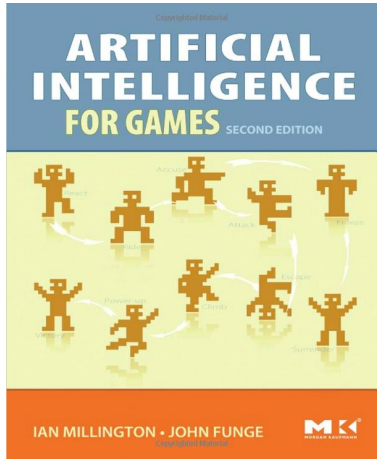
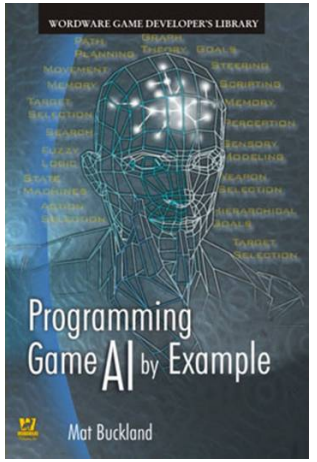


Disclaimer: I use these notes as a guide rather than a comprehensive coverage of the topic. They are neither a substitute for attending the lectures nor for reading the assigned material.



"I may not have gone where I intended to go, but I think I have ended up where I needed to be." –Douglas Adams

"All you need is the plan, the road map, and the courage to press on to your destination." –Earl Nightingale

"If I cease searching, then, woe is me, I am lost. That is how I look at it - keep going, keep going come what may." – Vincent van Gogh

Announcements

- HW2 (path network) due Sunday night, September 8 @ 11:55pm
- HW2 is more challenging than HW1. Start early!
  - You must write the code to generate the path network, as a set of edges between path nodes. An edge between path nodes exists when (a) there is no obstacle between the two path nodes, and (b) there is sufficient space on either side of the edge so that an agent can follow the line without colliding with any obstacles.
  - We will test path network using a random-walk navigator that moves the agent to the nearest path node and then follows a randomly generated path---sequence of adjacent path nodes.

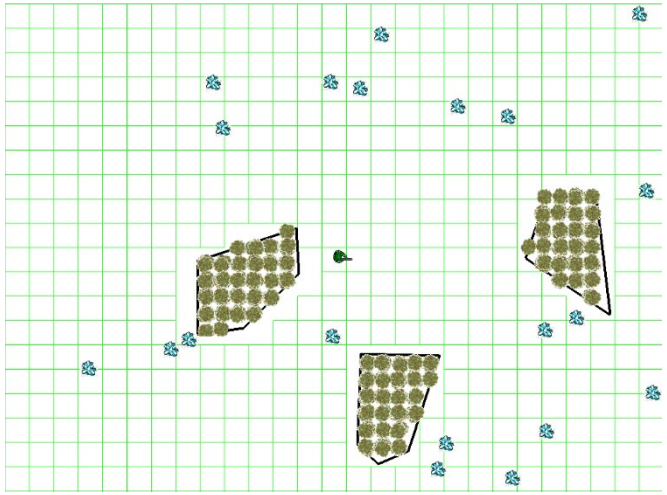
Verification of Participation (Deadline: Mon. Sept. 9, 2019 at 16:00): Verification of Participation is a process whereby instructional faculty report to the Registrar's Office and the Office of Scholarships & Financial Aid whether they have students enrolled in their classes who are not engaged with the course. This verification by faculty is a Federal Title IV requirement (visit https://registrar.gatech.edu/faculty-and-staff/verification-of-participation for more information about the requirement and what constitutes participation).

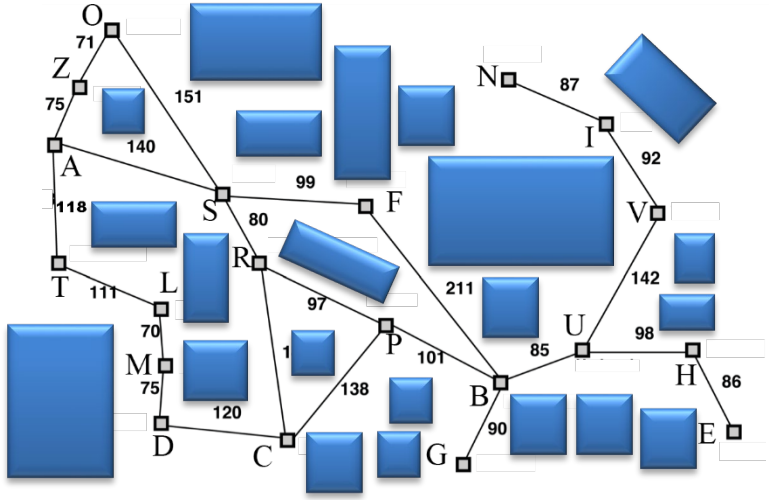
PREVIOUSLY ON…

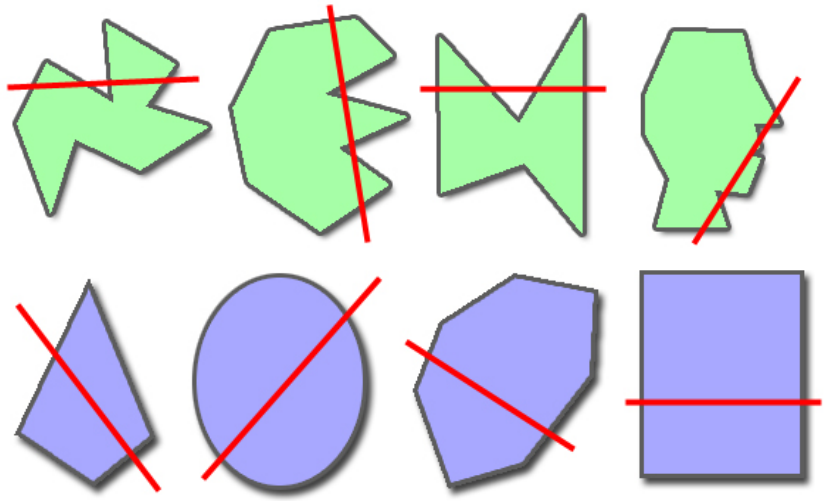
Graphs: Killer App in GAI

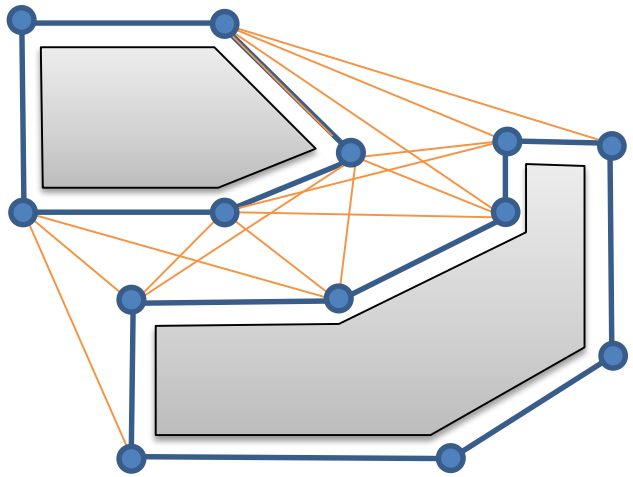
- Navigation / Pathfinding
- Navgraph: abstraction of all locations and their connections
- Cost / weight can represent terrain features (water, mud, hill), stealth (sound to traverse), etc
- What to do when …
  - Map features move
  - Map is continuous, or 100K+ nodes?
  - 3D spaces?

