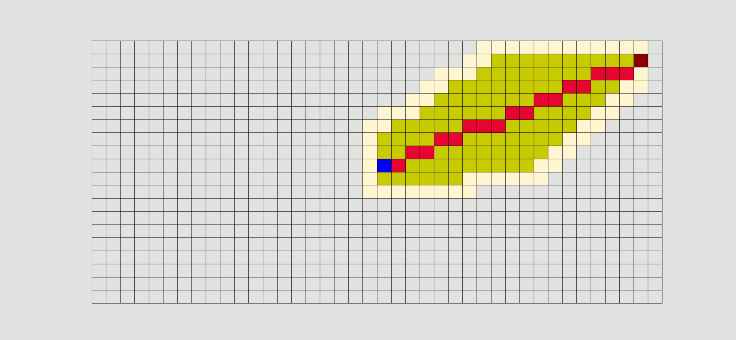


- Queue extractions: 167 (were 763 using Dijkstra's light algorithm)
- Shortest path length: ≈ 21.31
- Route length: ≈ 21.31

Using A* with Euclidean Distance on Grid Example

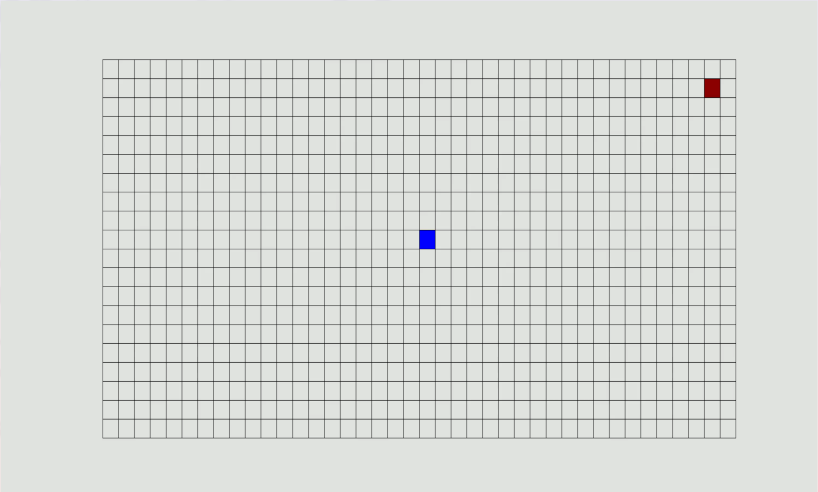


Using A* with Euclidean Distance on Grid Example

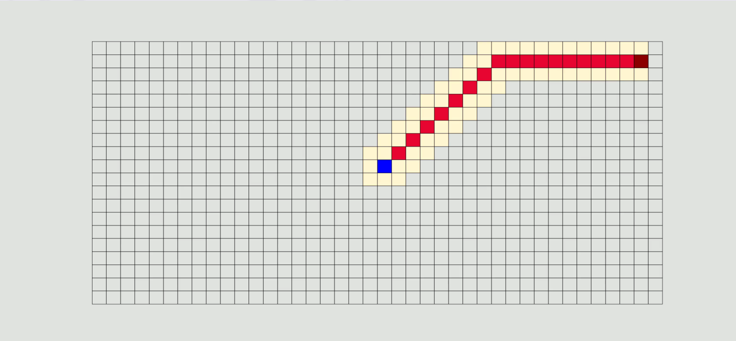


- Queue extractions: 113 (were 167 for A* with Chebyshev Distance)
- Shortest path length: ≈ 21.31
- Route length: ≈ 21.31

