CS2010: Data Structures and Algorithms II

Shortest paths in graphs

Ivana.Dusparic@scss.tcd.ie

Outline

› Shortest paths
  – Problem definition
  – Single source shortest path
    › BFS?
    › Topological sort – acyclic graphs but ok with negative weights
    › Dijkstra – non negative weights but ok with cycles
    › Bellman-Ford – non-negative cycles
  – Single-pair shortest path
    › Uniform cost
    › Greedy Best first
    › A* search algorithm
  – All pairs shortest path
    › Floyd-Warshall

Shortest paths in an edge-weighted digraph

Given an edge-weighted digraph, find the shortest path from s to t.

edge-weighted digraph

```
4->5   0.35
5->4   0.35
4->7   0.37
5->7   0.28
7->5   0.28
5->1   0.32
0->4   0.38
0->2   0.26
7->3   0.39
1->3   0.29
2->7   0.34
6->2   0.40
3->6   0.52
6->0   0.58
6->4   0.93
```
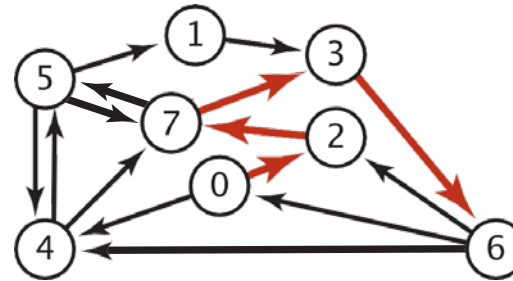

shortest path from 0 to 6

```
0->2   0.26
2->7   0.34
7->3   0.39
3->6   0.52
```

Shortest path variants

Which vertices?

- Single source: from one vertex s to every other vertex.
- Single sink: from every vertex to one vertex t.
- Source-sink: from one vertex s to another t.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.

Cycles?

- No directed cycles.
- No negative cycles.

Properties

› *Paths are directed.* A shortest path must respect the direction of its edges.

› *The weights are not necessarily distances.* Geometric intuition can be helpful, but the edge weights might represent time or cost.

› *Not all vertices need be reachable.* If t is not reachable from s, there is no path at all, and therefore there is no shortest path from s to t.

› *Negative weights introduce complications.* For the moment, we assume that edge weights are positive (or zero).

› *Shortest paths are normally simple.* Our algorithms ignore zero-weight edges that form cycles, so that the shortest paths they find have no cycles.

› *Shortest paths are not necessarily unique.* There may be multiple paths of the lowest weight from one vertex to another; we are content to find any one of them.

› *Parallel edges and self-loops may be present.* In the text, we assume that parallel edges are not present and use the notation v->w to refer to the edge from v to w

Types of edges recap

› Undirected
  – DFS, BFS

› Directed
  – DFS, BFS

› Weighted undirected
  – MSTs

› Here: Weighted directed

Weighted edge API

```
public class DirectedEdge

                DirectedEdge(int v, int w, double weight)
    double  weight()                              weight of this edge
       int  from()                                vertex this edge points from
       int  to()                                  vertex this edge points to
    String  toString()                            string representation
```
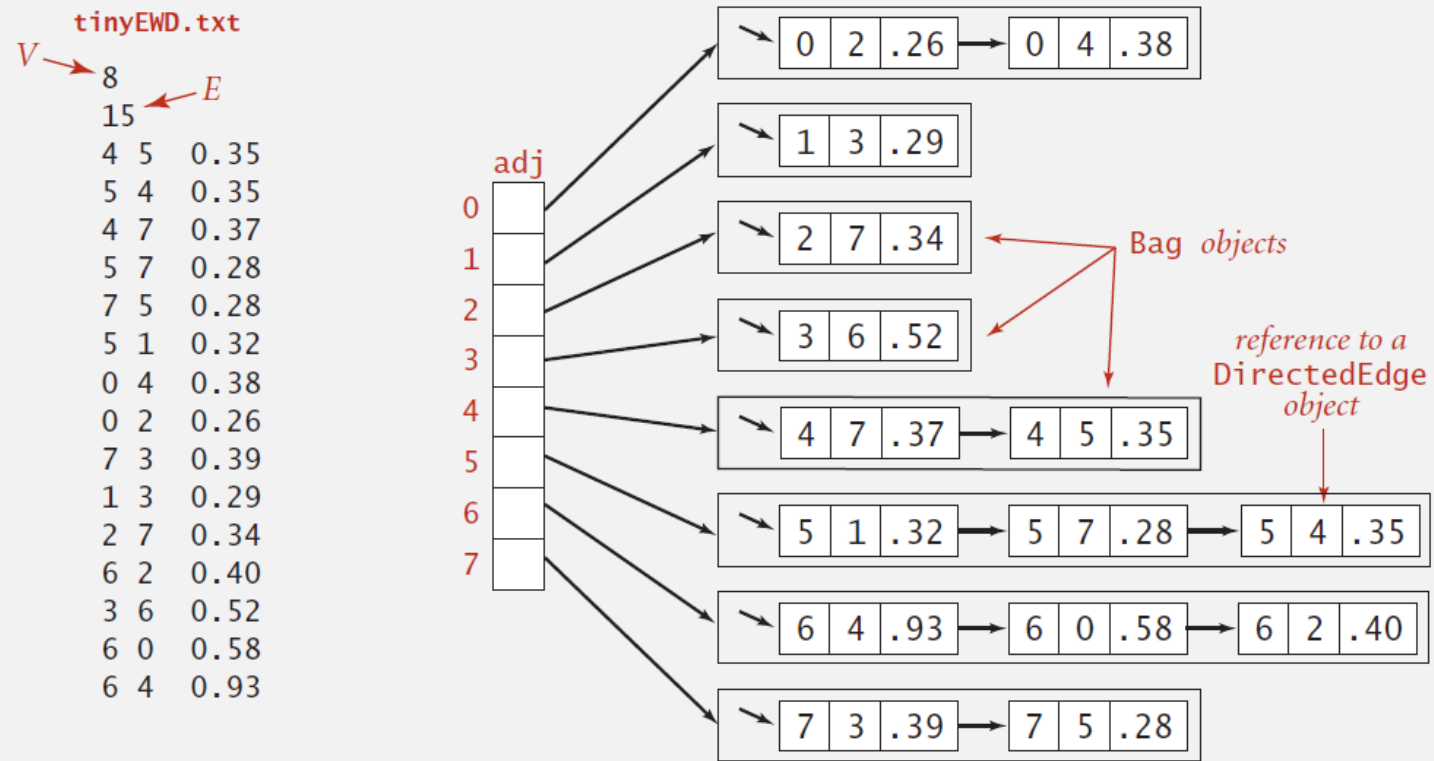
Edge weighted graph API

public class	**EdgeWeightedDigraph**	
	EdgeWeightedDigraph(int V)	*empty V-vertex digraph*
	EdgeWeightedDigraph(In in)	*construct from* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(DirectedEdge e)	*add* e *to this digraph*
Iterable\<DirectedEdge\>	adj(int v)	*edges pointing from* v
Iterable\<DirectedEdge\>	edges()	*all edges in this digraph*
String	toString()	*string representation*

Shortest path API

```
public class SP

              SP(EdgeWeightedDigraph G, int s)    constructor

    double    distTo(int v)                        distance from s
                                                   to v, ∞ if no path

   boolean    hasPathTo(int v)                     path from s to v?

Iterable<DirectedEdge>  pathTo(int v)              path from s to v,
                                                   null if none
```

Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V → 8
15 ← E
4 5 0.35
5 4 0.35
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32
0 4 0.38
0 2 0.26
7 3 0.39
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93

adj

0
1
2
3
4
5
6
7

| 0 | 2 | .26 | → | 0 | 4 | .38 |

| 1 | 3 | .29 |

| 2 | 7 | .34 |

Bag objects

| 3 | 6 | .52 |

reference to a DirectedEdge object

| 4 | 7 | .37 | → | 4 | 5 | .35 |

| 5 | 1 | .32 | → | 5 | 7 | .28 | → | 5 | 4 | .35 |

| 6 | 4 | .93 | → | 6 | 0 | .58 | → | 6 | 2 | .40 |

| 7 | 3 | .39 | → | 7 | 5 | .28 |

Data structures needed

› Two vertex-indexed arrays

- distTo[v] is length of shortest path from s to v.

- edgeTo[v] is last edge on shortest path from s to v.
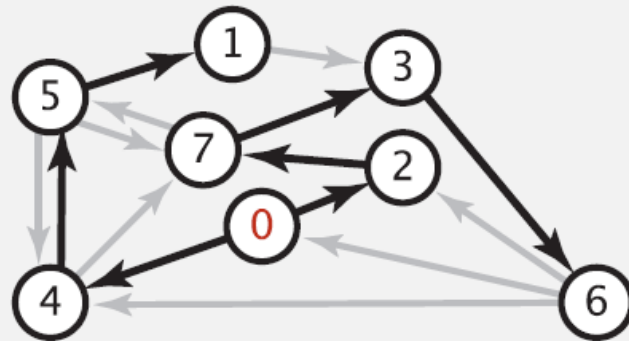
	edgeTo[]		distTo[]
0	null		0
1	5->1	0.32	1.05
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49
7	2->7	0.34	0.60

shortest–paths tree from 0

parent–link representation

Edge relaxation

› Our shortest-paths implementations are based on an operation known as relaxation.

› Initialize distTo[s] to 0 and distTo[v] to infinity for all other vertices v.

› To relax an edge v->w means to test whether the best known way from s to w is to go from s to v, then take the edge from v to w, and, if so, update our data structures.

Edge relaxation

```java
private void relax(DirectedEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Edge relaxation

Relax edge $e = v \rightarrow w$.

- distTo[v] is length of shortest known path from s to v.
- distTo[w] is length of shortest known path from s to w.
- edgeTo[w] is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v,
  update both distTo[w] and edgeTo[w].

v→w successfully relaxes

black edges
are in edgeTo[]
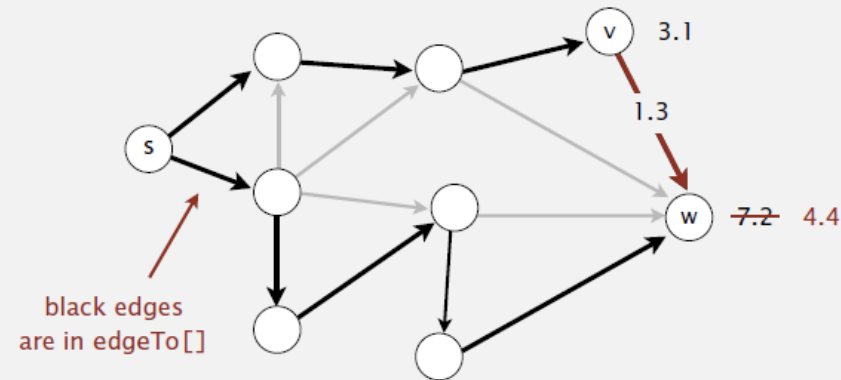
Vertex relaxation

› *Vertex relaxation -* relax all the edges pointing from a given vertex.

```java
private void relax(EdgeWeightedDigraph G, int v) {
    for (DirectedEdge e : G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from s iff:

- `distTo[s]` = 0.

- For each vertex v, `distTo[v]` is the length of some path from s to v.

- For each edge e = v→w, `distTo[w]` ≤ `distTo[v]` + `e.weight()`.

Pf. ⟸ [necessary]

- Suppose that `distTo[w]` > `distTo[v]` + `e.weight()` for some edge e = v→w.