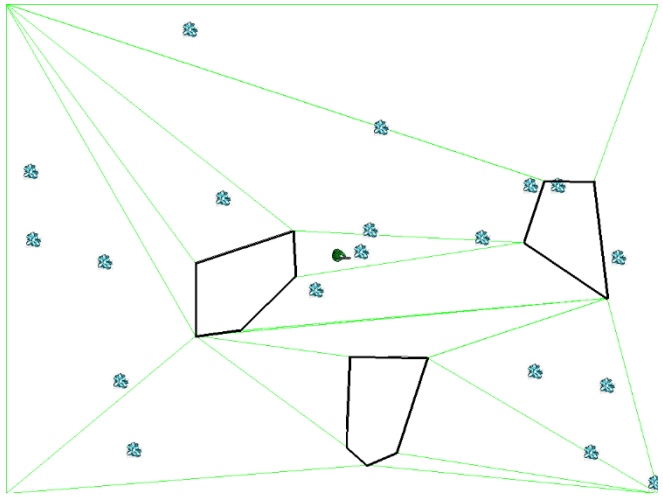
Modelling and Navigating the Game World

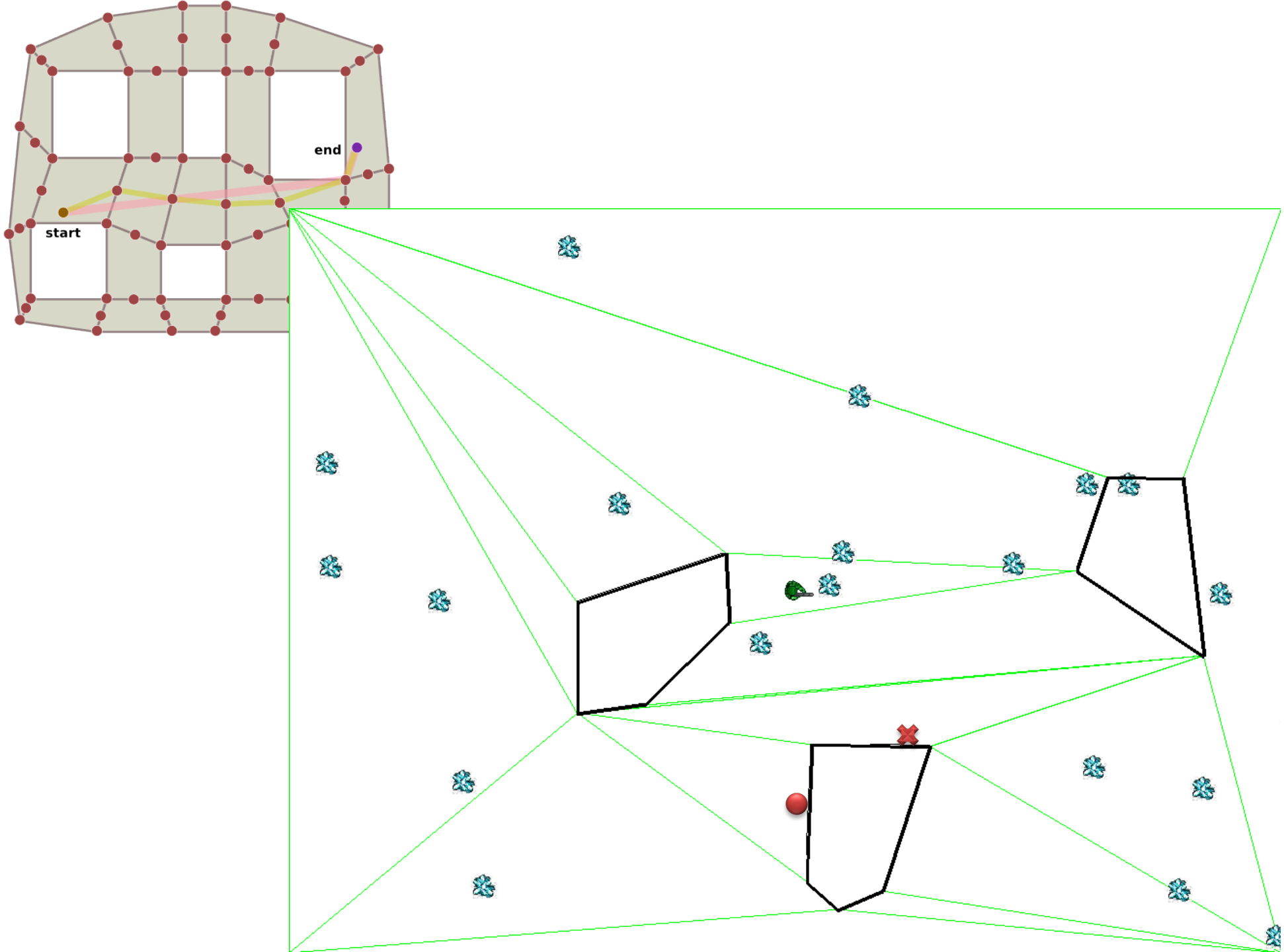


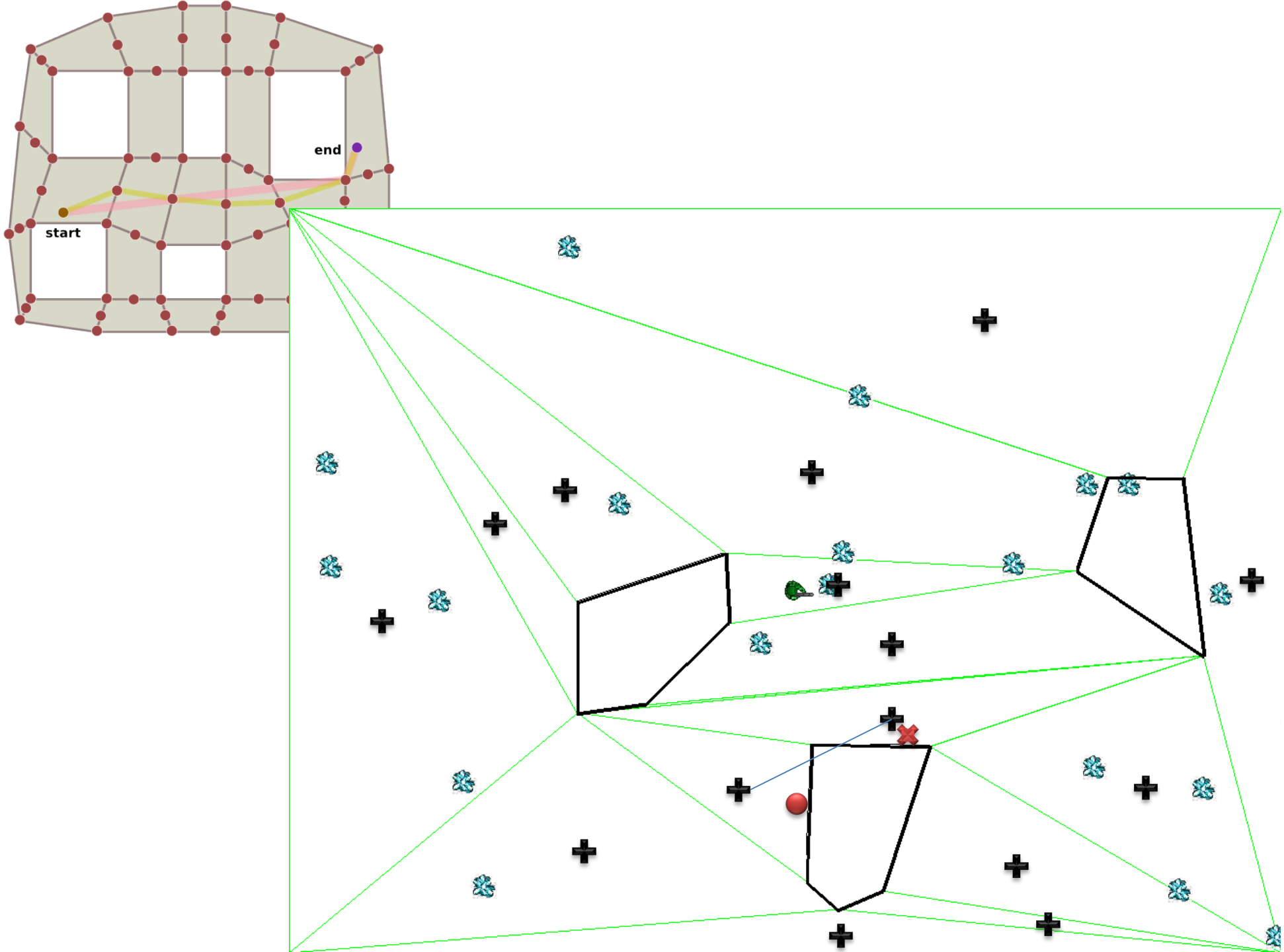


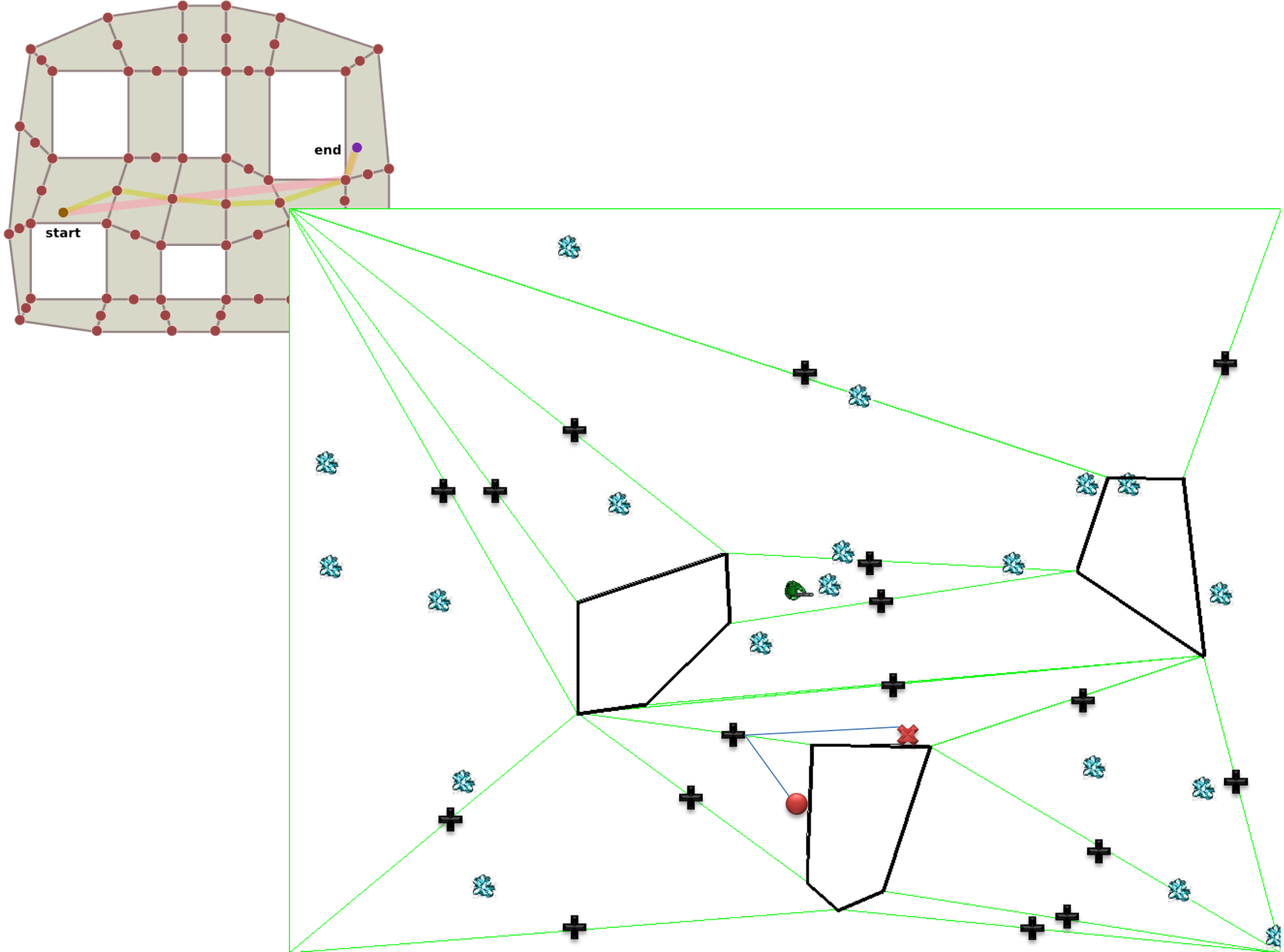


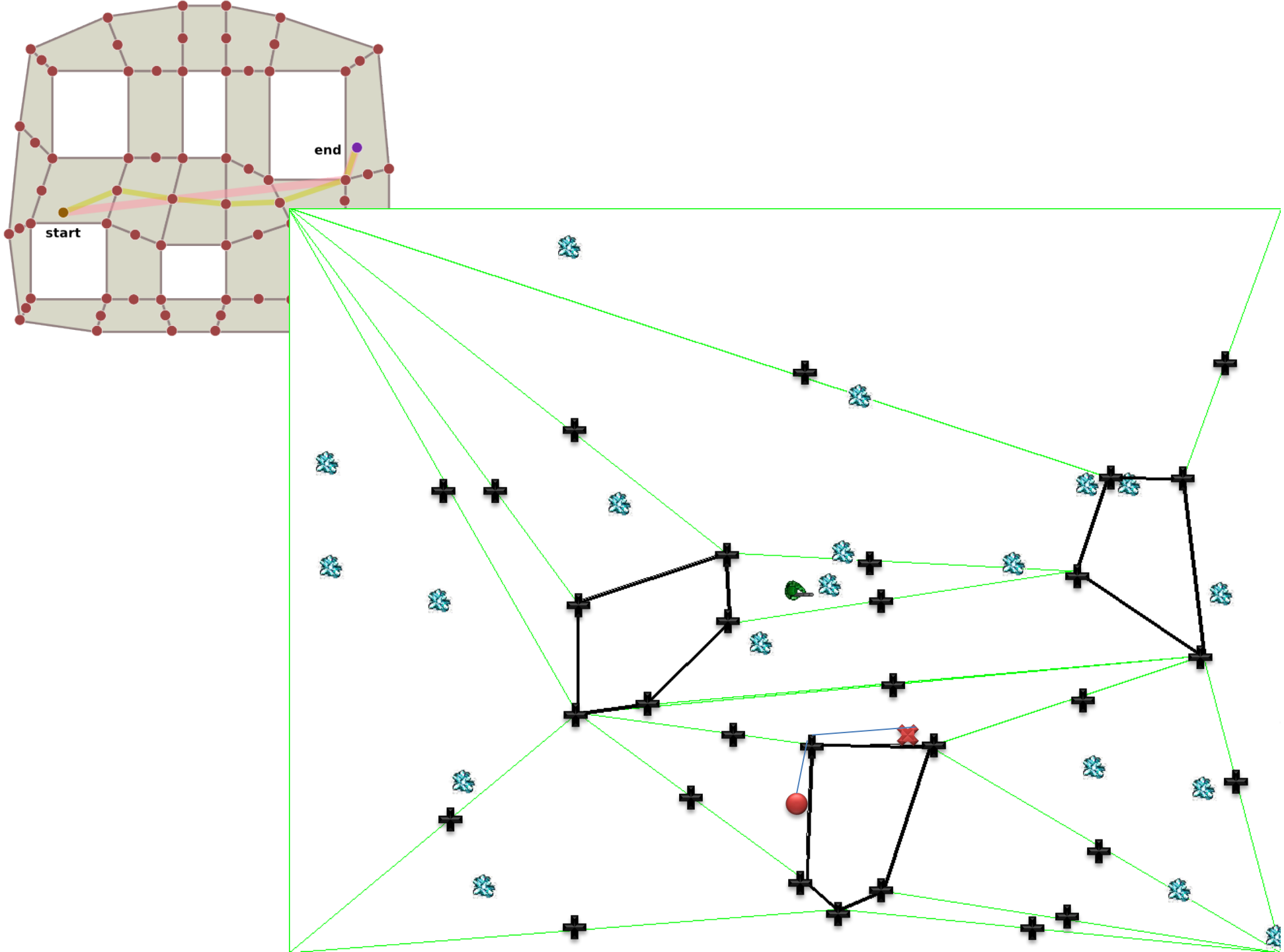


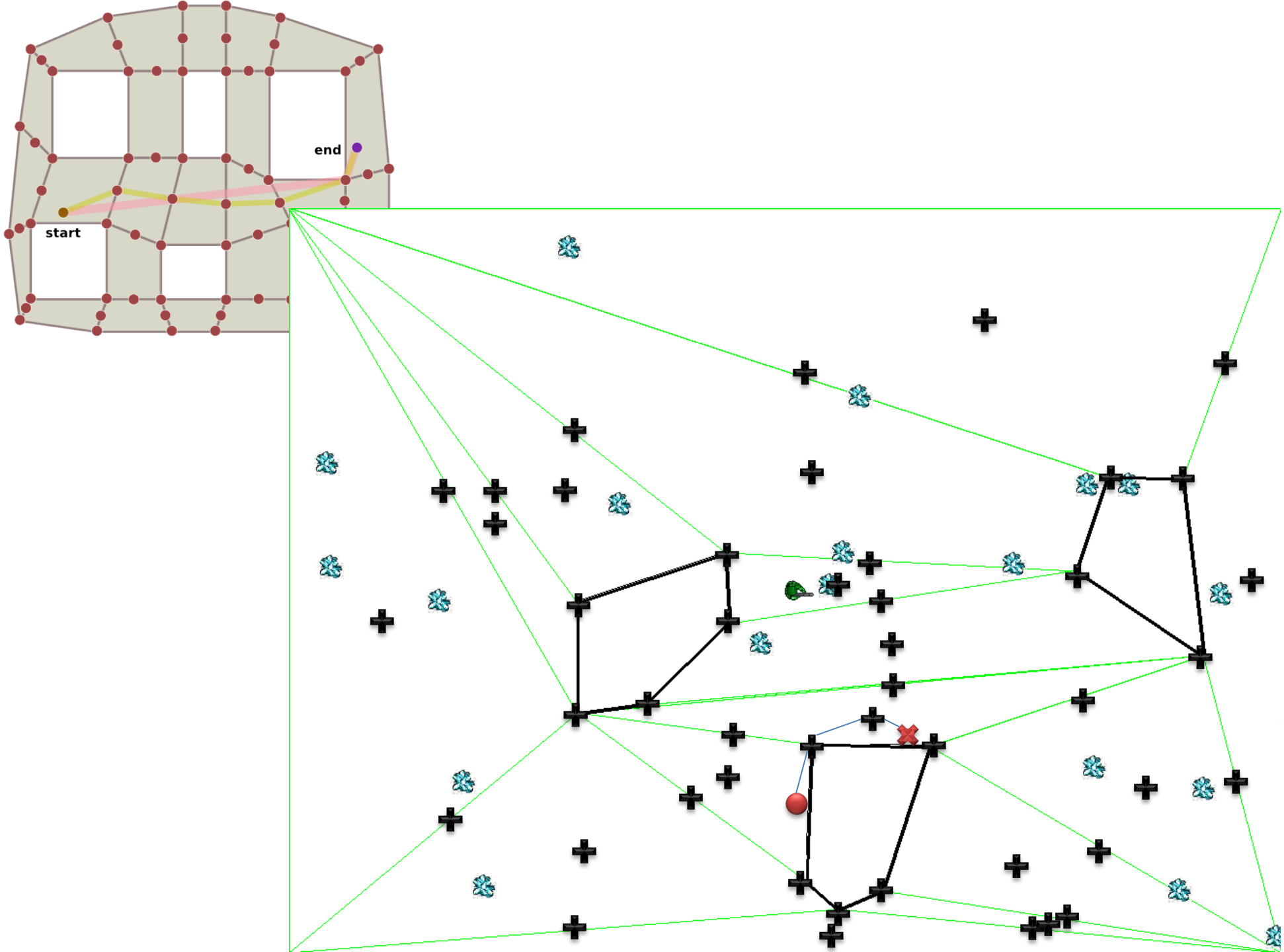



end
start


end
start


end
start


end
start

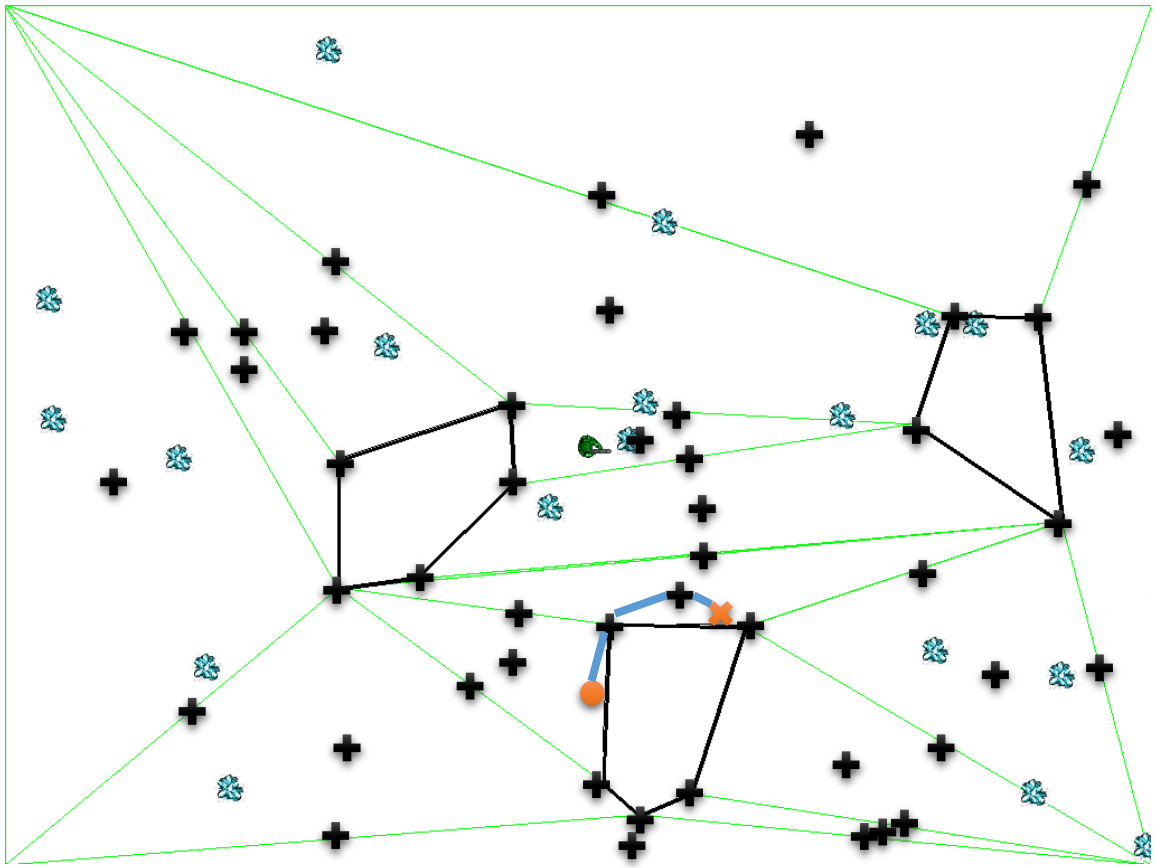

end
start

N-2: Grids, Path Networks

1. What's the intuition behind iterative deepening?
2. What are some pros/cons of grid navigation?
3. What are some benefits of path networks?
4. Cons of path networks?
5. What is the flood fill algorithm?
6. What is a simple approach to using path navigation nodes?
7. What is a navigation table?
8. How does the expanded geometry model work? Does it work with map gen features?
9. What are pros and cons of expanded geometry?

N-1: Nav Mesh

1. What are the major wins of a Nav Mesh?
2. How do we ensure convexity?
3. Would you calculate an optimal nav-mesh?
4. Do we still need waypoints? If so, where to place them?



Path finding models

1. **Tile-based graph – “grid navigation”**
   - Simplest topography
   - assume static obstacles
   - imaginary latice of cells superimposed over an environment such that an agent can be in one cell at a time.
   - Moving in a grid is relatively straightforward: from any cell, an agent can traverse to any of its four (or eight) neighboring cells
2. Path Networks / Points of Visibility NavGraph
3. Expanded Geometry
4. NavMesh

Path finding models

1. Tile-based graph – “grid navigation”
2. **Path Networks / Points of Visibility NavGraph**
   - 2 tier nav: Continuous, non-grid movement in local area
   - does not require the agent to be at one of the path nodes at all times. The agent can be at any point in the terrain.
   - When the agent needs to move to a different location and an obstacle is in the way, the agent can move to the nearest path node accessible by straight-line movement and then find a path through the edges of the path network to another path node near to the desired destination.
3. Expanded Geometry
4. NavMesh

Path finding models

1. Tile-based graph – “grid navigation”
2. Path Networks / Points of Visibility NavGraph
3. **Expanded Geometry**
   - Discretization of space can be smaller
   - 2 tier nav: Continuous, non-grid movement in local area
   - Can work with auto map generation
   - Can plan nicely with “steering behaviors”
4. NavMesh

Path finding models

1. Tile-based graph – “grid navigation”
2. Path Networks / Points of Visibility NavGraph
3. Expanded Geometry
4. **NavMesh**
   - Win: compact rep, fast search, auto create
   - Each node (list of edges) is a convex polygon
   - Convex = Any point w/in polygon is unobstructed from any other
   - Can be generated from the polygons used to define a map

Generating the Mesh: Greedy/Simple Approach

For point a in world points:

    For point b in world points:

        For point c in world points:

            if (it is a valid triangle) and !exists:

                add triangle to mesh

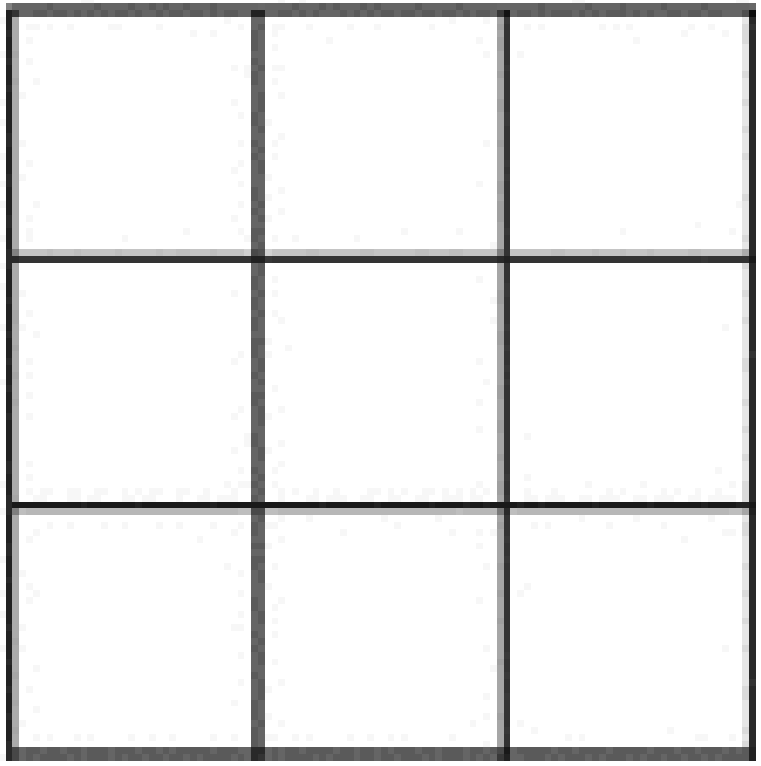
Iterate through triangles to merge to quads

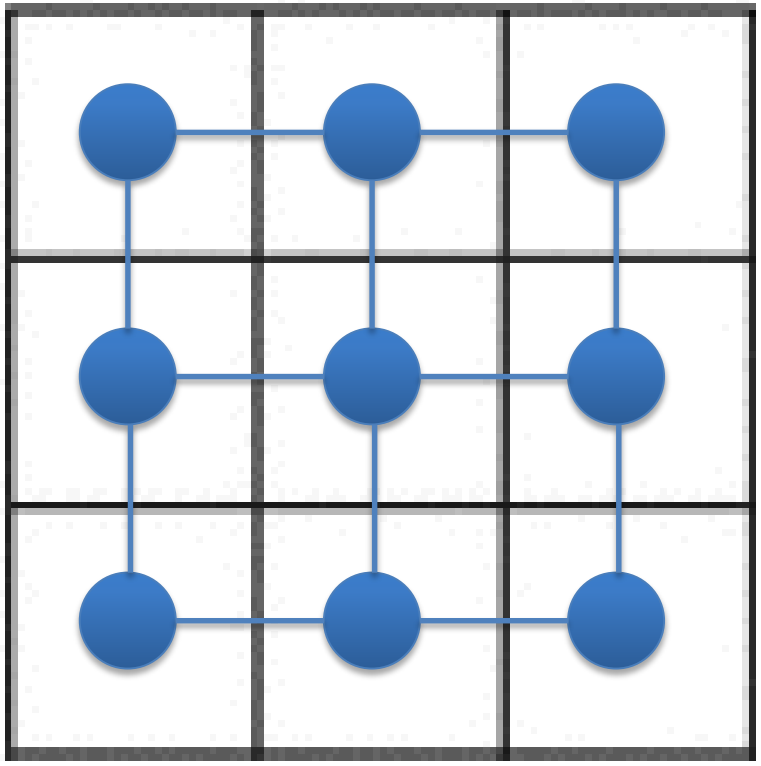
Iterate through...

On Convexity

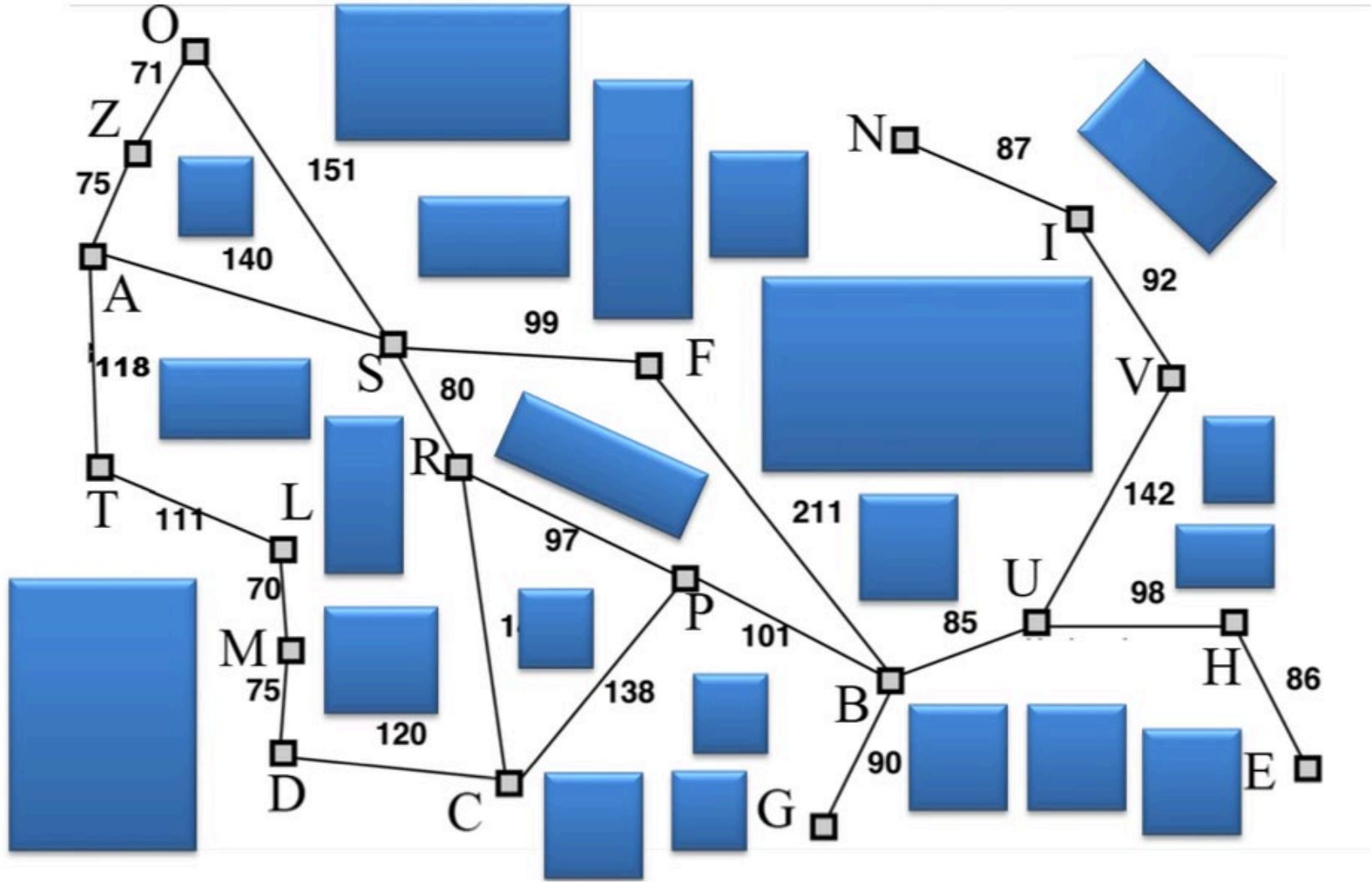
- by definition: convex polygons have all internal angles of less than 180 degrees
  - all diagonals are contained within the polygon
  - a line drawn through a convex polygon in any direction will intersect at exactly two points
- utils.py has isConvex(points): returns true if all the angles made up by a list of points are convex. Points is a list (point1, point2, ..., pointN) and a point is a tuple (x, y). The list must contain at least three points.