Using A* with Manhattan Distance on Grid Example

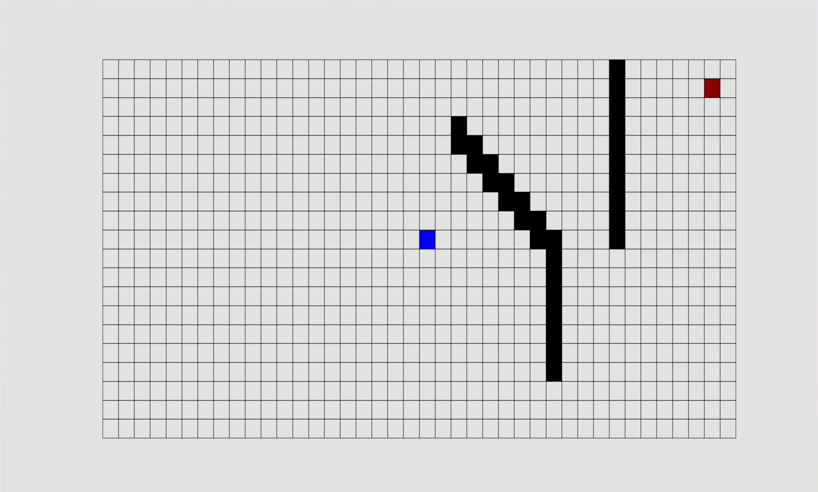


Using A* with Manhattan Distance on Grid Example

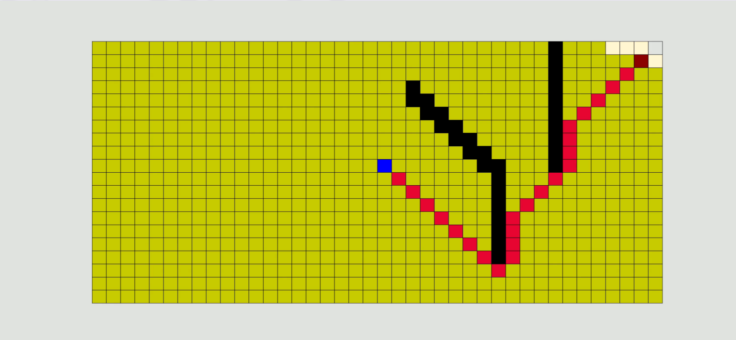


- Queue extractions: 19 (were 113 for A* with Euclidean Distance)
- Shortest path length: ≈ 21.31
- Route length: ≈ 21.31

