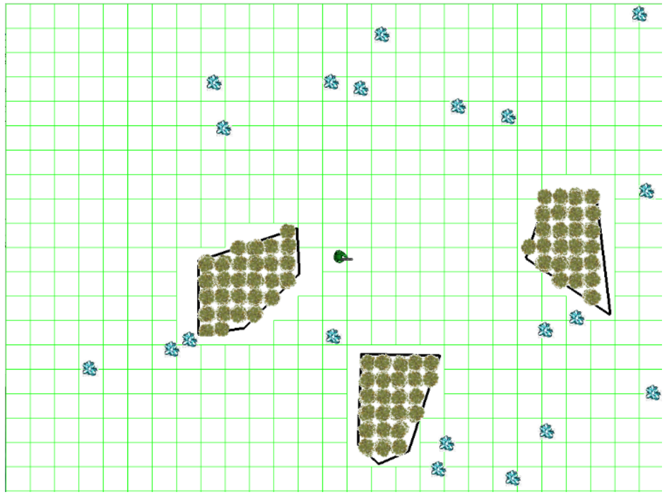
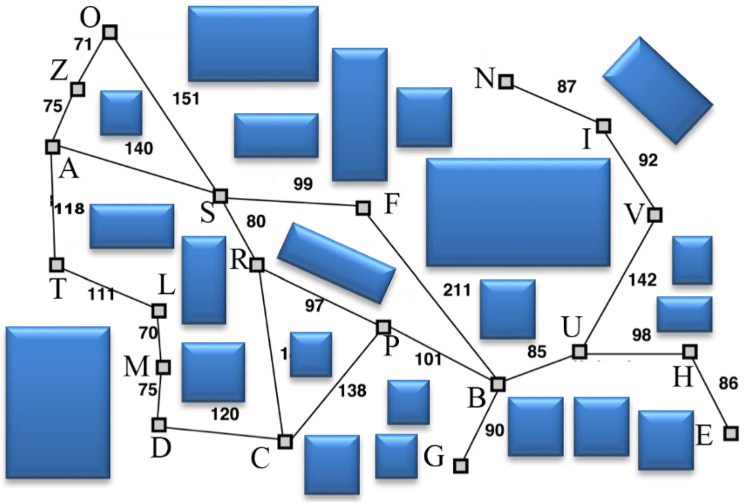
Announcements

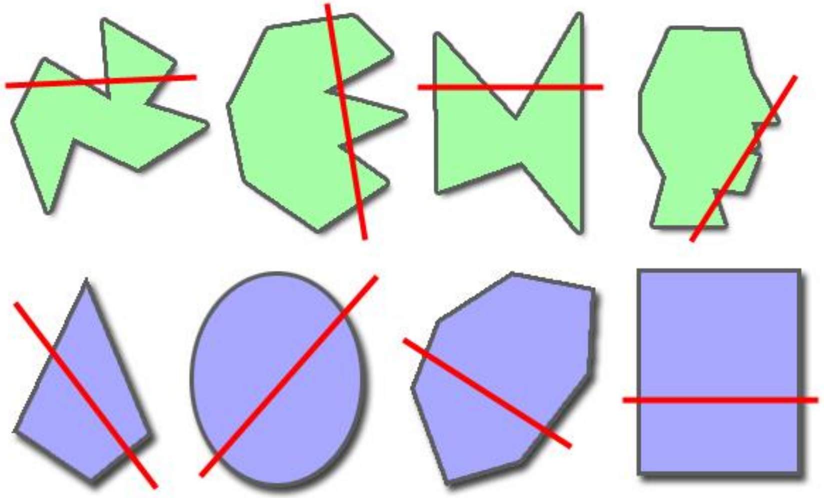
- HW1 grades, distribution in flux
- HW2 (path network) due Sunday night, January 28 @ 11:55pm
- HW3 much more difficult. Start ASAP
- Verification of course participation – see piazza, and complete before 10am Friday, January 26
  - First poll: 149/179 (30 noshows). 10 to 15 minutes, or 23% said ‘any’
  - 2nd poll: 82 votes so far
- Webpage!
  - https://www.cc.gatech.edu/~surban6/2018sp-gameAI/

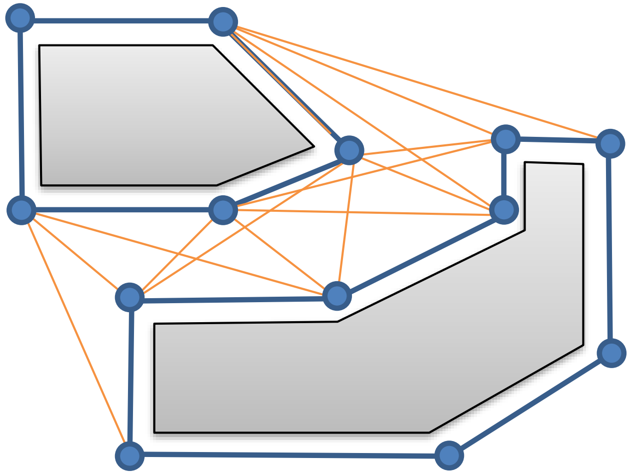
PREVIOUSLY ON…

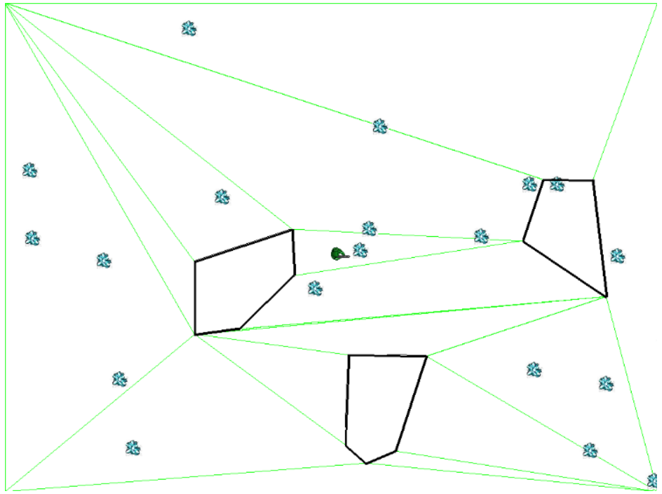
Modelling and Navigating the Game World

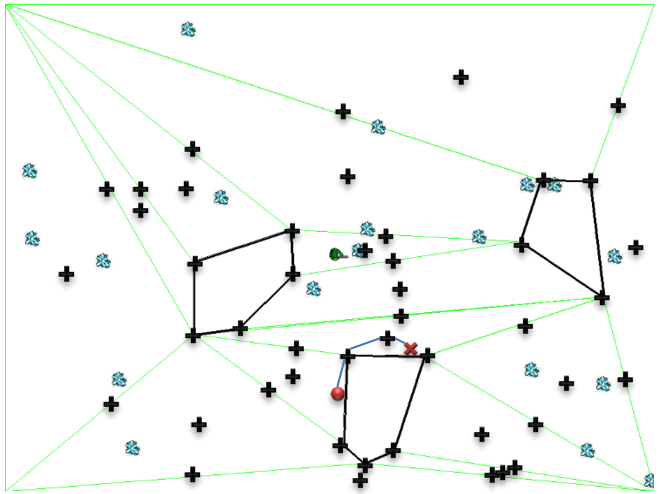




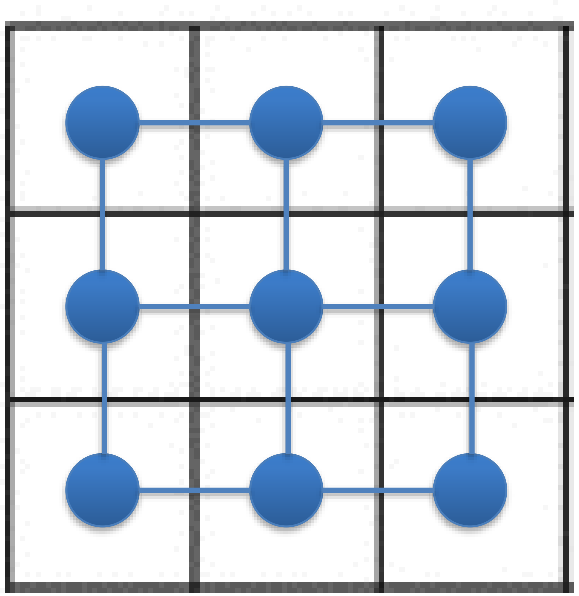


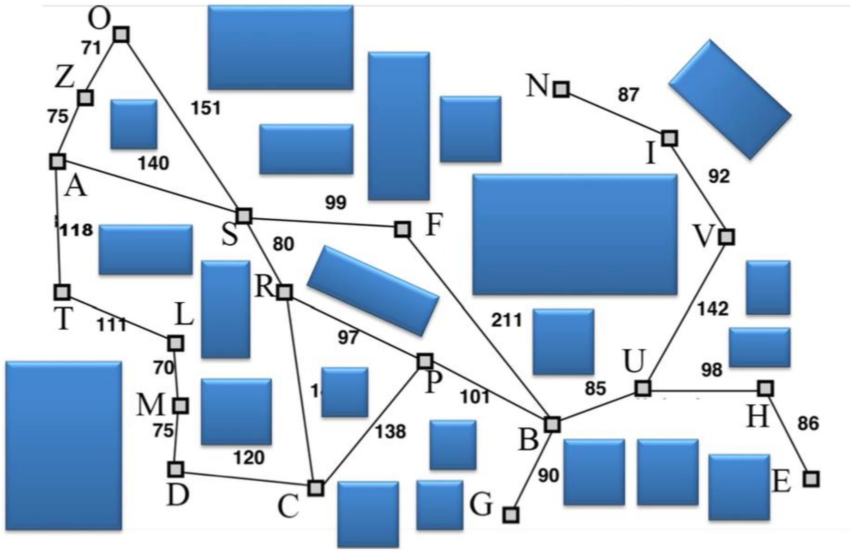


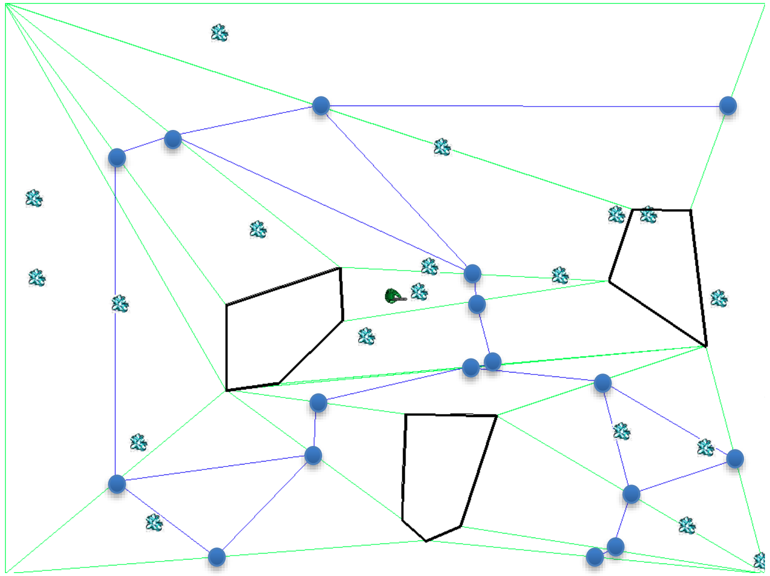




Graphs, Graphs, Graphs…

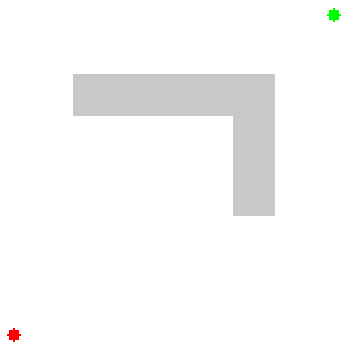






Graph Search: Sorting Successors

- Uninformed (all nodes are same)
  - Greedy
  - DFS (stack – lifo), BFS (queue – fifo)
  - Iterative-deepening (Depth-limited)
- Informed (pick order of node expansion)
  - Dijkstra – guarantee shortest path ($E\log_2 N$)
  - Floyd-Warshall
  - A* (IDA*)…. Dijkstra + heuristic
  - D*
- Hierarchical can help



http://en.wikipedia.org/wiki/A*_search_algorithm

N-1

1. What kind of solution does greedy search find? Why might this be useful?
2. What kind of solution does A* find?
3. What are some of the insights behind A*?
4. What’s a good data structure to use with A*? Why?

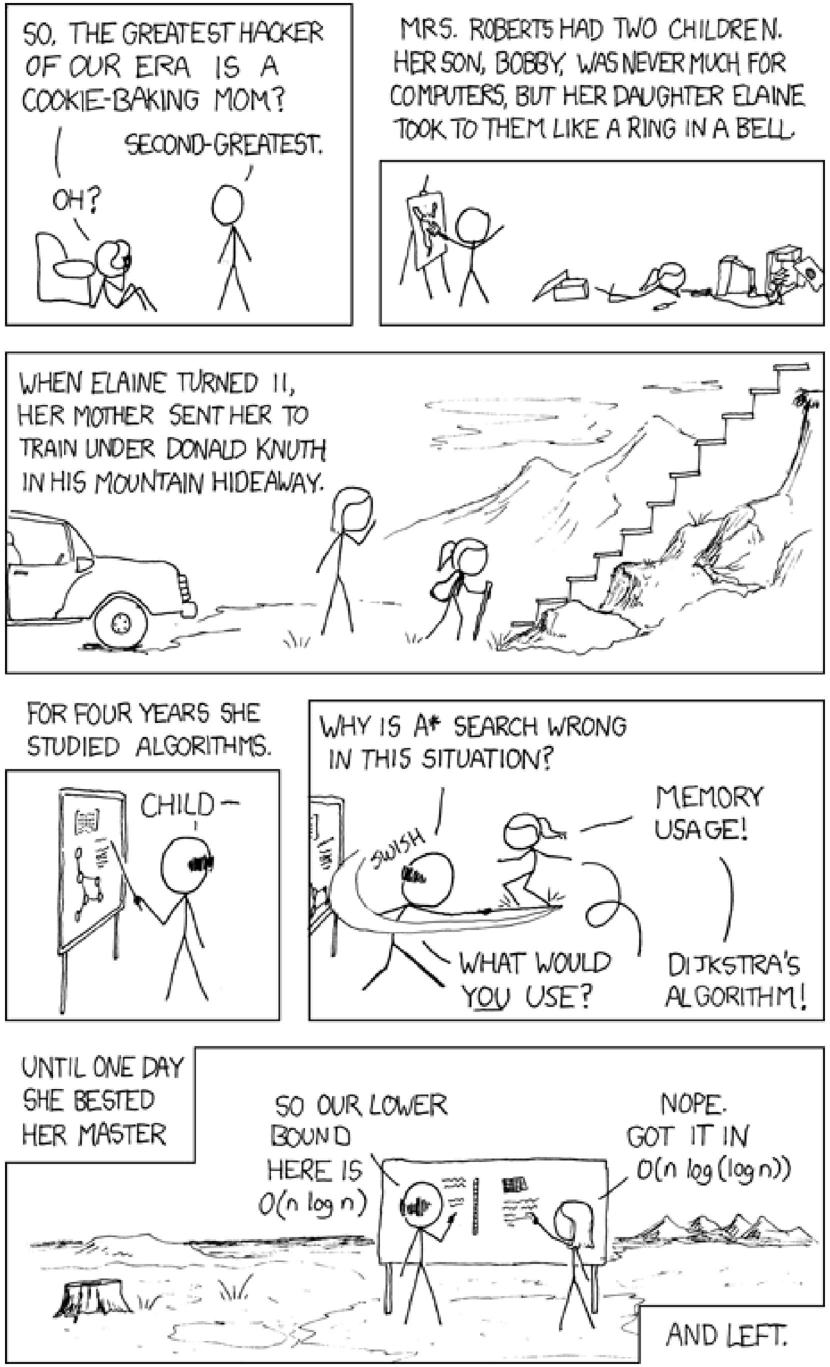
Crawl down the Wikipedia rabbit hole rather than books for this one

AI Game Programming wisdom 2, CH 2

Buckland CH 8

Millington CH 4

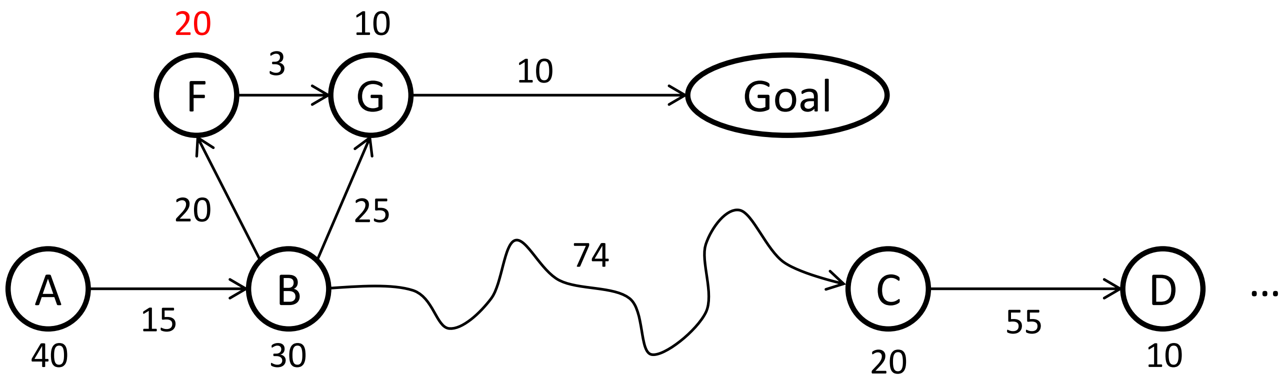
SEARCH CONTINUED



https://www.xkcd.com/342/

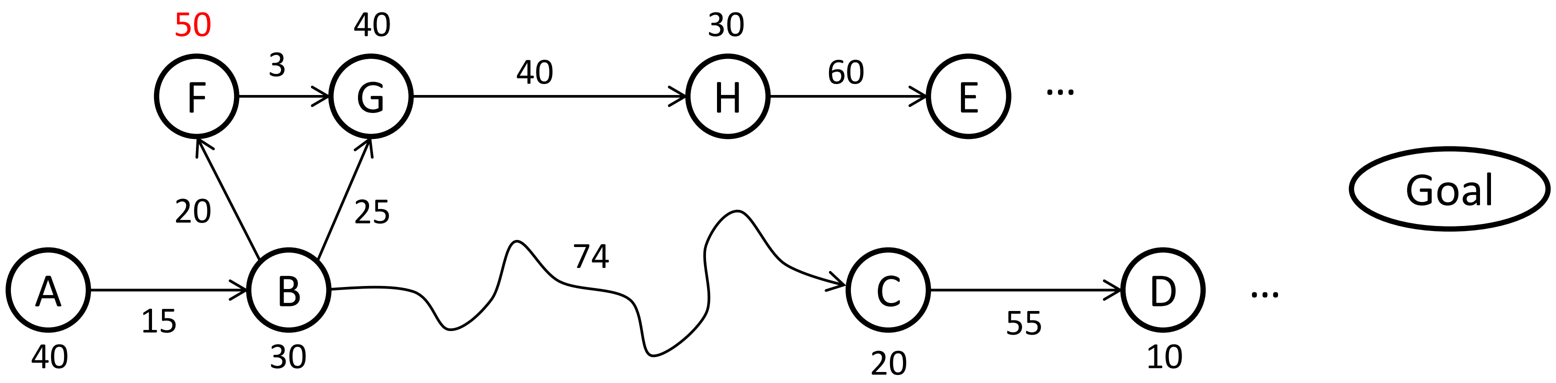
Non-Admissible Heuristics

- What happens if you have a non-admissible heuristic?



