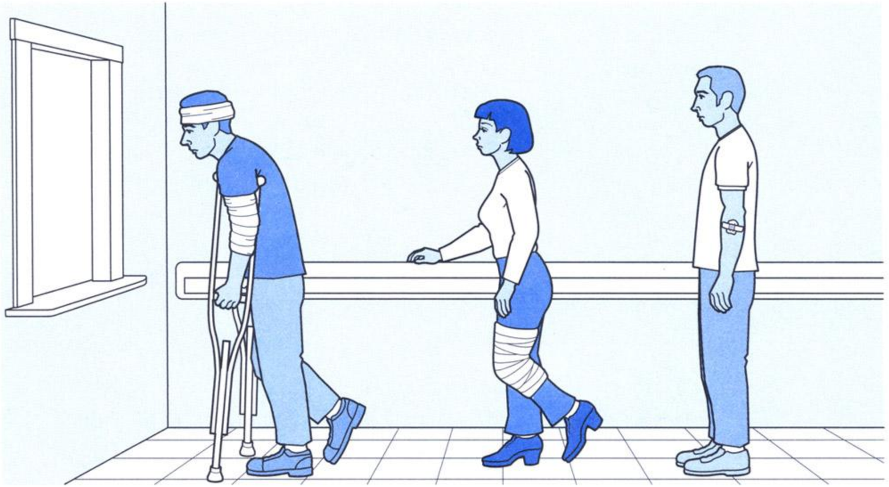
Priority Queue Applications:

Pathfinding in Weighted Graphs

EECS 214 Spring 2017

Priority Queues

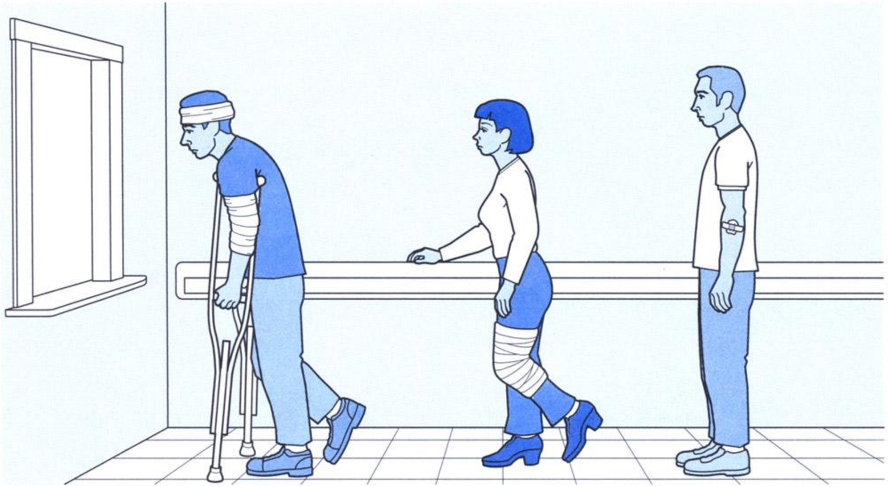
- Like normal queues
  - objects wait in line to be processed
- But objects have an associated **numeric priority**
  - which is specified when the object is inserted
  - objects are dequeued in order of priority
- Different API
  - **Insert(***object*, *priority***)**
    - adds *object* with specified *priority*
  - **ExtractMax()**
    - except in most applications, we sort in ascending order, so...