Dijkstra's Algorithm on a Different Grid Example

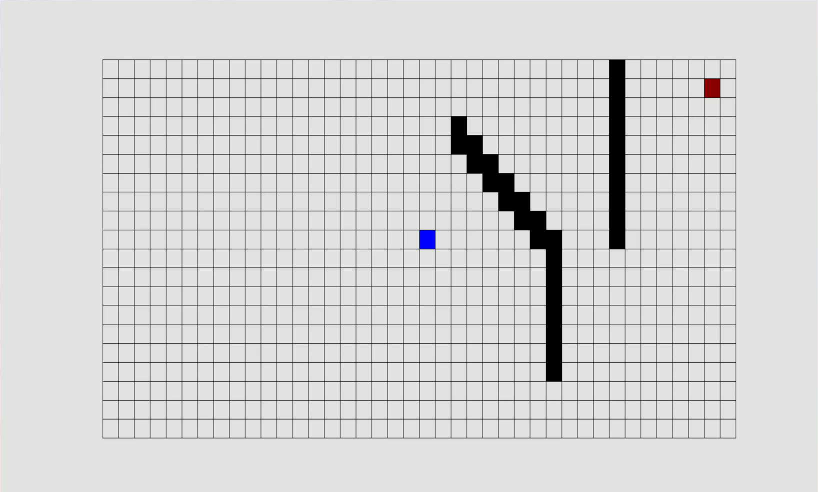


Dijkstra's Algorithm on a Different Grid Example

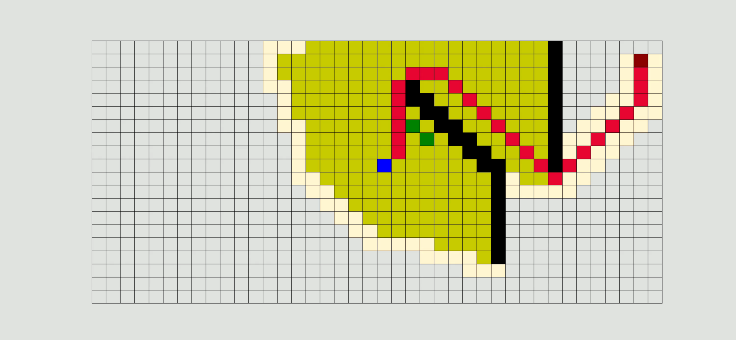


- Queue extractions: 765 (were 763 using Dijkstra's light algorithm on the other example)
- Route length: ≈ 31.46

Using Manhattan Distance on a Different Grid Example

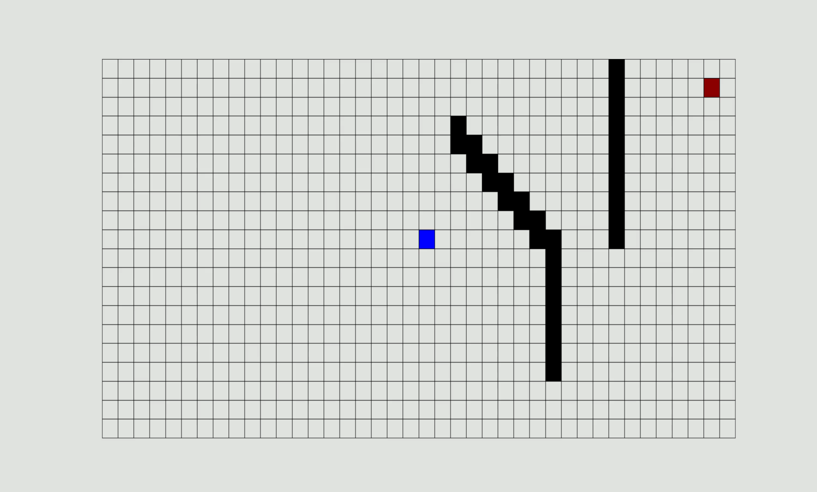


Using Manhattan Distance on a Different Grid Example

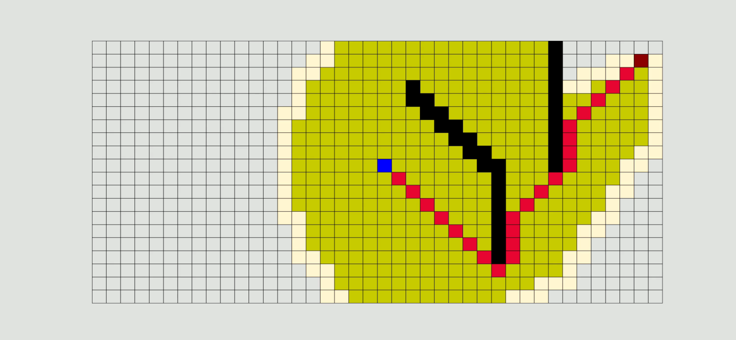


- Queue extractions: 231 (were 765 using Dijkstra's light algorithm)
- Shortest path length: ≈ 31.46
- Route length: ≈ 32.62