Non-Admissible Heuristics

- What happens if you have a non-admissible heuristic?
- You get “short-cuts”: find a path to a node on the closed list that is shorter than before
- Shortcuts happen because h(n) overestimates



Non-admissible heuristics

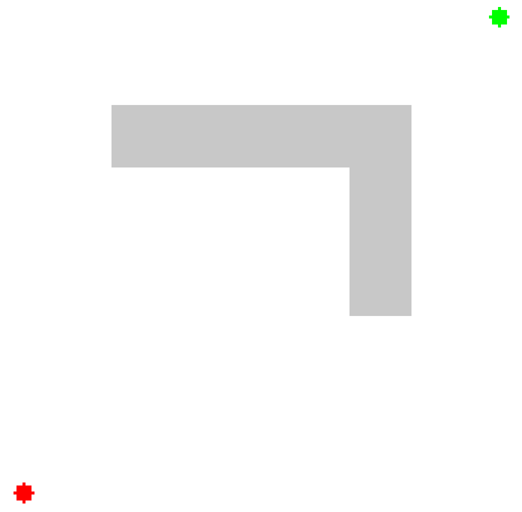
- Discourage agent from being in particular states
- Encourage agent to be in particular states

Dijkstra's algorithm

- 1956: A single-source, multi-target shortest path algorithm
- Tells you path from any one node to all other nodes
- Special case of A*, where the heuristic is always zero.
- Time complexity for single vertex: O(E log V)
  - Run for each vertex: O(VE Log V) which can go (V^3 Log V) in worst case
- "This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights." (source: Wikipedia)

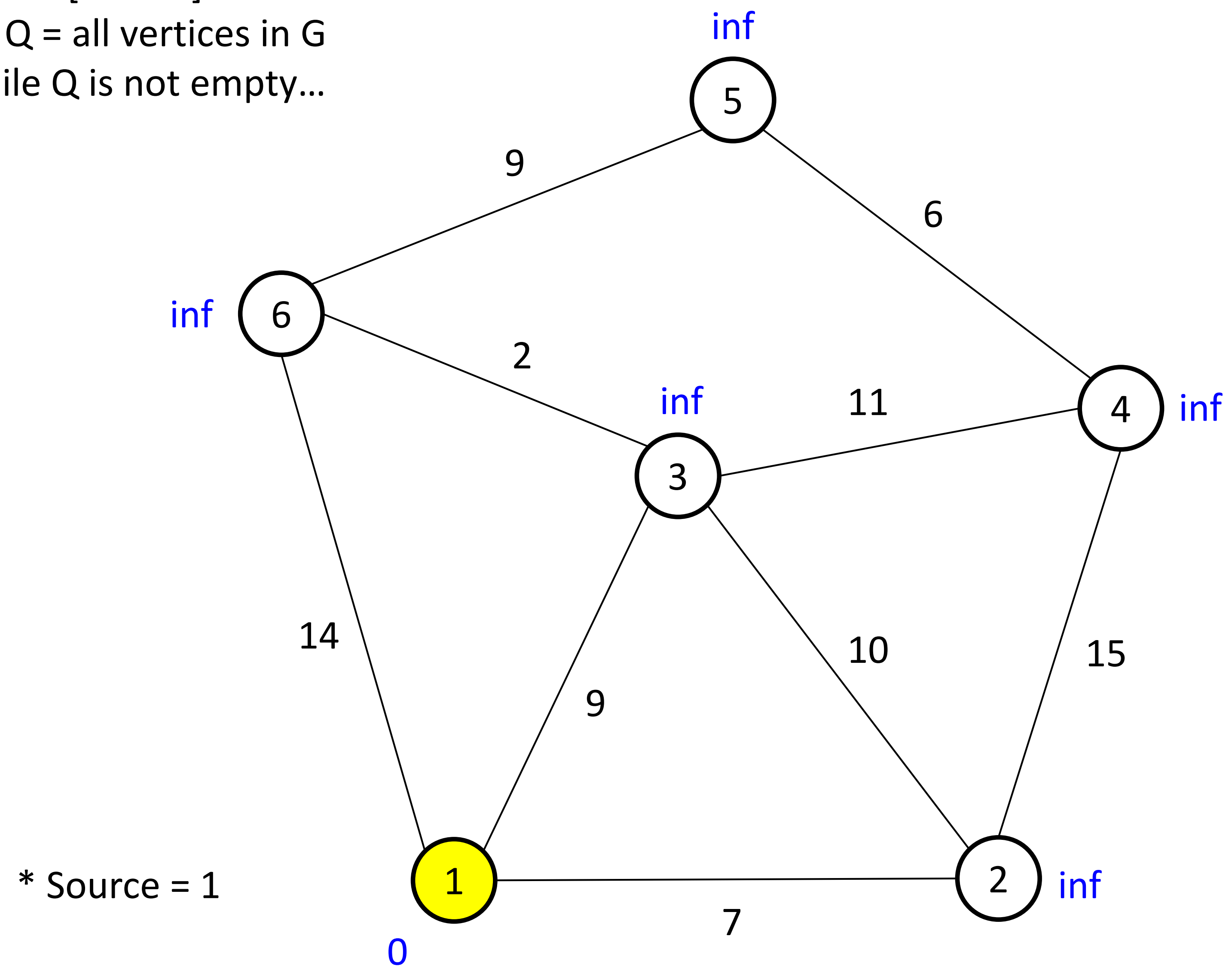
```
Given: G=(V,E), source

For each vertex v in G, set dist[v] to infinity
Set dist[source] = 0
Let Q = all vertices in G
While Q is not empty:
        Let u = get vertex in Q with smallest distance value
        Remove u from Q
        For each neighbor v of u:
                d = dist[u] + distance(u, v)
                if d < dist[v] then:
                        dist[v] = d
                        parent[v] = u
Return dist[]
```



Source: wikipedia

For each vertex v in G, set dist[v] to infinity
Set dist[source] = 0
Let Q = all vertices in G
While Q is not empty…



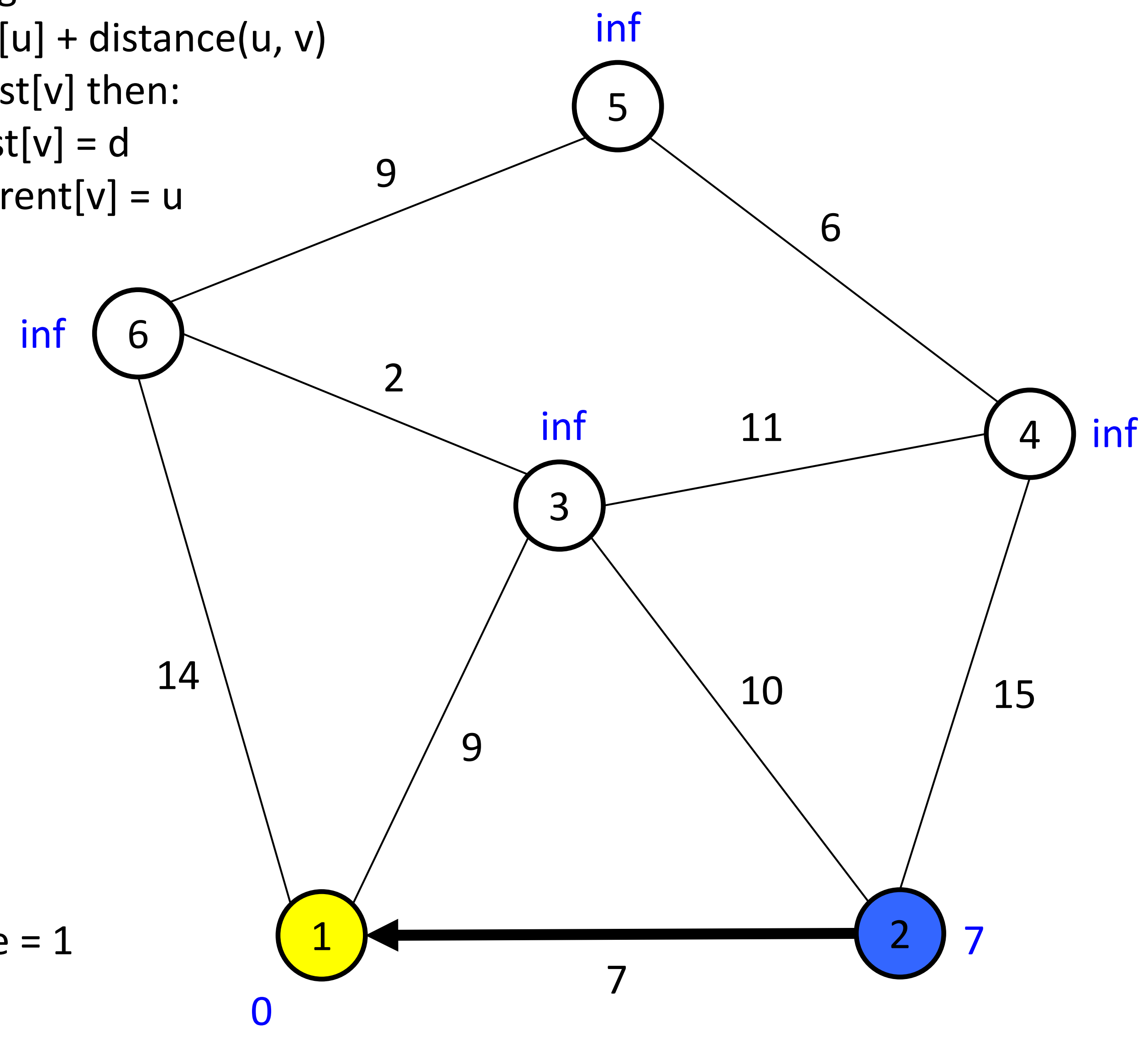
Dest	Cost	Parent
1	0	
2	Inf	
3	Inf	
4	Inf	
5	Inf	
6	Inf	

Q=[1,2,3,4,5,6]
U=1

Let u = get vertex in Q with smallest distance value (node 1)

dist[v]

Remove u (node 1) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



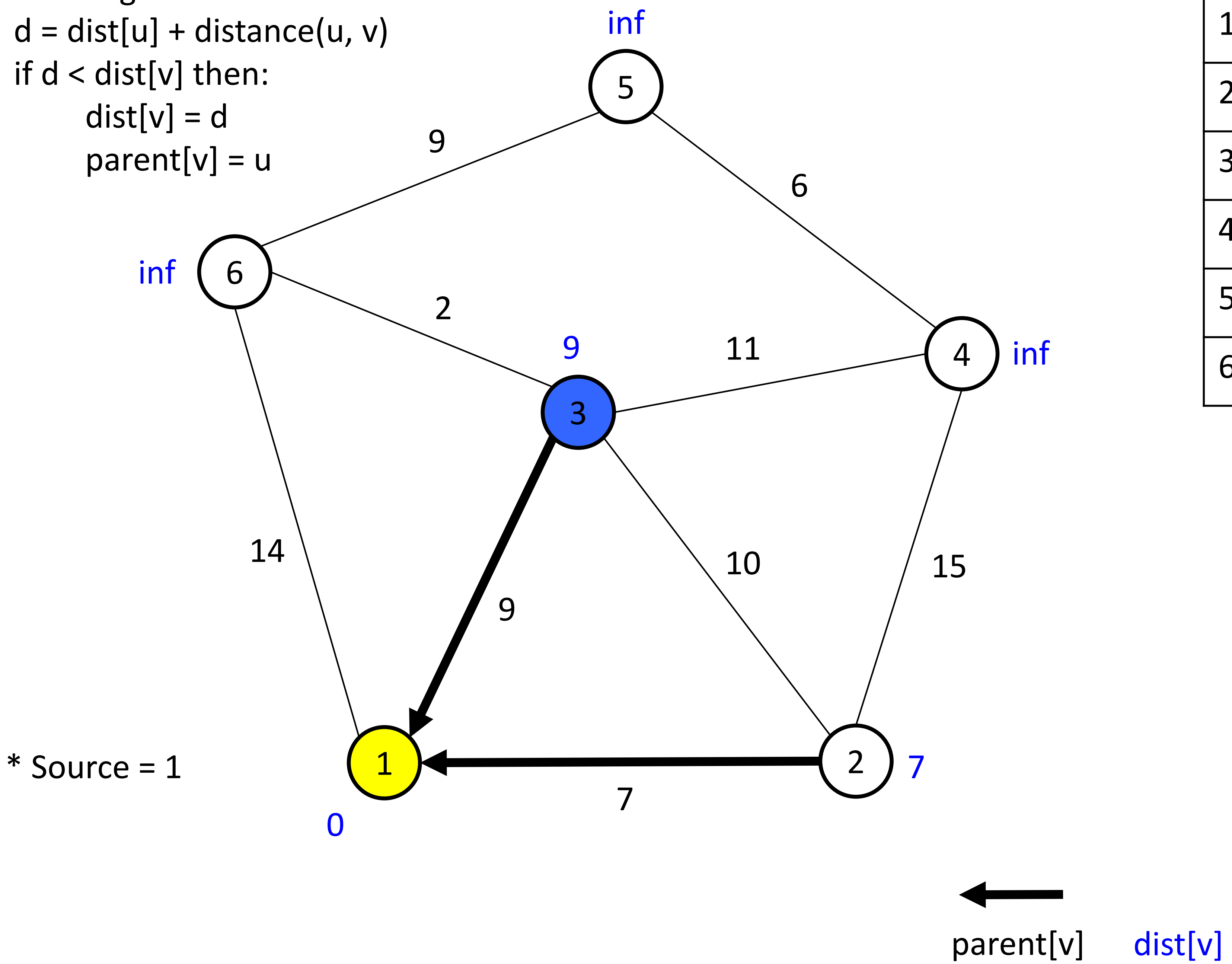
* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	Inf	
4	Inf	
5	Inf	
6	Inf	

Q=[~~1,~~2,3,4,5,6]
U=1
V=2

parent[v]   dist[v]

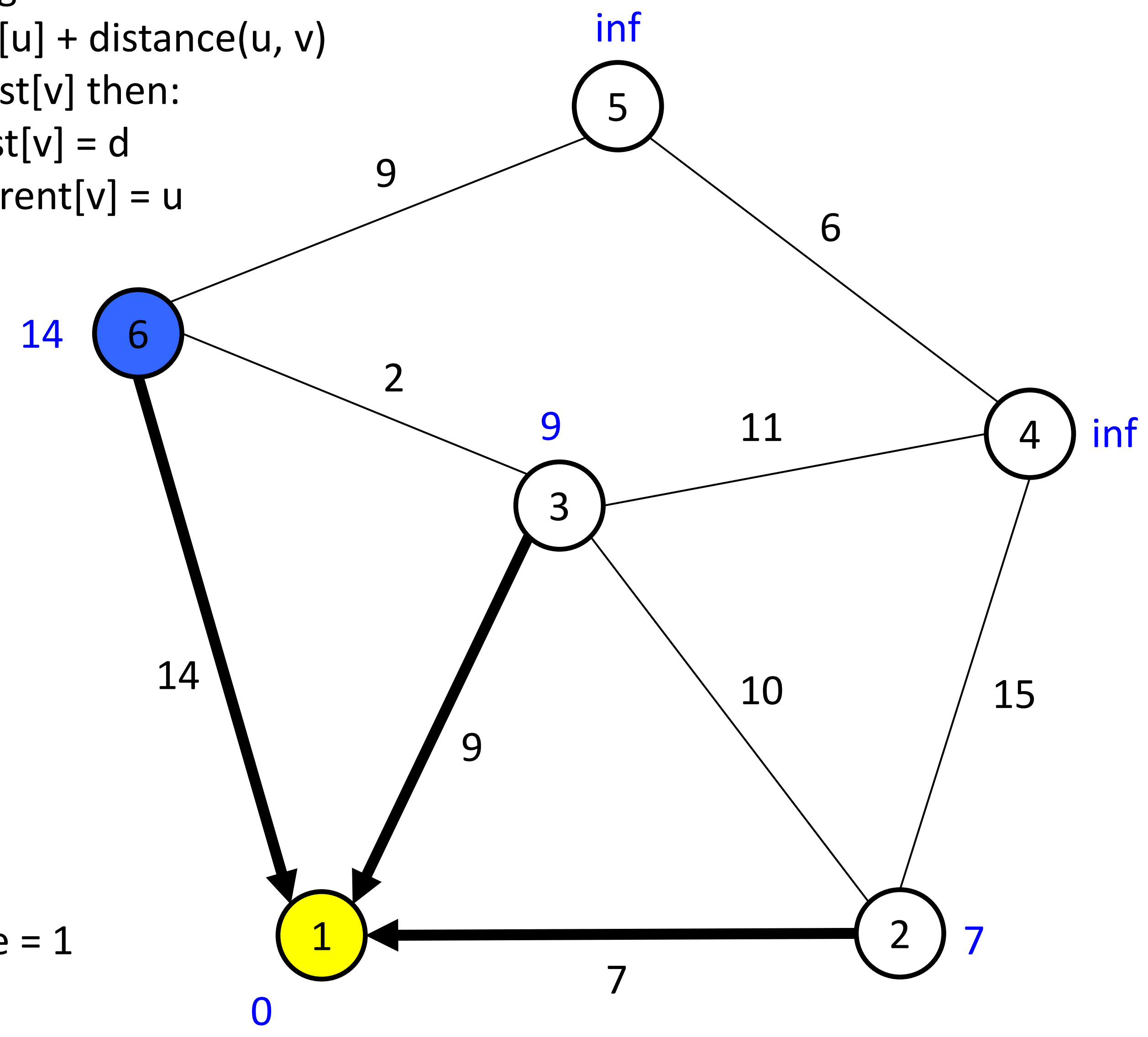
Remove u (node 1) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	Inf	
5	Inf	
6	Inf	