Grid as Graph



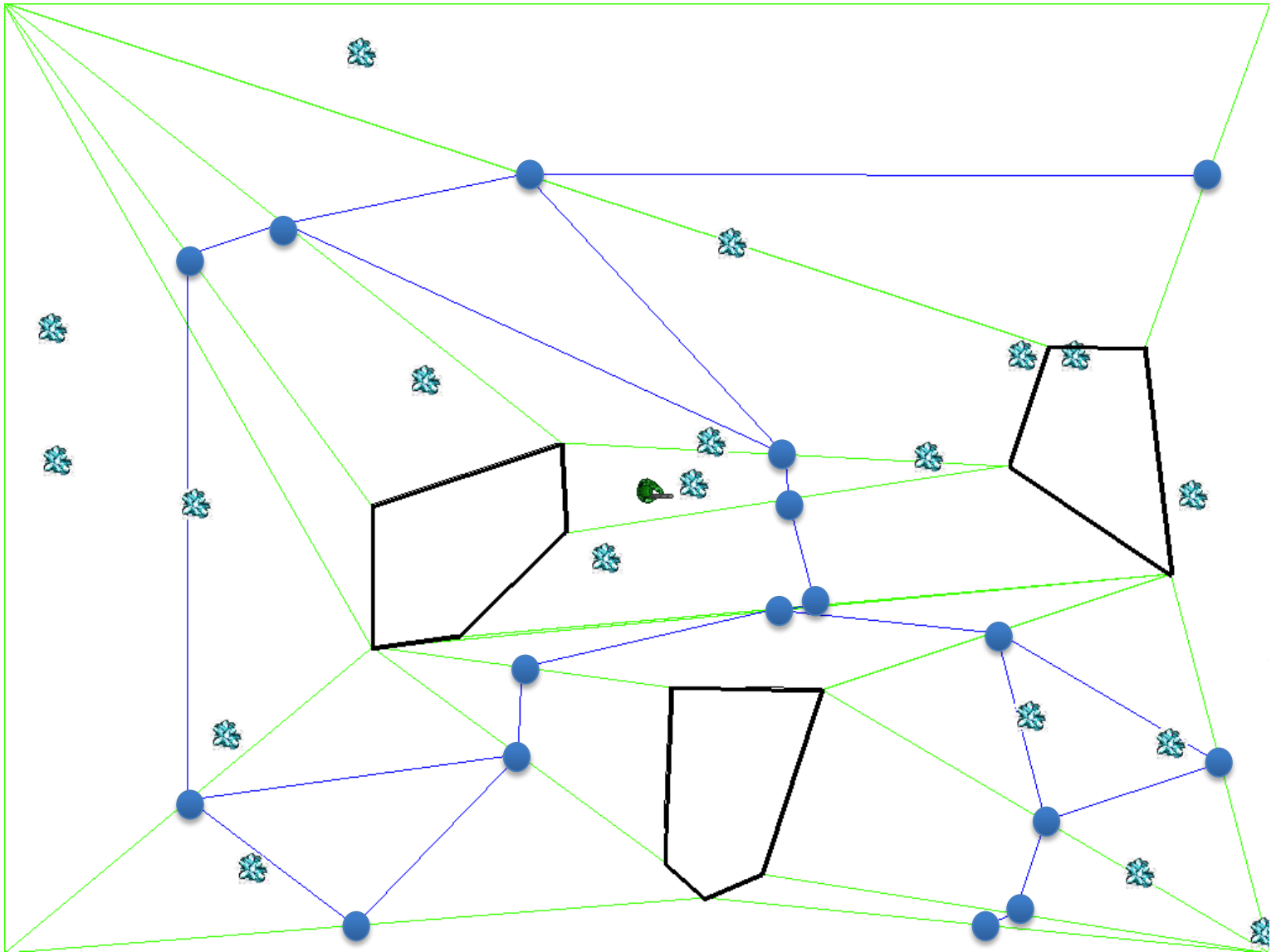


Path Network as Graph



Nav Mesh as Graph

(well actually path network again)



Why talk about these as graphs?

- Standard, abstract way to discuss different spatial representations
- Allows for quantifiable comparison between different spatial representations (e.g. number of edges/nodes)
- Allows us to discuss different search approaches without worrying about the exact spatial representation

Path finding problem solved, right?

- Compilation
  - http://www.youtube.com/watch?v=lw9G-8gL5o0
- Sim City (1, 2 … 5)
  - https://www.youtube.com/watch?v=zHdyzx_ecbQ
- Half-Life 2
  - http://www.youtube.com/watch?v=WzYEZVI46Uw
- Fable III
- DOTA (Defense of the ancients) 1+2
  - https://www.youtube.com/watch?v=p585DHI0qh4
- WoW (World of Warcraft)
- Minecraft Bedrock Edition
  - https://www.youtube.com/watch?v=qR5M5v0XDM0
- Fallout 4
  - https://www.youtube.com/watch?v=M7TicvLXrQo
- DARPA robotics challenge:
  - https://www.youtube.com/watch?v=g0TaYhjpOfo

Excellent slides now available here: https://cs.stanford.edu/people/abisee/gs.pdf

Supporting interactive animations: https://cs.stanford.edu/people/abisee/tutorial

We can also thank Stanford for A*

GRAPH SEARCH – PATH PLANNING

Path Planning Algorithms

- Must Search the state space to move NPC to goal state
- Computational Issues:
  - Completeness
    - Will it find an answer if one exists?
  - Time complexity
  - Space complexity
  - Optimality
    - Will it find the best solution

Search Strategies

- Blind search
  - No domain knowledge.
  - Only goal state is known
- **Heuristic search**
  - Domain knowledge represented by **heuristic rules**
  - Heuristics drive low-level decisions
  - Video games provide domain knowledge that can be leveraged!

Graph Search: Sorting Successors

- Uninformed / Blind (no domain knowledge; all nodes are same)
  - DFS (stack – lifo), BFS (queue – fifo)
  - Iterative-deepening DFS (Depth-limited)
- Informed / Heuristic (pick order of node expansion w/ heuristic)
  - Greedy Best First
  - Dijkstra – guarantee shortest path ($E\log_2 N$)
  - Floyd-Warshall
  - A* (IDA*)…. Dijkstra + heuristic
  - D*
- Hierarchical can help (run speed)

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

Path Planner

- Initial state (cell), Goal state (cell)
- Each cell is a state agent can occupy
- Sort successors, try one at a time (backtrack)
- Heuristic: Manhattan or straight-line distance
- Each successor stores who generated it

What problem are all of these approaches solving?

or

What differentiates these approaches?

Pathfinding List (Open and Closed Sets)

- Critical Operations:
  - Adding an entry to the list
  - Removing an entry from the list
  - Finding the smallest element
  - Finding an entry in the list corresponding to a particular node (find() or contains())
- Must find a balance between these four operations for best performance

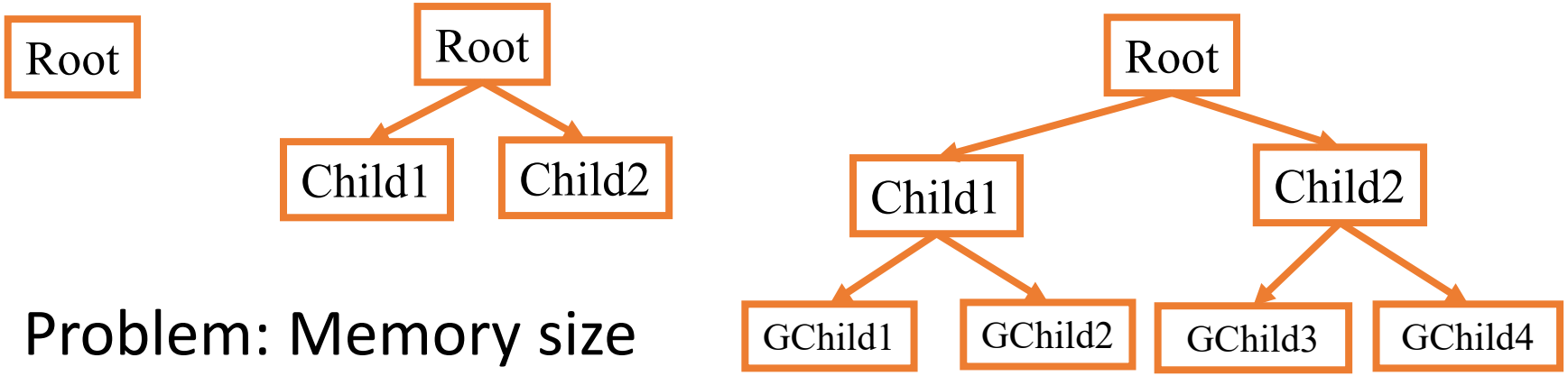
Pathfinding List

- Naïve: Simple linked list (finding may visit all elements)
- Priority Queue: Sorted List. Finding is efficient but cost to add increases
- Priority Heap: array-based tree structure. Smallest element head of tree. Remove smallest element and adding new element take *O(log n)*
- Bucketed Priority Queue: Sorted buckets of unsorted lists. Can be tuned for application

Uninformed

Breadth-First Search

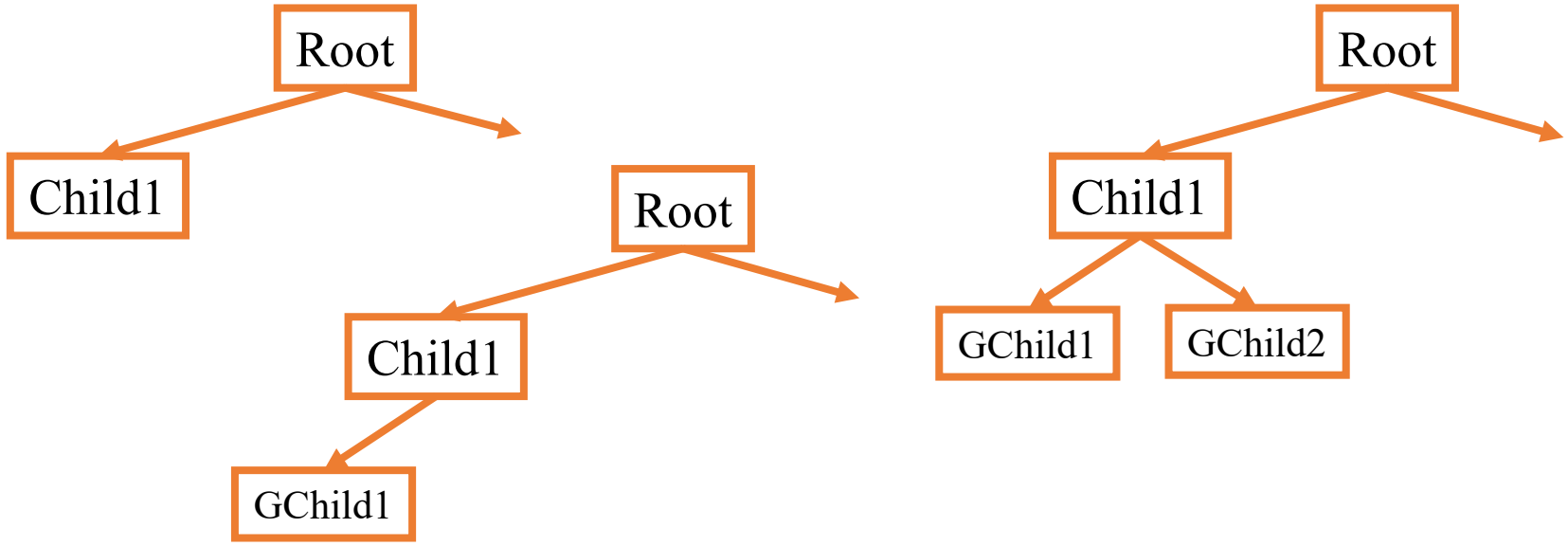
- Expand **Root node**
  - Expand all **Root node's children**
    - Expand all **Root node's grandchildren**



- Problem: Memory size

Depth First Search

- Always expand the node that is deepest in the tree
- Not best solution (if you stop at first found)



Iterative Deepening

- mid 1970s
- Idea: perform depth-limited DFS repeatedly, with an increasing depth limit, until a solution is found.
- Each repetition of depth-limited DFS needlessly duplicates all prior work?
  - Duplication is not significant because a branching factor b > 1 implies that the number of nodes at depth k exactly is much greater than the total number of nodes at all depths k-1 and less.
  - That is: most nodes are in the bottom level.
- The space required by DFS is O(tree depth); BFS is O(#tree nodes ~= $branching^{depth}$); IDDFS O(depth * branching factor)

Number of nodes

- Full, complete, balanced binary tree, height *h*
  - Total number of nodes *N* = 2^{h+1} – 1
    - Leaf nodes at height 0: 2^0 = 1
    - Leaf nodes at height 1: 2^1 = 2
    - Leaf nodes at height 2: 2^2 = 4
    - Leaf nodes at height 3: 2^3 = 8
    - N = 1 + 2 + 2^2 + 2^3 + … + 2^h
      = (2^{h+1} – 1) / (2 – 1) = 2^{h+1} - 1
  - Number of leaves *L* = 2^h
  - Height 42, *N = 8,796,093,022,207*
    L = 4,398,046,511,104

Informed

Heuristics

- [dictionary]*"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."*
- *h(n)* = estimated cost of cheapest path from n to goal (with goal == 0)

Heuristic Search

- Find shortest path from a single source to a single destination
- Heuristic function:
  - We have some knowledge about how far away any given state from the goal, in terms of operation cost
  - For navigation: Euclidean distance, Manhattan distance

Greedy Search

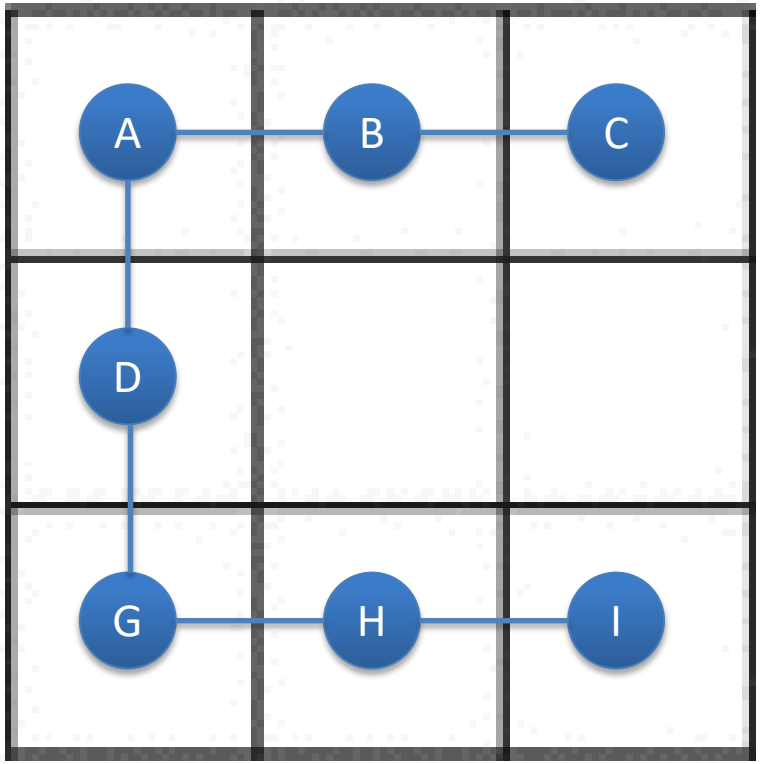
- Expand the node that yields the minimum cost
  - Expand the node that is closest to target
  - Depth first
  - Minimize the function *h(n)* the heuristic cost function
- Not Complete!
- “Greedy” implies that working solution is not revised
  - Note, A* is best-first, but not greedy (incorporates distance from start)
- Local Minima/Maxima (dead end)

- https://cs.stanford.edu/people/abisee/tutorial

Greedy Algorithm Review

Find a path from start to goal node

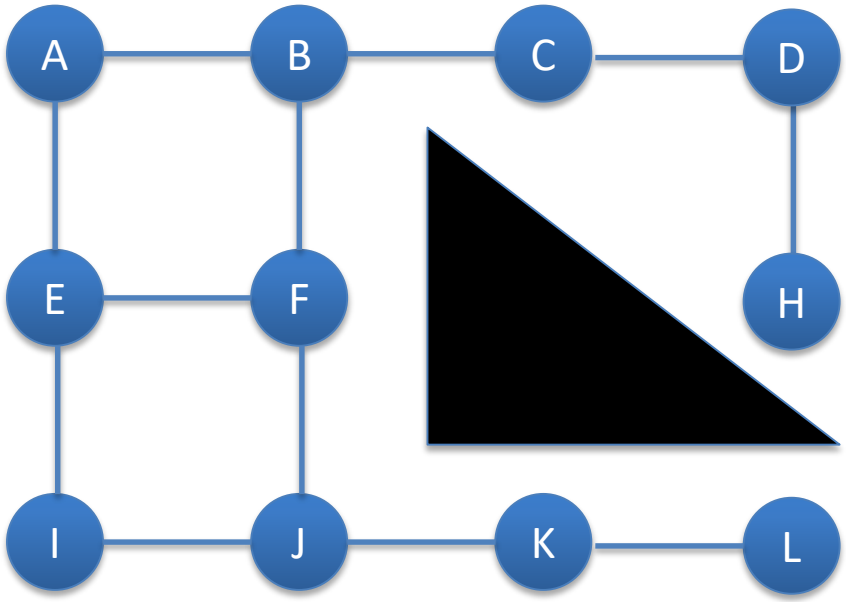
1. Add the neighbors of the current node to some open set list
   - We can get here!
2. Pick next current node from open set
3. If next node is goal, backtrack to start for path

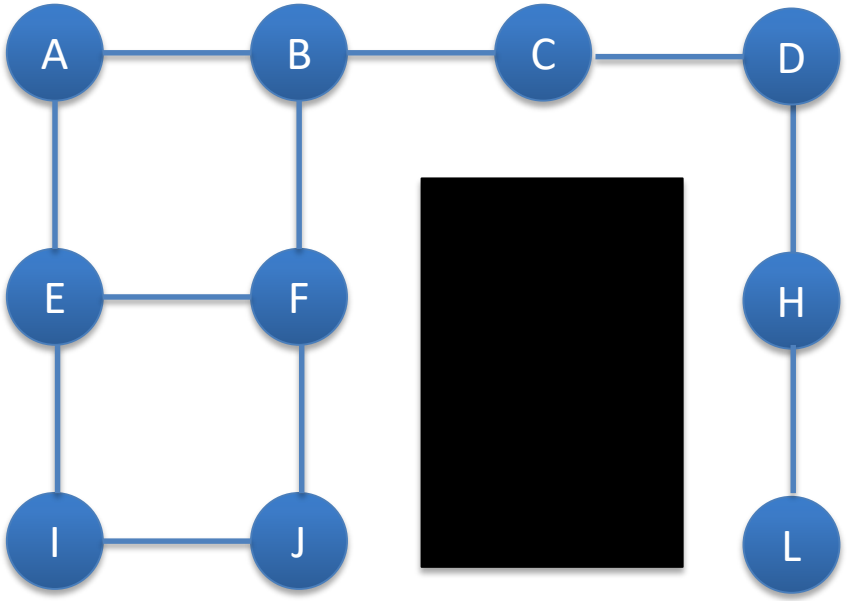


Question

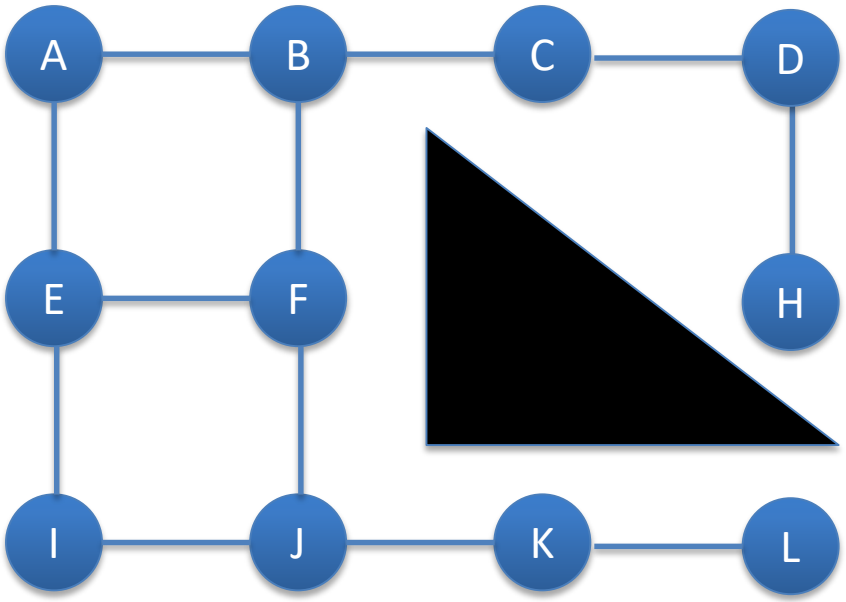
What heuristic could be used to get from B to L in both graphs the fastest?