- Then, e gives a path from s to w (through v) of length less than `distTo[w]`.
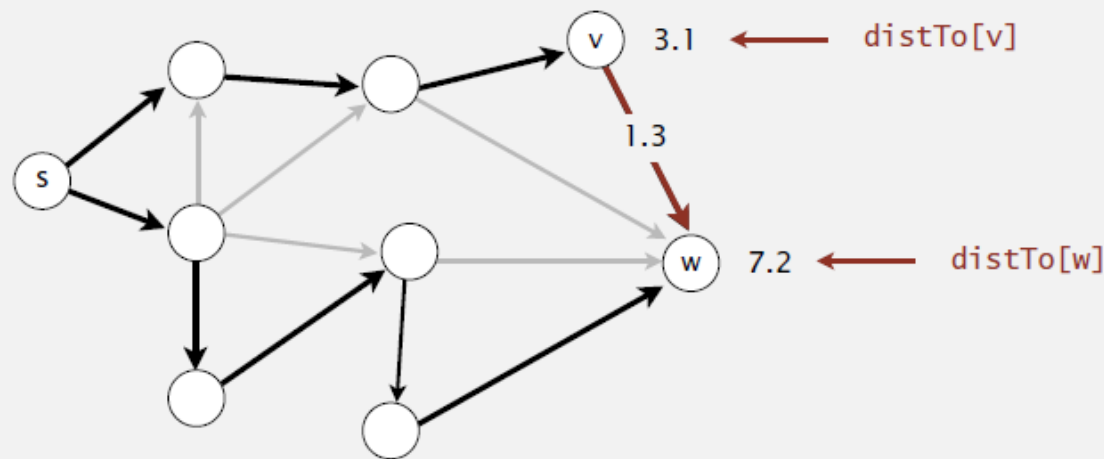
Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.
Then `distTo[]` are the shortest path distances from s iff:

- `distTo[s]` = 0.
- For each vertex v, `distTo[v]` is the length of some path from s to v.
- For each edge $e = v \rightarrow w$, `distTo[w]` $\leq$ `distTo[v]` + `e.weight()`.

Pf. $\Rightarrow$ [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = w$ is a shortest path from s to w.
- Then,

  `distTo[v₁]` $\leq$ `distTo[v₀]` + `e₁.weight()`

  `distTo[v₂]` $\leq$ `distTo[v₁]` + `e₂.weight()`

  $\ldots$

  `distTo[vₖ]` $\leq$ `distTo[vₖ₋₁]` + `eₖ.weight()`

  $e_i = i^{th}$ edge on shortest path from s to w

- Add inequalities; simplify; and substitute `distTo[v₀]` = `distTo[s]` = 0:

  `distTo[w]` = `distTo[vₖ]` $\leq$ `e₁.weight()` + `e₂.weight()` + … + `eₖ.weight()`

  weight of shortest path from s to w

- Thus, `distTo[w]` is the weight of shortest path to w. ∎

  weight of some path from s to w

Generic shortest path algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:
- Relax any edge.

Proposition. Generic algorithm computes SPT (if it exists) from s.

Pf sketch.
- The entry distTo[v] is always the length of a simple path from s to v.
- Each successful relaxation decreases distTo[v] for some v.
- The entry distTo[v] can decrease at most a finite number of times. ∎

Generic shortest path algorithm

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).
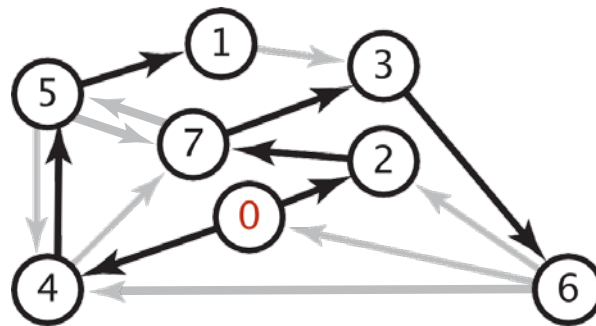
Dijkstra's shortest path algorithm

Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists.

Consequence. Can represent the SPT with two vertex-indexed arrays:
- distTo[v] is length of shortest path from s to v.
- edgeTo[v] is last edge on shortest path from s to v.


shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

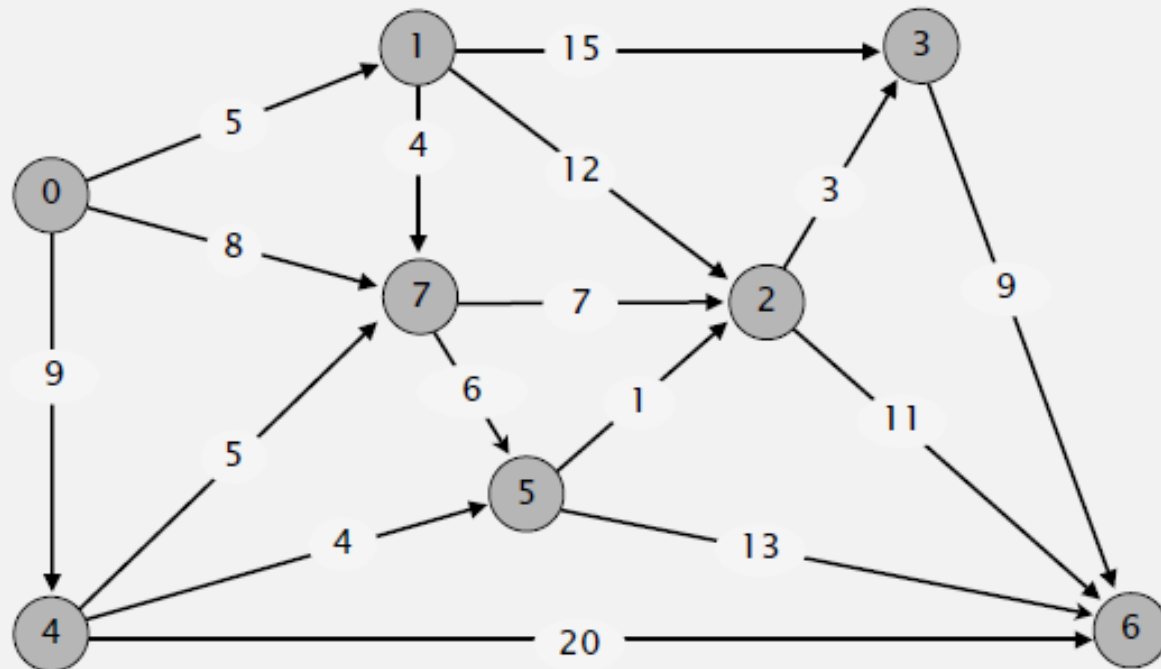# Relax edge $e = v \rightarrow w$.

- distTo[v] is length of shortest known path from s to v.
- distTo[w] is length of shortest known path from s to w.
- edgeTo[w] is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v,
  update both distTo[w] and edgeTo[w].

v→w successfully relaxes

3.1

1.3

w   7.2   4.4

black edges
are in edgeTo[]
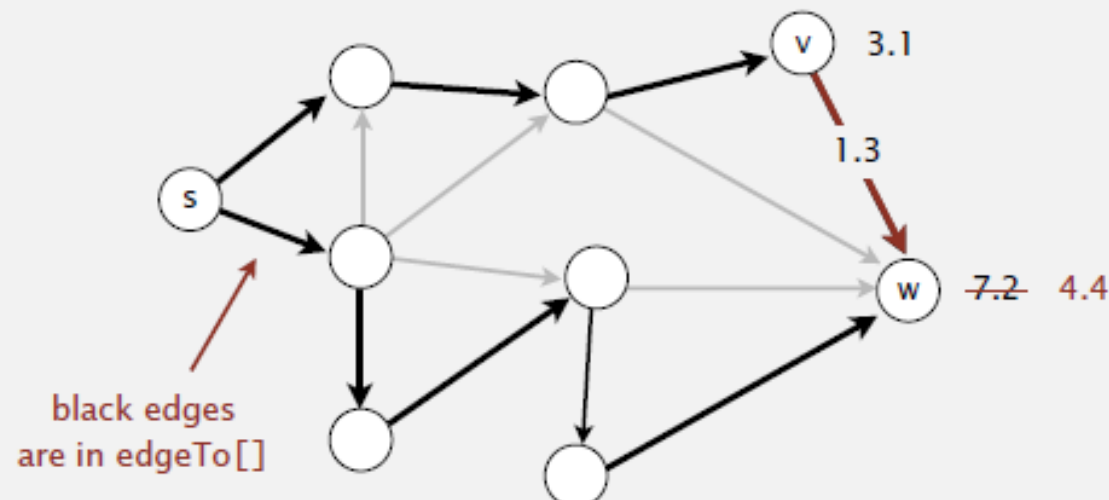
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

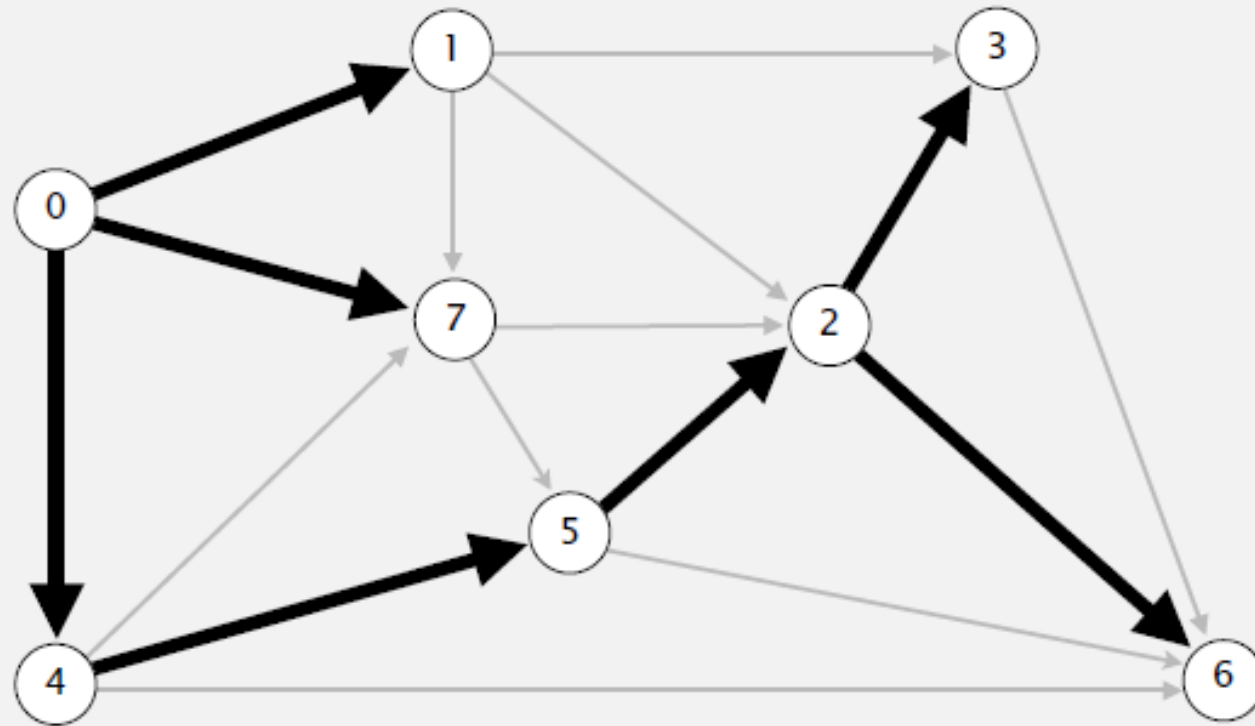an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0