Q=[~~1,~~2,3,4,5,6]
U=1
V=3

```
Remove u (node 1) from Q
For each neighbor v of u:
        d = dist[u] + distance(u, v)
        if d < dist[v] then:
                dist[v] = d
                parent[v] = u
```

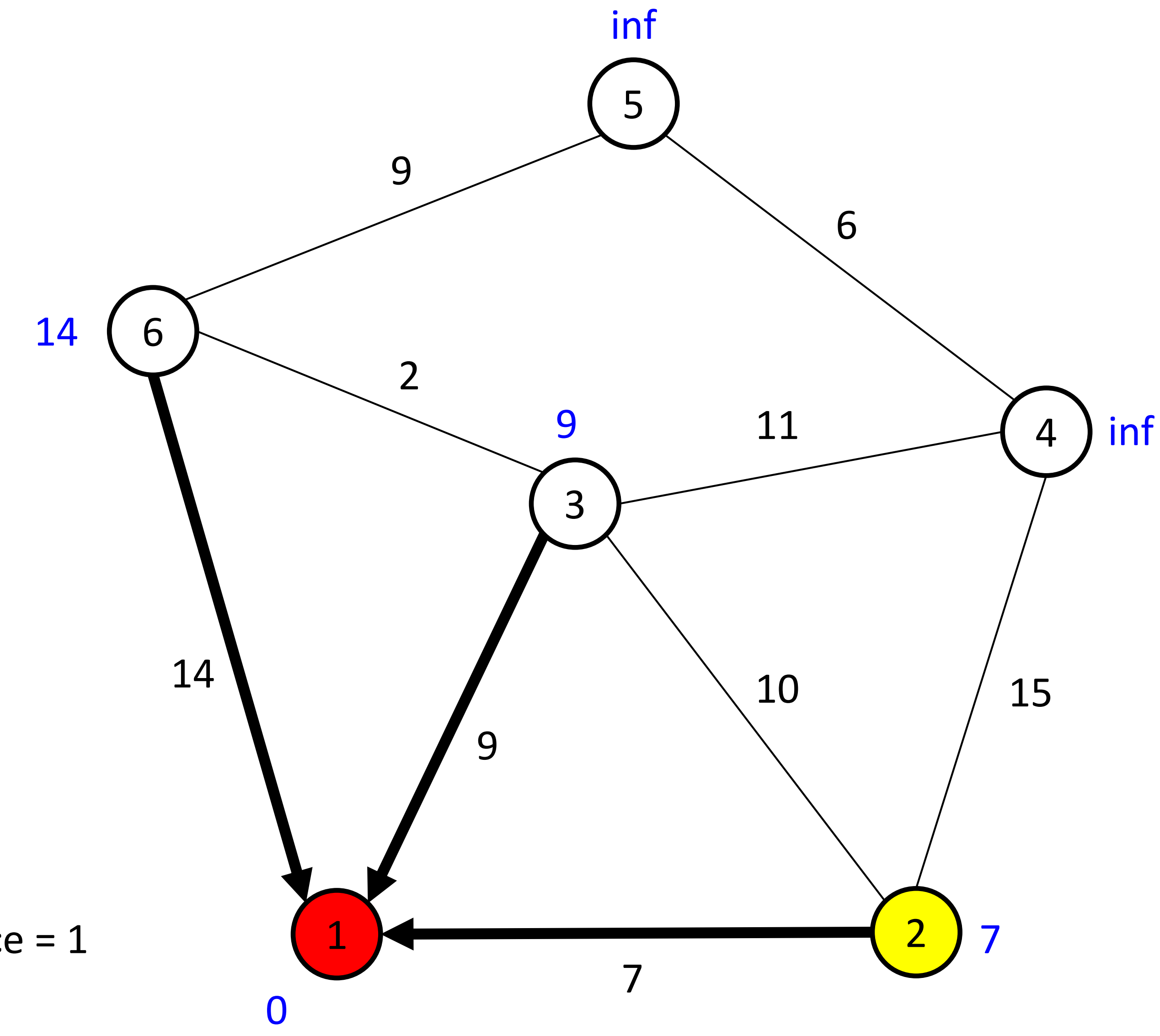


* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	Inf	
5	Inf	
6	14	1

Q=[~~1,~~2,3,4,5,6]
U=1
V=6

parent[v] dist[v]



* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	Inf	
5	Inf	
6	14	1

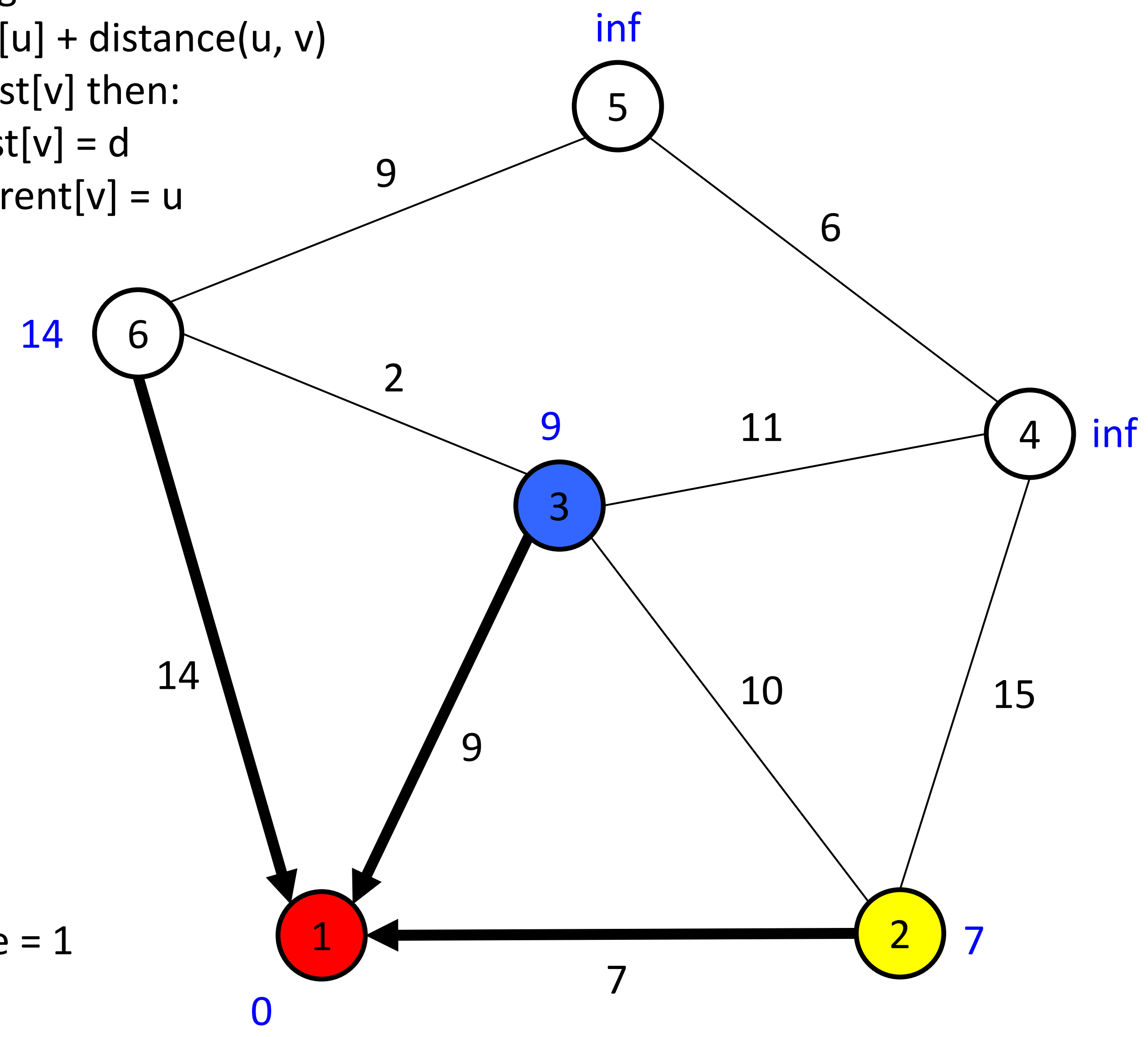
Q=[~~2,~~3,4,5,6]

U=2

V=

Let u = get vertex in Q with smallest distance value (node 2)

Remove u (node 2) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



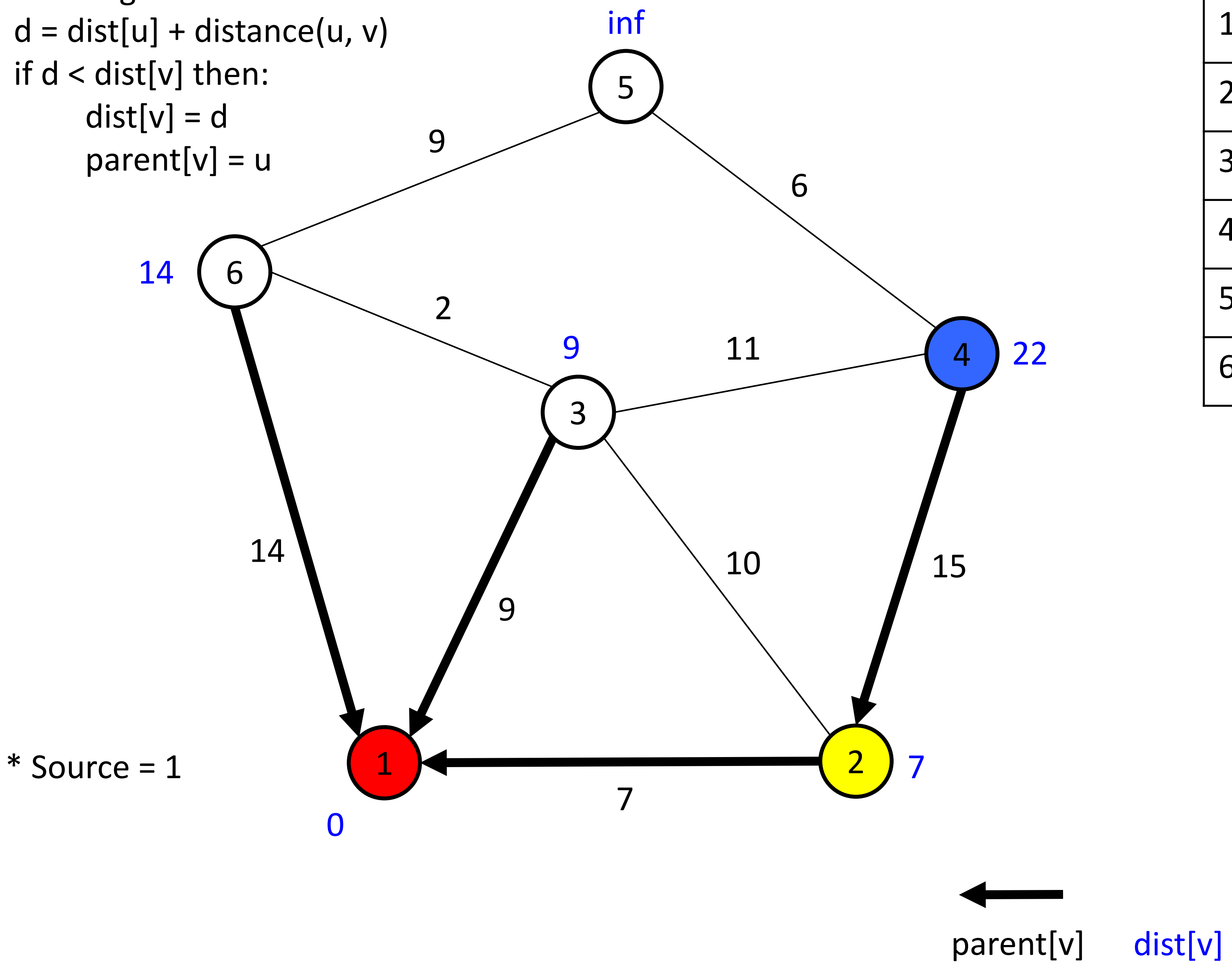
* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	Inf	
5	Inf	
6	14	1

Q=[~~2,~~3,4,5,6]
U=2
V=3

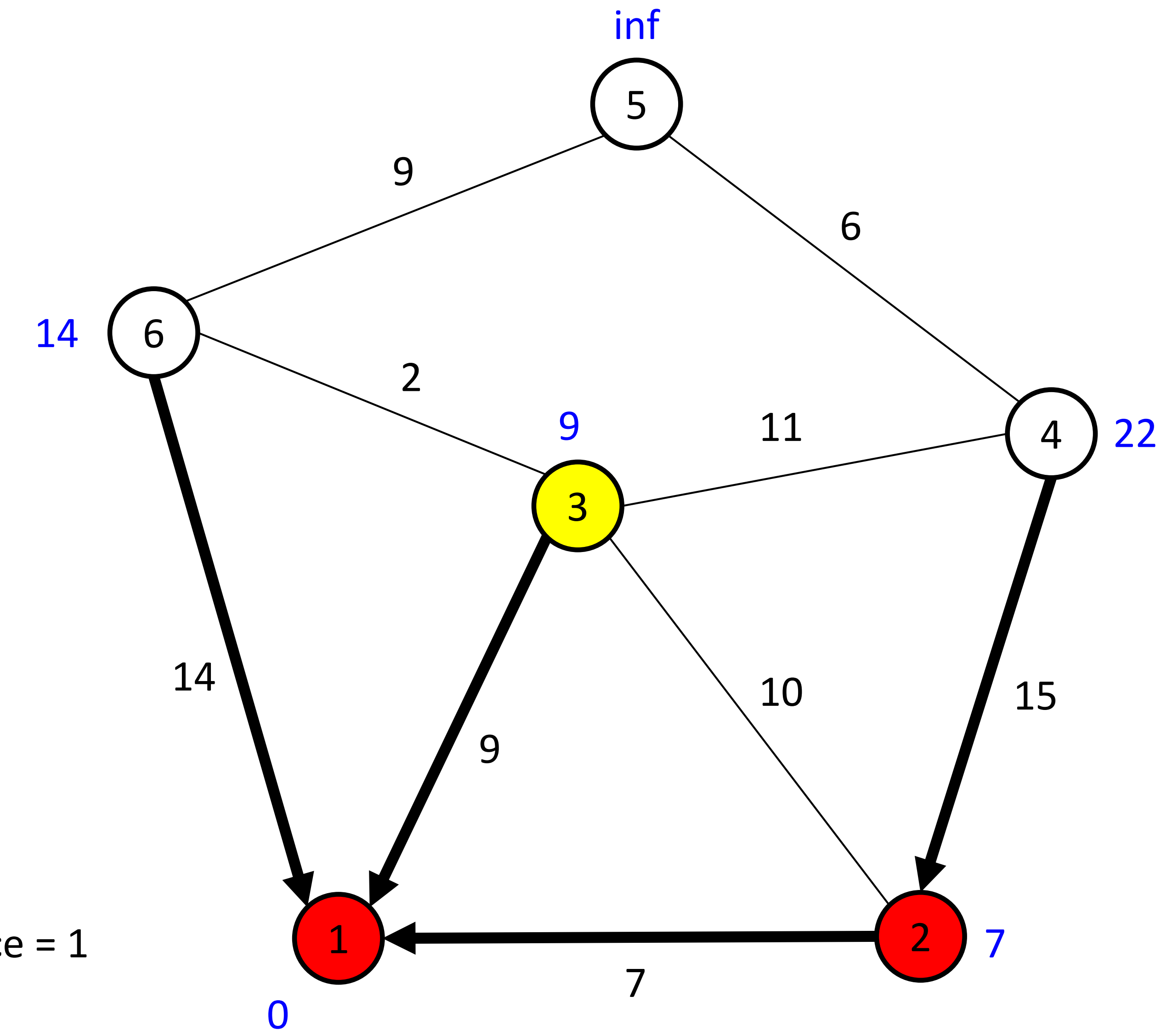
parent[v] dist[v]

Remove u (node 2) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	22	2
5	Inf	
6	14	1

Q=[~~2,~~3,4,5,6]
U=2
V=4



* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	22	2
5	Inf	
6	14	1

Q=[~~3,~~4,5,6]

U=3

V=

Let u = get vertex in Q with smallest distance value (node 3)