http://www.alpcentauri.info/fig548_01_0.jpg

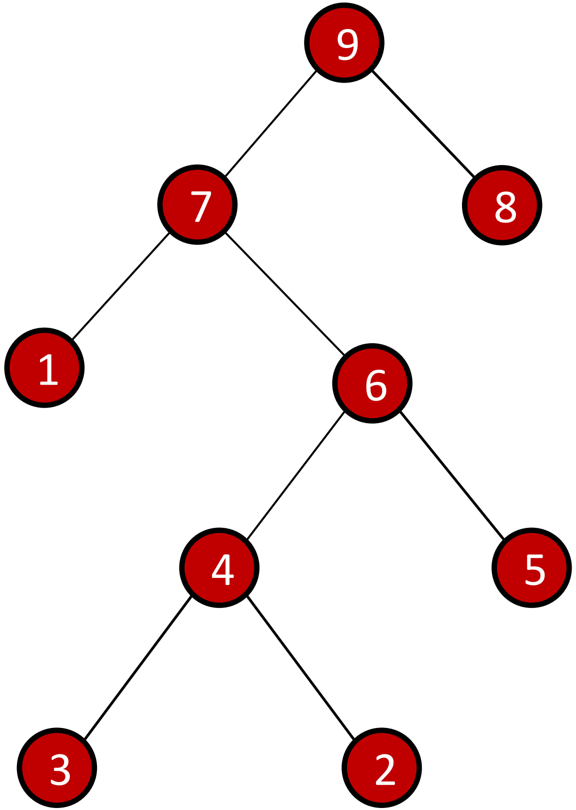
Priority Queues

- Like normal queues
  - objects wait in line to be processed
- But objects have an associated **numeric priority**
  - which is specified when the object is inserted
  - objects are dequeued in order of priority
- Different API
  - **Insert(***object*, *priority***)**
    - adds *object* with specified *priority*
  - **ExtractMin()**
    - min priority queues
    - returns lowest priority object



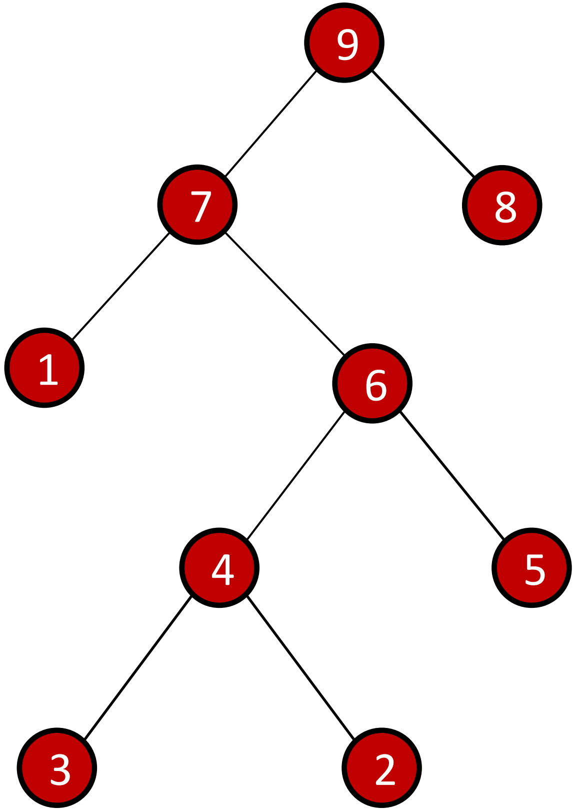
http://www.alpcentauri.info/fig548_01_0.jpg

Heaps Revisited



- Simple tree structure for implementing priority queues (they have other applications)
- We require that *parent nodes* be larger than their *children*
  - we don't require a sorted *in-order traversal*
- There are some crazyass types of heaps!
  - like the beap!
  - but we're going to focus on *binary heaps*
  - id est, complete binary trees
  - that also satisfy the heap property

Heaps Revisited



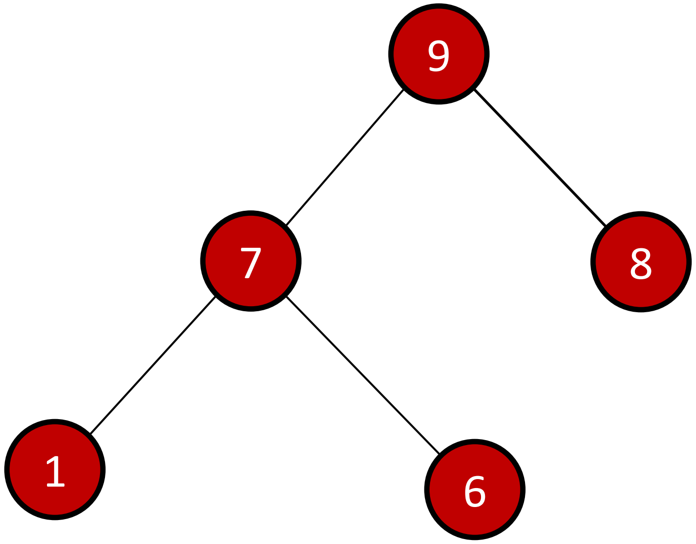
Proposition: the largest element of a heap is always its root