Using Chebyshev Distance on a Different Grid Example

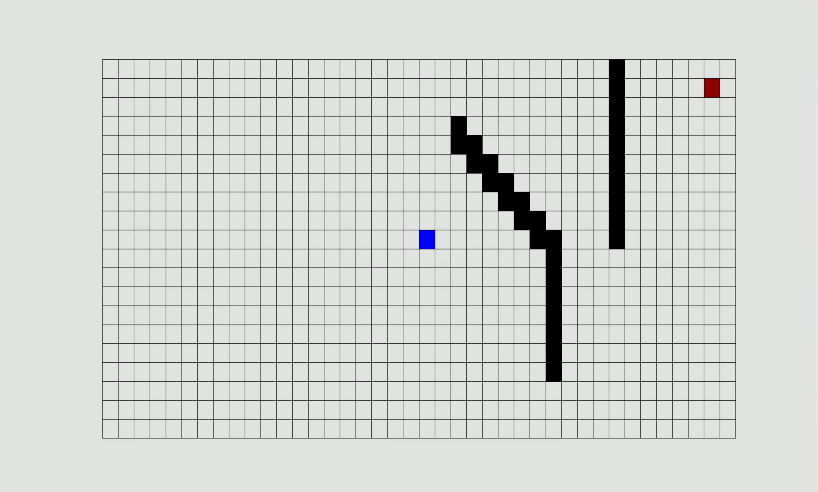


Using Chebyshev Distance on a Different Grid Example

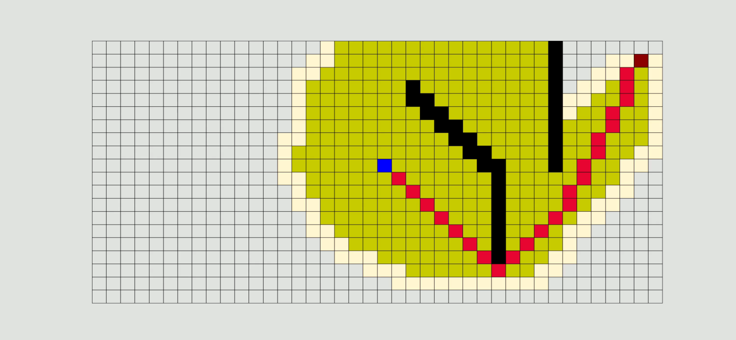


- Queue extractions: 372 (were 231 for A* with Manhattan Distance)
- Shortest path length: ≈ 31.46
- Route length: ≈ 31.46

Using Euclidean Distance on a Different Grid Example



Using Euclidean Distance on a Different Grid Example



- Queue extractions: 315 (were 372 for A* with Chebyshev Distance)
- Shortest path length: ≈ 31.46
- Route length: ≈ 31.46

Routing on World Scale Graphs

We aim to apply routing techniques also to handle

- internet packets moving between severs
- parcels delivered by multiple couriers
- travelers commuting between airplanes
- cars moving along a continent-wide route system

Routing on World Scale Graphs

We aim to apply routing techniques also to handle

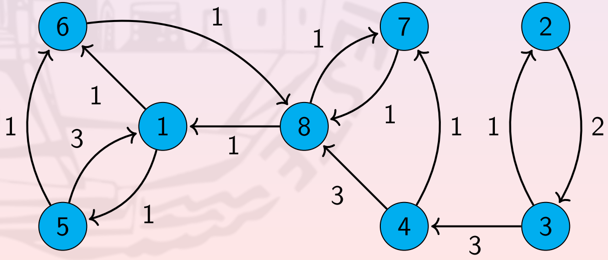
- internet packets moving between severs
- parcels delivered by multiple couriers
- travelers commuting between airplanes
- cars moving along a continent-wide route system

The graphs are too huge to be completely store in memory

neither Dijkstra nor A* can be applied

Shrinking Graphs Preserving Important Nodes

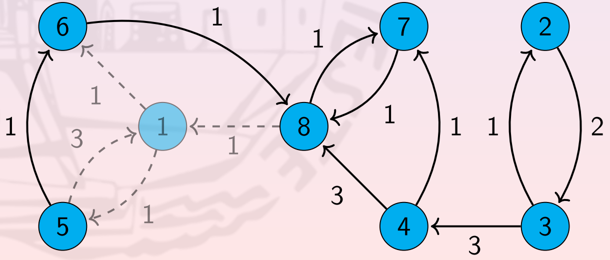
Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"



Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes



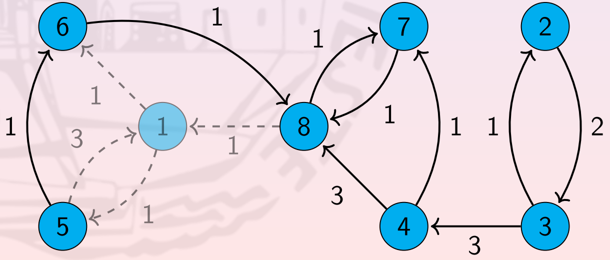
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



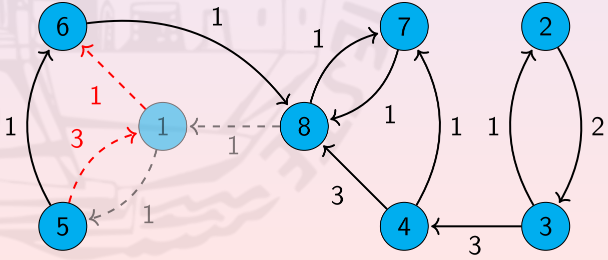
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \dots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



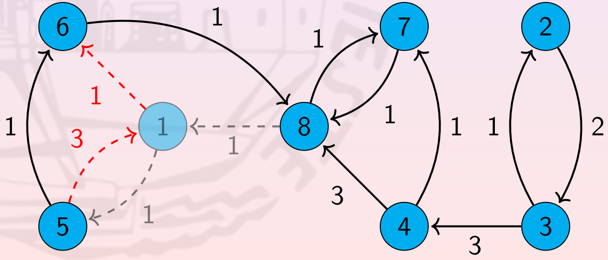
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



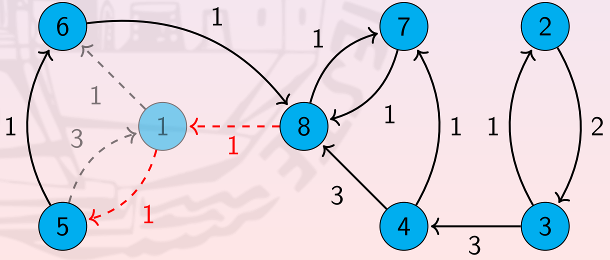
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



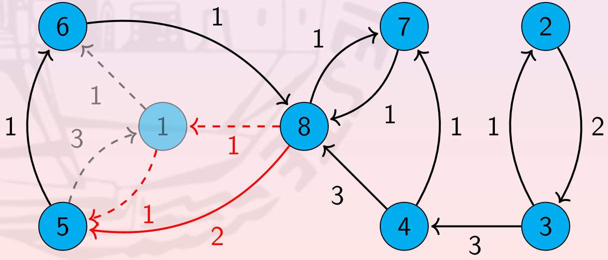
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



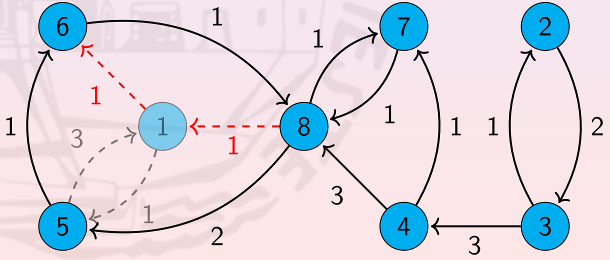
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \dots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



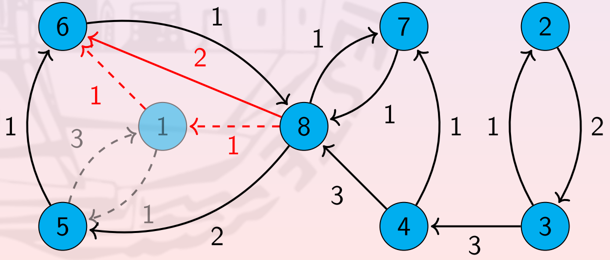
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