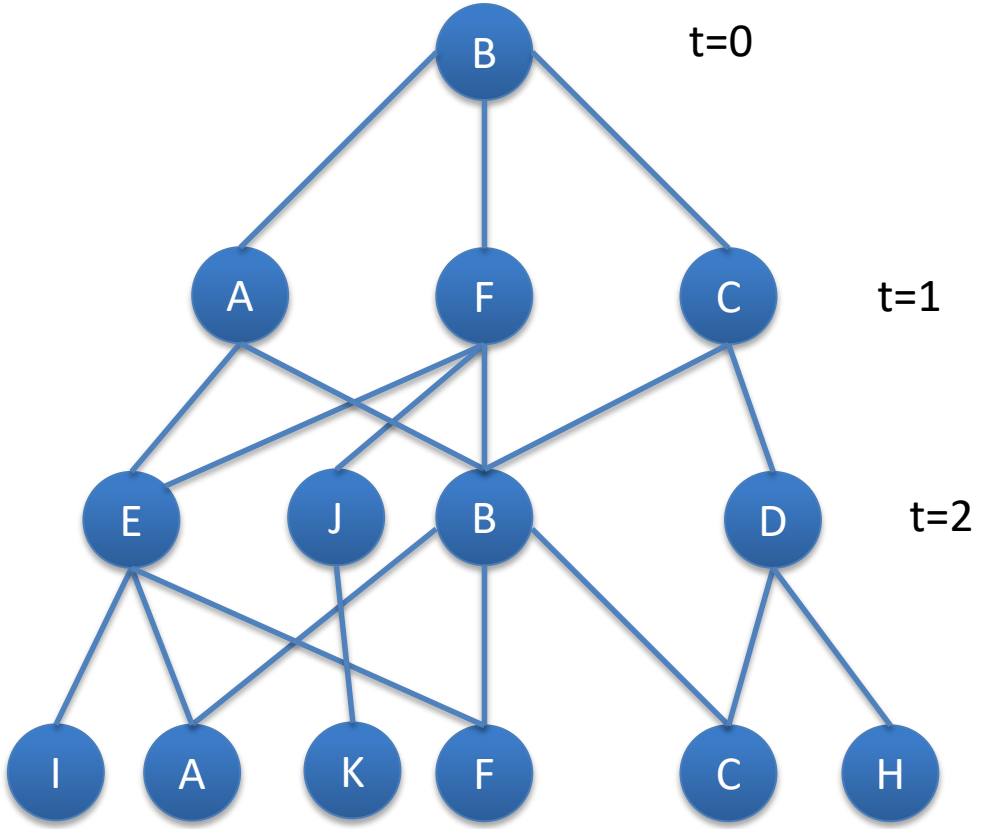
- Fastest meaning with fewest current nodes chosen





Greedy as a tree

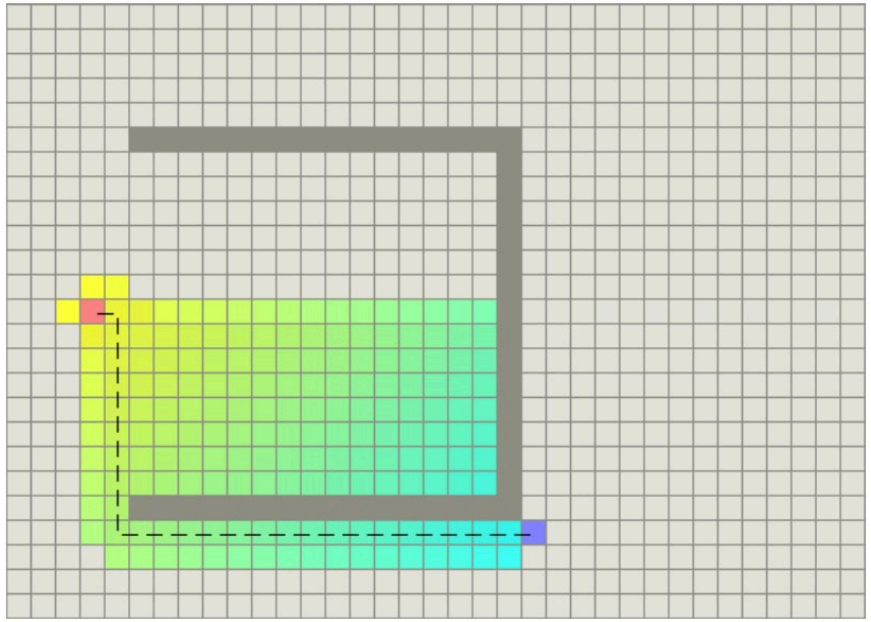




Improvement over Greedy

- Beyond improving the heuristic, how can we improve the greedy pathing algorithm?

- When does it fail?



A*

- Won’t just have an open set, but also a closed set (nodes already evaluated)
- Open set will be a priority queue, so if we discover a better node we can immediately pick it

- Priority Queue: A queue that automatically sorts itself so minimum cost is at the top

A* Search

- 1968: Single source, single target graph search
- Guaranteed to return the optimal path if the heuristic is admissible
- Evaluate each state: f(n) = g(n) + h(n)
- Open list: nodes that are known and waiting to be visited
- Closed list: nodes that have been visited

Heuristic Function for A*

- Computational performance is important
- Underestimate completely (0 heuristic) is Dijkstra!
- Perfect Heuristic: A* would go straight to correct node *O(p)* (but such a heuristic solves for exactly what we are looking for in the first place!)
- Overestimate: May not return the best path

A* Search

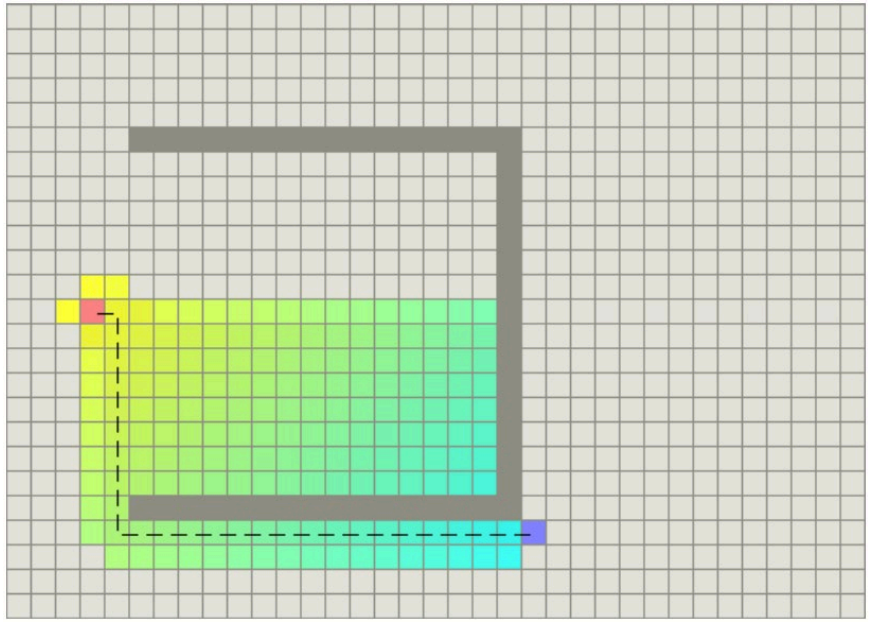
- A* is optimal…
- …but only if you use an **admissible** heuristic
- An admissible heuristic is mathematically guaranteed to underestimate the cost of reaching a goal
- What is an admissible heuristic for path finding on a path network?

Admissible Heuristic

- An *admissible heuristic* is one that guarantees that the shortest path can be found with the search because it *never overestimates the cost of reaching the goal*
  - A heuristic that does not overestimate is *admissible*
  - Otherwise we say a heuristic is *inadmissible*
- Euclidean Distance is *admissible*
- In games: perfectly acceptable to use either *admissible* or *inadmissible*
  - **"Overestimates can make A* faster if they are almost perfect but home in on the goal more quickly"** – M&F

A*

- Nodes will have two costs:
  - G score: Cost from getting from start to here
  - H score: Estimated cost of getting from here to goal
  - F score: G+H
- We will pick which node to choose next based on both of these scores

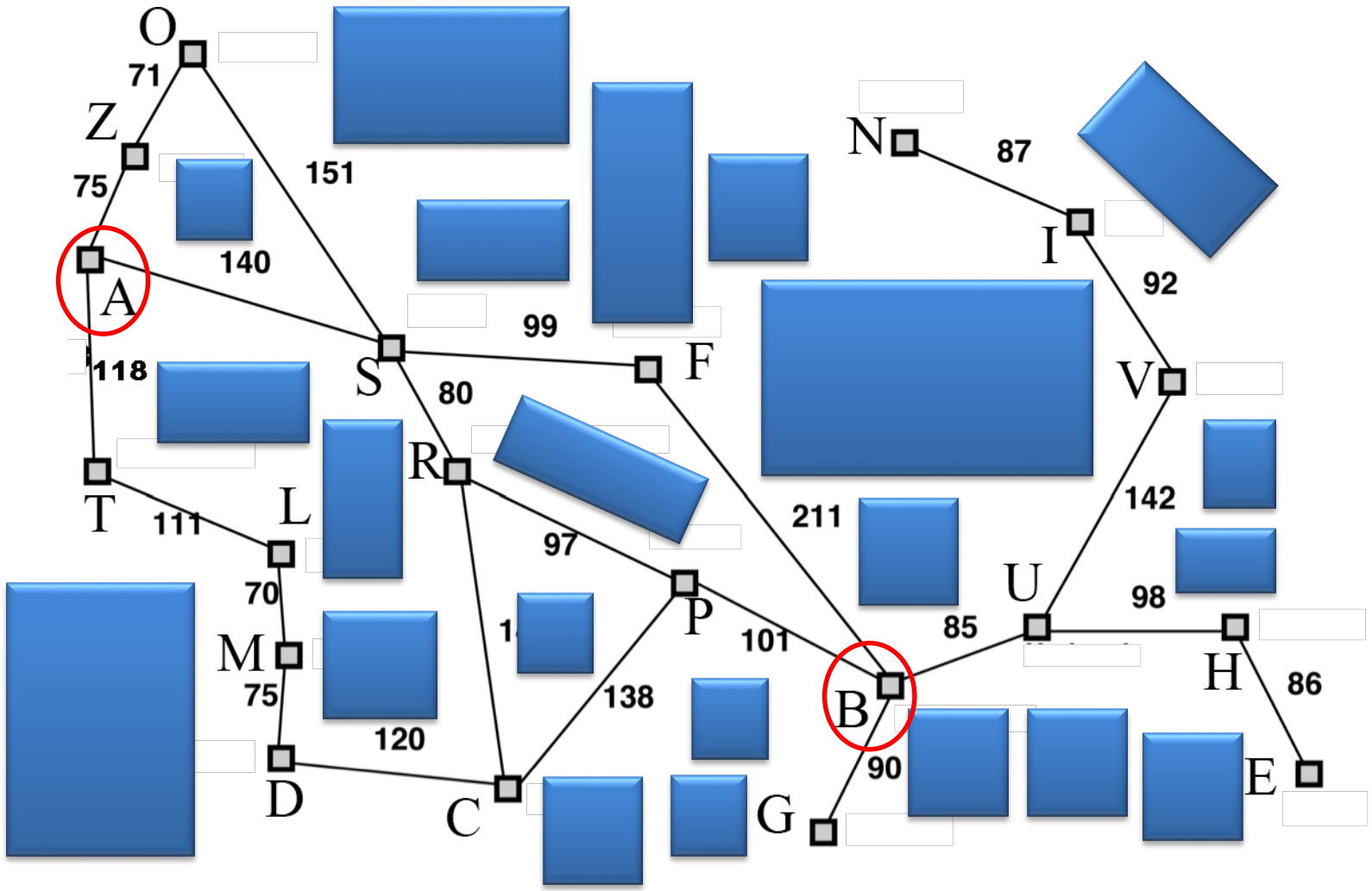


A*

```
add start to openSet
while openSet is not empty:
        current = openSet.pop()
        if current == goal:
                return reconstruct_path(current)
        closedSet.Add(current)
        for each neighbor of current:
                if neighbor in closedSet:
                        continue
                gScore = current.gScore + dist(current, neighbor)
                if neighbor not in openSet:
                        openSet.add(neighbor)
                else if gScore< openSet.get(neighbor).gScore
                        openSet.replace(openSet.get(neighbor), neighbor)
```

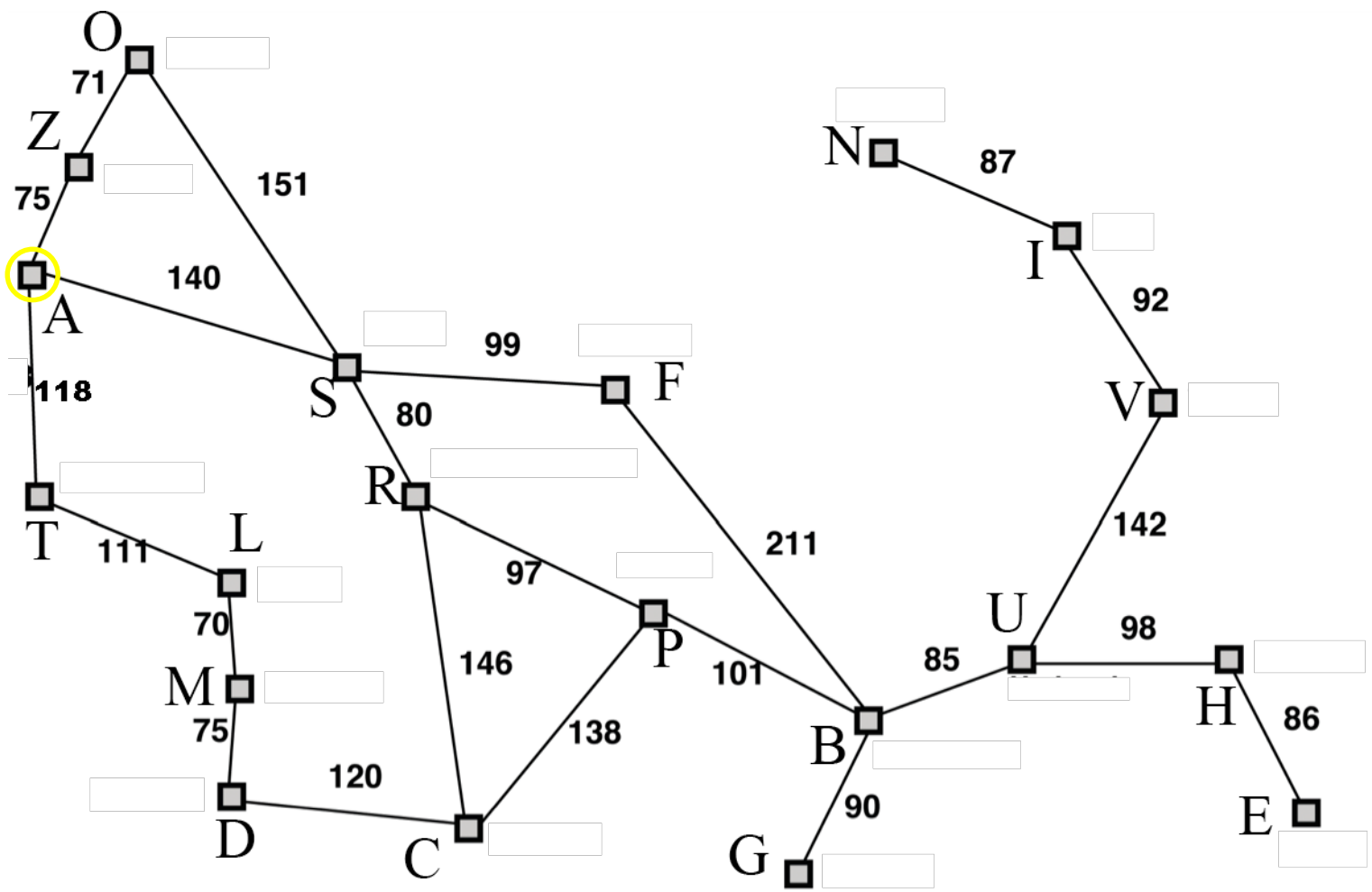
Heuristic Distance, A→B

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374



Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374

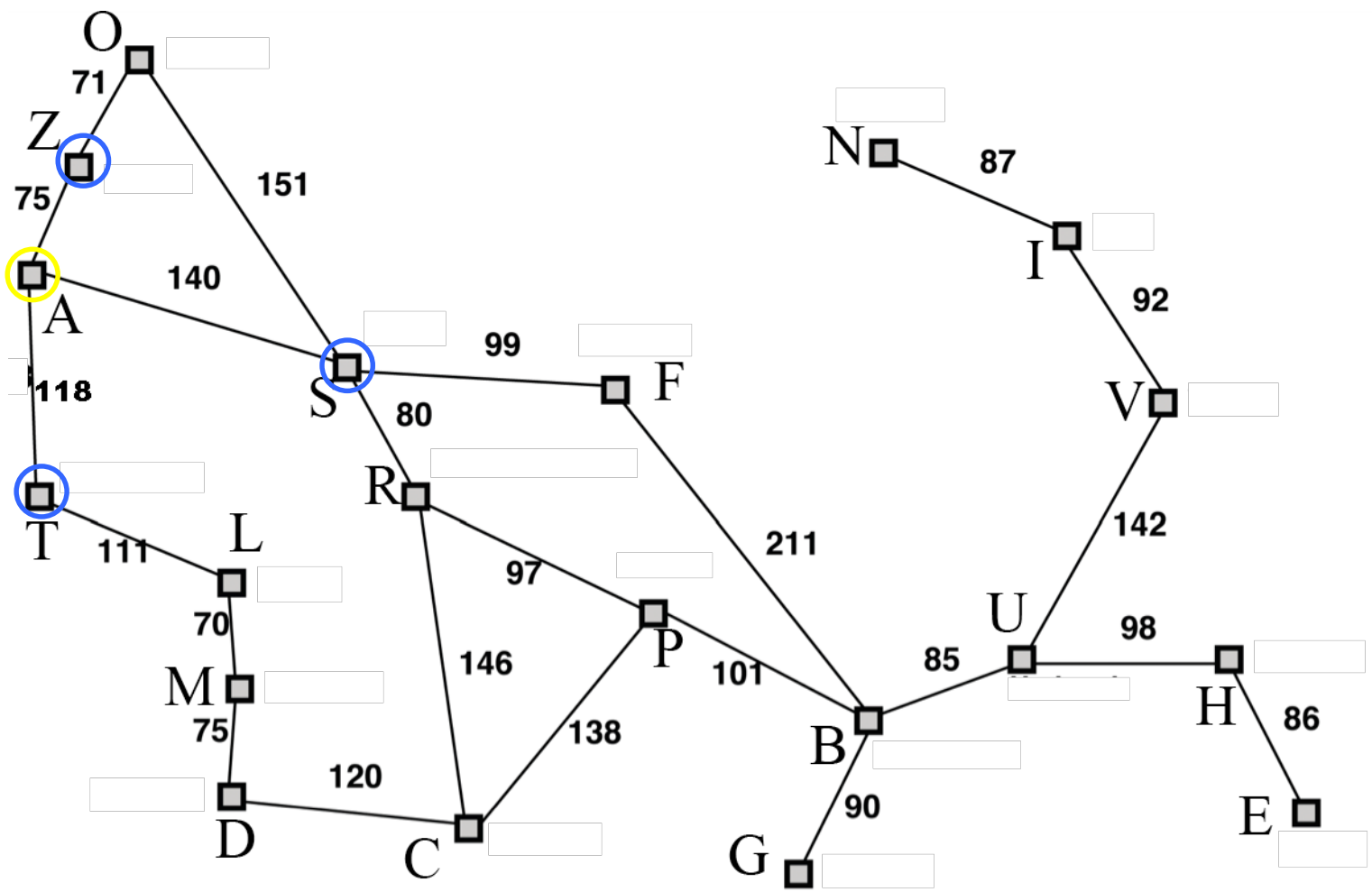


Open: A(366)

Closed:

Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374

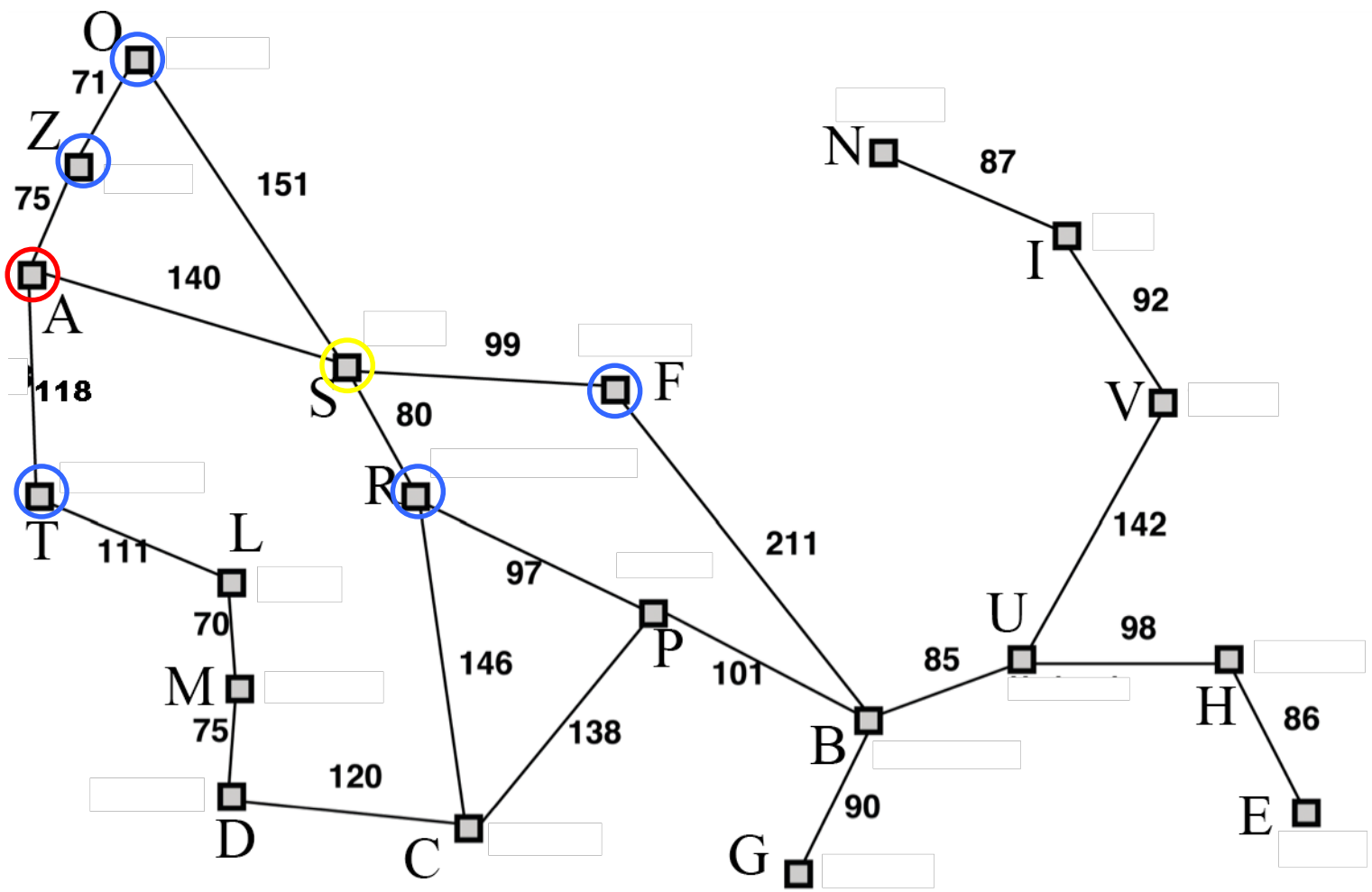


Open: S(253+140=393), T(329+118=447), Z(374+75=449)

Closed: A(366)

Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374

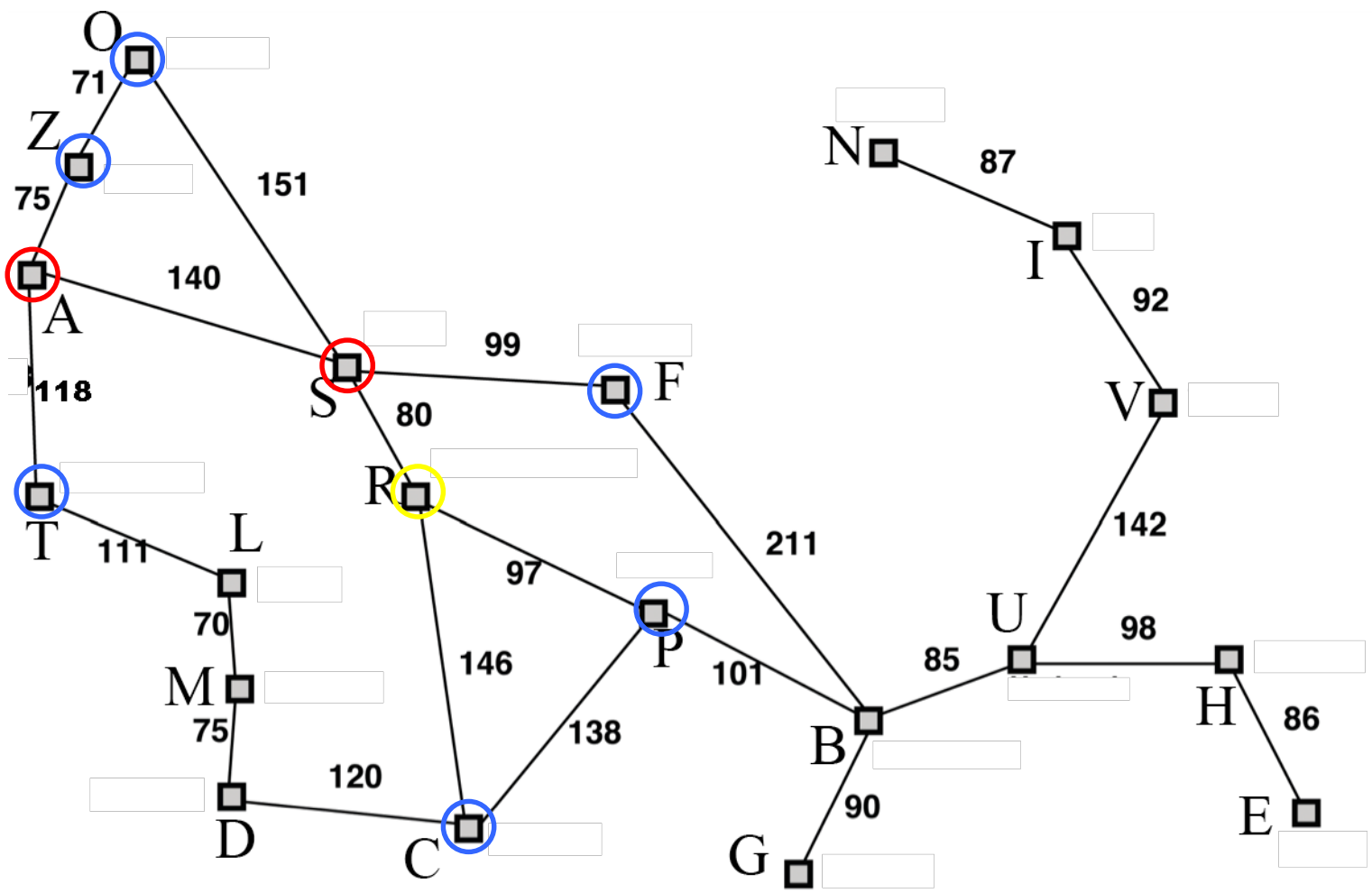


Open: R(220+193=413), F(239+176=415), T(329+118=447), Z(374+75=449), O(291+380=671)

Closed: S(393), A(366)

Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374

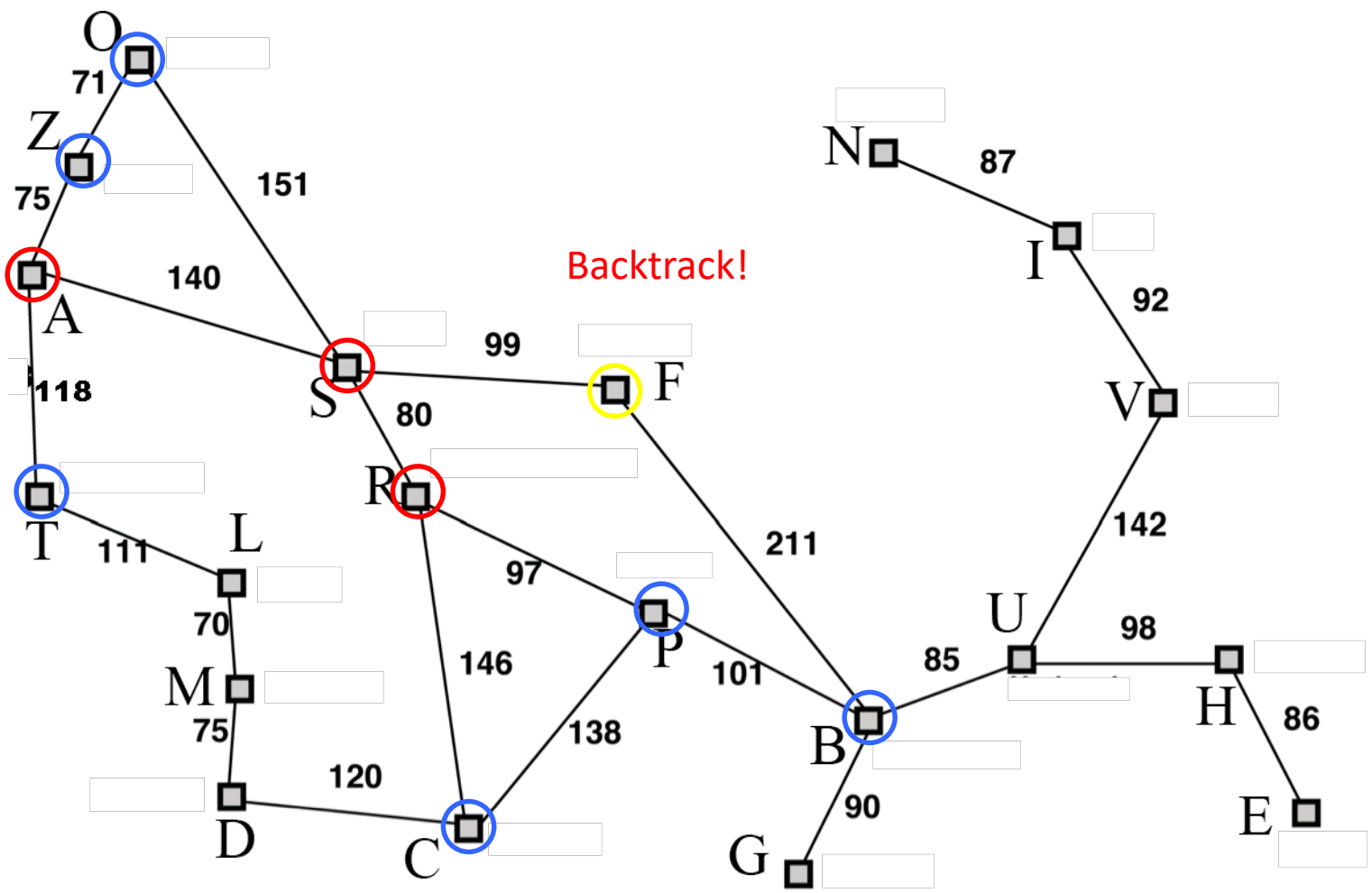


Open: F(239+176=415), P(317+100=417), T(329+118=447), Z(374+75=449), C(366+160=526), O(291+380=671)

Closed: R(413), S(393), A(366)

Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374

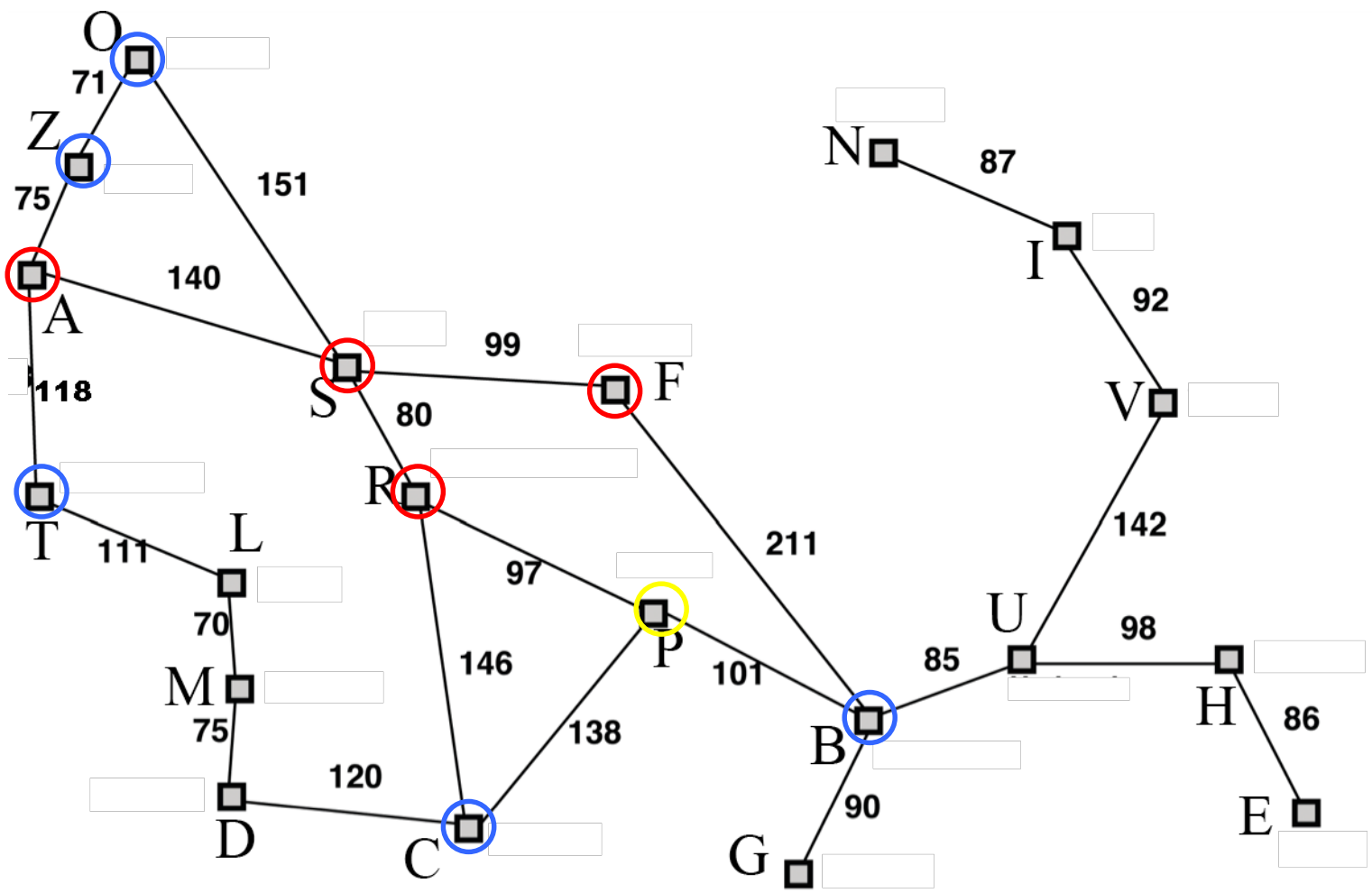


Open: P(317+100=417), T(329+118=447), Z(374+75=449), B(450+0=450), C(366+160=526), O(291+380=671)

Closed: F(415), R(413), S(393), A(366)

Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374

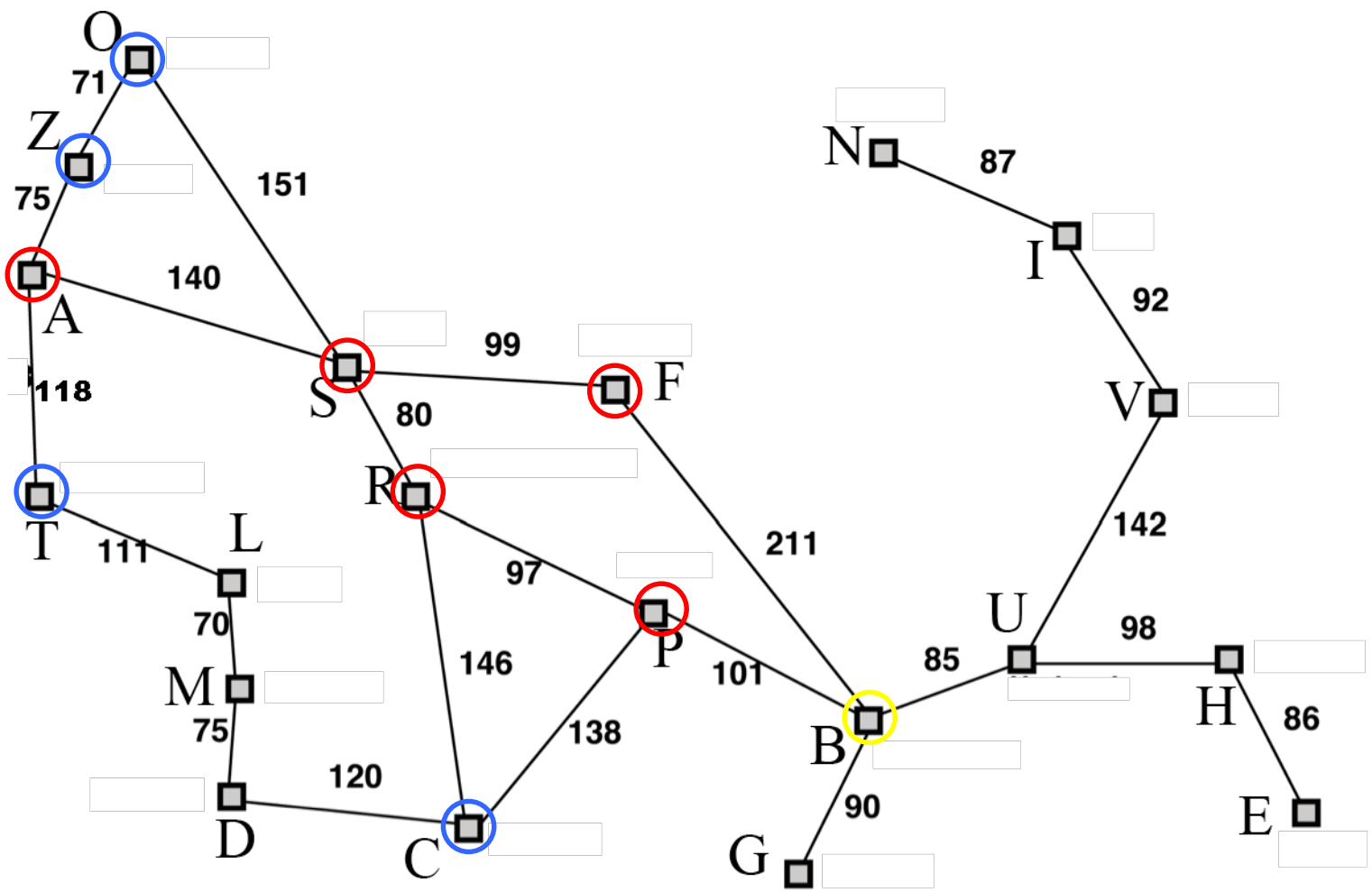


Open: B(418+0=418), T(329+118=447), Z(374+75=449), C(366+160=526), O(291+380=671)

Closed: P(417), F(415), R(413), S(393), A(366)

Evaluation function f(n) = g(n) + h(n)

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
S	253
T	329
U	80
V	199
Z	374



Solution: A-S-R-P-B

Open: T(329+118=447), Z(374+75=449), C(366+160=526), O(291+380=671)

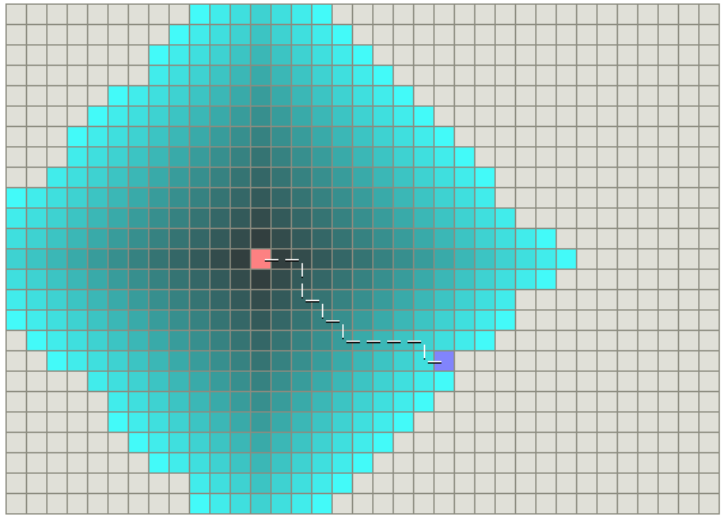
Closed: B(418), P(417), F(415), R(413), S(393), A(366)

Non-admissible heuristics

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

A*

Dijkstra’s Algorithm



Dijkstra's algorithm

- 1956: A single-source, multi-target shortest path algorithm
- Tells you path from any one node to all other nodes
- 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)
- Like BFS for weighted graphs; if all costs equal, Dijkstra = BFS

Given: G=(V,E), source

For each vertex v in G: set dist[v] to infinity, set parent[v] to null
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[], parent[]