Remove u (node 3) from Q

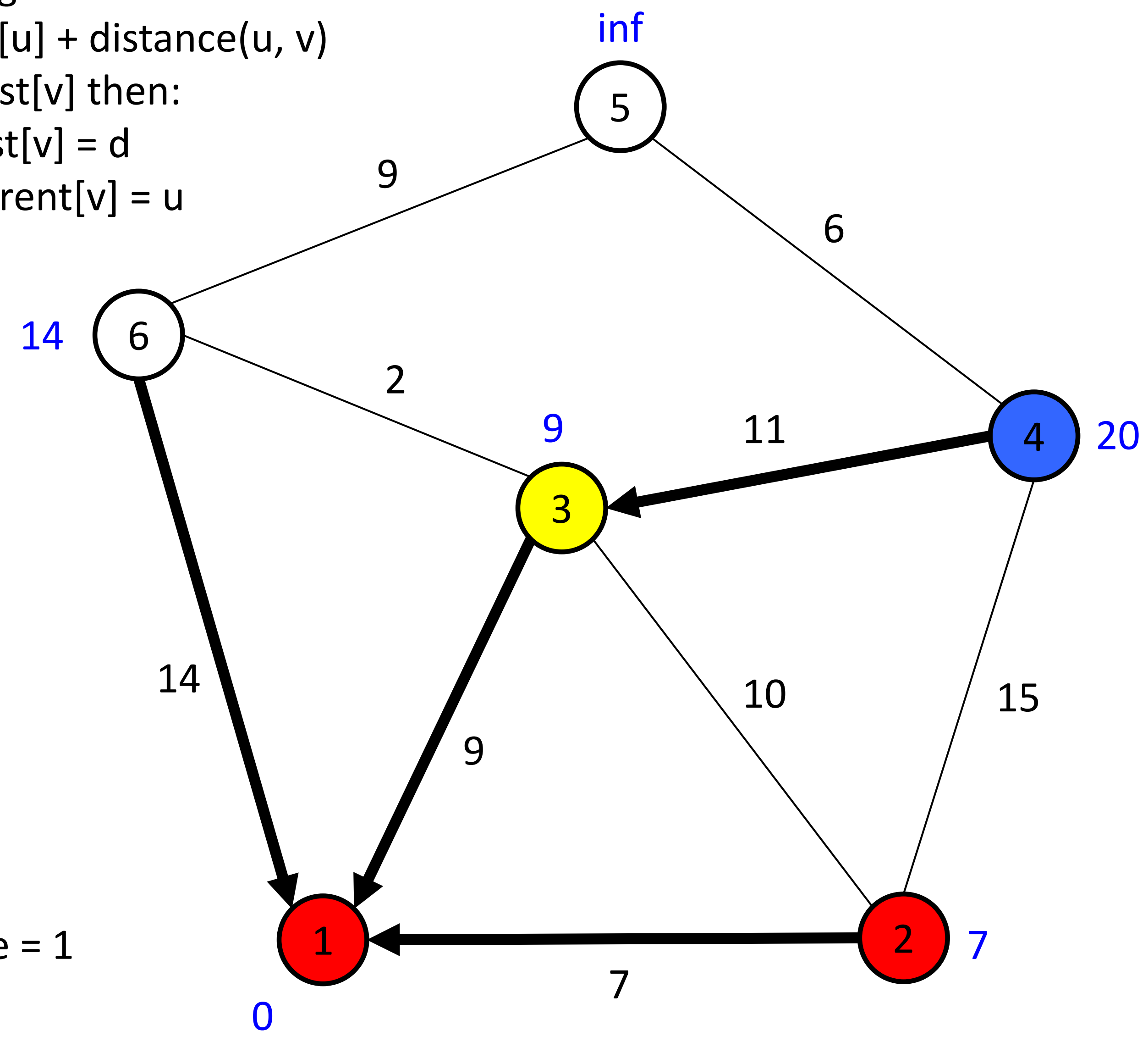
For each neighbor v of u:

    d = dist[u] + distance(u, v)

    if d < dist[v] then:

        dist[v] = d

        parent[v] = u



* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	Inf	
6	14	1

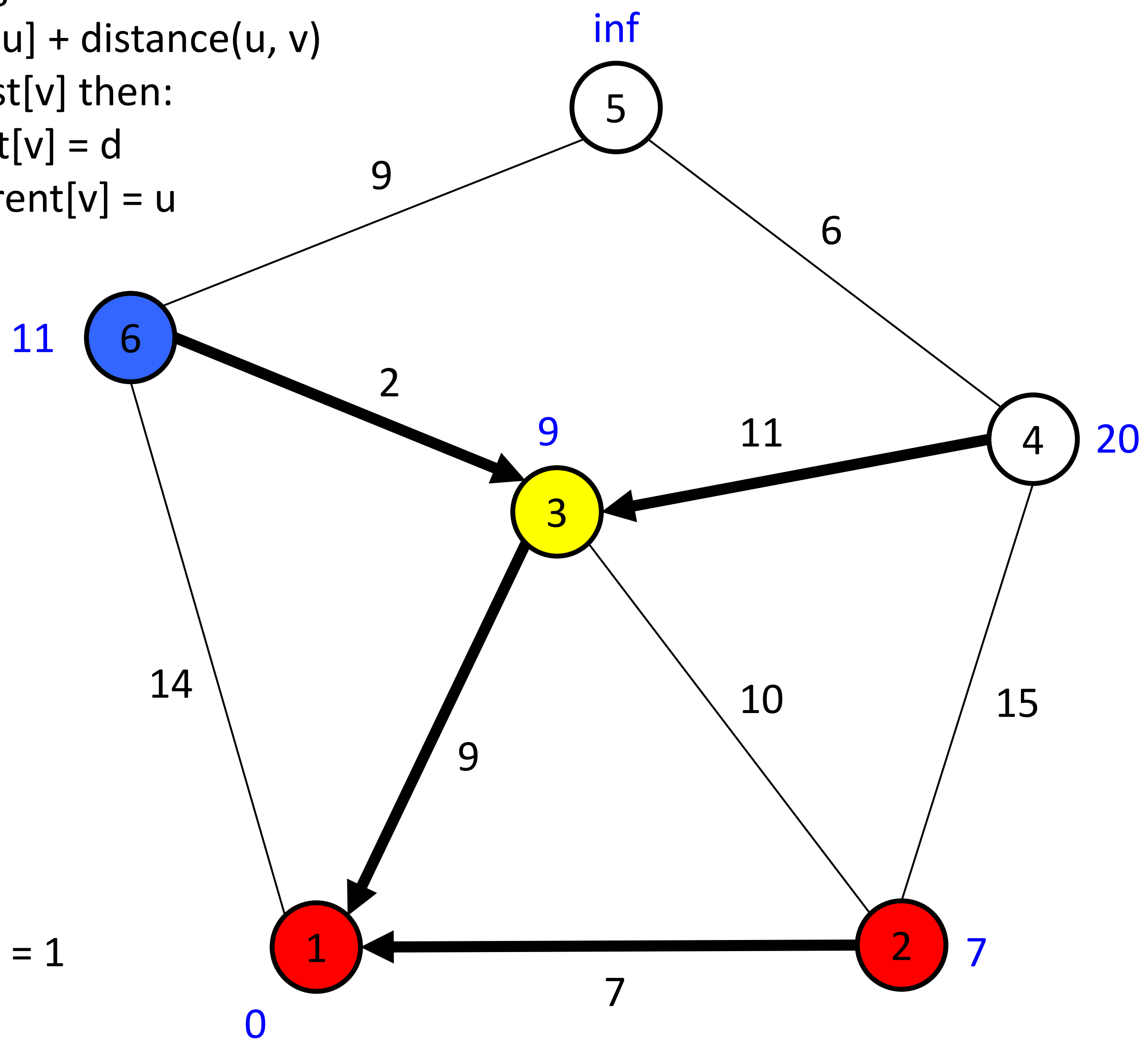
Q=[~~3,~~4,5,6]

U=3

V=4

parent[v]   dist[v]

Remove u (node 3) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u

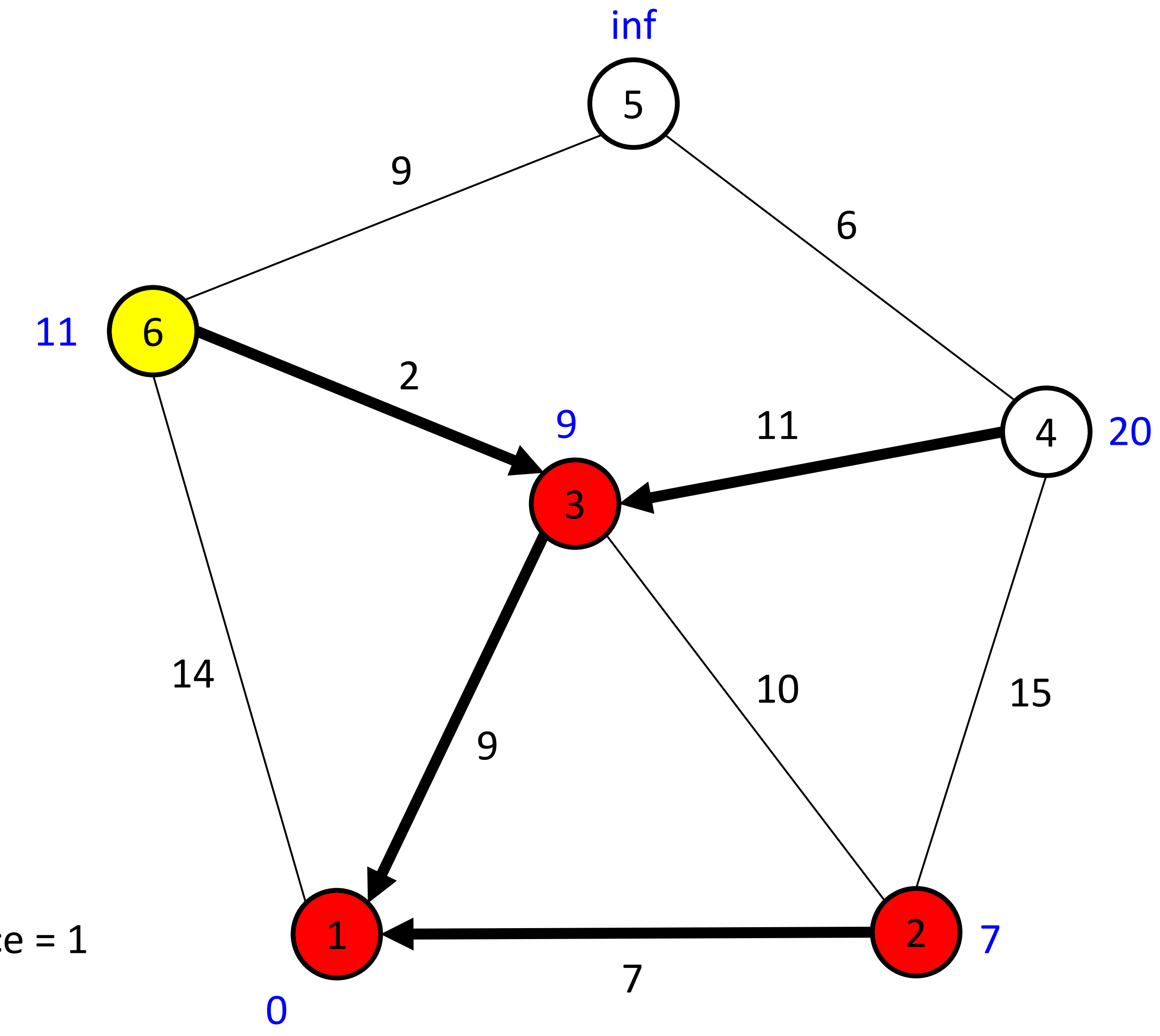


* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	Inf	
6	11	3

Q=[~~3,~~4,5,6]
U=3
V=6

parent[v] dist[v]



Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	Inf	
6	11	3

Q=[4,5,~~6~~]
U=6
V=

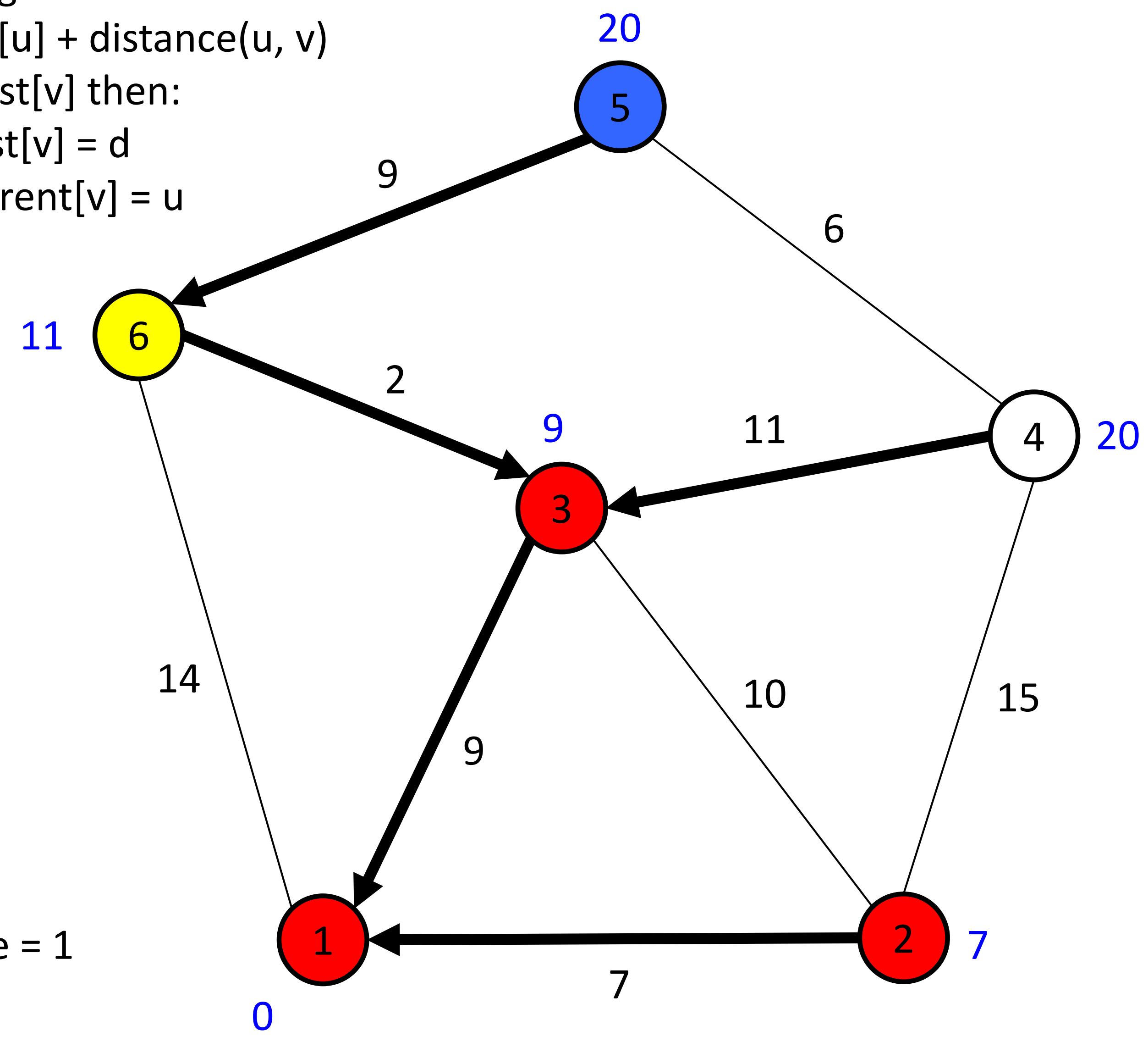
* Source = 1

Let u = get vertex in Q with smallest distance value (node 6)

Remove u (node 6) from Q

For each neighbor v of u:

    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



* Source = 1

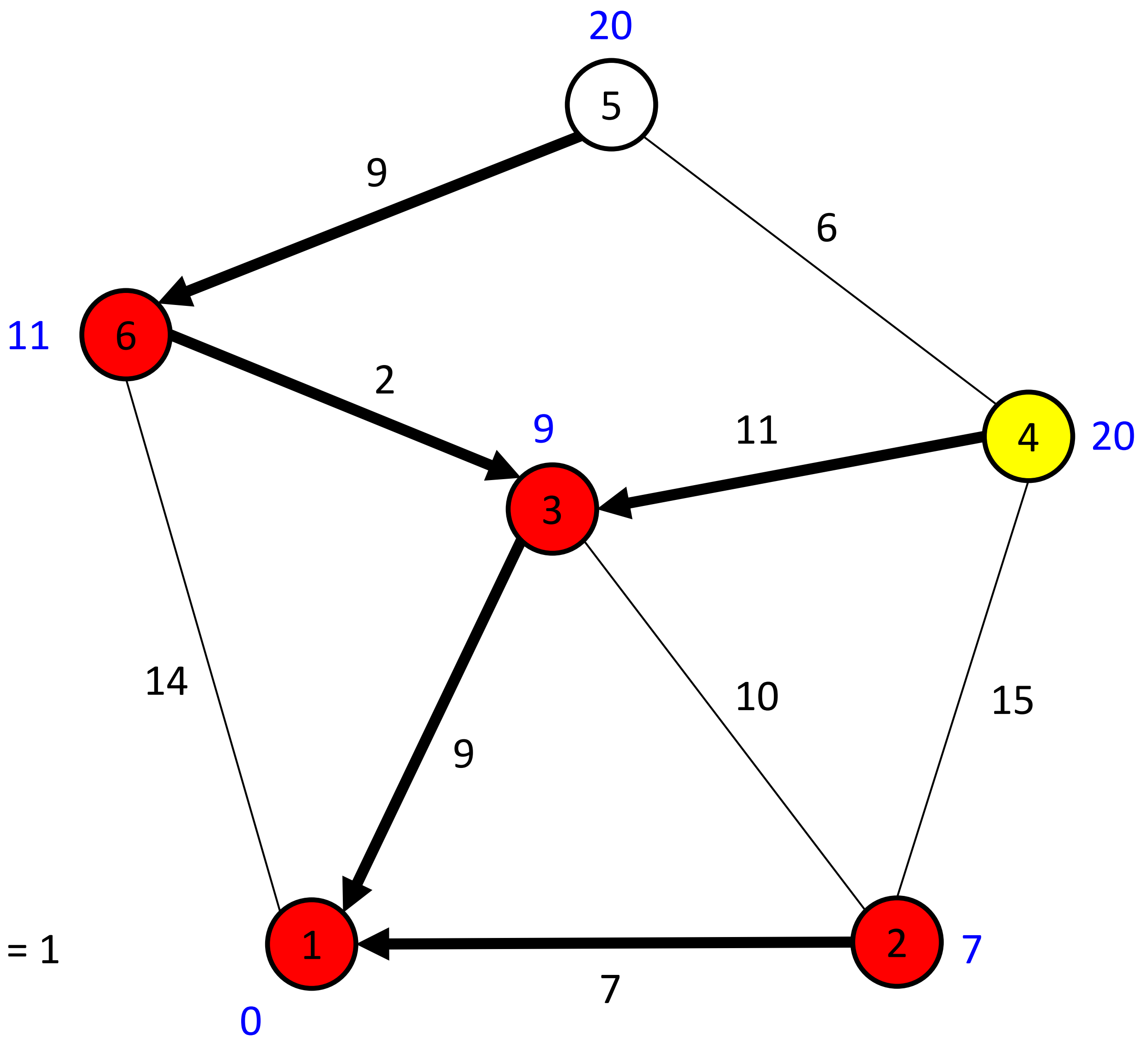
Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

Q=[4,5,~~6~~]

U=6

V=5

parent[v] dist[v]