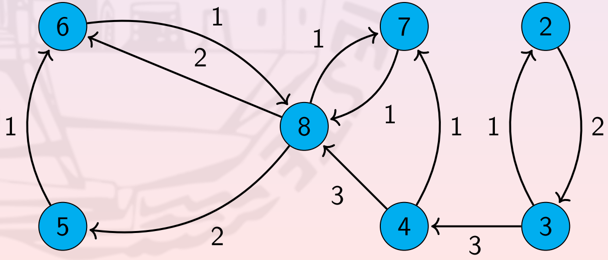
Shrinking Graphs Preserving Important Nodes

Let $V = \{1, \ldots, n\}$ be sorted by ascending "importance"

We can remove 1 preserving more important nodes

The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts (i, j) s.t.

$$W(i, j) = W(i, 1) + W(1, j)$$



Contractions and Overlay Graphs

The contraction of node k consists in:

- adding the needed shortcuts
- removing k

Contractions and Overlay Graphs

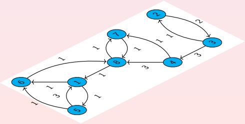
The contraction of node k consists in:

- adding the needed shortcuts
- removing k

The resulting graph is the overlay graph

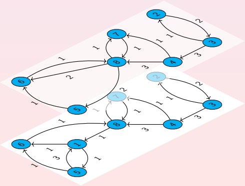
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



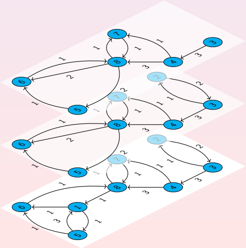
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



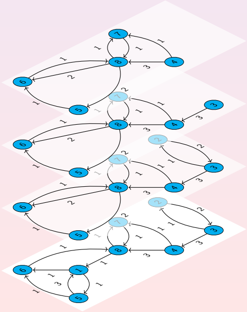
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



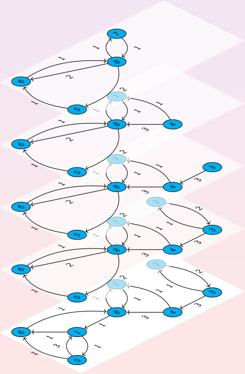
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



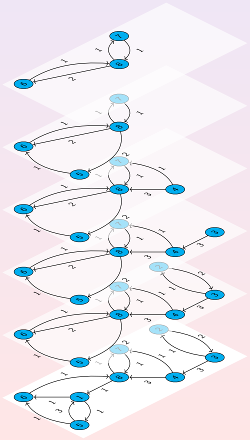
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



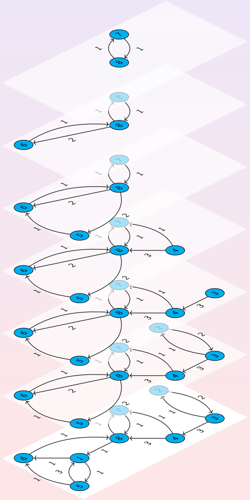
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



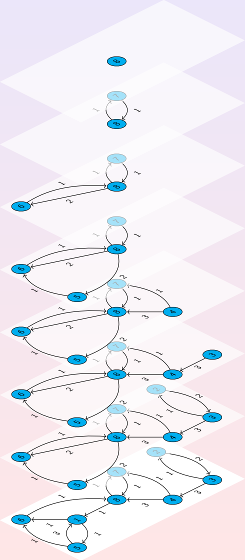
Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy



Contraction Hierarchy

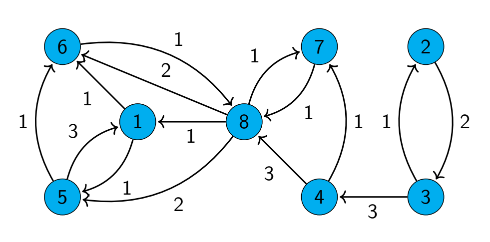
The sequence of the overlay graphs is a contraction hierarchy