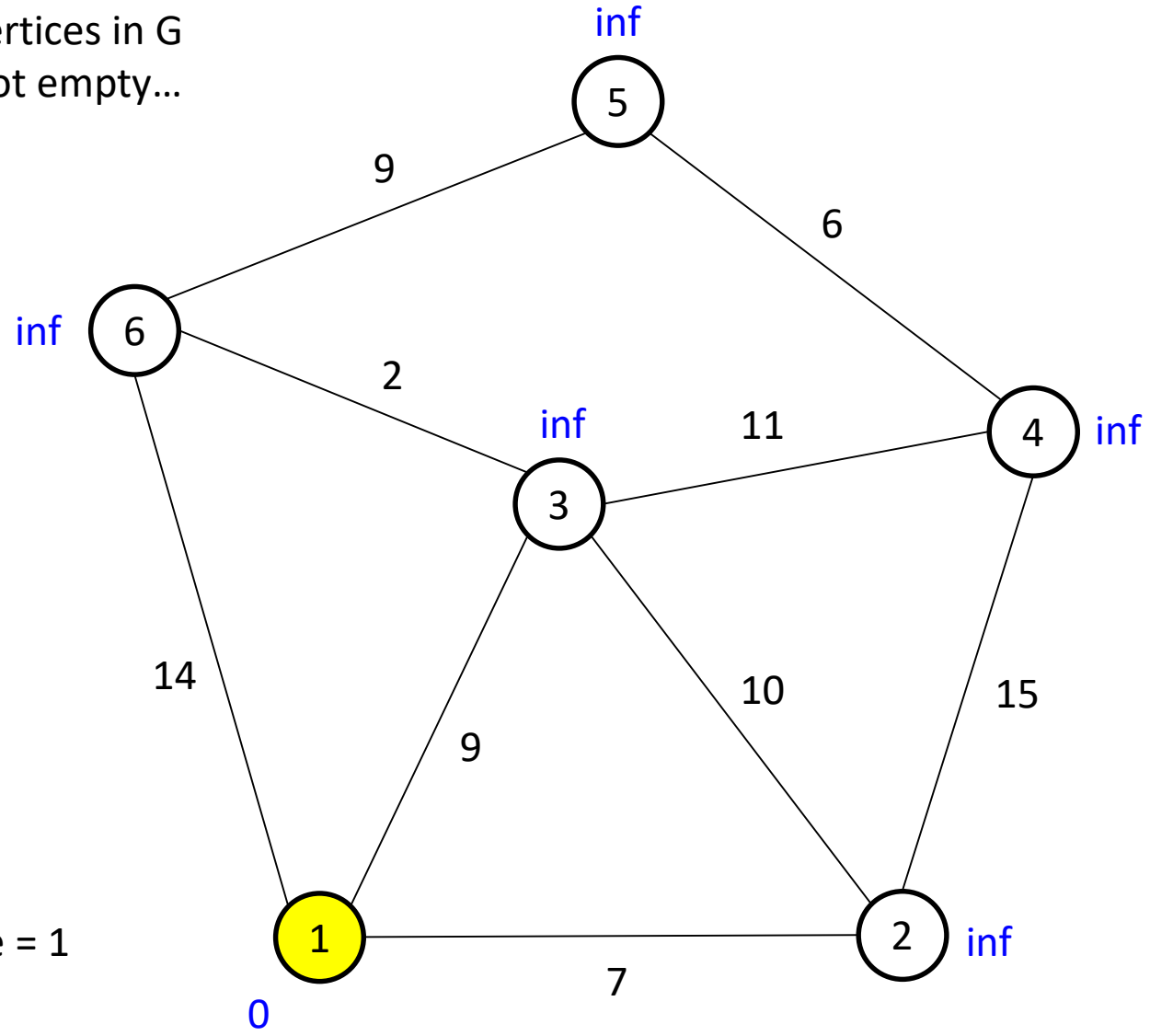
Source: wikipedia

Path reconstruction

Given: parent[], target

Set path = empty stack
Set curnode = target
while parent[curnode] is defined:
    path.push(curnode)
    curnode = parent[curnode]
path.push(curnode)

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	

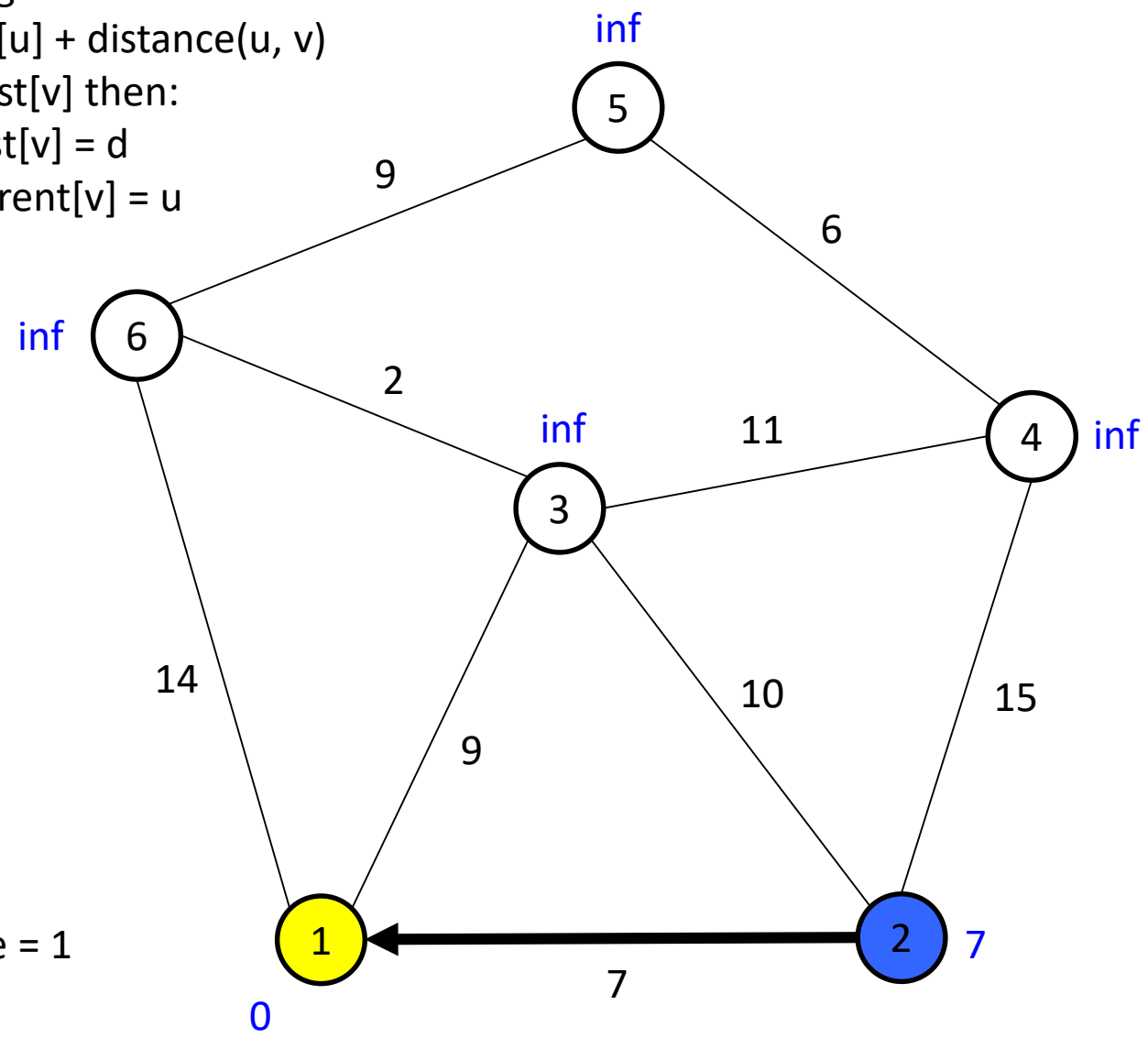
Q=[1,2,3,4,5,6]
U=1

* Source = 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



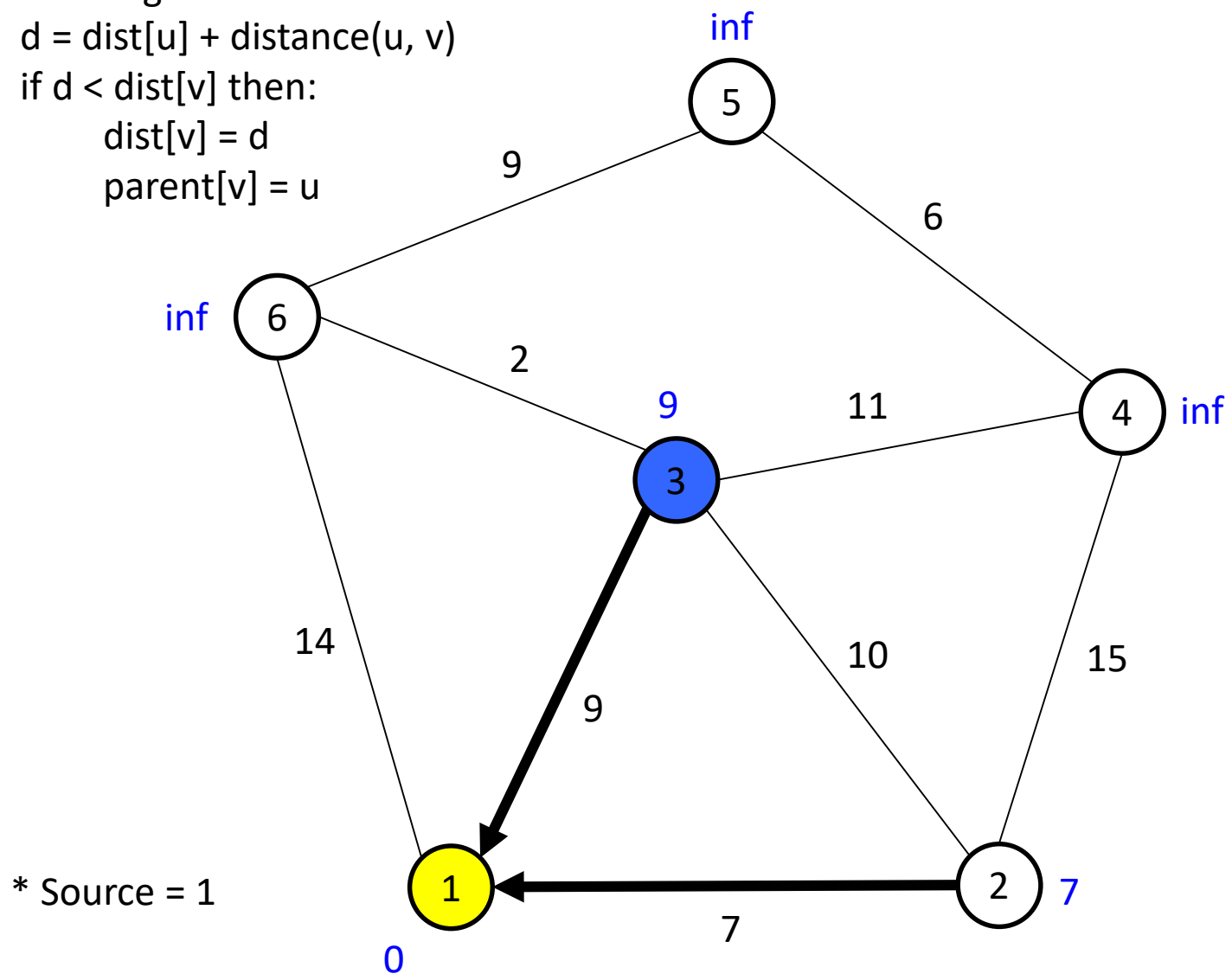
* Source = 1

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]

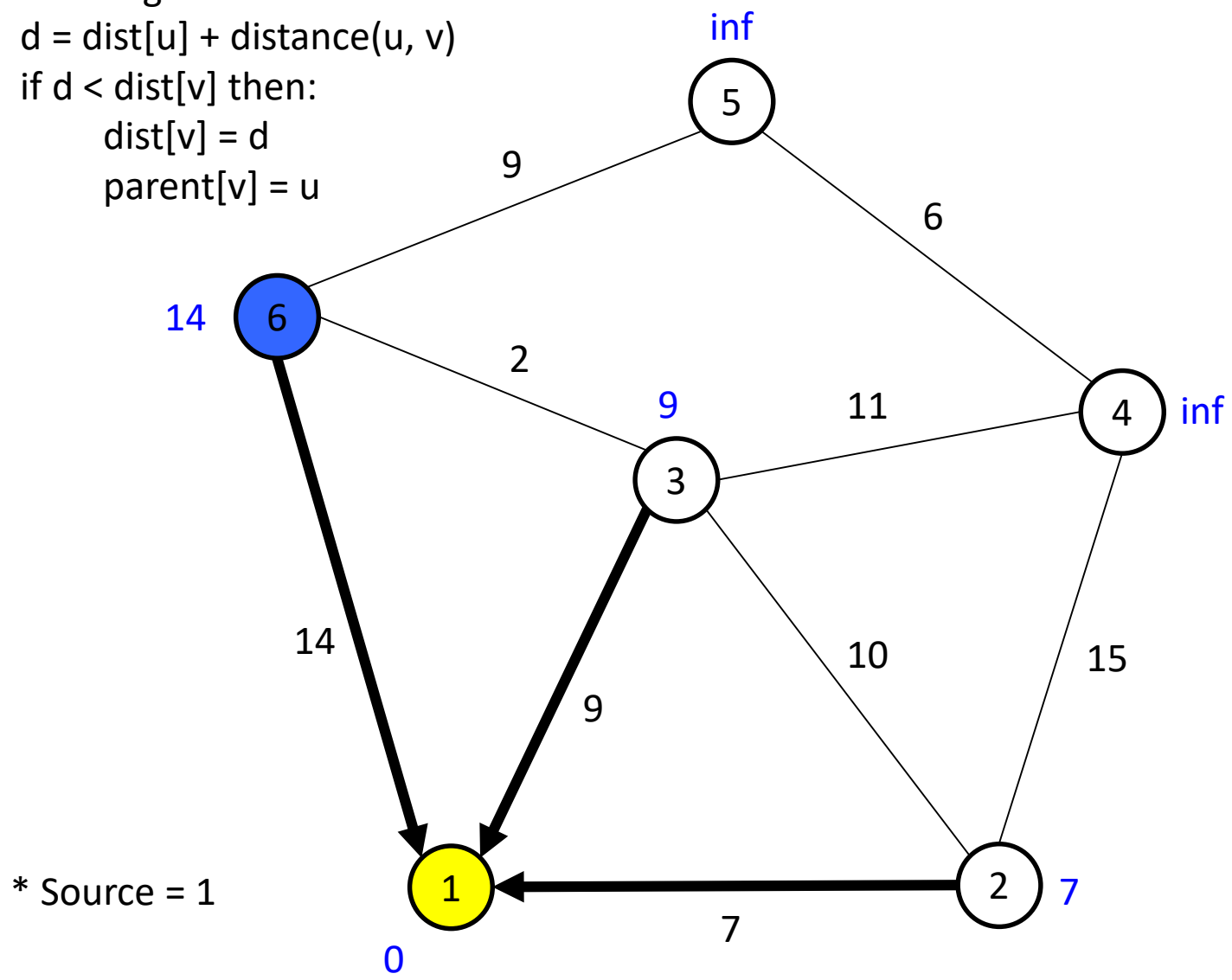
Remove u (node 1) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