Dijkstra Demo

https://algs4.cs.princeton.edu/lectures/44DemoDijkstra.pdf

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
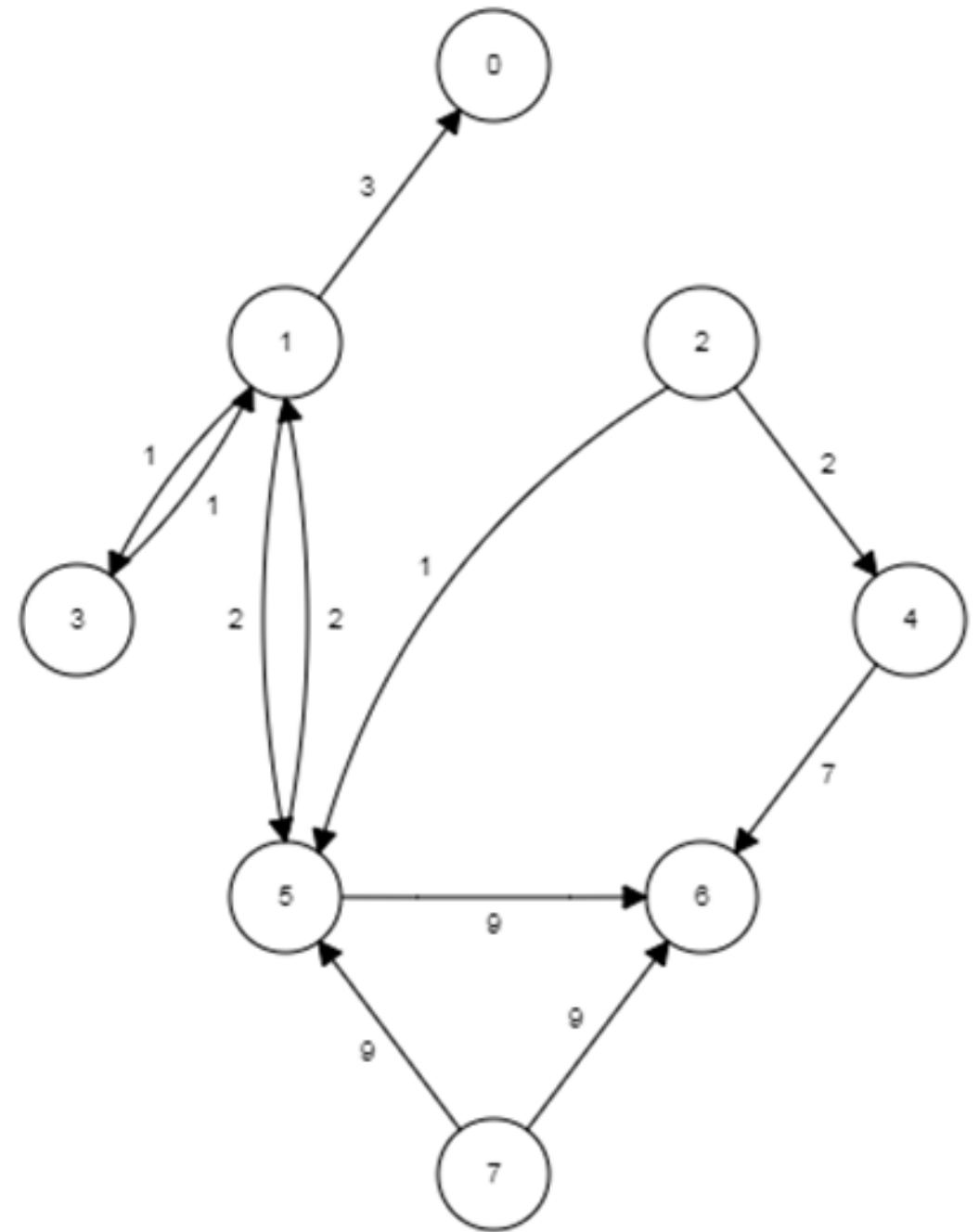- Add vertex to tree and relax all edges pointing from that vertex.

v	distTo[]	edgeTo[]
0	0.0	–
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest–paths tree from vertex s

Dijkstra exercise

Start from vertex 1

v	DistTo[]	EdgeTo[]
0		
1		
2		
3		
4		
5		
6		
7		

Solution

Vertex	Known	Cost	Path			
0	T	3	1	1	0	
1	T	0	-1	1		
2	F	INF	-1	No Path		
3	T	1	1	1	3	
4	F	INF	-1	No Path		
5	T	2	1	1	5	
6	T	11	5	1	5	6
7	F	INF	-1	No Path		

Dijkstra performance

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d–way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	$1^{\dagger}$	$\log V^{\dagger}$	$1^{\dagger}$	$E + V \log V$

† amortized

Bottom line.
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Dijkstra's algorithm seem familiar?

- Prim's algorithm is essentially the same algorithm.
- Both are in a family of algorithms that compute a spanning tree.

Main distinction: Rule used to choose next vertex for the tree.

- Prim: Closest vertex to the tree (via an undirected edge).
- Dijkstra: Closest vertex to the source (via a directed path).
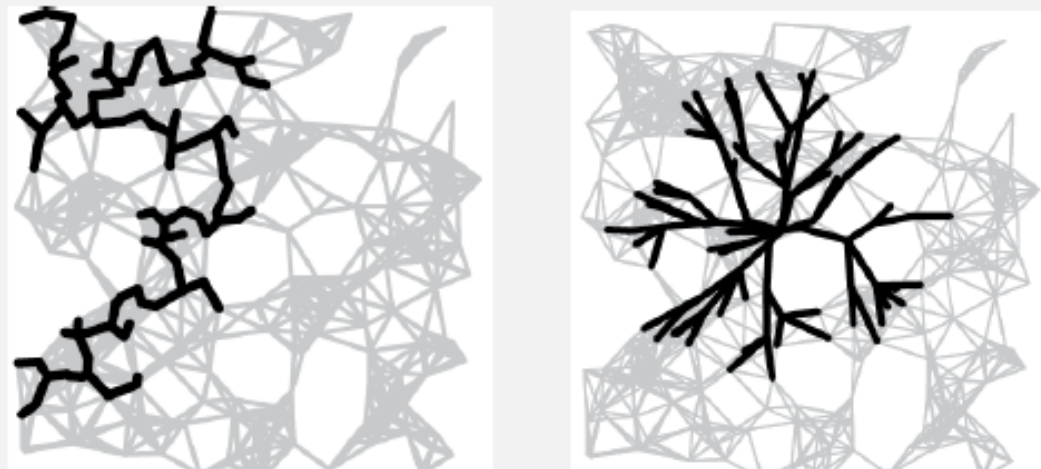
Dijkstra – why can't work with negative weights?

› Example?

› Consquence: picking the shortest candidate edge (local optimality) always ends up being correct (global optimality).
  – Greedy algorithm!
  – Wouldn't be true if used dijkstra with negative weights – use a different algorithm instead!

› Can be modified to work with negative weights, i.e., vertex can be en-queued more than once -> exponential worst case running time though!

Uniform-cost search

› Algorithm which is focused on finding a single shortest path to a single finishing point rather than the shortest path to every point

› Stops as soon as the finishing point is found
  – Found a GOAL/SOLUTION in AI
  – Check if at the goal only when selecting the vertex for expansion

› Summary: expand least-cost-so-far unexpanded vertex

Greedy best-first search

Maze example
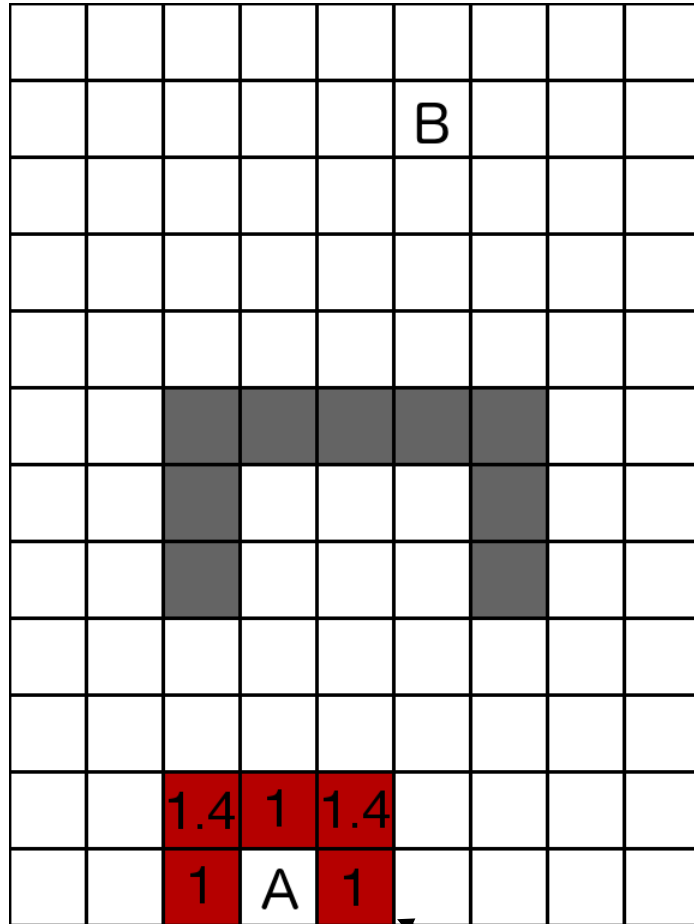
adjacent edge weights

› Which nodes will
  – DFS visit next?
  – BFS visit next?
  – Dijkstra's visit next?

› How could we write an algorithm that **prefers** edges **more likely** to be on the shortest path?

Uninformed vs informed search

› Dijkstra/uniform cost search - so-called "uninformed" or "blind" search

› BFS and DFS are blind
– As likely to go away from the destination than towards it

› Dijkstra
– Is looking for shortest paths, in all directions

› Often we have some knowledge about the destination
– Domain-specific
– Informed search
– What do we do with this knowledge? Heuristic function

Heuristic

› Need to estimate how close we are to the destination
  – Use a **heuristic** - an approximate measure of how close you are to the destination/target
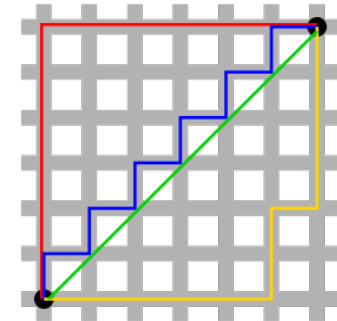 – Should be easy to compute!

› Examples
  – Google map route planning?
    › Euclidean distance
  – Mutate the word game
    › Number of letters correct

Want: CAT ⟶ DOG

CAT → COT → COG → DOG
CAD, CUT, PAT
COP, POT, DOT
CON, FOG, LOG
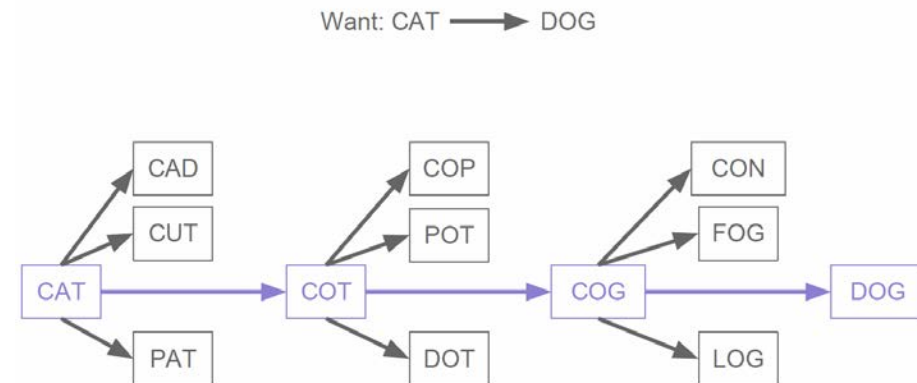
Maze example 2
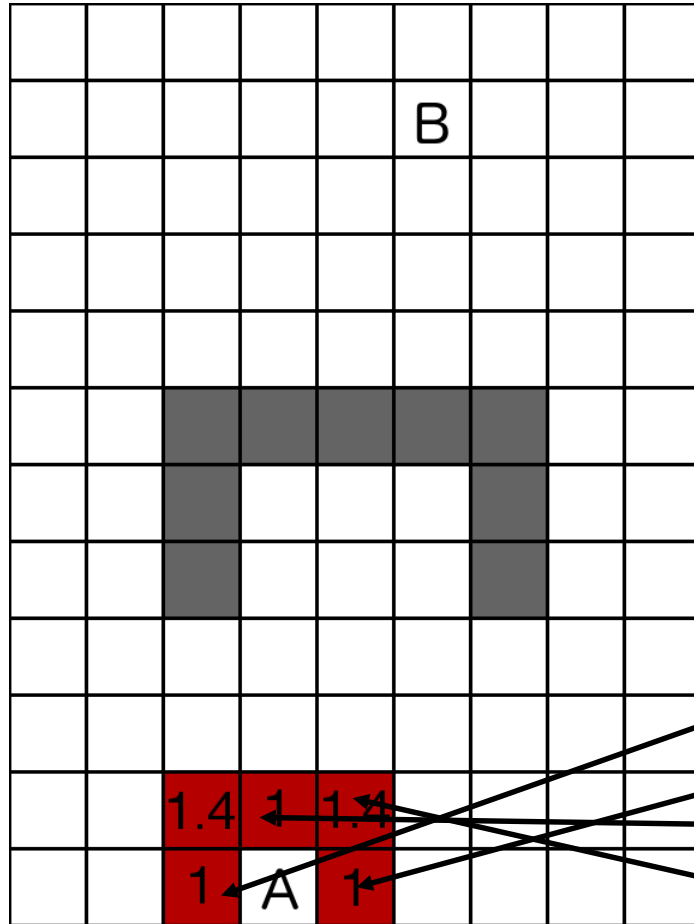
> › Manhattan distance
>> – Number of blocs up + number of blocs across