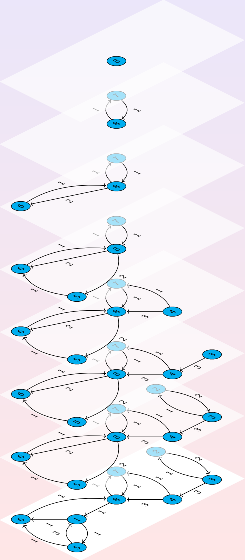


Contraction Hierarchy

The sequence of the overlay graphs is a contraction hierarchy

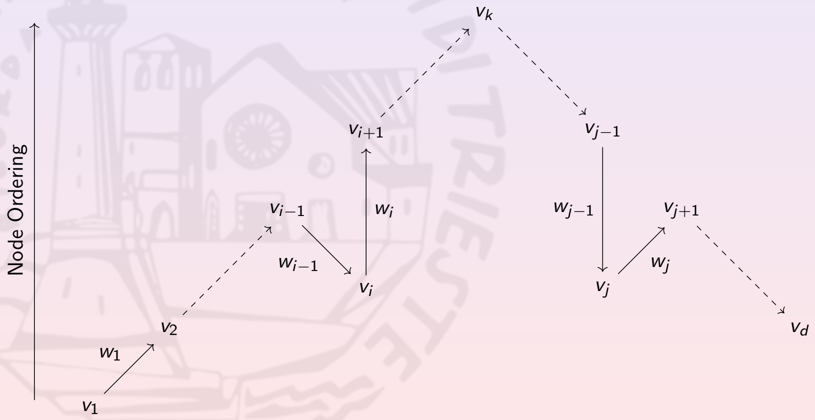
Looking it from the top, we get





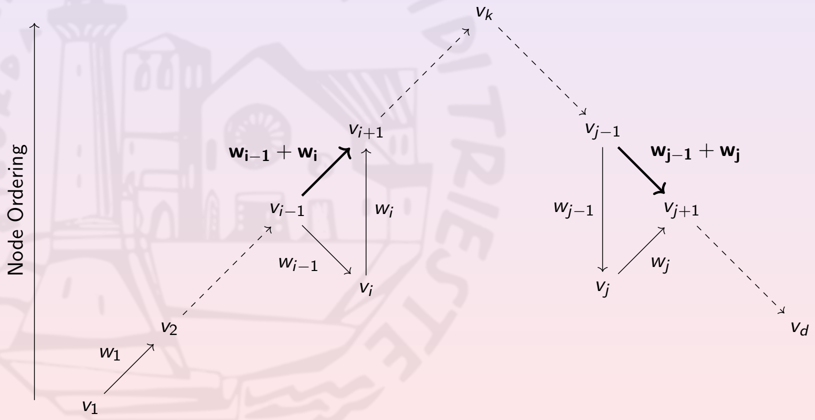
Shortest Paths on Contraction Hierarchy

Let $v_1, \dots v_d$ be a shortest path on the CH with $w_i = W[v_1, v_d]$



Shortest Paths on Contraction Hierarchy

Let $v_1, \ldots v_d$ be a shortest path on the CH with $w_i = W[v_1, v_d]$



Shortest Paths on Contraction Hierarchy (Cont'd)

The shortest paths on CH have the form $v_1, \ldots, v_k, \ldots, v_d$ where:

- $v_{i-1} < v_i$ for all $i \leq k$
- $v_{i-1} > v_i$ for all $i > k$

Shortest Paths on Contraction Hierarchy (Cont'd)

The shortest paths on CH have the form $v_1, \ldots, v_k, \ldots, v_d$ where:

- $v_{i-1} < v_i$ for all $i \leq k$
- $v_{i-1} > v_i$ for all $i > k$

Find them by building $G \uparrow = (V, E \uparrow)$ and $G \downarrow = (V, E \downarrow)$ where:

- $E \uparrow = \{(v, w) \in E \mid v < w\}$
- $E \downarrow = \{(v, w) \in E \mid v > w\}$