Proof:

- suppose some other element is the largest
- it *isn't* the root, so it has a parent
- it *is* the largest, so it is larger than its parent
- but this contradicts the definition of a heap, so this data structure is not a heap

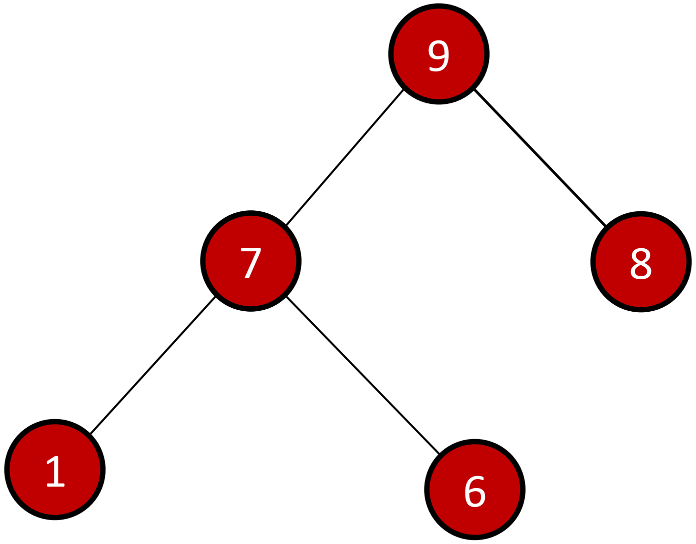
Binary Heaps

- A binary heap is
  - a complete binary tree
  - that satisfies the heap property
- 👌LIT👌
  - so how do we represent one?



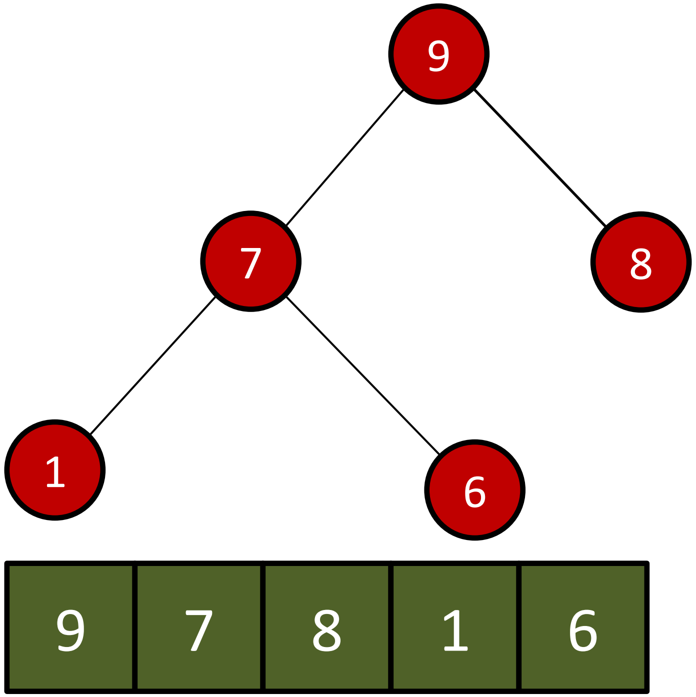
Embedding in an Array

- Turns out that a complete binary tree can be embedded in an array if we're clever about how we do it
- We can compute from an item's position
  - its parent's position
  - its children's positions