10 + 3 = 13
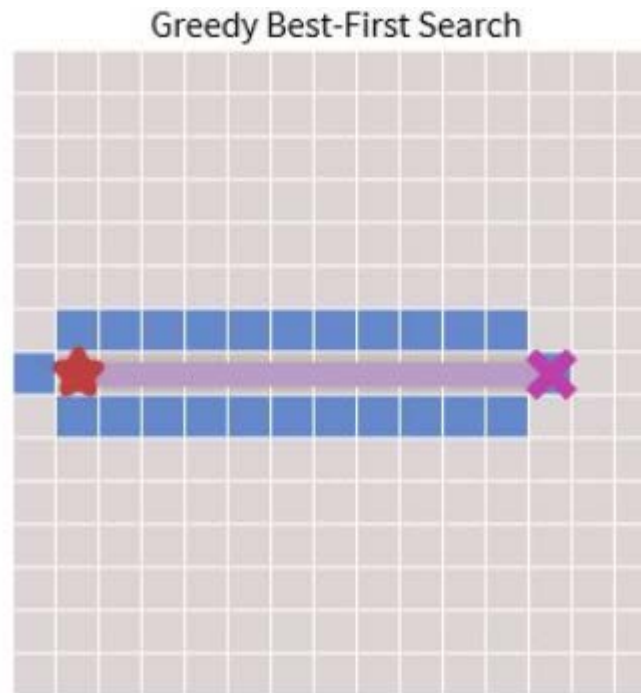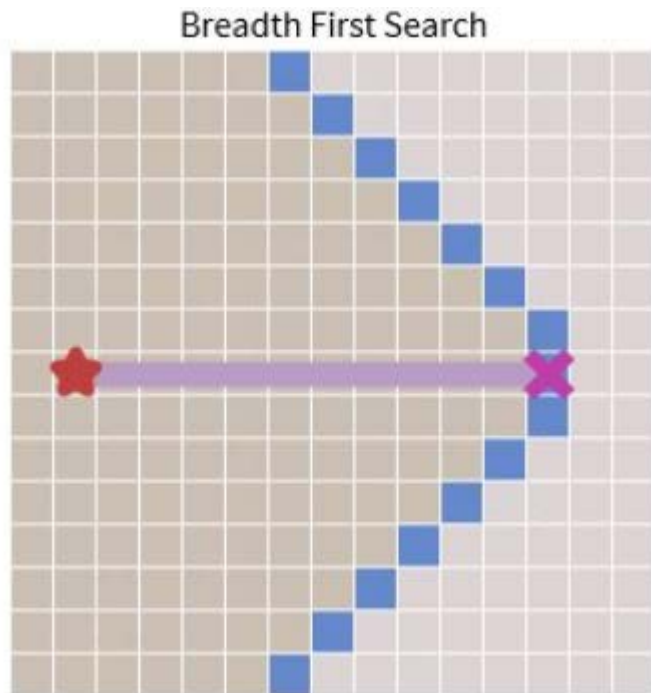
10 + 1 = 11

9 + 2 = 11

9 + 1 = 10

Best-first search

› Node selected for expansion based on a function $f(n)$

› $f(n)$ is a cost estimate, and the nodes with the lowest $f(n)$ are expanded first

› Choice of $f(n)$ = search strategy

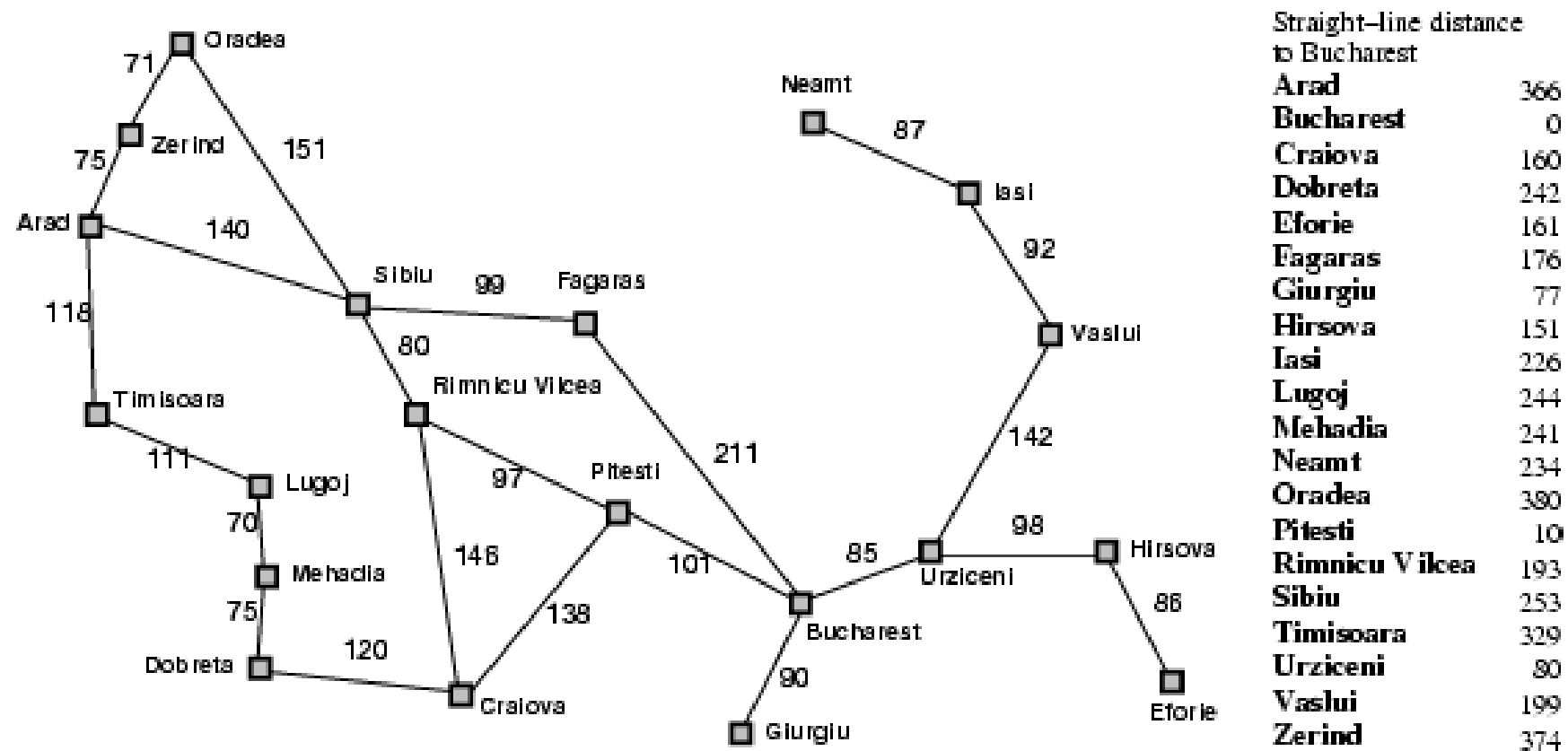› Special-cases: greedy search and A* search

Greedy best first search

› Like Breadth-First Search except...
  – queue of choices are ranked using a heuristic
    › priority queue
  – the parent stays in the queue so that it can back-track
  – stops when goal state found

› Greedy means choose the **local maximum** at each stage **hoping** to find the **global maximum**
  – Dijkstra looks at distance already travelled
  – GBFS looks at distance to go

Greedy best-first search

› Each step moving closer to the target, e.g.,

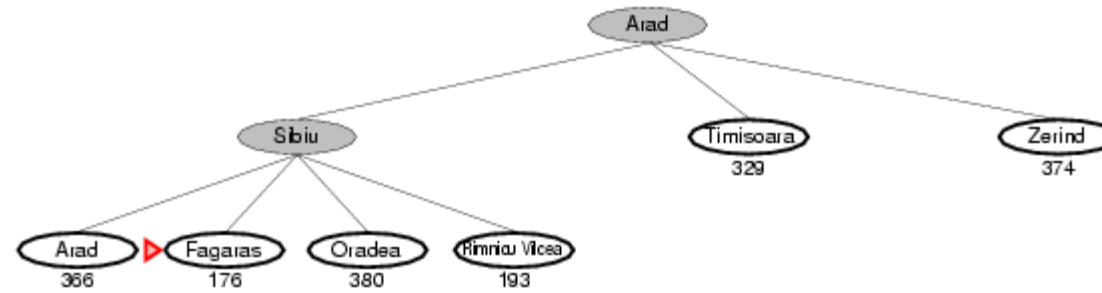# Greedy best-first search – Romania example

Straight–line distance
to Bucharest

|	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

Greedy best-first search example

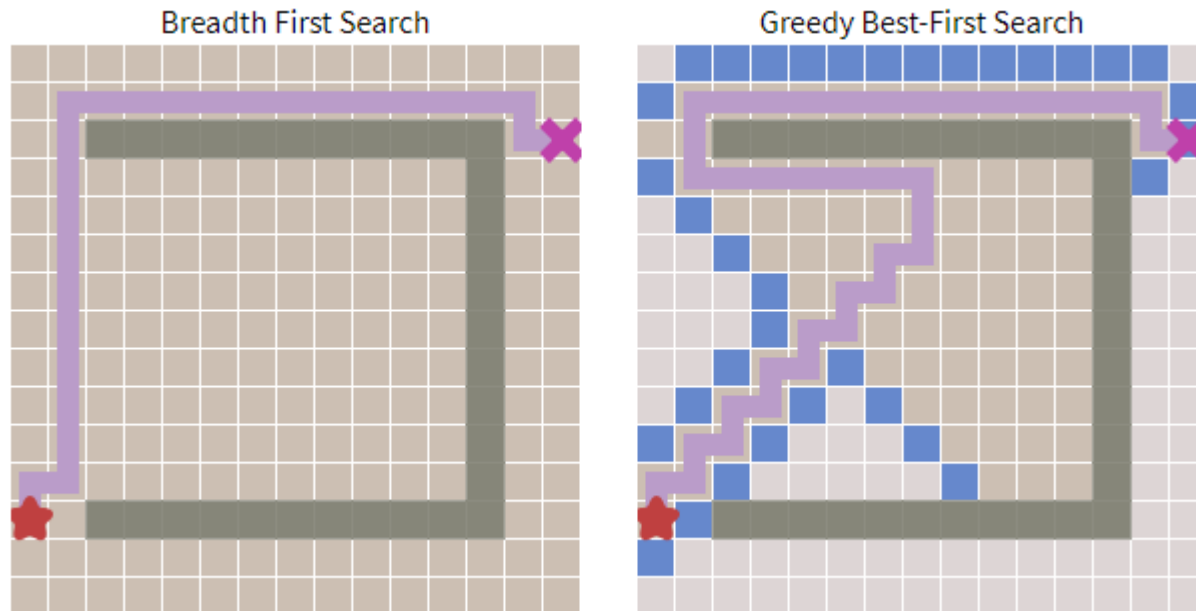# Greedy best-first search example

Greedy best-first search example
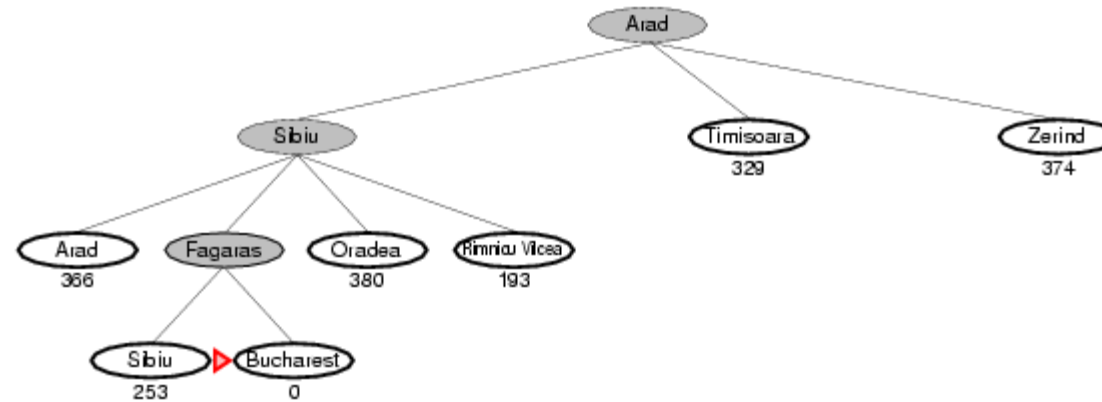
Optimality

› Demo of BFS vs Greedy best-first search

› https://cs.stanford.edu/people/abisee/tutorial/greedy.html

Breadth First Search

Greedy Best-First Search

Greedy Best First search performance

› Usually many fewer nodes visited than BFS and Dijkstra
  – Useful when need an answer quickly!