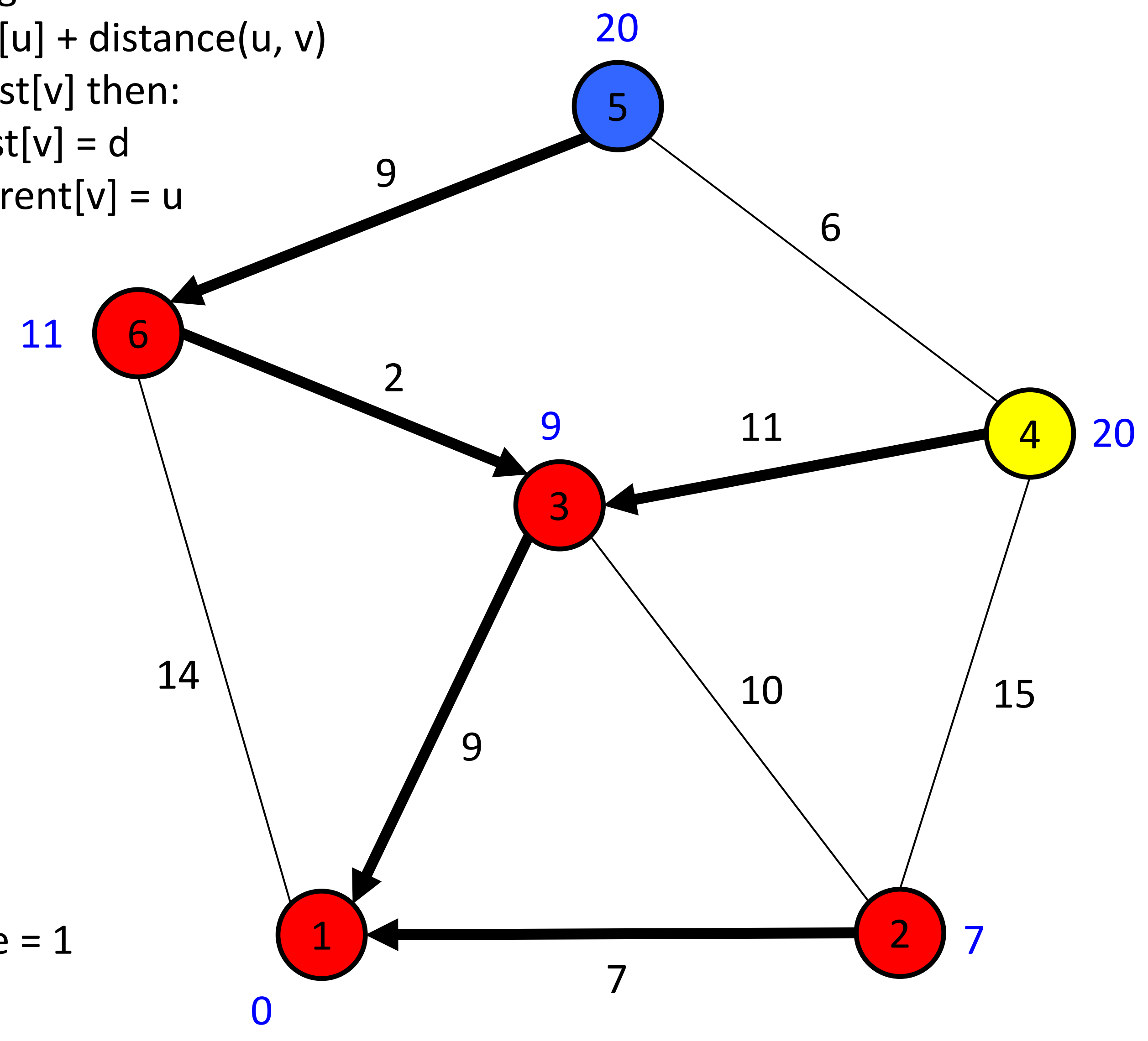
Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

Q=[~~4,~~5]
U=4
V=

* Source = 1

Let u = get vertex in Q with smallest distance value (node 4)

Remove u (node 4) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u

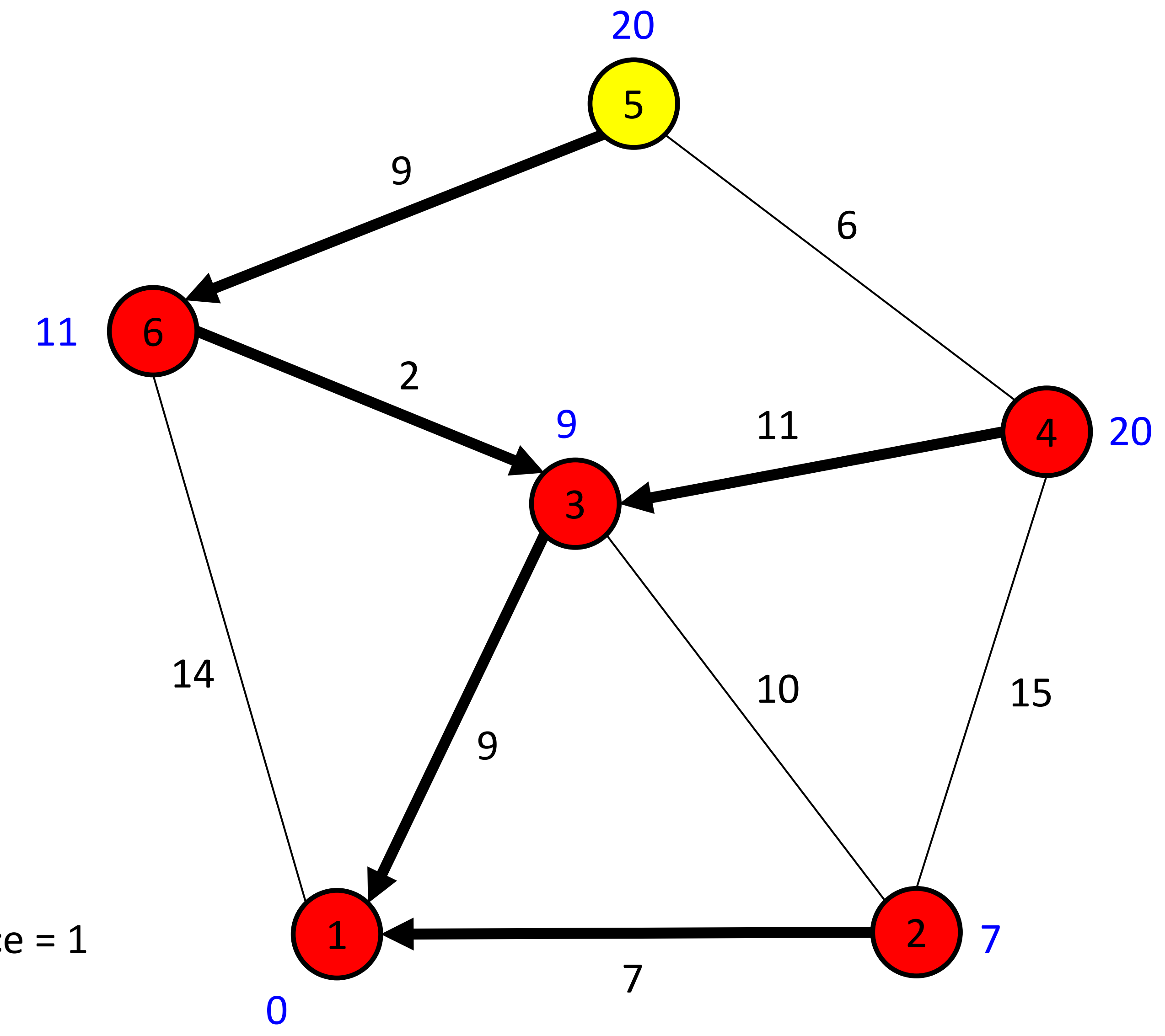


* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

Q=[~~4,~~5]
U=4
V=

parent[v]   dist[v]

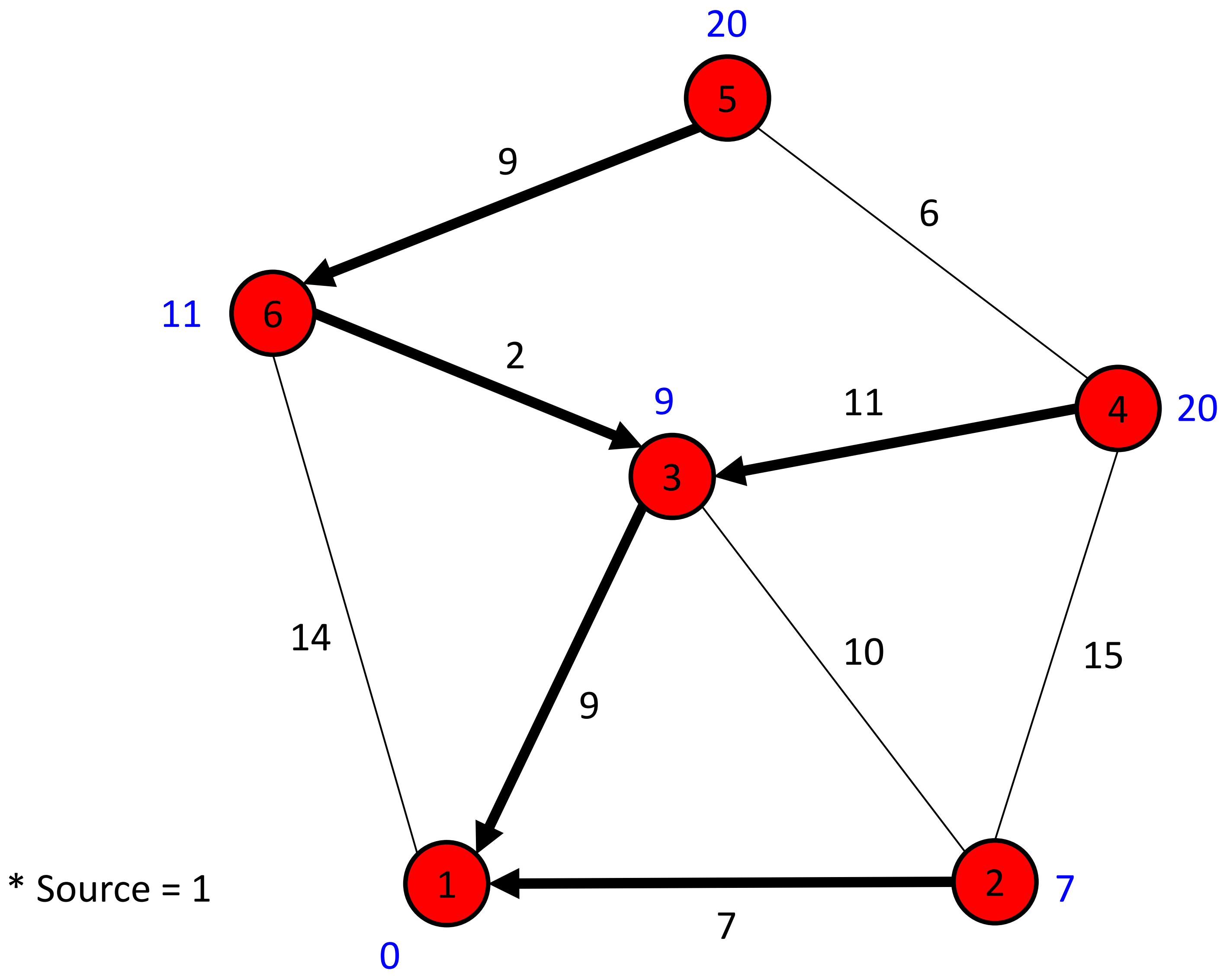


* Source = 1

Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

Q=[~~5~~]
U=5
V=

Let u = get vertex in Q with smallest distance value (node 5)



Dest	Cost	Parent
1	0	1
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

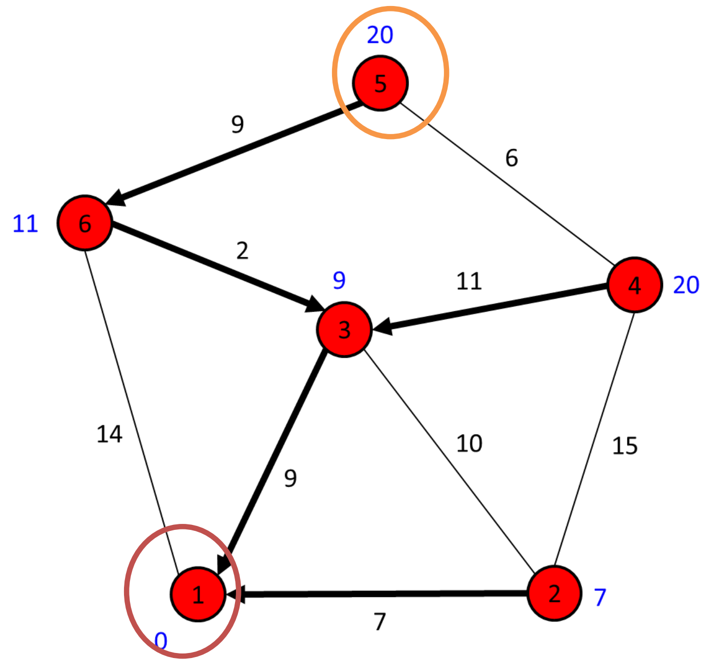
Q=[ ]
U=5
V=

* We now know the shortest distance and shortest path to all nodes from node 1.

Reconstructing the path from lookup table

Want to go from node 1 to v (e.g. v=5)

```
if parent[v] is empty then return null path
path = (v)
while v != 1 do:
        v = parent[v]
        path.prepend(v)
return path
```



Dest	Cost	Parent
1	0	1
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

All pairs shortest path (APSP)

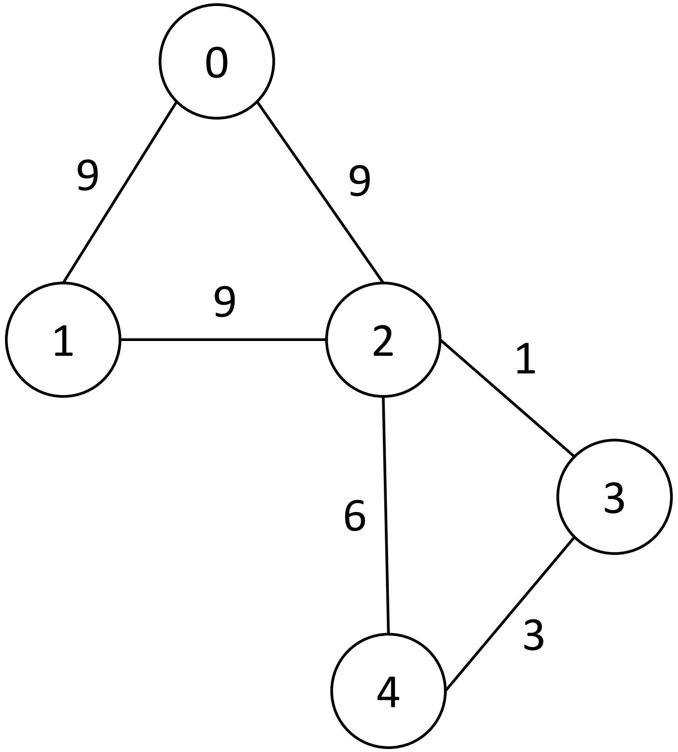
- We talked about this briefly as a “navigation table”
- A look-up table of the form table[node1,node2]-> node 3
  - Where node3 is the next node to go to if you want to go from node1 to node2
- Intuition: Find the shortest distance/path between all pairs of nodes
  - Use this to construct the look-up table

Floyd-Warshall algorithm

- 1962: All-pairs shortest path algorithm
- Tells you path from all nodes to all other nodes in weighted graph
- Positive or negative edge weights, but no negative cycles (edges sum to negative)
- Incrementally improves estimate
- $O(|V|^3)$
- Makes use of dynamic programming
- Compares all possible paths through the graph between each pair of vertices
- Use Dijkstra from each starting vertex when the graph is sparse and has non-negative edges

Given: G=(V,E), |V| = number of vertices

```
For each edge (u, v) do:
    dist[u][v] = weight of edge (u, v) or infinity
    next[u][v] = v
For k = 0 to |V| do:                    ← Intermediate node
    for i = 0 to |V| do:                ← Start node
        for j = 0 to |V| do:            ← End node
            if dist[i][k] + dist[k][j] < dist[i][j] then:
                dist[i][j] = dist[i][k] + dist[k][j]
                next[i][j] = next[i][k]
```



Distance

	0	1	2	3	4
0	INF	9	9	INF	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

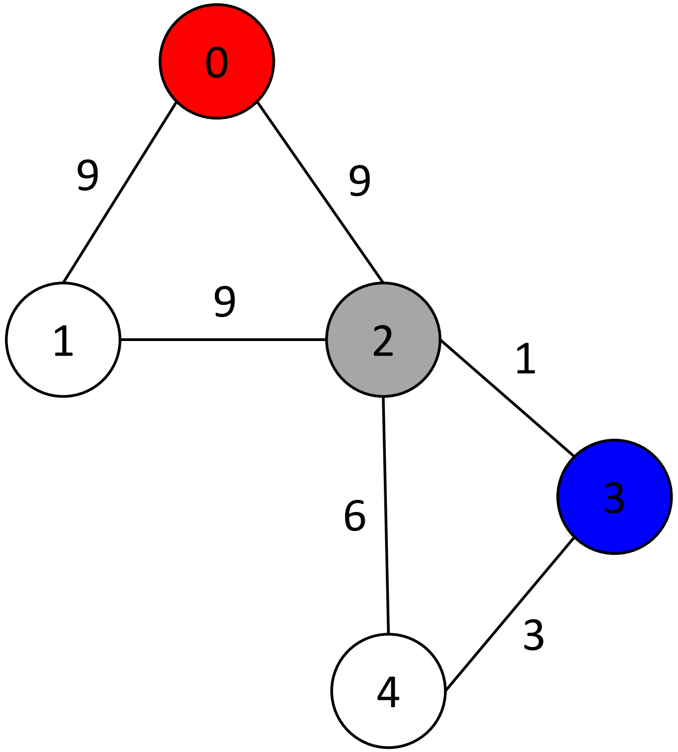
Next

	0	1	2	3	4
0		1	2		
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 0

i = 0

j = 0



Distance

	0	1	2	3	4
0	INF	9	9	INF	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

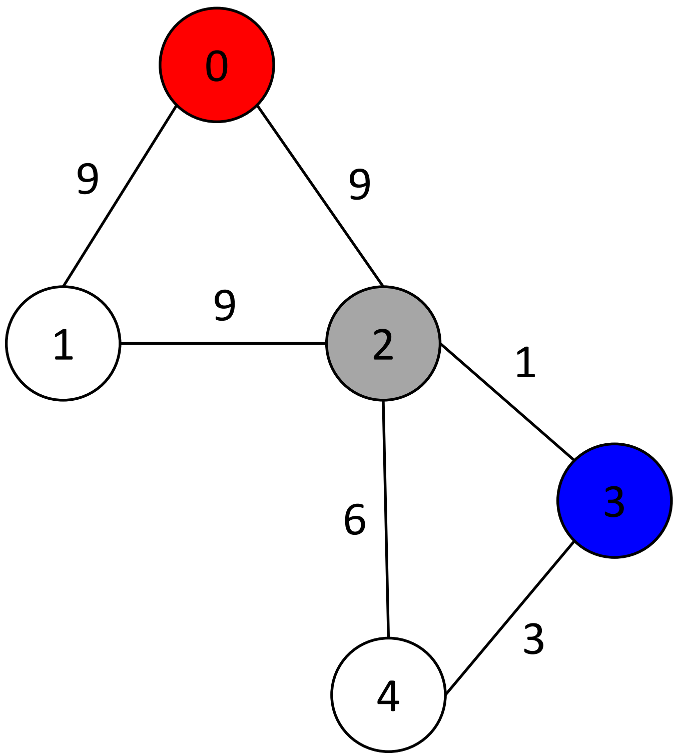
Next

	0	1	2	3	4
0		1	2		
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 3 $9 + 1 < \text{INF}$