Shortest Paths on Contraction Hierarchy (Cont'd)

The shortest paths on CH have the form $v_1, \ldots, v_k, \ldots, v_d$ where:

- $v_{i-1} < v_i$ for all $i \leq k$
- $v_{i-1} > v_i$ for all $i > k$

Find them by building $G\uparrow = (V, E\uparrow)$ and $G\downarrow = (V, E\downarrow)$ where:

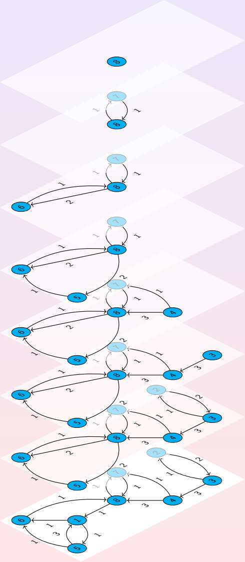
- $E\uparrow = \{(v, w) \in E \,|\, v < w\}$
- $E\downarrow = \{(v, w) \in E \,|\, v > w\}$

and using a bidirectional version of Dijkstra on $G\uparrow$ and $G\downarrow$

It search forward on $G\uparrow$ and backward on $G\downarrow$ until the two searches finalize the same node

Contraction Hierarchy and Graph Size

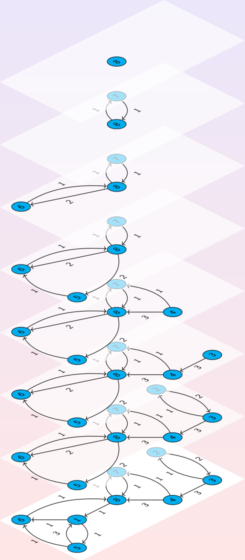
Many overlay graphs shares a large set of edges



Contraction Hierarchy and Graph Size

Many overlay graphs shares a large set of edges

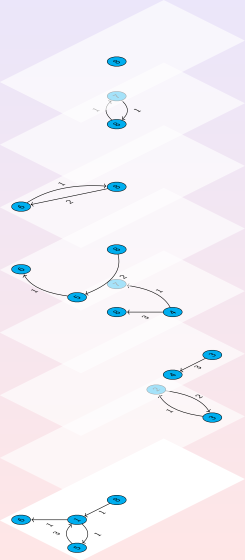
We can store only those that are about to disappear and the involved nodes



Contraction Hierarchy and Graph Size

Many overlay graphs shares a large set of edges

We can store only those that are about to disappear and the involved nodes

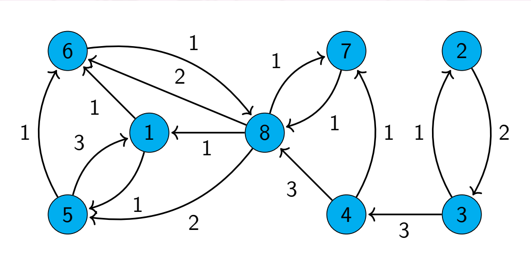


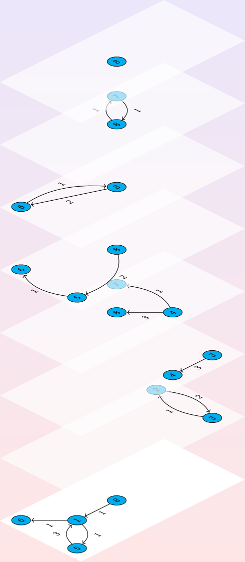
Contraction Hierarchy and Graph Size

Many overlay graphs shares a large set of edges

We can store only those that are about to disappear and the involved nodes

Looking it from the top, we get again





Partition Huge Graphs

By merging subsequent layers, we endup with graphs which have high probability to be disconnected

Partition Huge Graphs

By merging subsequent layers, we endup with graphs which have high probability to be disconnected

Huge graphs can be parted into subgraphs at the lowest levels and connect them by using highest levels