Embedding in an Array

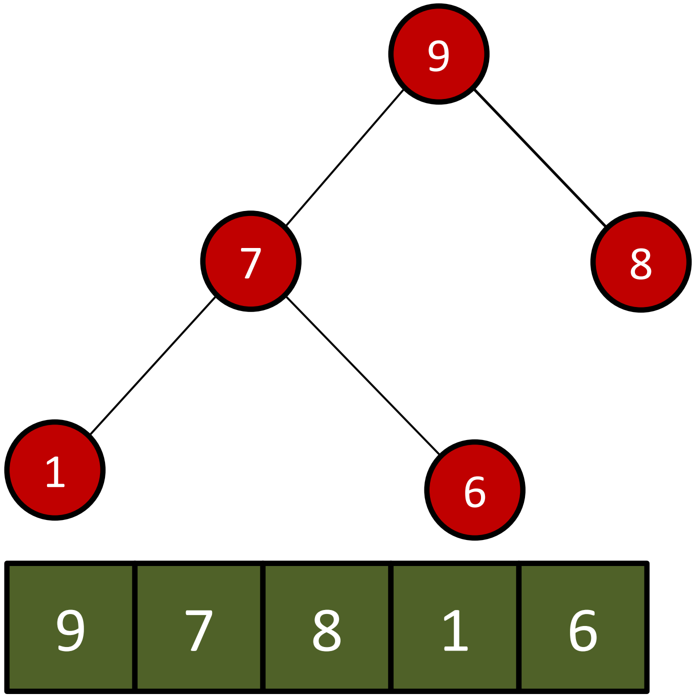
- Store it in breadth-first order
  - store the root in the first element (element 0)
  - for any node:
    - let i be its array index
    - its left child: $2i + 1$
    - its right child: $2i + 2$
    - its parent: $(i - 1) / 2$



Insertion

HeapInsert(*A*, *Val*):

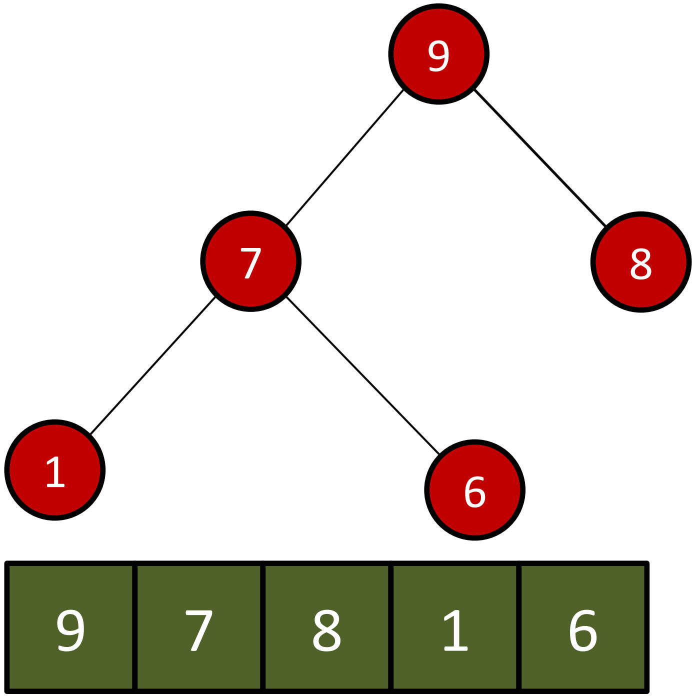
```
A.size++
i = A.size
while i > 0 and A[Parent(i)] < Val:
    A[i] = A[Parent(i)]
    i = Parent(i)
A[i] = Val
```



Extraction (Maximum)

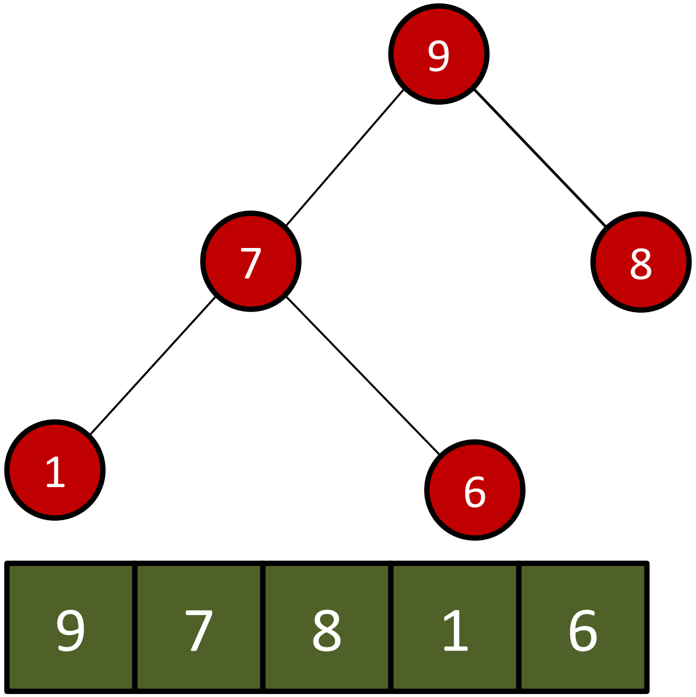
HeapExtractMax(*A*):

```
max = A[0] // root === maximum
A[0] = A[A.size]
A.size--
Heapify(A, 0)
return max
```