Distance

	0	1	2	3	4
0	INF	9	9	10	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

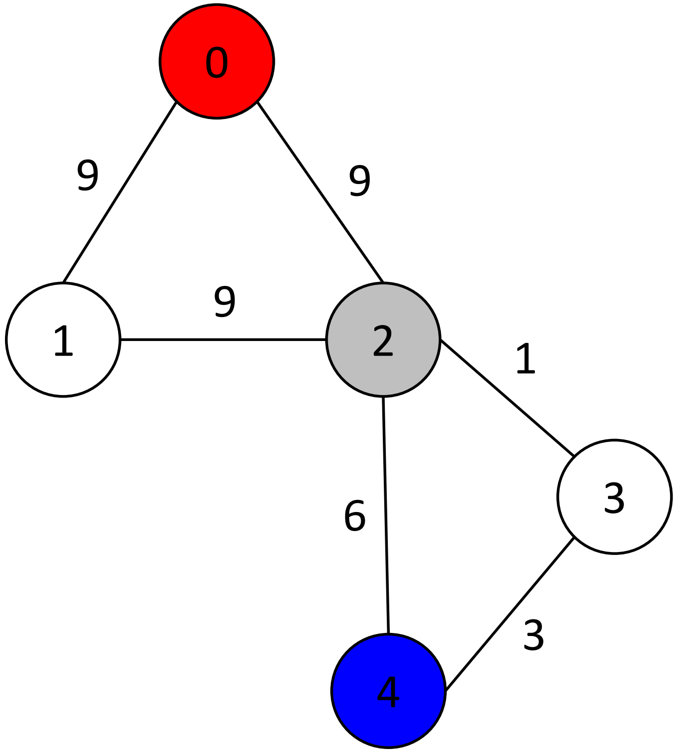
Next

	0	1	2	3	4
0		1	2	2	
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 3 0 -> 3: dist 10, goto 2



Distance

	0	1	2	3	4
0	INF	9	9	10	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

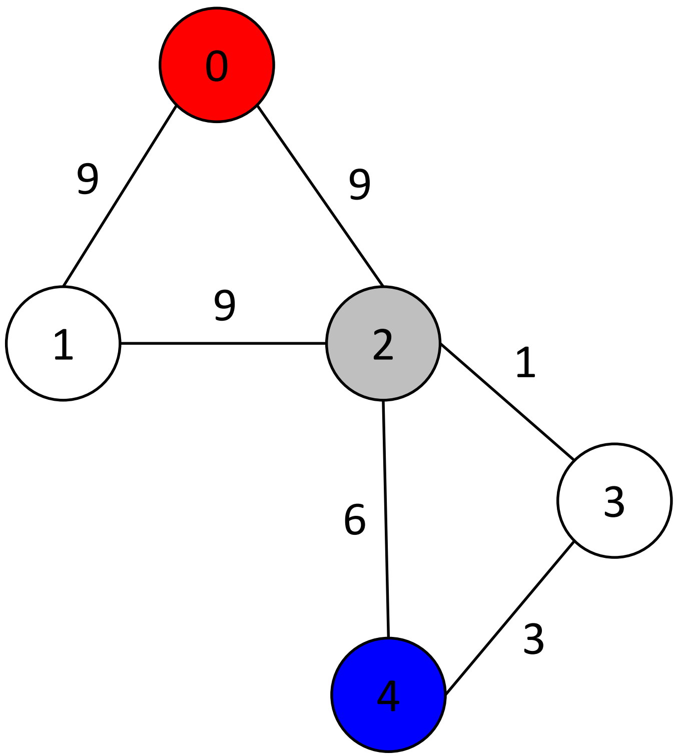
Next

	0	1	2	3	4
0		1	2	2	
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 4 9 + 6 < INF



Distance

	0	1	2	3	4
0	INF	9	9	10	15
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

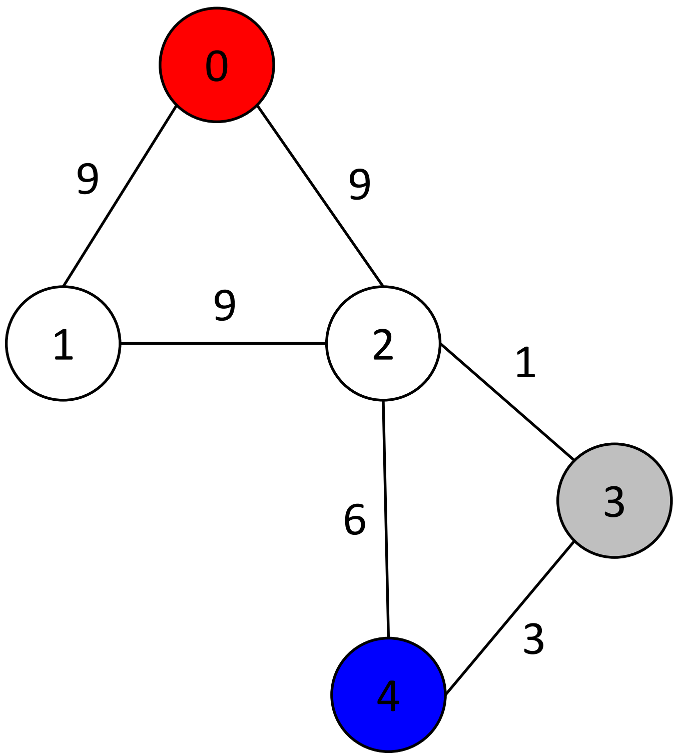
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 4 0 -> 3: dist 10, goto 2



Distance

	0	**1**	**2**	**3**	**4**
0	INF	9	9	10	15
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

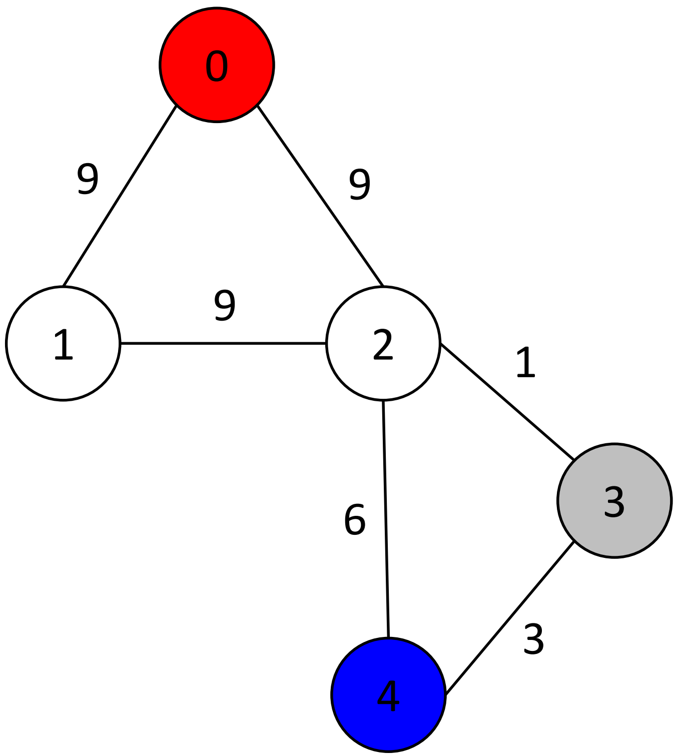
Next

	0	**1**	**2**	**3**	**4**
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 3

i = 0

j = 4 10 + 3 < 15



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

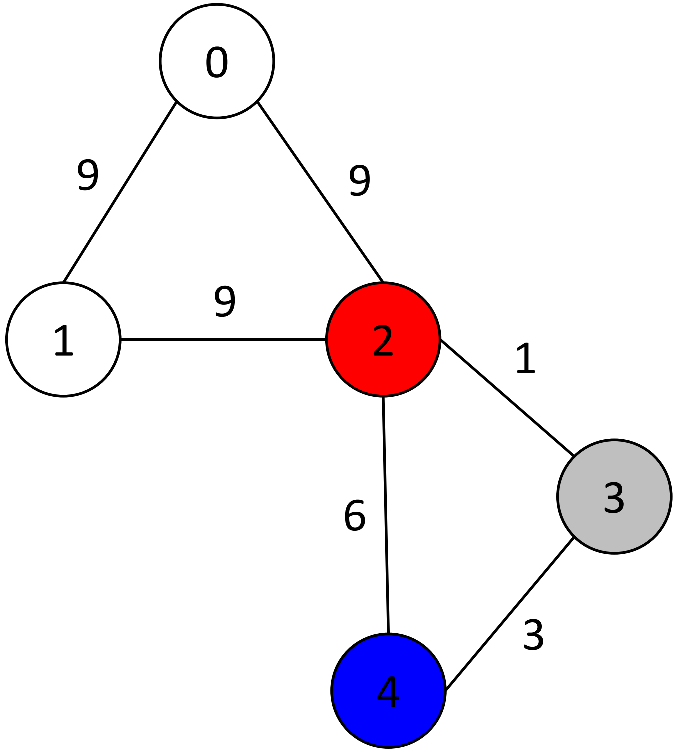
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 3

i = 0

j = 4 0 -> 4: dist 13, goto 2



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

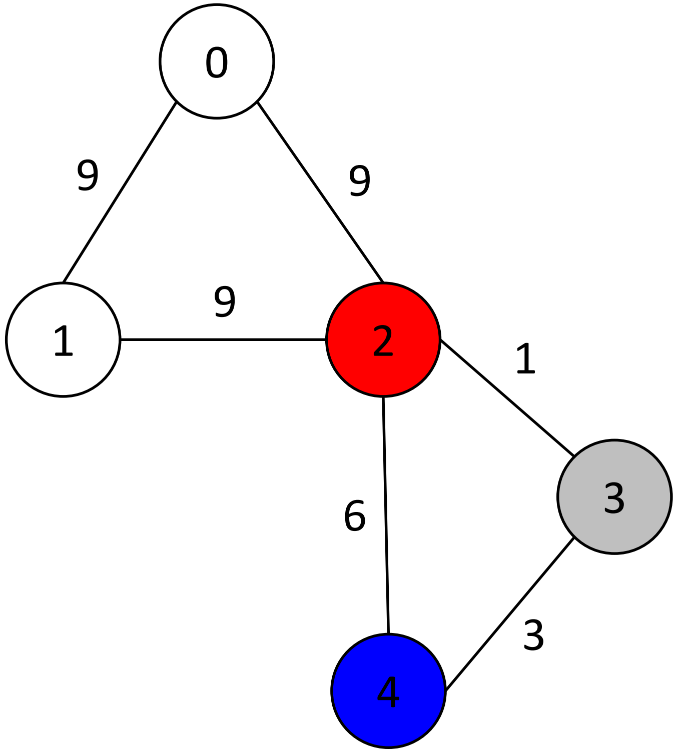
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 3

i = 2

j = 4 $1 + 3 < 6$



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	INF	INF
2	9	9	INF	1	4
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

Next

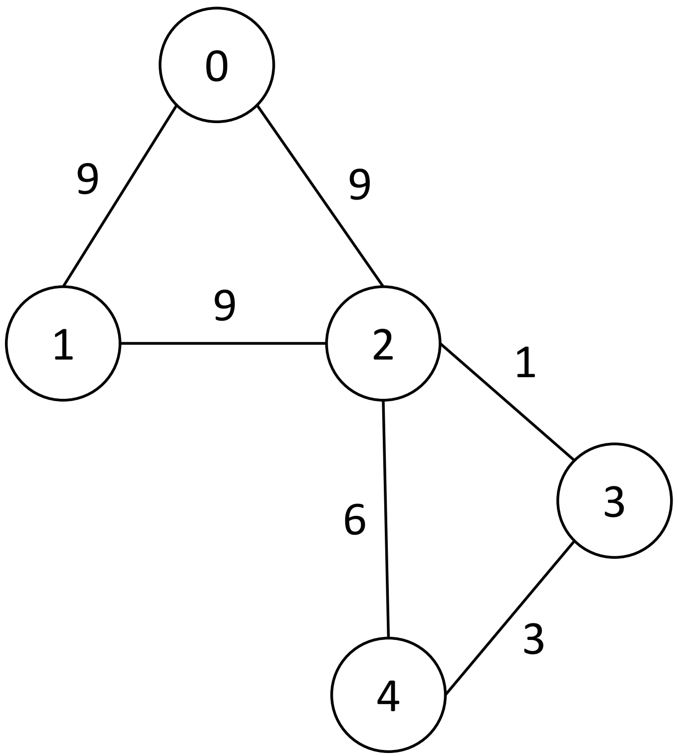
	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	3
3			2		4
4			2	3	

k = 3

i = 2

j = 4   2 -> 4: dist 4, goto 3

Finally...



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	10	13
2	9	9	INF	1	4
3	10	10	1	INF	3
4	13	13	4	3	INF

Next

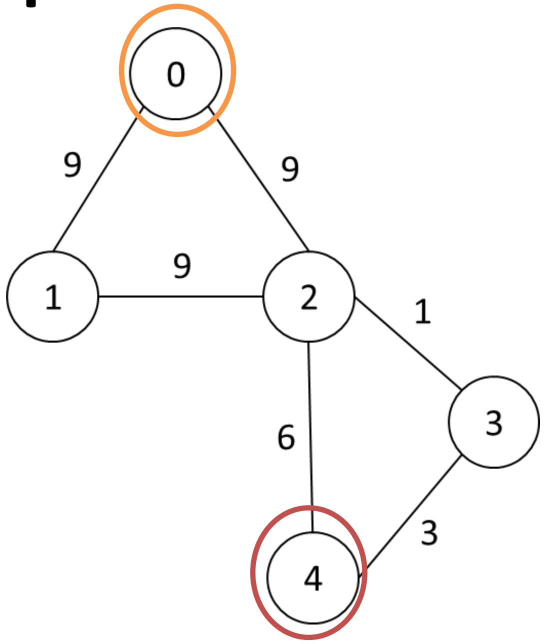
	0	1	2	3	4
0		1	2	2	2
1	0		2	2	2
2	0	1		3	3
3	2	2	2		4
4	3	3	3	3	

k = 3

i = 2

j = 4 2 -> 4: dist 4, goto 3

Reconstructing the path from lookup table



Want to go from u to v (E.g. u=0, v=4)

if next[u][v] is empty then return null path

path = (u)

path=0

while u <> v do:

    u = next[u][v]

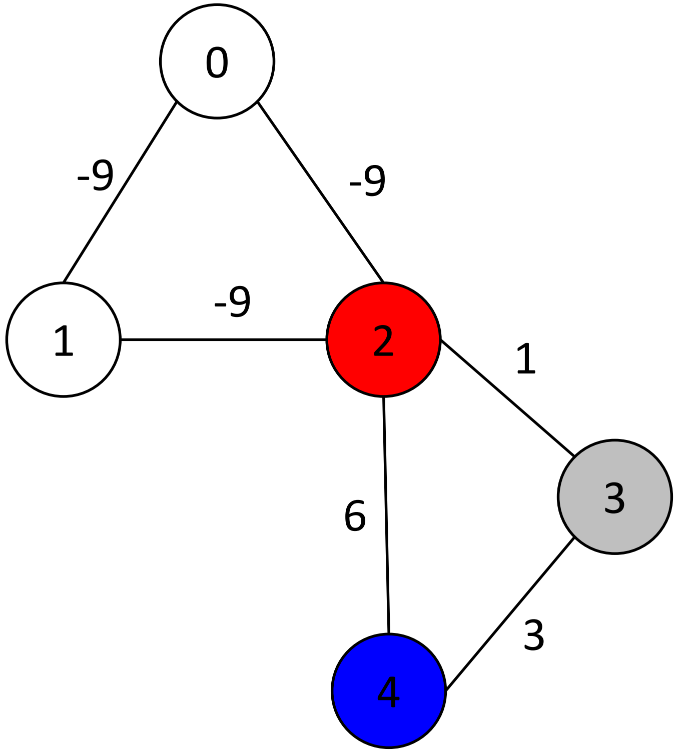
    path.append(u)

u=next[0][4]=2; path=0,2
u=next[2][4]=3; path=0,2,3
u=next[3][4]=4; path=0,2,3,4

return path

	0	**1**	**2**	**3**	**4**
0		1	2	2	2
1	0		2	2	2
2	0	1		3	3
3	2	2	2		4
4	3	3	3	3	

Detecting Negative Cycles

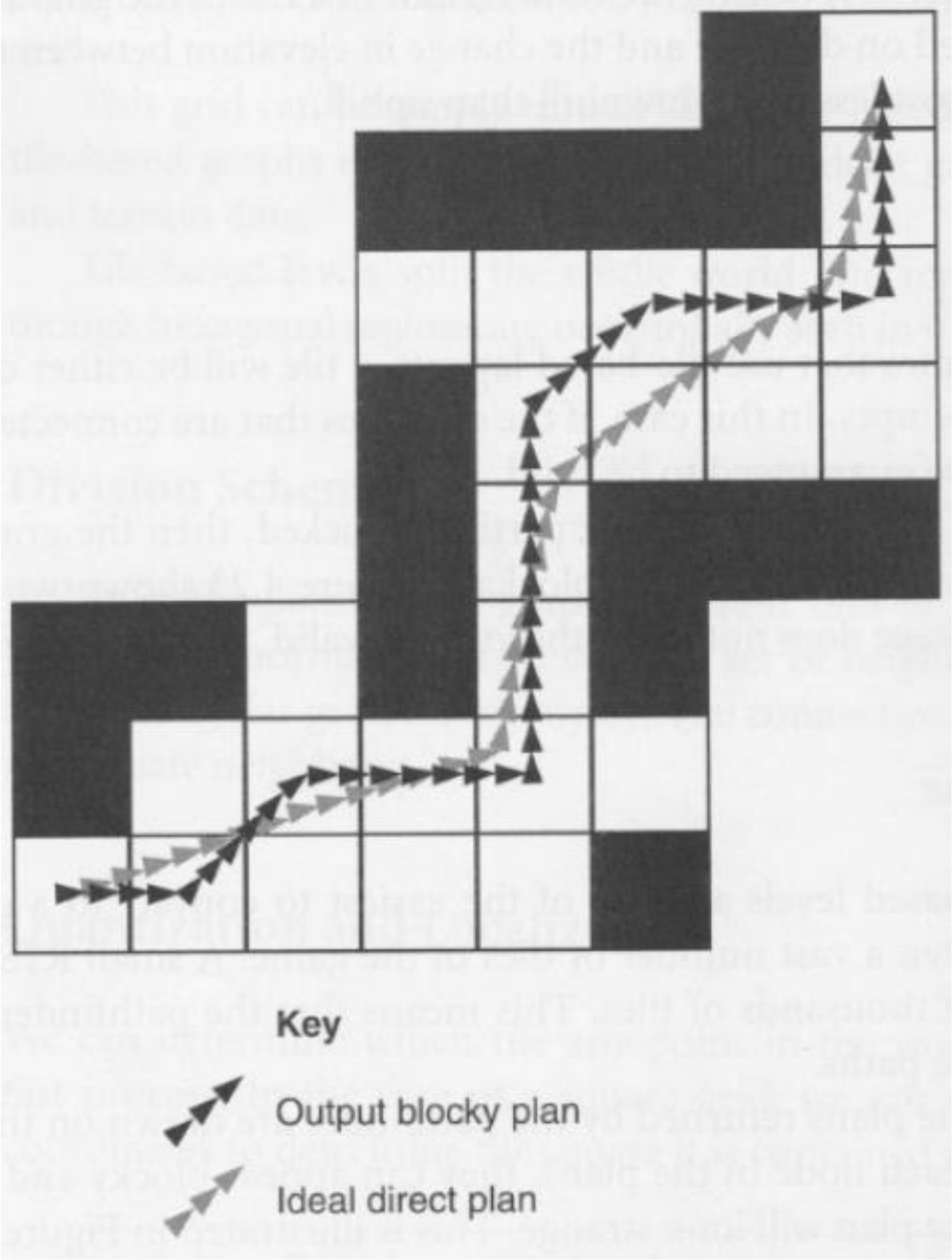


Distance

	0	1	2	3	4
0	-3	-9	-9	8	4
1	-9	-3	-9	INF	INF
2	-9	-9	-3	1	4
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

When to use A* and APSP

1. If the environment is small and static?
2. If the environment is dynamic?
3. If the environment is large and static?
   1. If runtime memory is an issue?
   2. If runtime memory isn't an issue?
4. If the environment is large and dynamic?