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

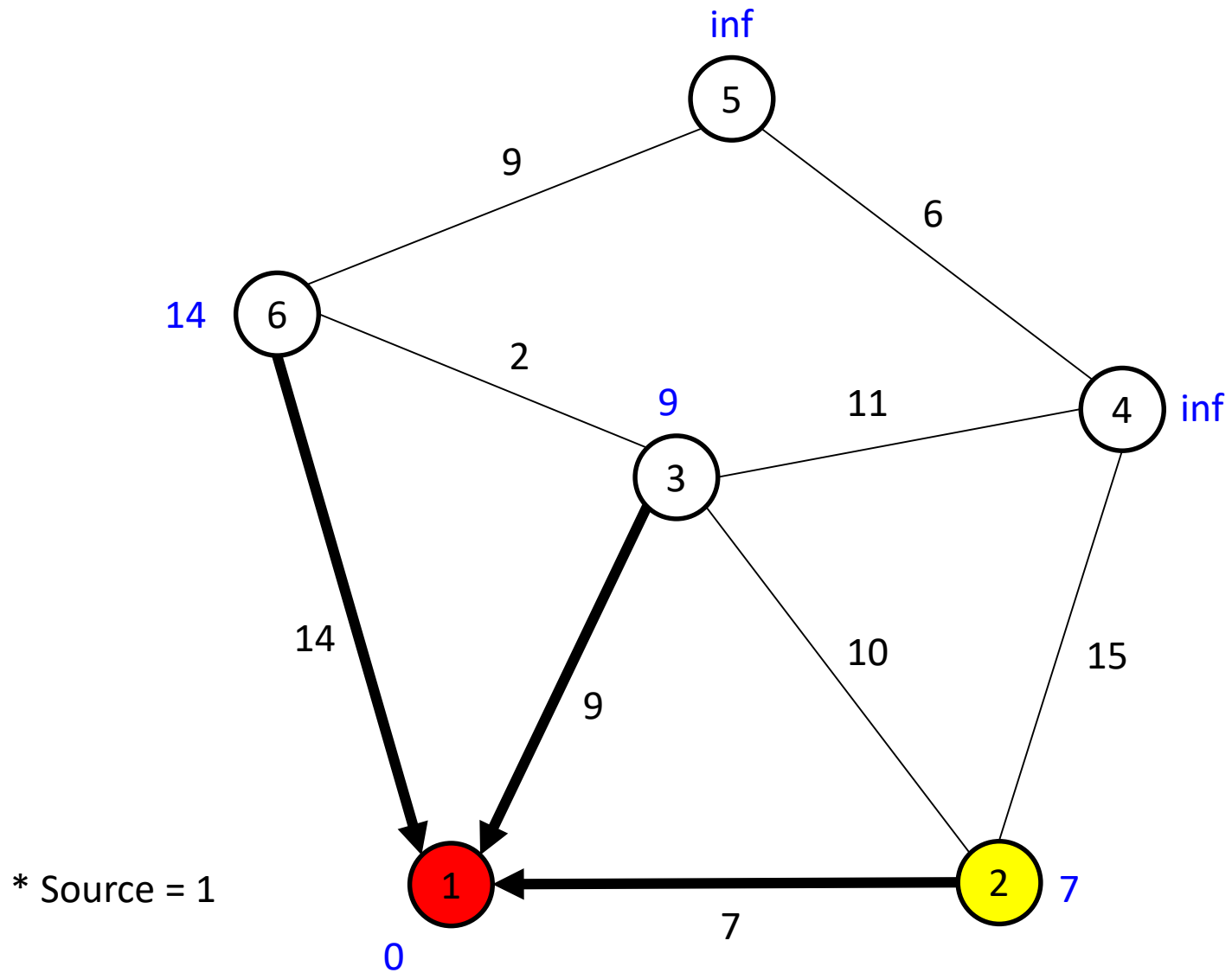
Q=[~~1~~,2,3,4,5,6]
U=1
V=3

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



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

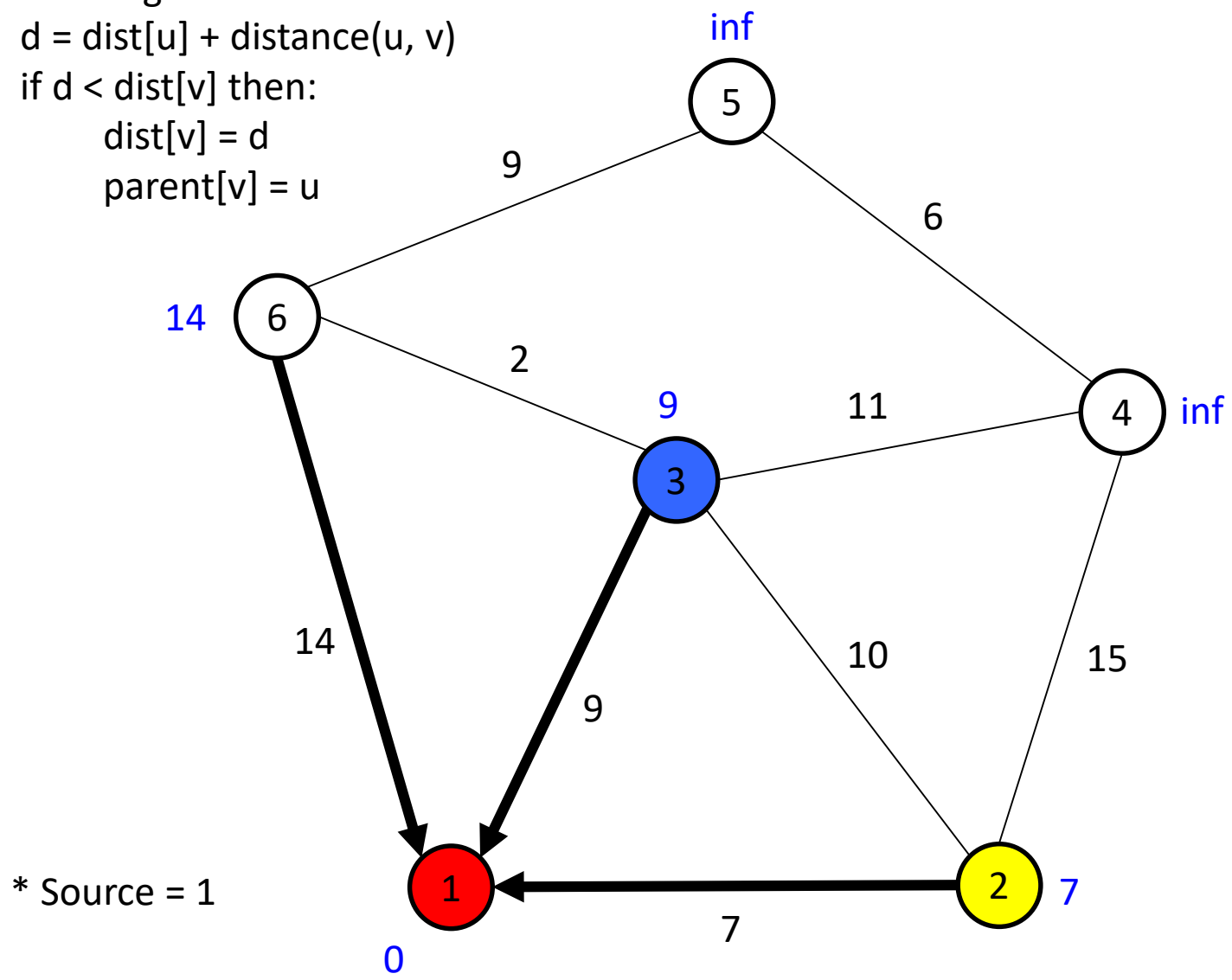


Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	Inf	
5	Inf	
6	14	1

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

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

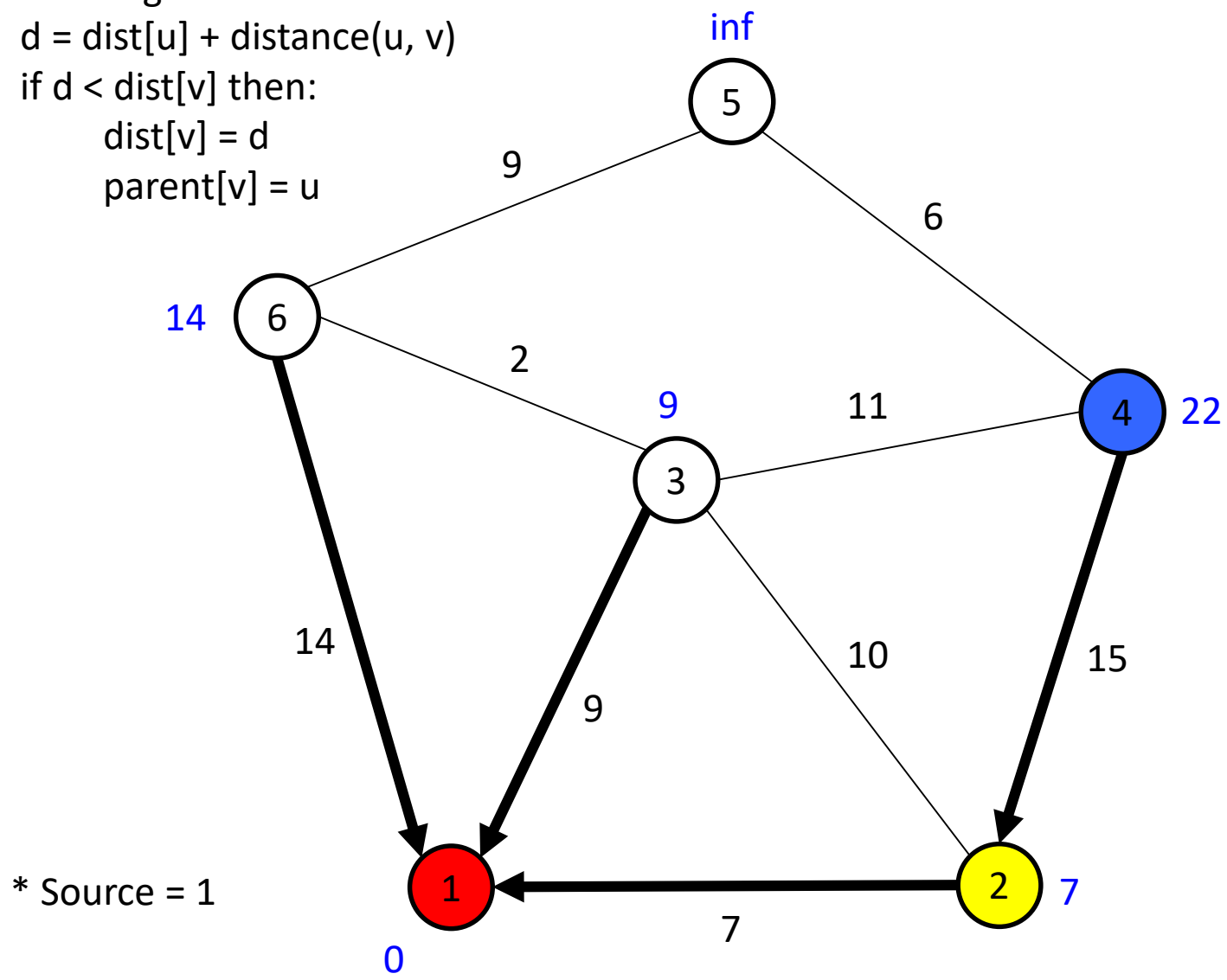
Remove u (node 2) from Q
For each neighbor v of u:
    d = dist[u] + distance(u, v)
    if d < dist[v] then:
        dist[v] = d
        parent[v] = u



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

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

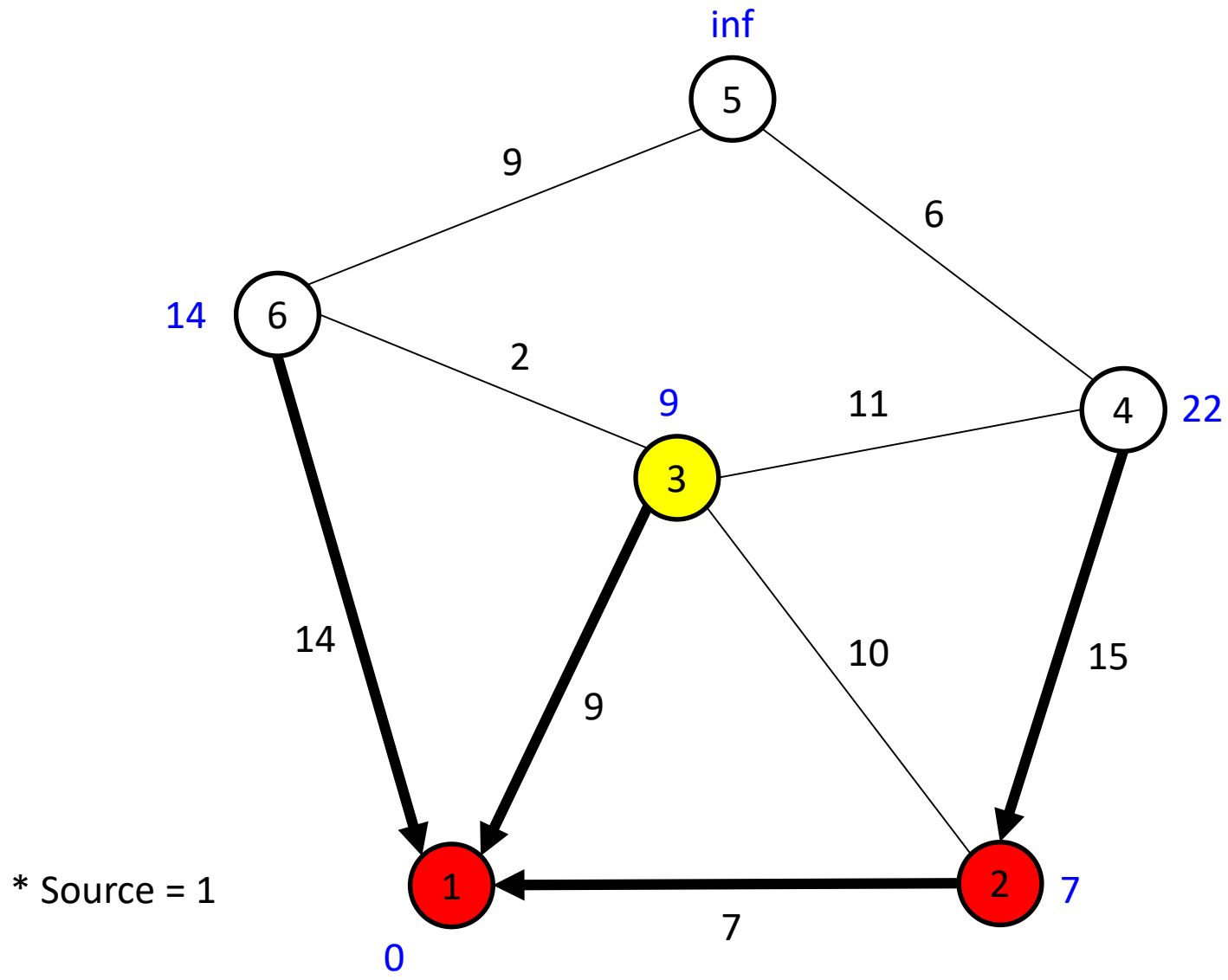


* Source = 1

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

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

parent[v] dist[v]

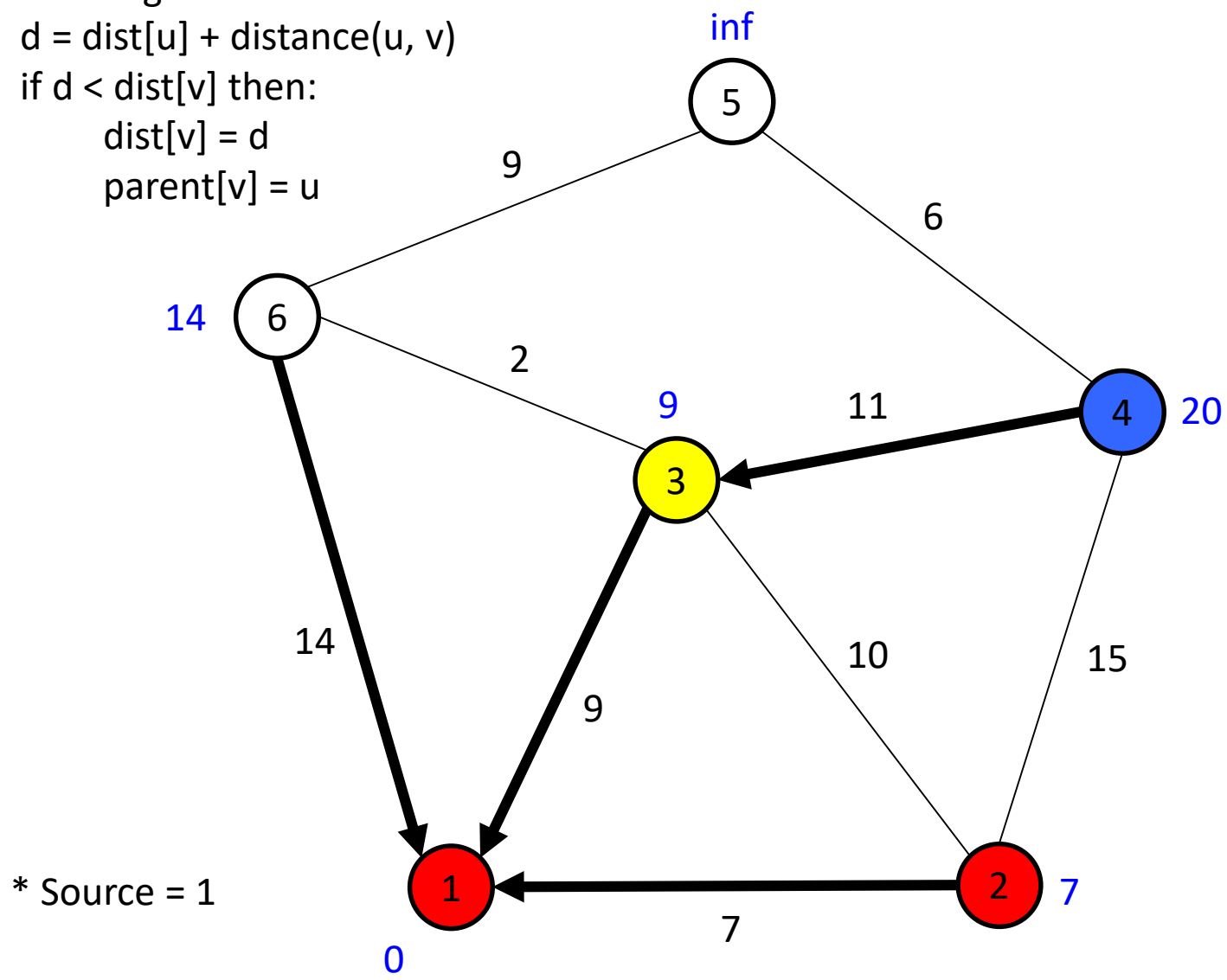


Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	22	2
5	Inf	
6	14	1

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



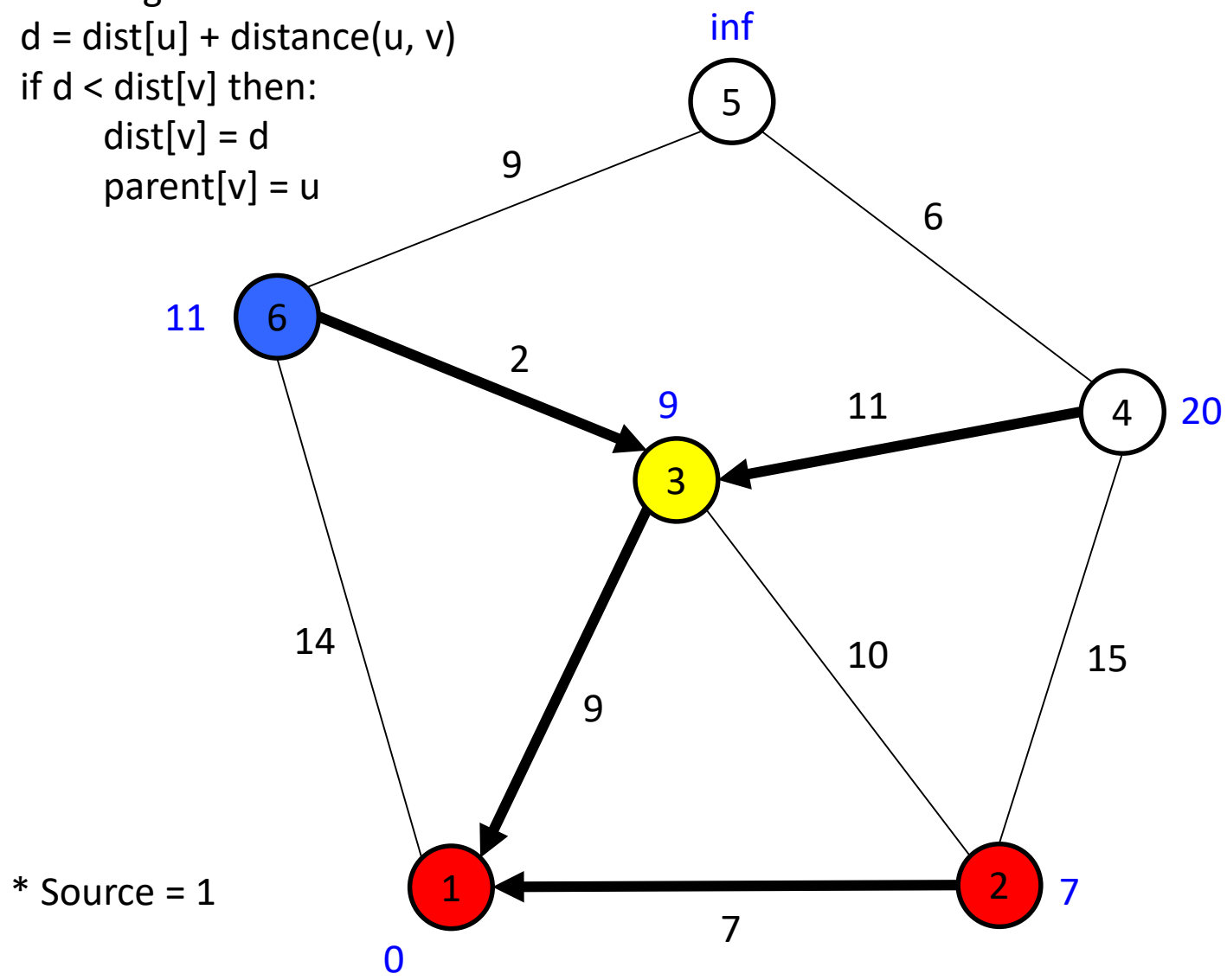
* Source = 1

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

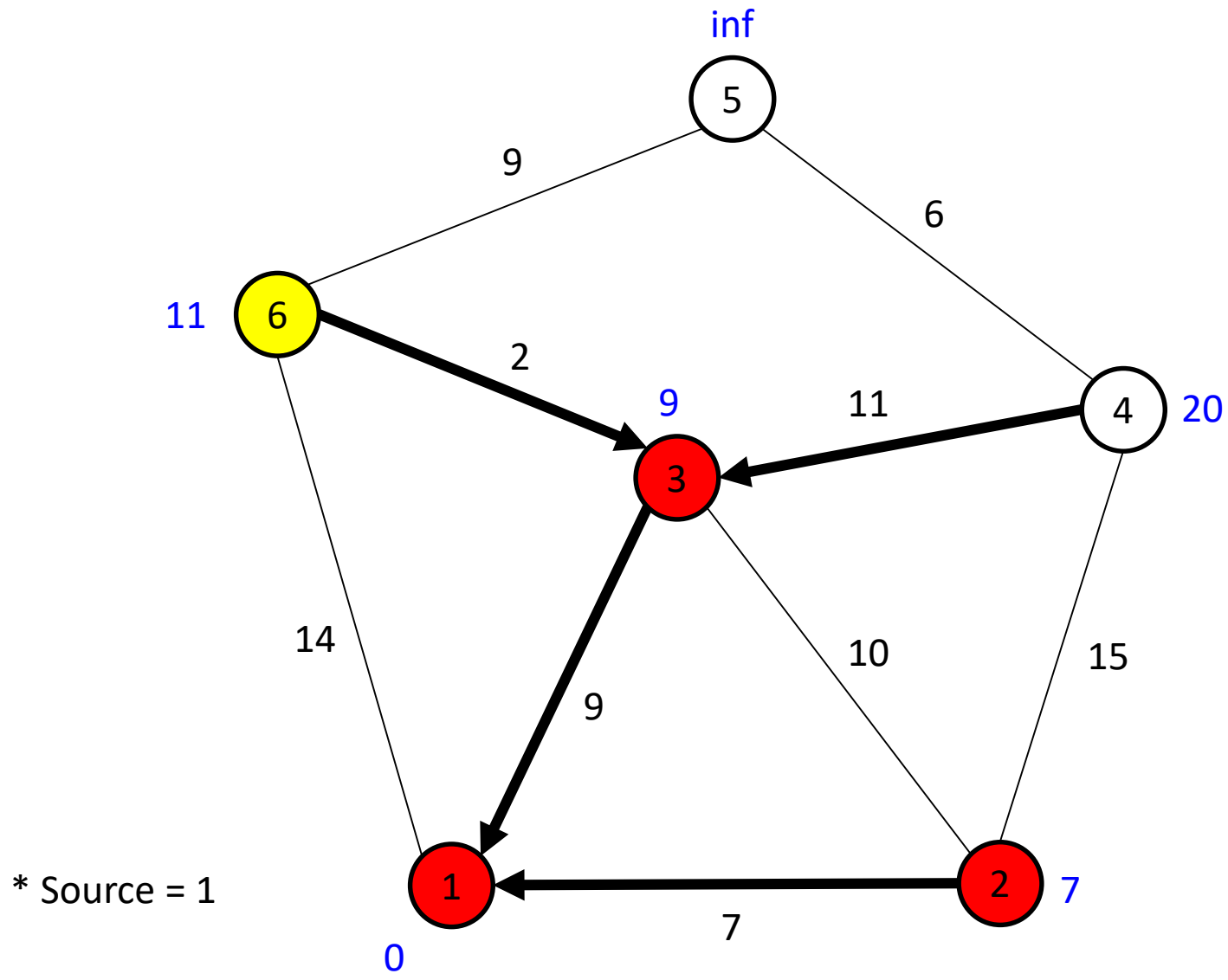


* Source = 1

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]