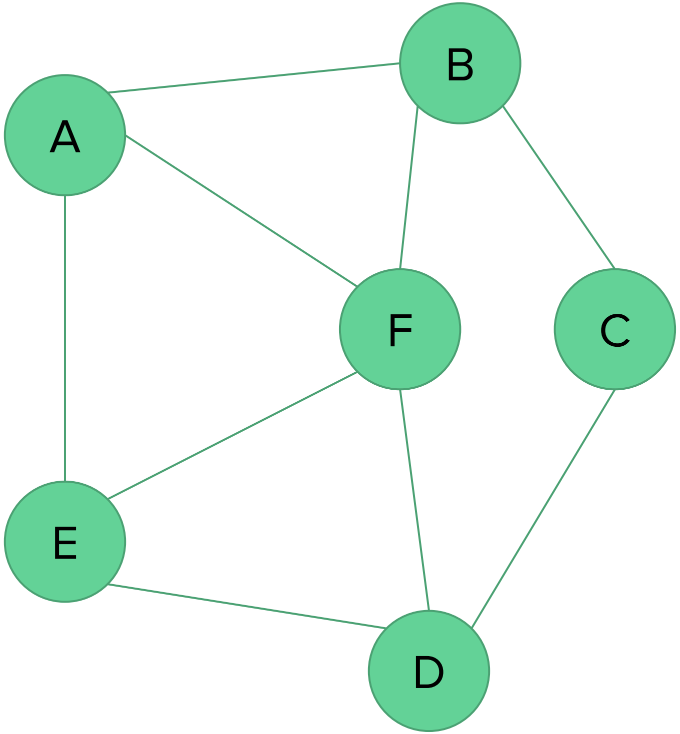
Extraction (Maximum)

```
Heapify(A, i):
    left = left(i); right = right(i)

    if left <= A.size and A[left] > A[i]:
        largest = left
    else
        largest = i

    if right <= A.size and A[right] > A[largest]:
        largest = r

    if largest != i:
        swap(A[i], A[largest]
        Heapify(A, largest)
```



Finding the Shortest Path

- We've used breadth-first search to find a shortest path before
- But BFS assumes an equal weight for each edge
- When path planning, different edges get different weights or costs
- Costs can be:
    - time
    - distance
    - difficulty
    - anything, really, that you want to minimize



Pathfinding with Edge Costs

- BFS is super great, but it searches nodes naïvely
- So how do we make a version that searches nodes:
  - in order of increasing total path cost
  - rather than increasing number of edges

- That's right! Priority queues!
- A node's priority = cost of path from start to that node

Dijkstra's Least-cost Pathfinding Algorithm

```
Dijkstra(G, s, e):
    PQ = new PriorityQueue() // starter bullshit
    set all node costs to infinity
    s.cost = 0
    foreach node n in G:
        PQ.Insert(n, n.cost)
    while not PQ.IsEmpty:
        u = PQ.ExtractMin()
        if u == e: return;
        foreach neighbor in u.neighbors:
            w = edge(u, v).weight // edge weight between nodes u and v
            newcost = u.cost + w
            if newcost < v.cost:
                PQ.DecreaseKey(v, newcost)
                v.cost = newcost
                v.predecessor = u
```

Hold up. Decrease Key?

- We need a new operation in our API that lets us change our priority queue: decreasing the priority of a given object already in the queue
  - N.B.: we're using a min-priority queue ('cause we're doing ExtractMin), so decreasing a queue moves it *forward* in the priority queue
- Now, how should we implement DecreaseKey?