Fixing awkward agent movement:

- String pulling
- Splines
- Hierarchical A*

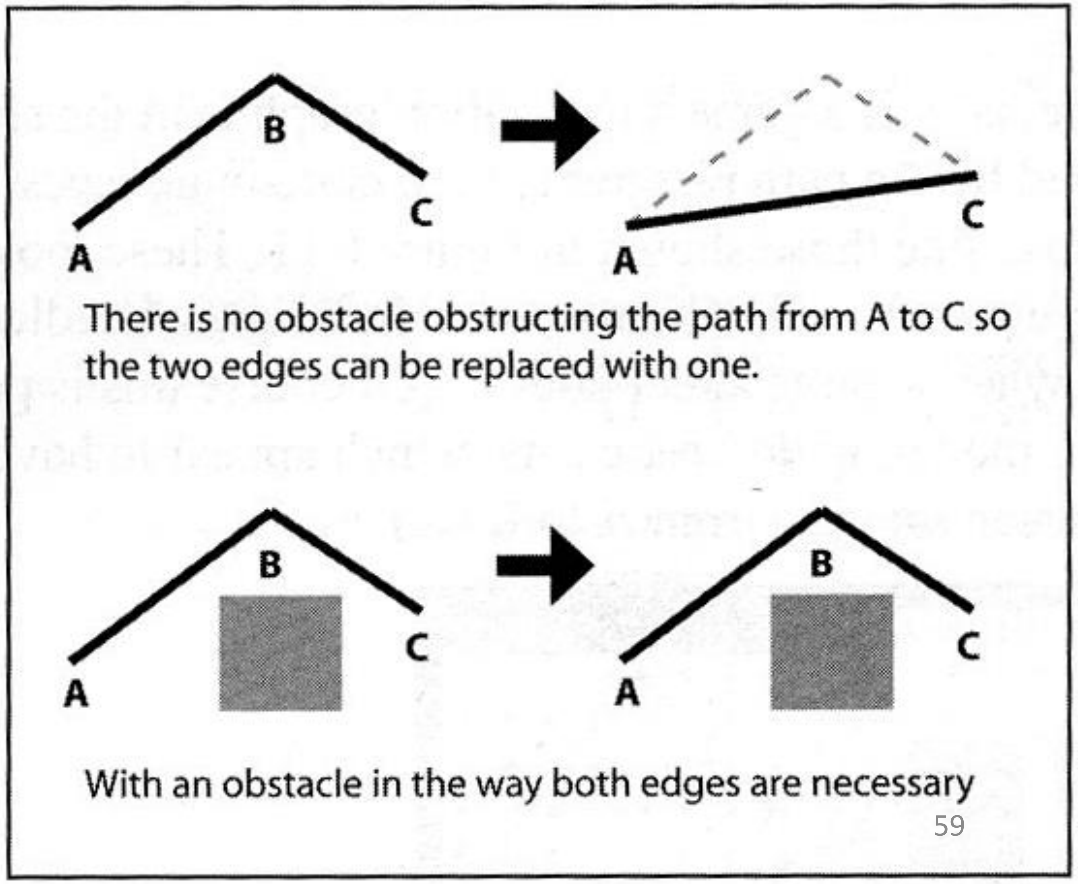
SOLVING WEIRD FINAL PATHS

Weird: path doesn’t consider environment

- Add extra heuristic to mark certain grid cells as more “costly” to step through.
  - Cells near obstacles
  - Cells that an agent can get “caught” on
  - Cells that an agent can get “trapped” in

Path Smoothing via "String pulling"

- Zig-zagging from point to point looks unnatural
- Post-search smoothing can elicit better paths



Quick Path-Smoothing

- Given a path, look at first two edges, E1 & E2
  1. Get E1_src and E2_dest
  2. If unobstructed path between the two, set E1_dest = E2_dest, then delete E2 from the path. **Set next edge as E2.**
  3. Else, increment E1 and E2.
  4. Repeat until E2_dest == goal.

Slow Path-Smoothing

- Given a path, look at first two edges, E1 & E2
  1. Get E1_src and E2_dest
  2. If unobstructed path between the two, set E1_dest = E2_dest, then delete E2 from the path. **Set E1 and E2 from beginning of path.**
  3. Else, increment E1 and E2.
  4. Repeat until E2_dest == goal.

SOLVING LONG PATH SEARCH TIMES

Solution to Long Search Times

- Precompute paths (if you can)
  - Dijkstra: Single source shortest path (SSSP; O(E log V))
    - Run for each vertex: O(VE Log V) which can go (V^3 Log V) in worst case
  - Floyd-warshall: All pairs shortest path (APSP, O($|V|^3$))
- Register search requests
  - Works best with lots of agents. Prevents heavy load to CPU.
  - Let agents wander or seek toward a goal while waiting for a search response. (Although they might wander in the wrong direction)
- Hierarchical Path Planning

Precomputing Paths

- Faster than computation on the fly esp. large maps and many agents
- Use Dijkstra's or Floyd-warshall algorithm to create lookup tables
- Lookup cost tables
- What is the main problem with pre-computed paths?

Sticky Situations: Movable objects, fog of war, memory issues, and other burps – precomputed paths do no good

SOLVING WHEN WE CAN’T PRECOMPUTE

Sticky Situations

- Dynamic environments can ruin plans; memory issues can inhibit precomputing

- What do we do when an agent has been pushed back through a doorway that it has already “visited”?

- What do we do in “fog of war” situations?

- What if we have a moving target?

Dynamic environments

- Terrain can change
  - Jumpable?
  - Kickable?
  - Too big to jump/kick?
- Typically: destructible environments
- Path network edges can be eliminated
- Path network edges can be created

Other Heuristic Search Speedups

- A* (Iterative deepening)
- Hierarchical A*
- Real-time A*
- Real-time A* with lookahead
- D* lite

Hierarchical Path Planning

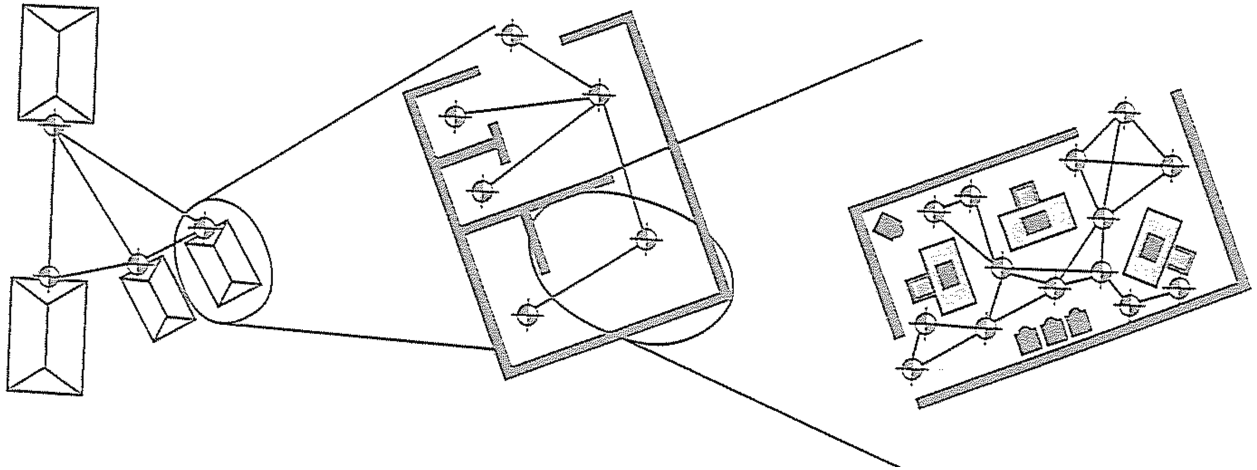
- Used to reduce CPU overhead of graph search
- Plan with coarse-grained and fine-grained maps
- Example: Planning a trip to NYC based on states, then individual roads

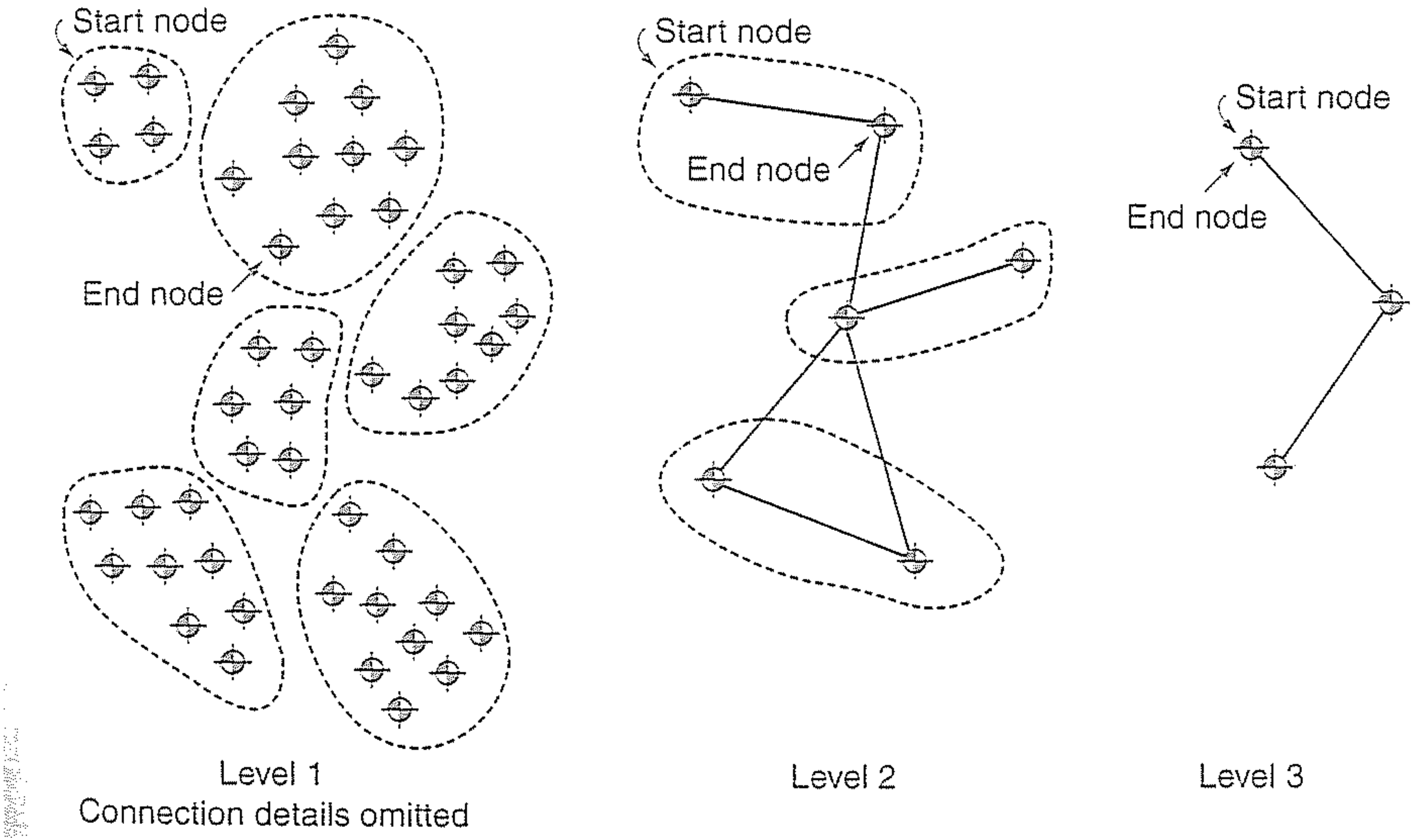
Hierarchical A*

- http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf
- http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/
- Within 1% of optimal path length, but up to 10 times faster

Hierarchical A*

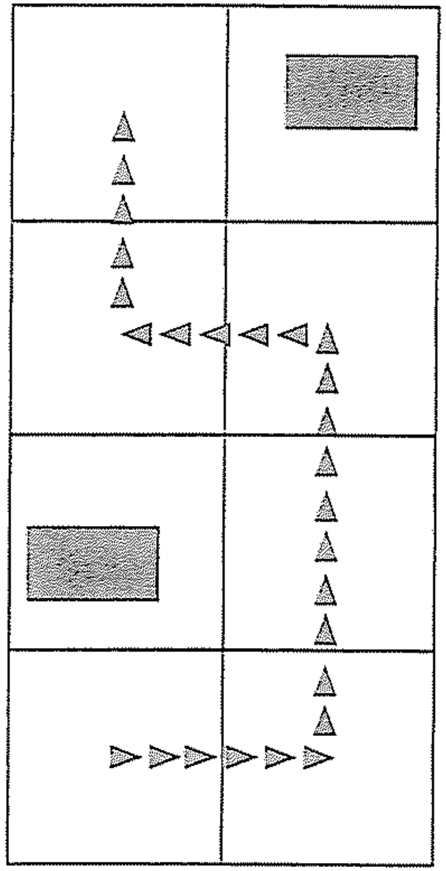
- People think hierarchically (more efficient)
- We can prune a large number of states



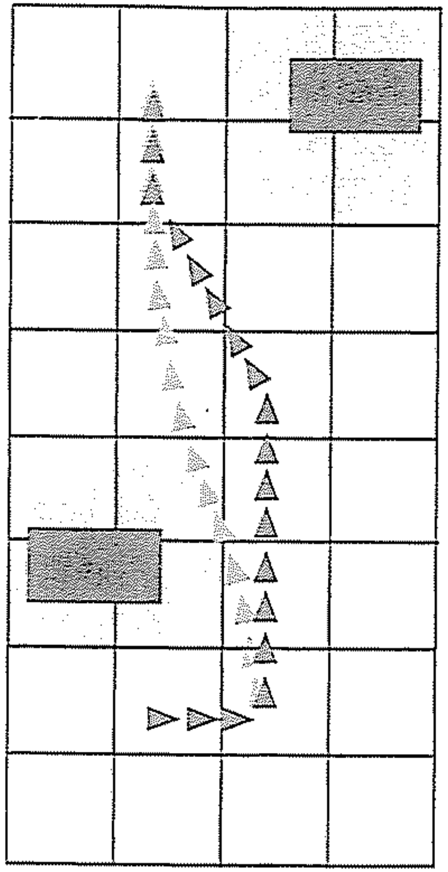


How high up do you go? As high as you can without start and end being in the same node.