› **Does not guarantee a shortest path** like Dijkstra's
  – Greedy = choose what appears best
  – Vulnerable to **local maxima** traps

A*

A* shortest path

› Single source, single destination/goal

› Combine Dijkstra's algorithm/Uniform-cost search with greedy best first search
– Dijkstra looks at minimum cost so far – backward cost
– Greedy best first looks at minimum estimated cost – forward cost

› Uses heuristics to guide its search and achieve better performance
– prioritizes paths that seem to be leading closer to the goal

A*

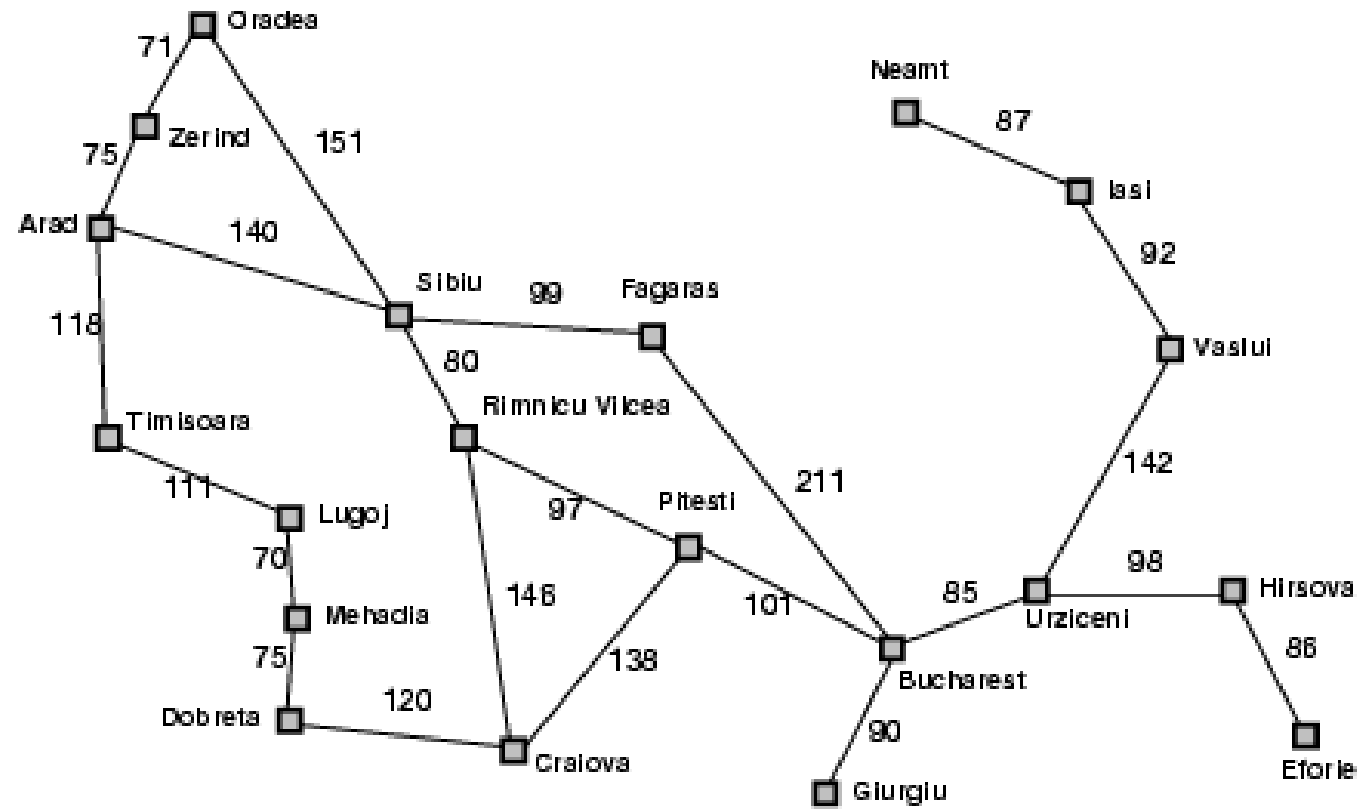› actual cost of path so far + estimated cost to goal

 – f(n) = g(n) + h(n)

› This helps avoid local maxima traps

› May be slower than Greedy Best-First
 – But guarantees shortest path
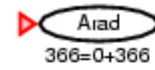
› Main idea: avoid paths that have already been expensive

Heuristic function

› solving a problem more quickly when classic methods are too slow

› finding an approximate solution when classic methods fail to find any exact solution

› trading optimality, completeness, accuracy, or precision for speed.

› a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow
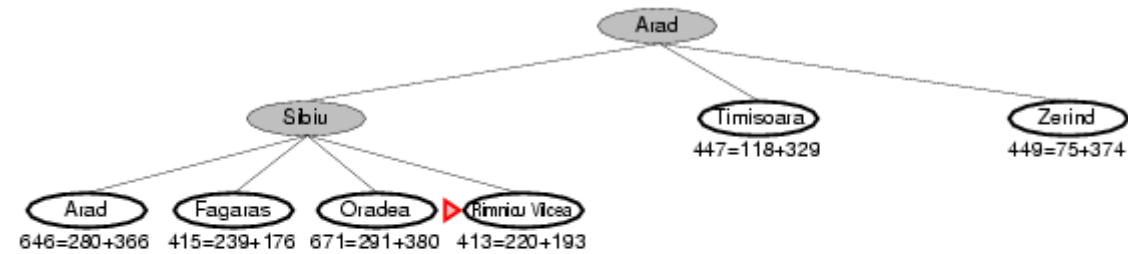
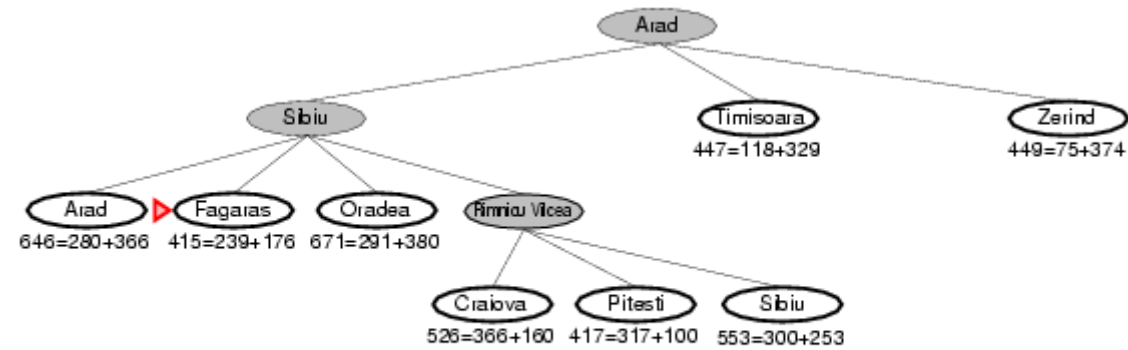– Eg approximate exact solution

A* – Romania example

Straight–line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example
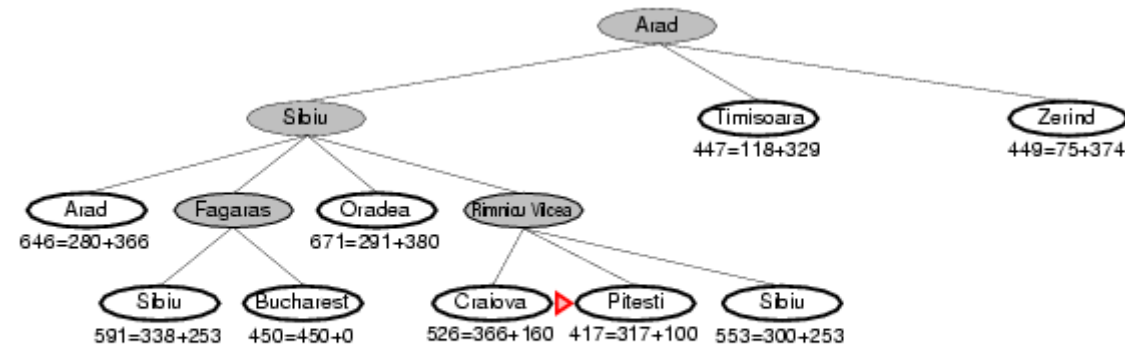

Arad
366=0+366

A* search example
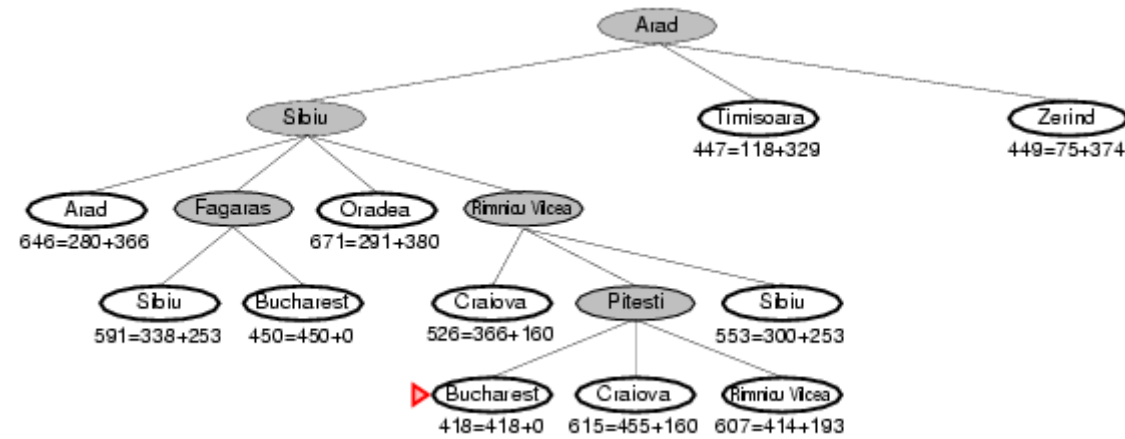
A* search example

A* search example

A* search example

A* search example

Heuristic function h(n)

› The 2 most important properties:
  – relatively cheap to compute
  – relatively accurate estimator of the cost to reach a goal. Usually a "good" heuristic is if ½ opt(n) <h(n)<opt(n)

› Admissible heuristic

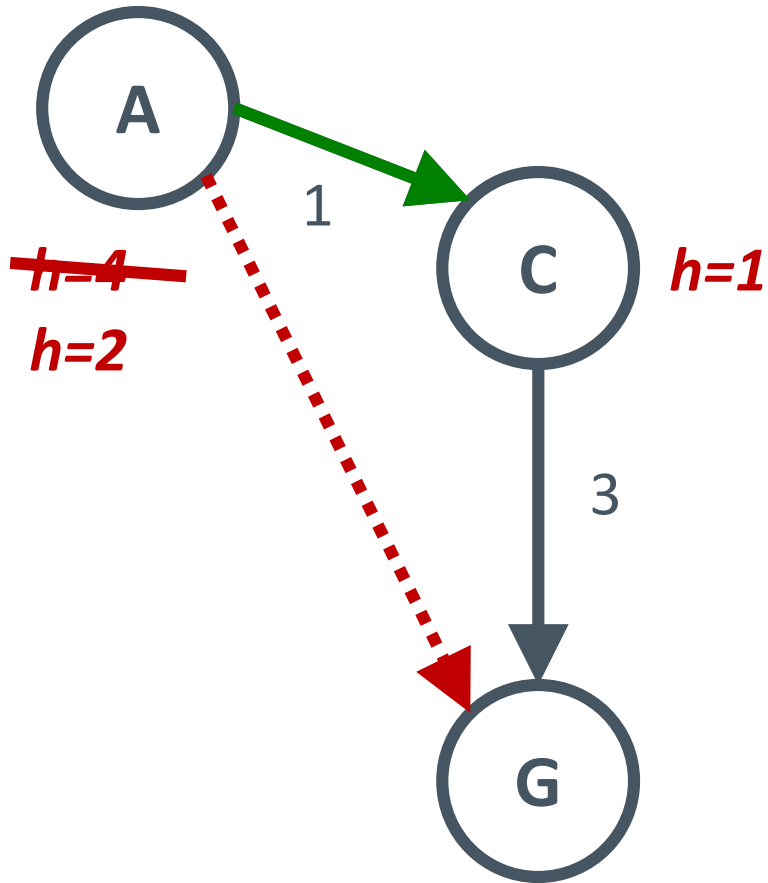# Admissible heuristics

› A heuristic $h(n)$ is admissible if for every node n,

$h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.