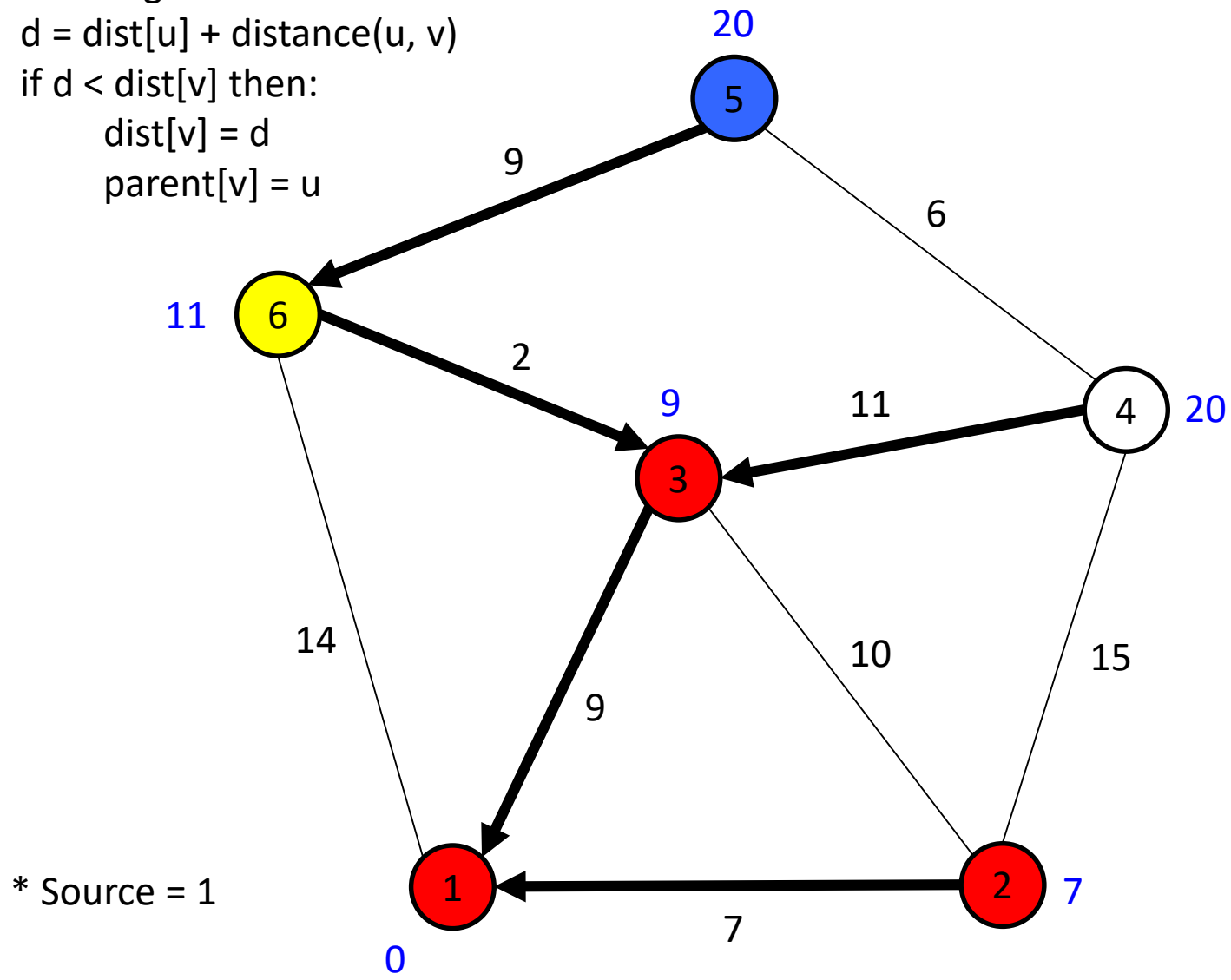
Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	Inf	
6	11	3

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

* Source = 1

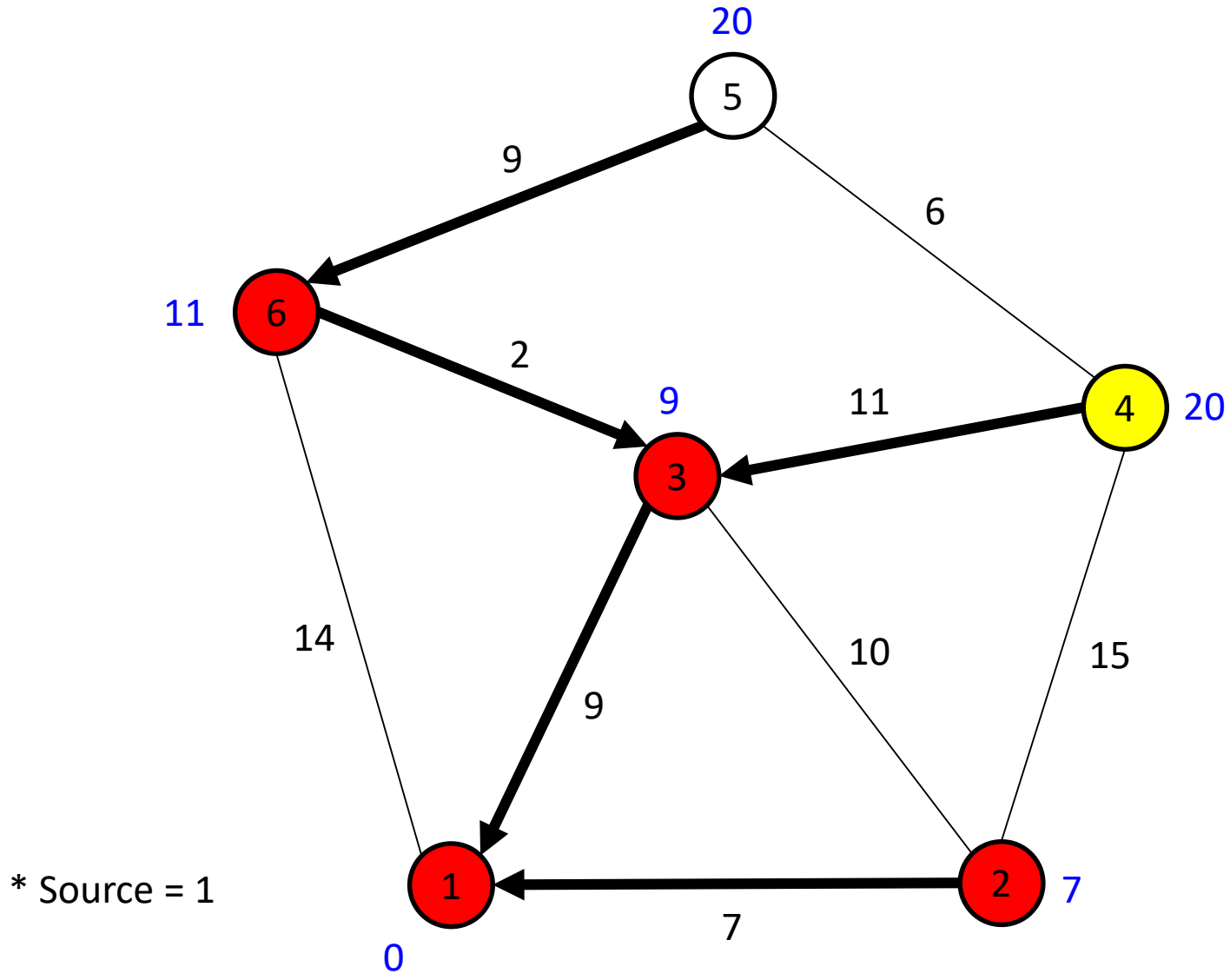
Let u = get vertex in Q with smallest distance value (node 6)

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



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



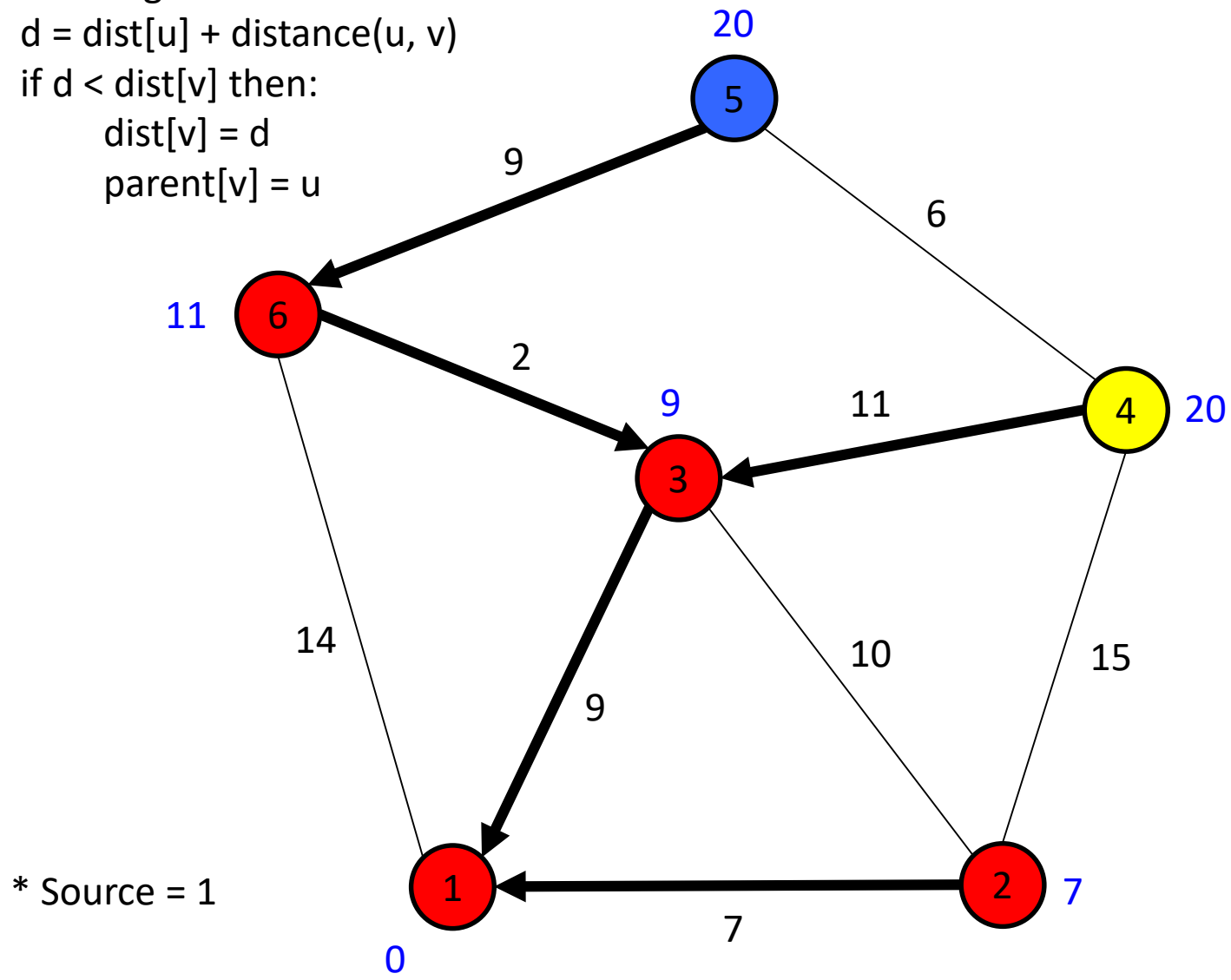
Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

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

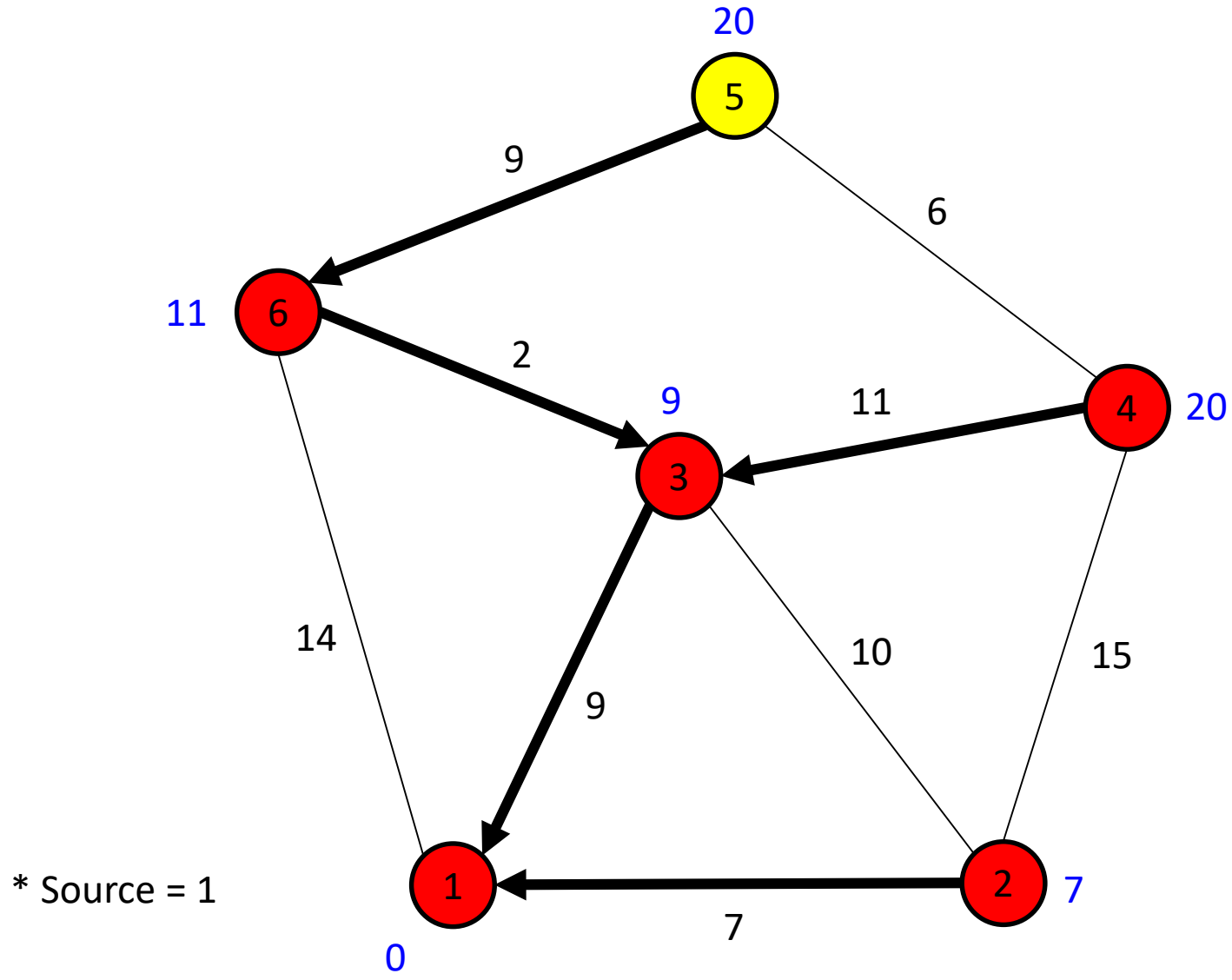


* Source = 1

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]

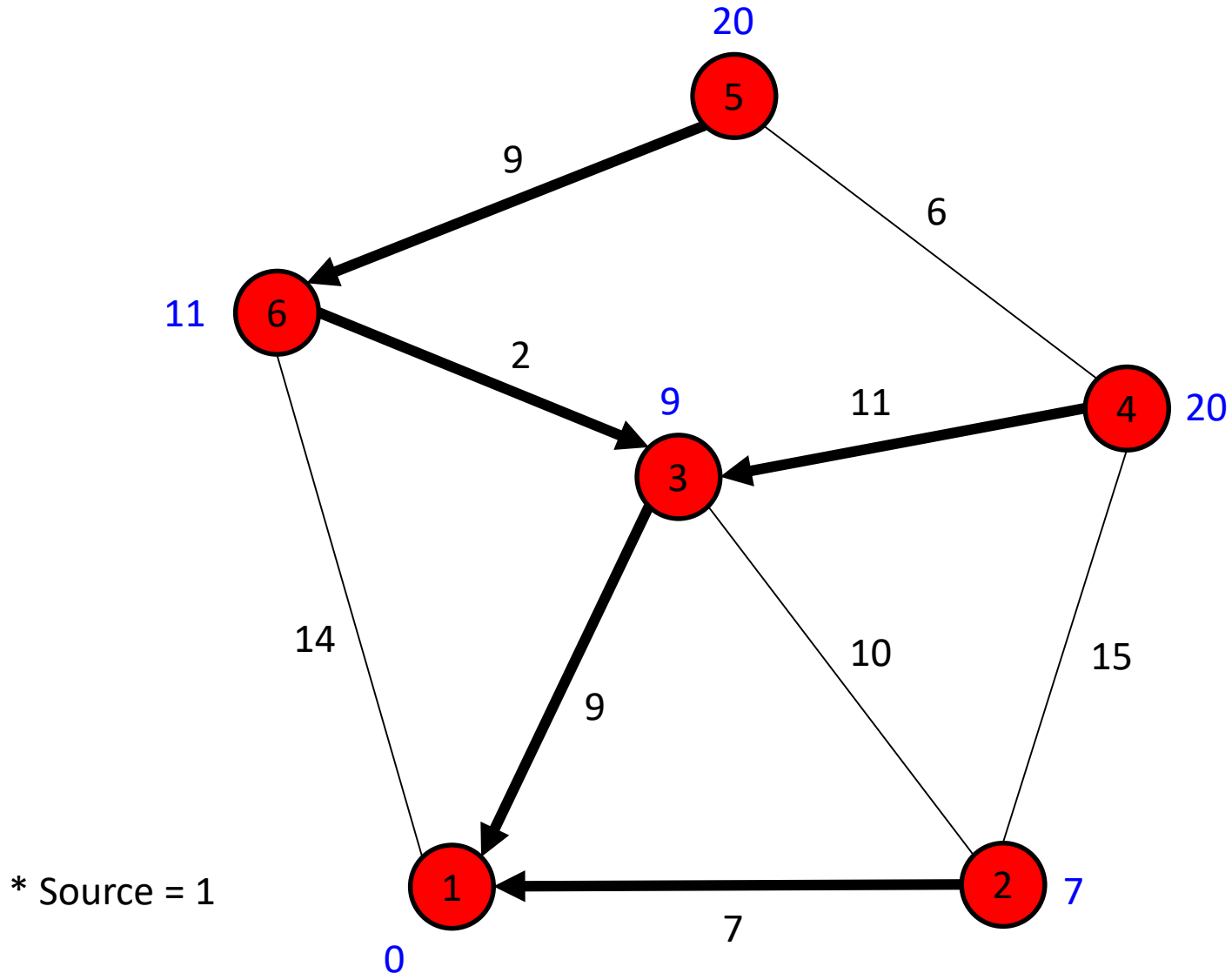


Dest	Cost	Parent
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

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

* Source = 1

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=

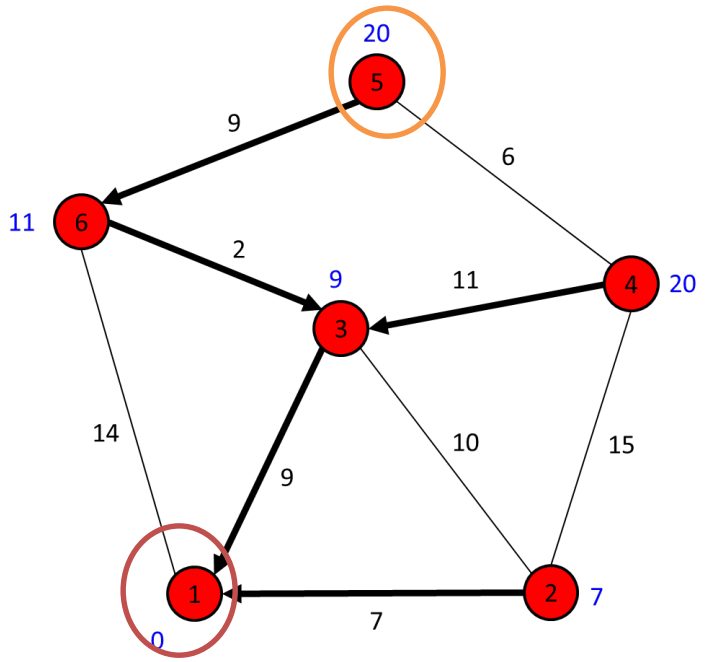
* Source = 1

* 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