Path Smoothing in Hierarchical A*

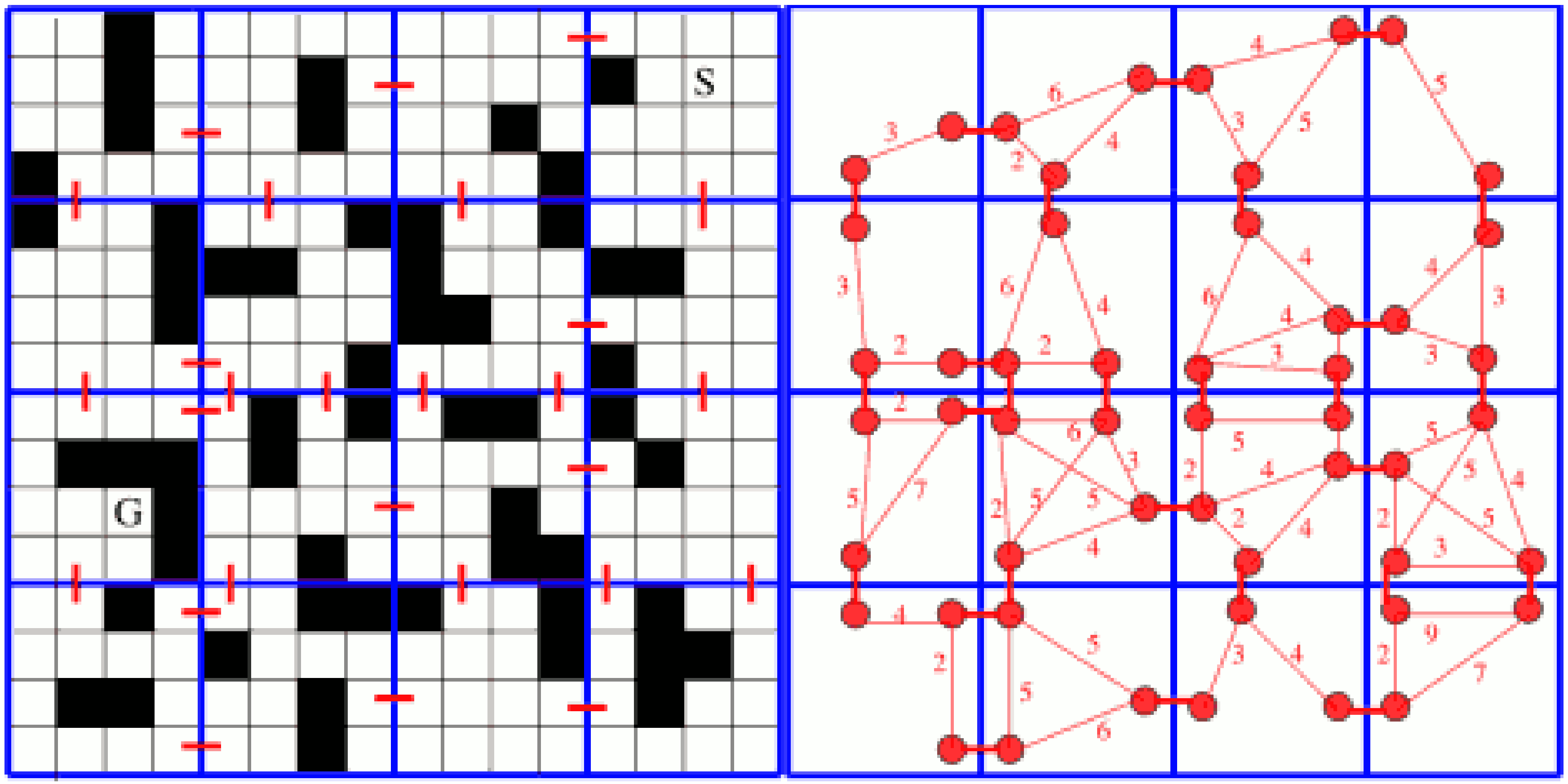


High-level plan



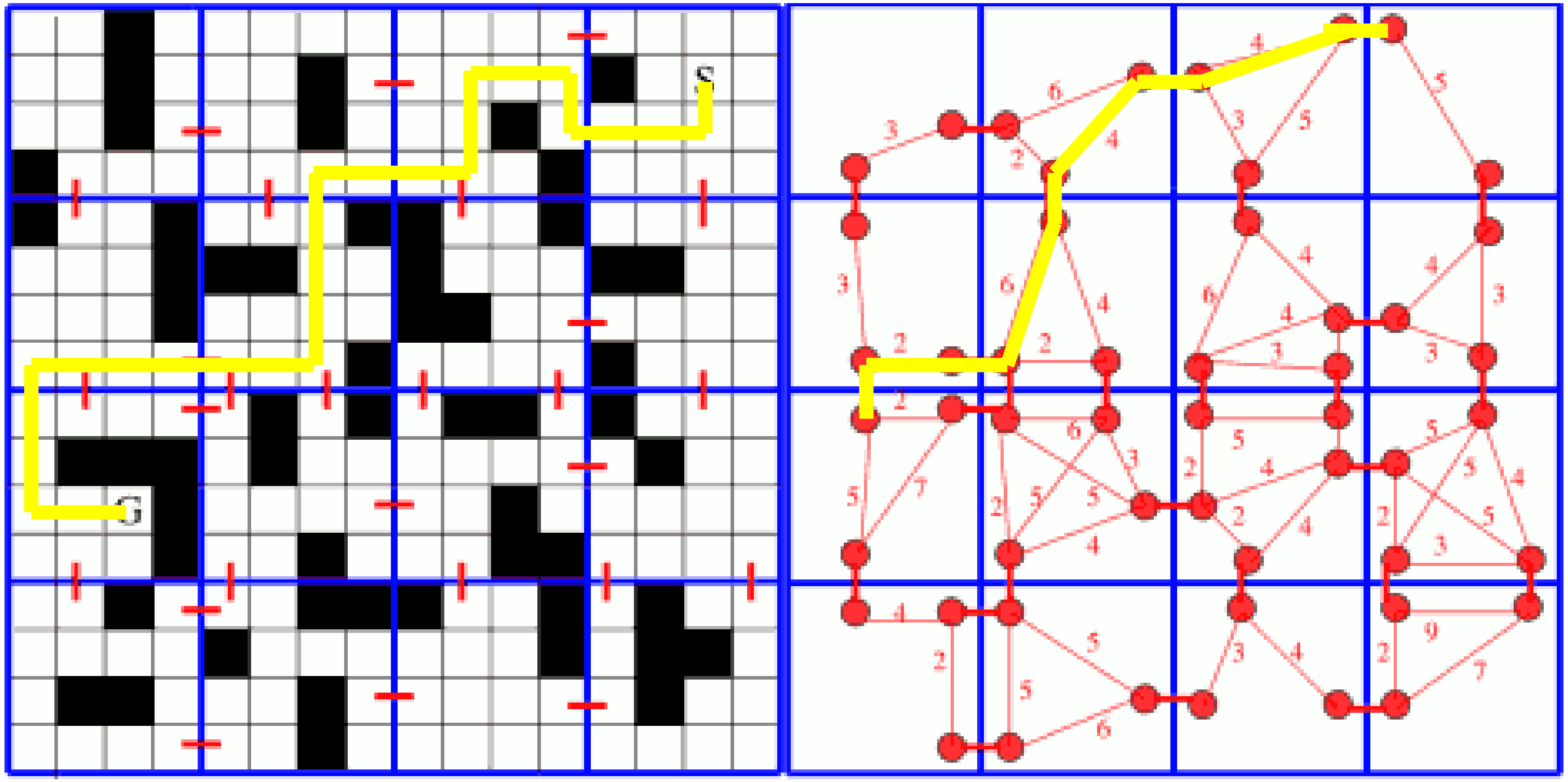
Low-level plan

1. Build clusters. Can be arbitrary
2. Find transitions, a (possibly empty) set of obstacle-free locations.
3. Inter-edges: Place a node on either side of transition, and link them (cost 1).
4. Intra-edges: Search between nodes inside cluster, record cost.
   * Can keep optimal intra-cluster paths, or discard for memory savings.



1. Start cluster: Search within cluster to the border
2. Search across clusters to the goal cluster
3. Goal cluster: Search from border to goal
4. Path smoothing

* Really just adds start and goal to the hierarchy graph



Real Time A*

- Reduces execution time of A* by limiting search horizon of A*
- Online search: execute as you search
  - Because you can't look at a state until you get there
  - You can't backtrack
  - No open list
- Modified cost function f()
  - g(n) is actual distance from n to current state (instead of initial state)
- Use a hash-table to keep track of h() for nodes you have visited (because you might visit them again)
- **Pick node with lowest f-value from immediate successors**
- **Execute move immediately**
- After you move, update previous location
  - h(prev) = second best f-value
  - Second best f-value represents the estimated cost of returning to the previous state (and then add g)

RTA* with lookahead

- At every node you can see some distance
- DFS, then back up the value (think of it as minimin with alpha-pruning)
- **Search out to known limit**
- **Pick best, then move**
- Repeat, because something might change in the environment that change our assessment
  - Things we discover as our horizon moves
  - Things that change behind us

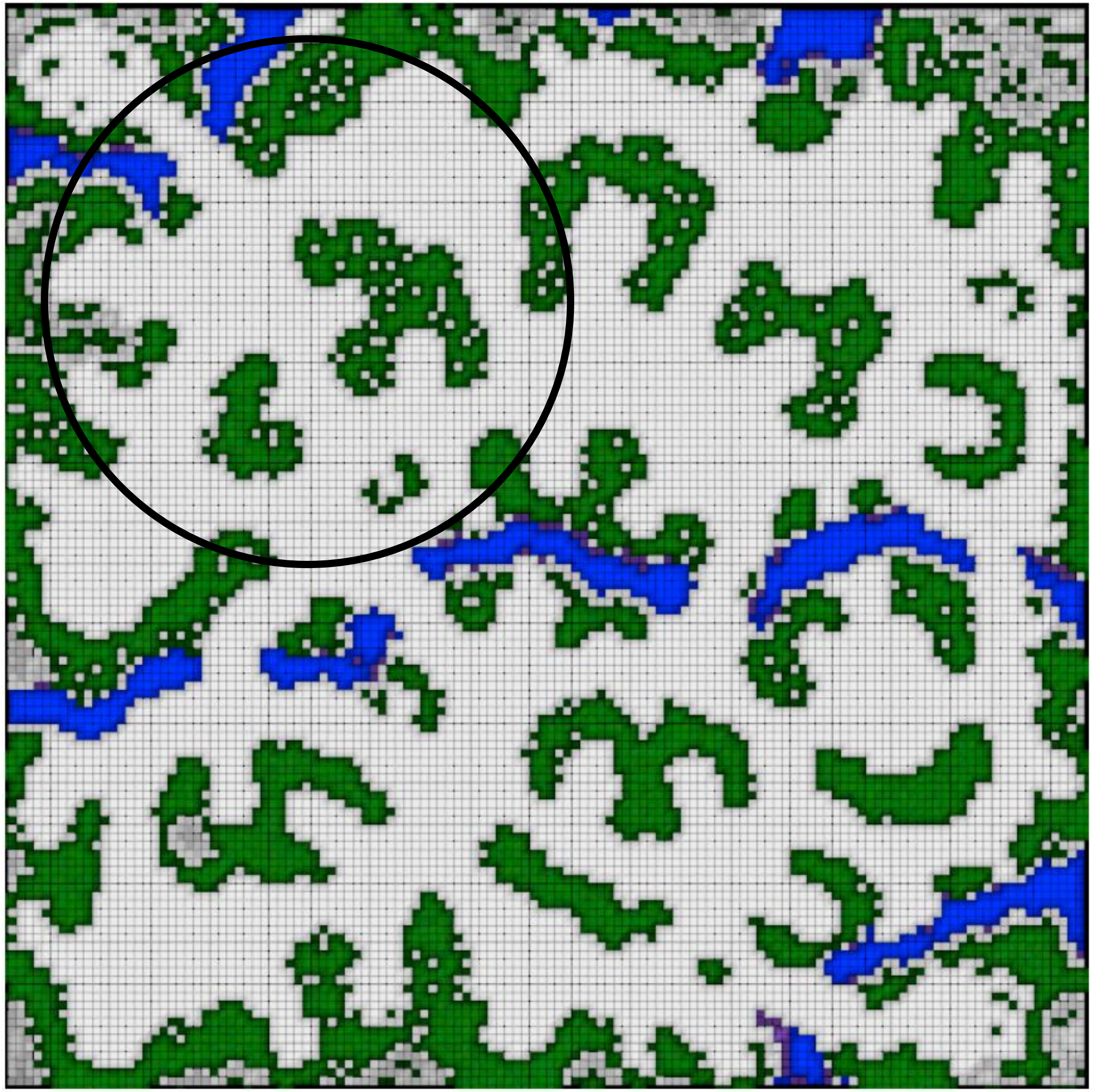
D* Lite

- 1994: Incremental search: replan often, but reuse search space if possible
- In unknown terrain, assume anything you don’t know is clear (optimistic)
- Perform A*, execute plan until discrepancy, then replan
- D* Lite achieves 2x speedup over A* (when replanning)

D* Lite

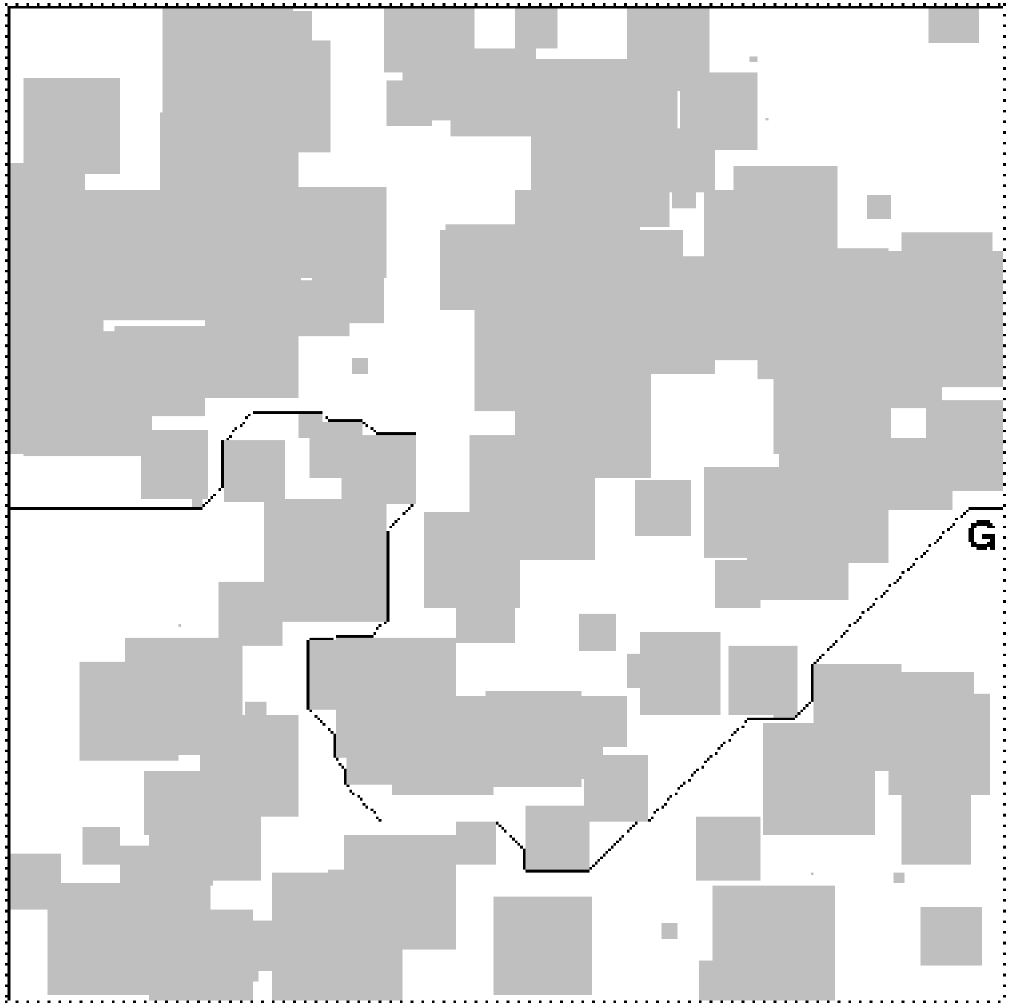
Sven Koenig
College of Computing
Georgia Institute of Technology
Atlanta, GA 30312-0280
skoenig@cc.gatech.edu

Maxim Likhachev
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
maxim+@cs.cmu.edu

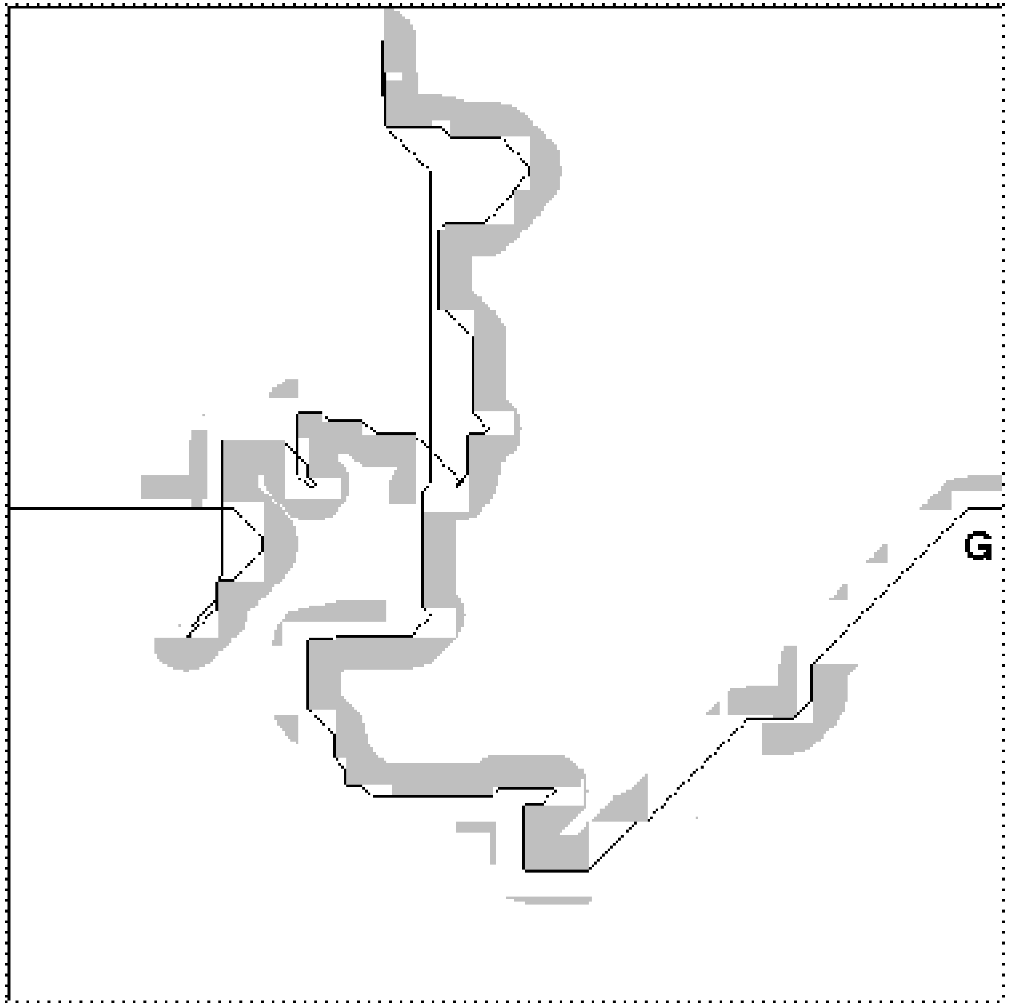


Fog of war

“Omniscient optimal”: given complete information



“Optimistic optimal”: assume empty for parts you don’t know.



Heuristic Search Recap

- A*
  - Use when we can't precompute
    - Dynamic environments
    - Memory issues
  - Optimal when heuristic is admissible (and assuming no changes)
  - Replanning can be slow on really big maps
- Hierarchical A* is the ~same, but faster
  - Within 1% of A* optimality but up to 10x faster
- Real-time A*
  - Stumbling in the dark, 1 step lookahead
  - Replan every step, but fast!
  - Realistic? For a blind agent that knows nothing
  - Optimal when completely blind
- Real-time A* with lookahead
  - Good for fog-of-war
  - Replan every step, with fast bounded lookahead to edge of known space
  - Optimality depends on lookahead

Heuristic Search Recap

- D* Lite
  - Assume everything is open/clear
  - Replan when necessary
  - **Worst case: Runs like Real-Time A***
  - **Best case: Never replans**
  - Optimal including changes

See Also

- AI Game Programming wisdom 2, CH 2
- Buckland CH 8
- Millington CH 4
- Wikipedia rabbit hole
- Monte Carlo Tree Search (MCTS)