› An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

› Example: *Romania example straight line distance* - never overestimates the actual road distance

Admissible heuristics

› Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe

Consistency of Heuristics

› Main idea: estimated heuristic costs ≤ actual costs

  – Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G

  – Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)

› Consequences of consistency:

  – The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)

  – A* graph search is optimal

A*

› How to pick a heuristic function?

› Domain knowledge – some information about a goal

› Eg in route planning:
  – Manhattan distance – on a square grid that allows 4 directions of movement
  – Diagonal distance – 8-direction square
  – Euclidean distance – any direction

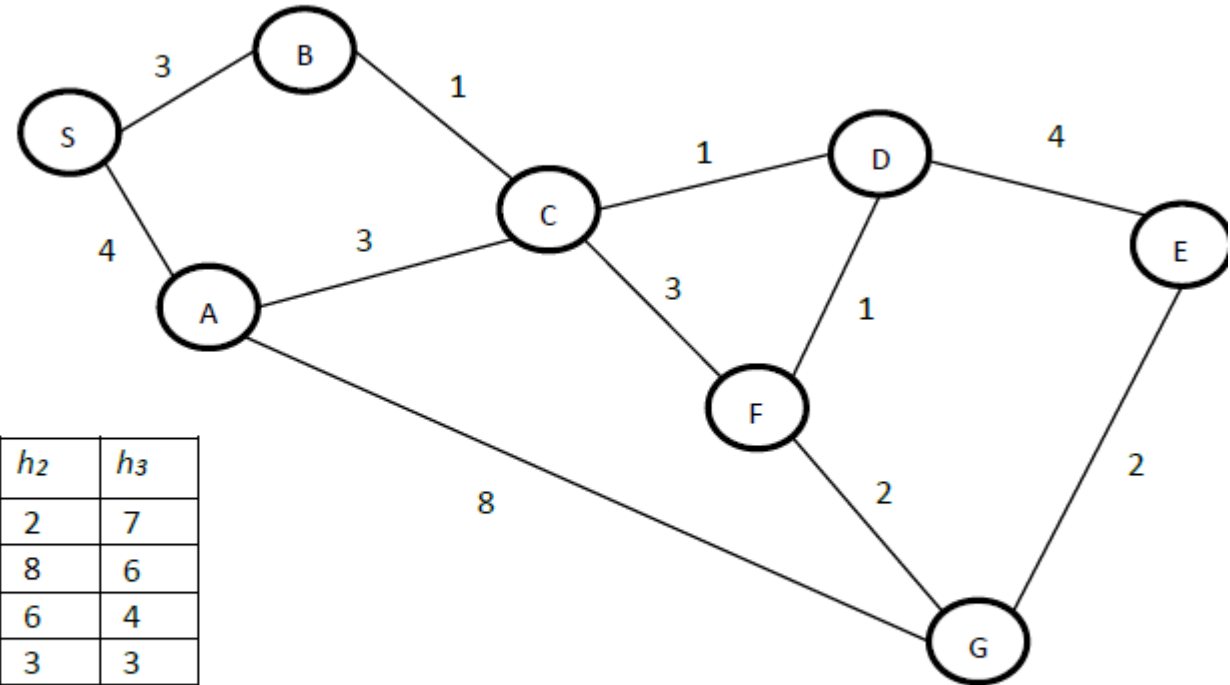 – A good example on route planning in games http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

A* vs Dijkstra visualisation

https://www.youtube.com/watch?v=g024lzsknDo

A* exercise

- Are heuristics h1, h2, h3
  - Admissible?
  - Consistent?
- Fine the shortest path from S to G using one of the consistent heuristics

	h_1	h_2	h_3
S	3	2	7
A	6	8	6
B	2	6	4
C	3	3	3
D	2	3	3
E	1	1	1
F	2	2	2
G	0	0	0

Topological sort shortest path

Shortest path in acyclic graphs

› Is it easier than in graphs with cycles?

› Yes! Linear time

› Use topological sort (which only works in DAGs – directional acyclic graphs) to find shortest path

› If we have an edge from x to y, the ordering visits x before y

› Shortest path visits/relaxes edges in topological order

› Topological sort – identify a vertex with no incoming edges, add it to the ordered list, remove it, repeat…
  – (decrease by 1 and conquer)

Topological sort shortest path demo

https://algs4.cs.princeton.edu/lectures/44DemoAcyclicSP.pdf

So far

› No negative weights – Dijkstra

› No cycles – Topological sort shortest path

› What if the graph has negative weights and cycles?

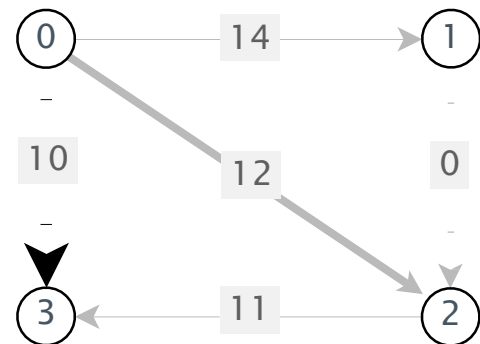› What about negative cycles?

Bellman-Ford Shortest Path Algorithm

Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.
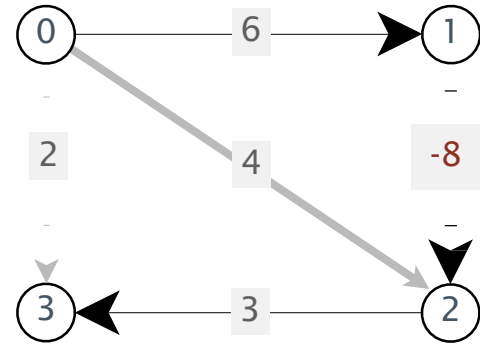
Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Add a constant to every edge weight doesn't work.

Adding 8 to each edge weight changes the
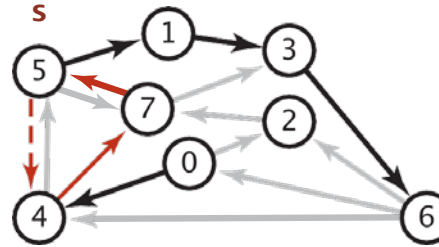shortest path from 0→1→2→3 to 0→3.

Conclusion. Need a different algorithm.

Negative cycles

Def. A negative cycle is a directed cycle whose sum of edge weights is negative.

digraph

```
4->5   0.35
5->4  -0.66
4->7   0.37
5->7   0.28
7->5   0.28
5->1   0.32
0->4   0.38
0->2   0.26
7->3   0.39
1->3   0.29
2->7   0.34
6->2   0.40
3->6   0.52
6->0   0.58
6->4   0.93
```


negative cycle (-0.66 + 0.37 + 0.28)
5->4->7->5

shortest path from 0 to 6
0->4->7->5->4->7->5...->1->3->6

Proposition. A SPT exists iff no negative cycles.

assuming all vertices reachable from s

Bellman-Ford algorithm

Bellman-Ford algorithm

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.
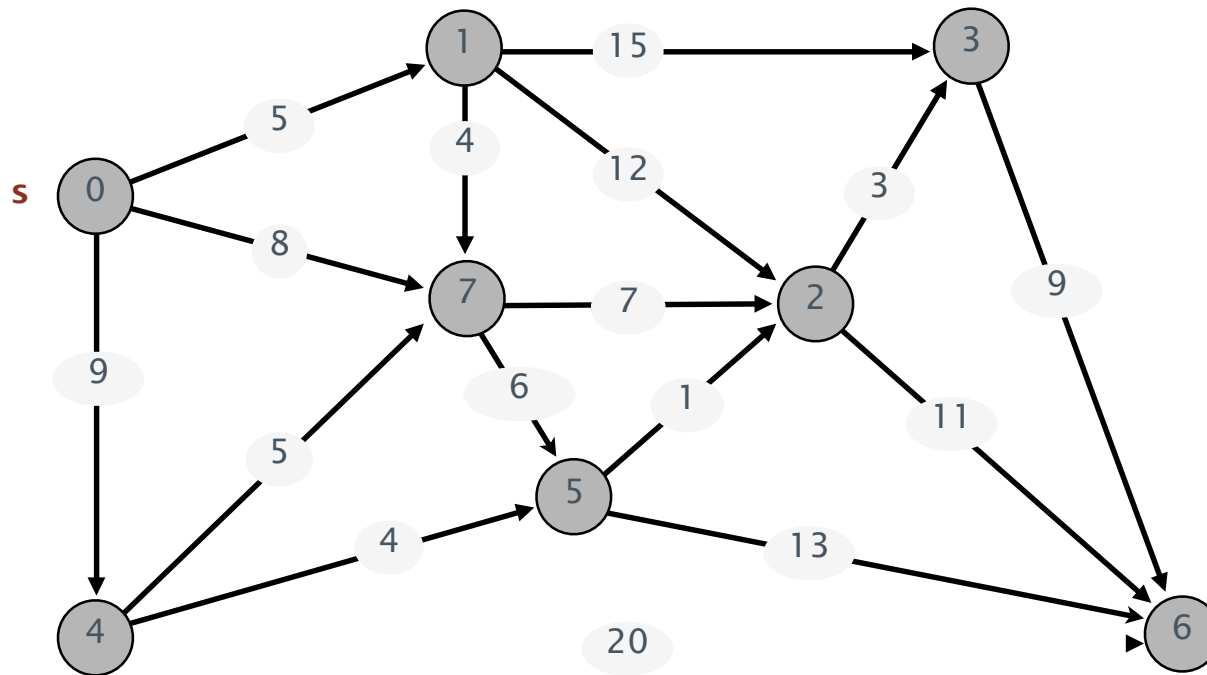
Repeat V times:

    – Relax each edge.

```
for (int i = 0; i < G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

pass i (relax each edge)

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

s

an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Bellman-Ford Demo

› https://algs4.cs.princeton.edu/lectures/44DemoBellmanFord.pdf

› Only positive weights in this example – paper exercise with negative weights

Bellman-Ford Exercise
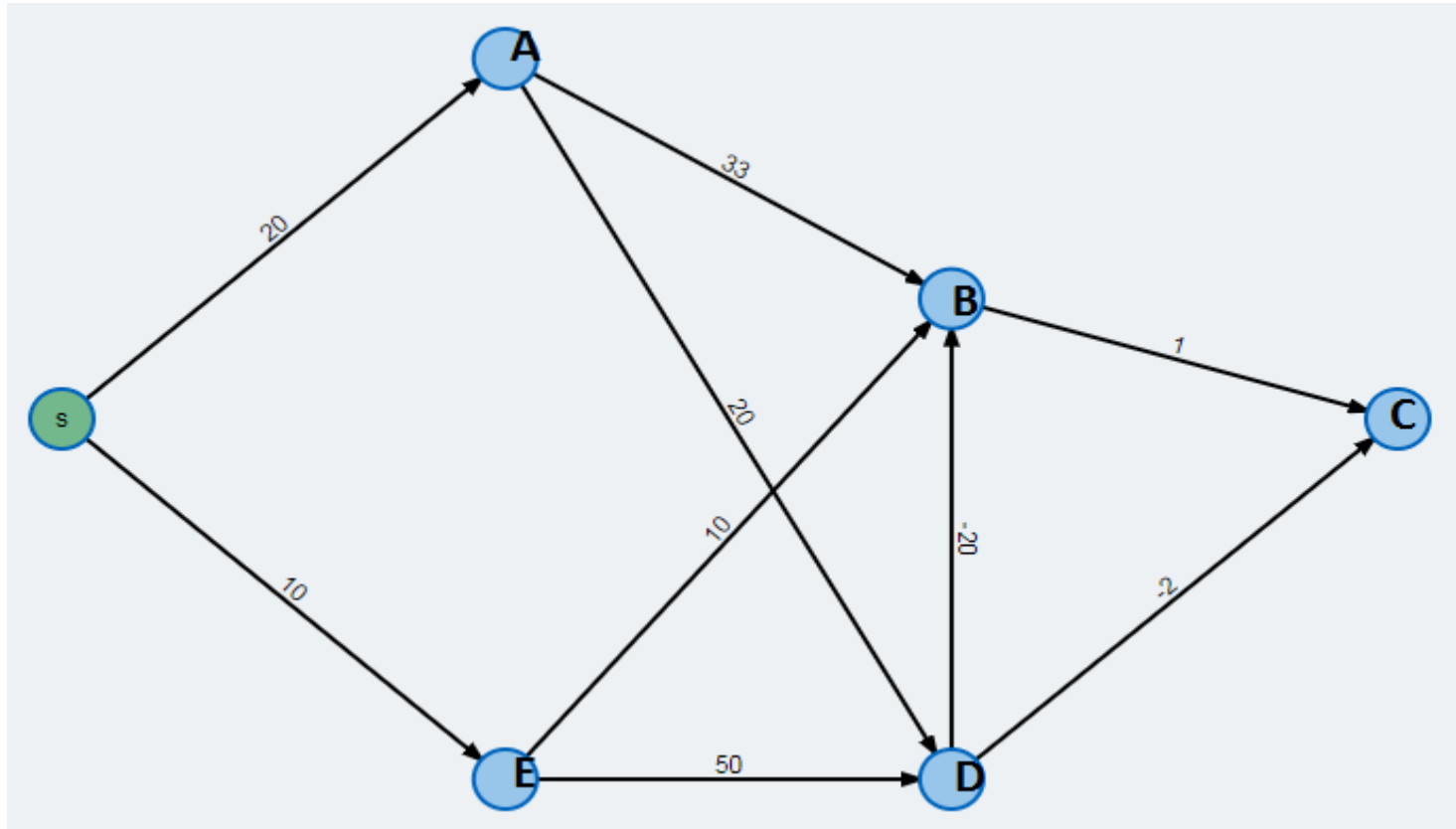
Bellman-Ford Exercise Table

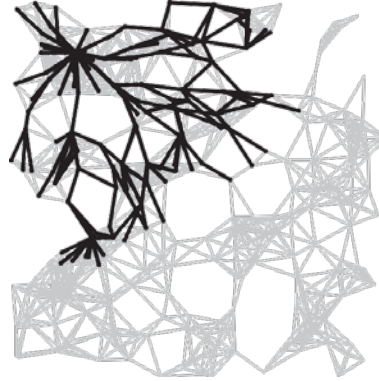| Iteration	Dist to S	A	B	C	D	E
0	0	inf	inf	inf	inf	Inf
1						
2						
3						
4						
5						

Keep track of distance and parent node eg 5 (via B) for each node for each iteration

Bellman-Ford algorithm: visualization
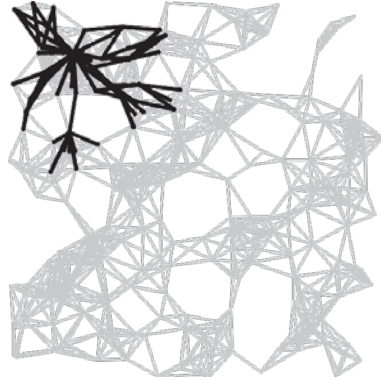
Bellman-Ford algorithm: analysis

Bellman-Ford algorithm

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat V times:

   – **Relax each edge.**

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After pass `i`, found path that is at least as short as any shortest path containing `i` (or fewer) edges.

Bellman-Ford algorithm: practical improvement

Observation. If `distTo[v]` does not change during pass `i`,
no need to relax any edge pointing from `v` in pass `i+1`.

FIFO implementation. Maintain queue of vertices whose `distTo[]` changed.

↑

be careful to keep at most one copy
of each vertex on queue (why?)

Overall effect.
- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Single source shortest path summary

| algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	$E + V$	$E + V$	V
Dijkstra (binary heap)	no negative weights	$E \log V$	$E \log V$	V
Bellman–Ford	no negative cycles	$E\,V$	$E\,V$	V
Bellman–Ford (queue–based)		$E + V$	$E\,V$	V

Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.
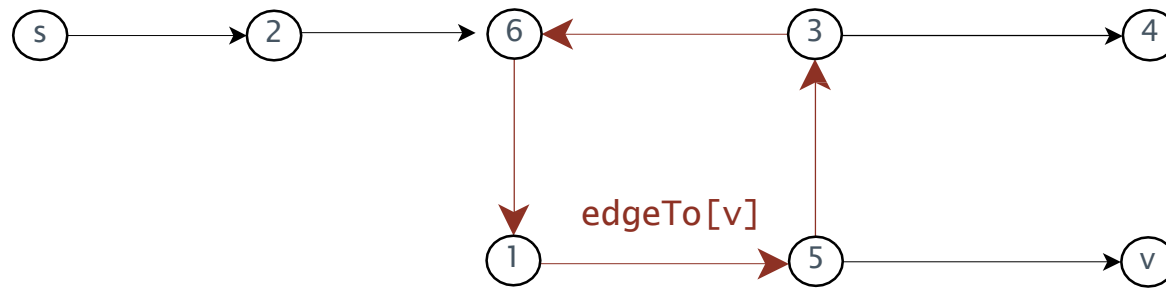
Proposition. If any vertex `v` is updated in pass `V`, there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

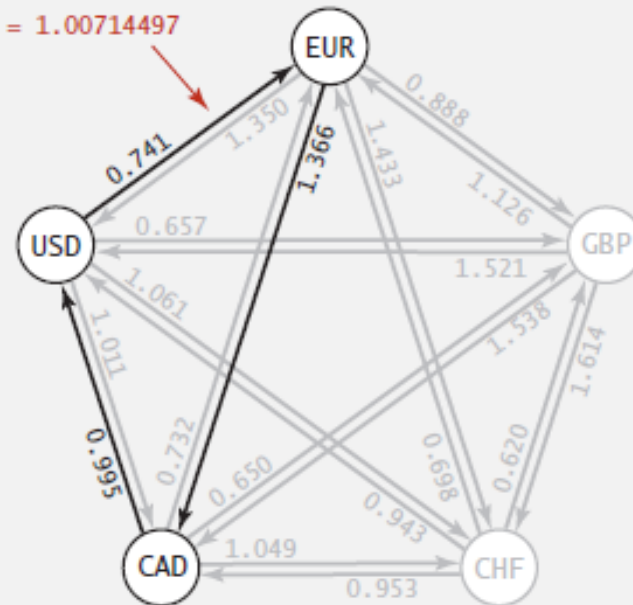Ex. $1,000 ⇒ 741 Euros ⇒ 1,012.206 Canadian dollars ⇒ $1,007.14497.

$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1.


0.741 * 1.366 * .995 = 1.00714497

Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \to w$ be $-ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0.
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).

$-ln(.741)$ $-ln(1.366)$ $-ln(.995)$

$.2998 - .3119 + .0050 = -.0071$

replace each weight w with $-ln(w)$
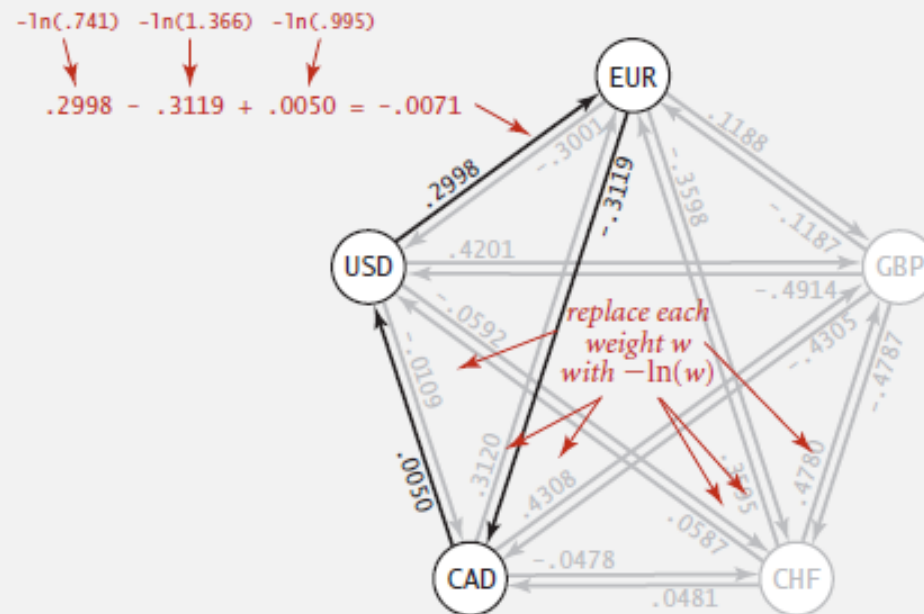
Remark. Fastest algorithm is extraordinarily valuable!

Floyd-Warshall shortest path algorithm

All-pairs shortest path

› Run Dijkstra for each vertex?

› Run Bellman-Ford each vertex?

› Worst case scenario – assume fully connected graph, so $E = V^2$

› Dijkstra = V times $V^2 \log V = V^3 \log V$

› Bellman-Ford = V times $V^3 = V^4$

› Other options?

› Floyd-Warshall = V^3 (but V^2 space)

Floyd-Warshall all-pairs shortest path

› For a path p = {v1, v2, …, vl}, vertices v2 to vl-1 are intermediate vertices

› Path consisting of a single edge has no intermediate vertices

› $d_{ij}^{(k)}$ is a shortest path from i to j such that any intermediate vertices on the path are chosen from the set {1, 2, …, k} – in any order, any subset of them
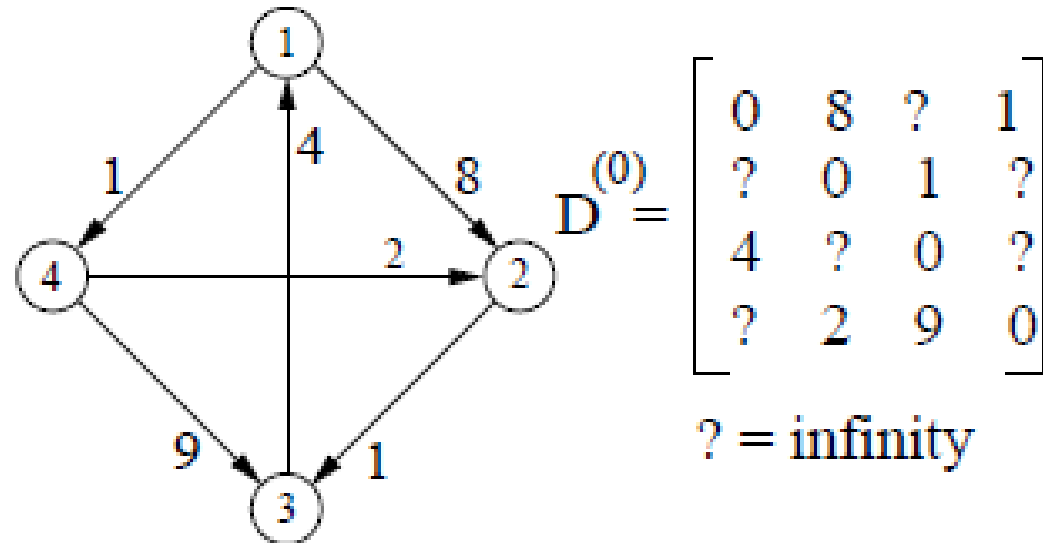
Floyd-Warshall

› Assume we know $d_{ij}^{(k-1)}$ – how do we calculate $d_{ij}^{(k)}$

› The path either goes through k, or it does not

  – If it does not, then shortest path $d_{ij}^{(k)} = d_{ij}^{(k-1)}$

  – If it does, it means it passes through it exactly once, and it consists of shortest path from i to k, and then the shortest path from k to j $d_{ij}^{(k)} = d_{ik}^{(k-1)-} + d_{kj}^{(k)-1}$

$$d_{ij}^{(0)} = w_{ij},$$

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \qquad \text{for } k \geq 1.$$
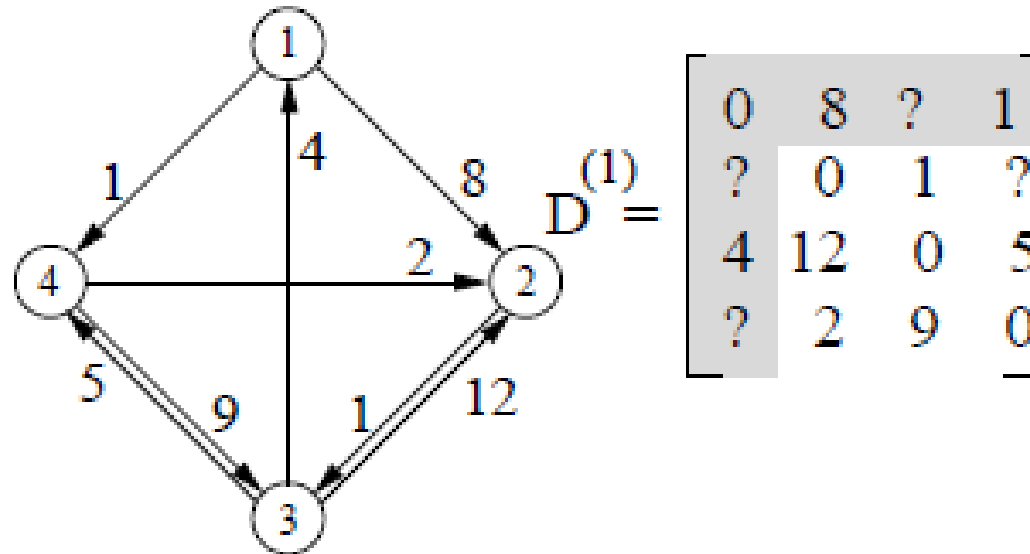
Floyd-Warshall pseudocode

for $k = 1$ to n do

    for $i = 1$ to n do

        for $j = 1$ to n do

            if $(d[i, k] + d[k, j] < d[i, j])$

                $\{d[i, j] = d[i, k] + d[k, j];$

                  $pred[i, j] = k;\}$

return $d[1..n, 1..n];$

Floyd Warshall example

$$D^{(0)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix}$$

? = infinity

Floyd Warshall example

$$D^{(1)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 9 & 0 \end{bmatrix}$$

Floyd Warshall example

$$D^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 3 & 0 \end{bmatrix}$$

Floyd Warshall example

$$D^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Floyd Warshall example

$$D^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Floyd-Warshall demo

› https://www.cs.usfca.edu/~galles/visualization/Floyd.html
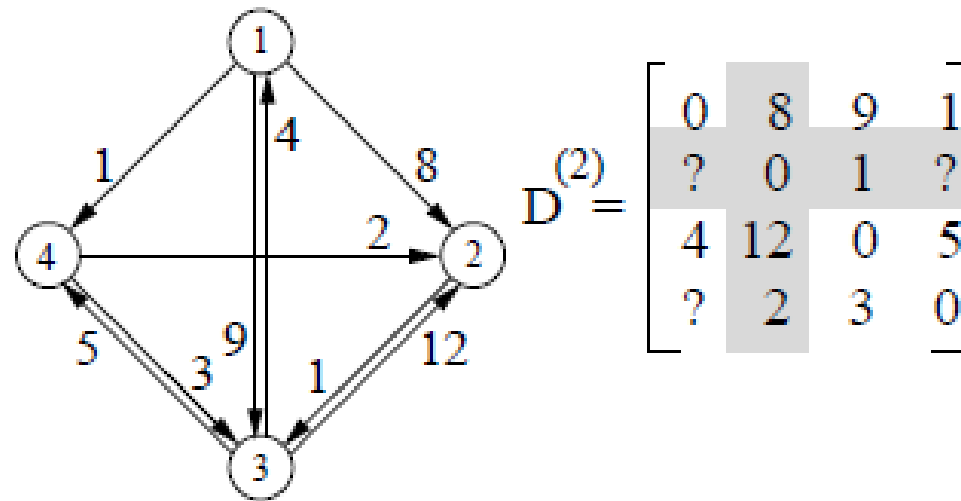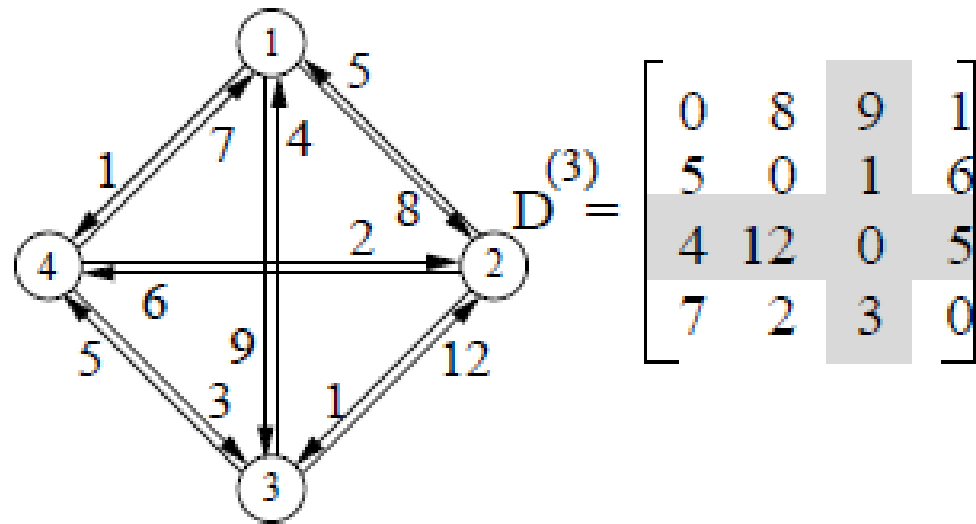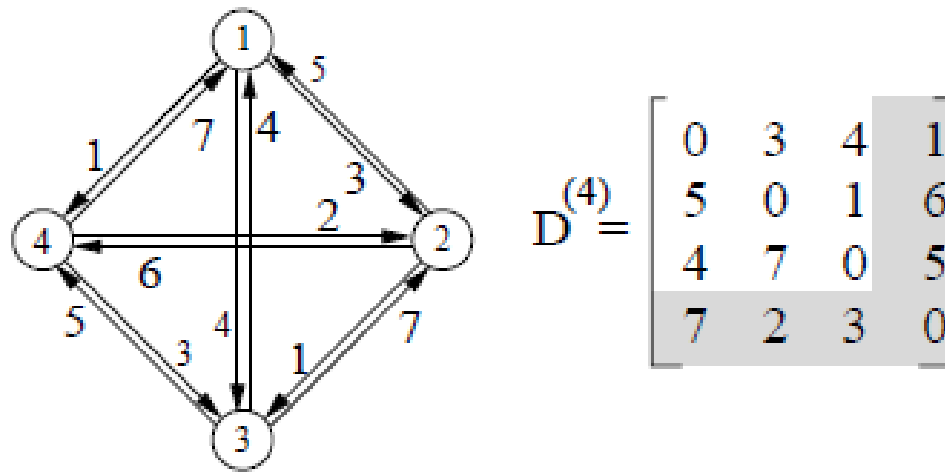
Floyd-Warshall exercise
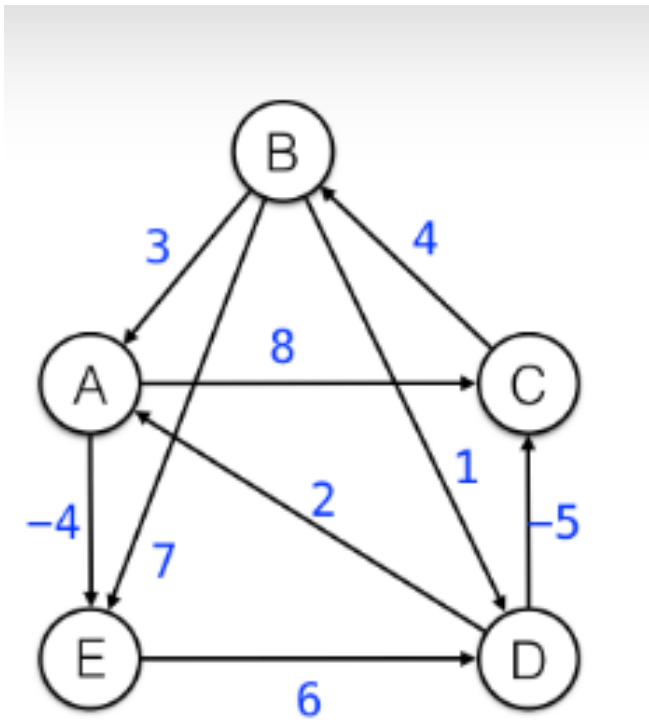
D⁰	A	B	C	D	E
A	0	3	8	INF	-4
B	INF	0	INF	1	7
C	INF	4	0	INF	INF
D	2	INF	-5	0	INF
E	INF	INF	INF	6	0

Exercise solution
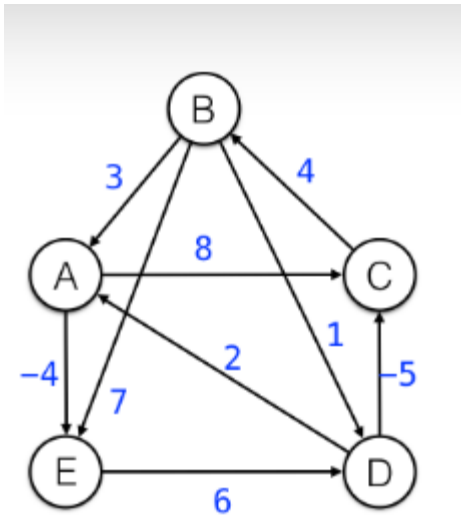
D^5	A	B	C	D	E
A	0	0	1	-3	**-4**
B	3	0	-4	1	**-1**
C	7	4	0	5	**3**
D	2	-1	-5	0	**-2**
E	**8**	**5**	**1**	**6**	**0**

Dynamic Programming

Dynamic programming

> Algorithm Design, Kleinberg and Tardos, Pearson 2014

> Introduction to Design and Analysis of Algorithms, Levitin. Pearson 2012

Dynamic programming

› Examples:
  – Bellman-Ford
  – Floyd-Warshall

› Examines the full search space but implicitly, by breaking up the problem into a series of subproblems, and then building up the solution to larger and larger subproblems

› Typically overlapping subproblems – instead of over and over calculating solutions to a subproblem, record it in a table and look up when needed

› So why a fancy name if this is all that it's doing?

› Richard Bellman on the Birth of Dynamic Programming, by Stuart Dreyfus

https://pubsonline.informs.org/doi/pdf/10.1287/opre.50.1.48.17791

Dynamic programming

› Informal guidelines:
– There are only a polynomial number of subproblems
– The solution to the original problem can be easily computed from the solution of the subproblems
– There is a natural order of subproblems from smallest to largest together with easy to compute recurrence that allows building a solution to a subproblem from smaller subproblems

Shortest paths challenges

› http://www.diag.uniroma1.it/challenge9/

› http://www.diag.uniroma1.it/challenge9/download.shtml