Implementing DecreaseKey

- One option:
  - we could remove it
  - then reinsert it at a lower priority
- But the insertion algorithm:
  - adds at the bottom of the heap
  - then swaps it upward until its priority is lower than its parent's (for a min heap)

HeapInsert(*A*, *Val*):

```
A.size++
i = A.size
while i > 0 and A[Parent(i)] > Val:
    A[i] = A[Parent(i)]
    i = Parent(i)
A[i] = Val
```

N.B: we're checking if *Parent* > *Val* now; it's a min-priority queue

Implementing DecreaseKey

- So DecreaseKey is actually super easy:
  - just move a node up
  - until it's in the right place
- We're going to copy the code for insert and remove the bits that put it at the end

```
DecreaseKey(A, i, Val):
    while i > 0 and A[Parent(i)] > Val:
        A[i] = A[Parent(i)]
        i = Parent(i)
    A[i] = Val
```

Implementing DecreaseKey

- Hey, we added an argument!
- Unfortunately, we need to remember where the node is in the damn heap
    - id est, its array index
- Best done by storing it in the graph node itself
- Have to remember to update it any time the node moves around in the queue

```
DecreaseKey(A, i, Val):
    while i > 0 and A[Parent(i)] > Val:
        A[i] = A[Parent(i)] // here
        i = Parent(i)
    A[i] = Val // and here
```

Proving Correctness

- There are two kinds of nodes:
  - those still in the queue
  - those that have been removed from the queue

```
Dijkstra(G, s, e):
    PQ = new PriorityQueue()
    set all node costs to infinity
    s.cost = 0
    foreach node n in G:
        PQ.Insert(n, n.cost)
    while not PQ.IsEmpty:
        u = PQ.ExtractMin()
        if u == e:
            return;
        foreach neighbor in u.neighbors:
            w = edge(u, v).weight
            newcost = u.cost + w
            if newcost < v.cost:
                PQ.DecreaseKey(v, newcost)
                v.cost = newcost
                v.predecessor = u
```

Invariants

For all nodes *V*:

- if *V* is not in the queue
    - dist(*V*) is the length of the shortest path from the start node to *V*
- otherwise
    - dist(*V*) is the length of the shortest path from the start node to *V* using nodes not in the queue

```
Dijkstra(G, s, e):
    PQ = new PriorityQueue()
    set all node costs to infinity
    s.cost = 0
    foreach node n in G:
        PQ.Insert(n, n.cost)
    while not PQ.IsEmpty:
        u = PQ.ExtractMin()
        if u == e:
            return;
        foreach neighbor in u.neighbors:
            w = edge(u, v).weight
            newcost = u.cost + w
            if newcost < v.cost:
                PQ.DecreaseKey(v, newcost)
                v.cost = newcost
                v.predecessor = u
```

Proof

For all nodes *V*:

- if *V* is not in the queue
  - dist(*V*) is the length of the shortest path from the start node to *V*
- otherwise
  - dist(*V*) is the length of the shortest path from the start node to *V* using nodes not in the queue

Base Case: only the start node has been removed from the queue

Trivially true:

- dist(*start*) = 0
- dist(*anything else*) = float.infinity

Proof

For all nodes *V*:

- if *V* is not in the queue
  - dist(*V*) is the length of the shortest path from the start node to *V*
- otherwise
  - dist(*V*) is the length of the shortest path from the start node to *V* using nodes not in the queue

Inductive case:

- assume it's true for the currently visited nodes
- let *V* be the next node in the queue

Then:

1. *V* has the minimum dist() of all the nodes in the queue
2. dist(*V*) is the length of the shortest path to *V* using nodes not in the queue

Proof

Inductive case:

- assume it's true for the currently visited nodes
- let *V* be the next node in the queue

Then:

1. v has the minimum dist() of all the nodes in the queue
2. dist(*V*) is the length of the shortest path to *V* using nodes not in the queue

Claim:

- dist(*V*) is the length of the shortest path using any nodes

Proof:

- Assume there exists a shorter path
- Case 1: the shorter path *contains a node in the queue*
  - then that node must be closer to the start than *V*
  - this contradicts (1)

Proof

Inductive case:

- assume it's true for the currently visited nodes
- let *V* be the next node in the queue

Then:

1. v has the minimum dist() of all the nodes in the queue
2. dist(*V*) is the length of the shortest path to *V* using nodes not in the queue

Claim:

- dist(*V*) is the length of the shortest path using any nodes

Proof:

- Assume there exists a shorter path
- Case 2: the shorter path *contains no nodes from the queue*
    - this contradicts (2)

Proof

Inductive case:

- assume it's true for the currently visited nodes
- let *V* be the next node in the queue

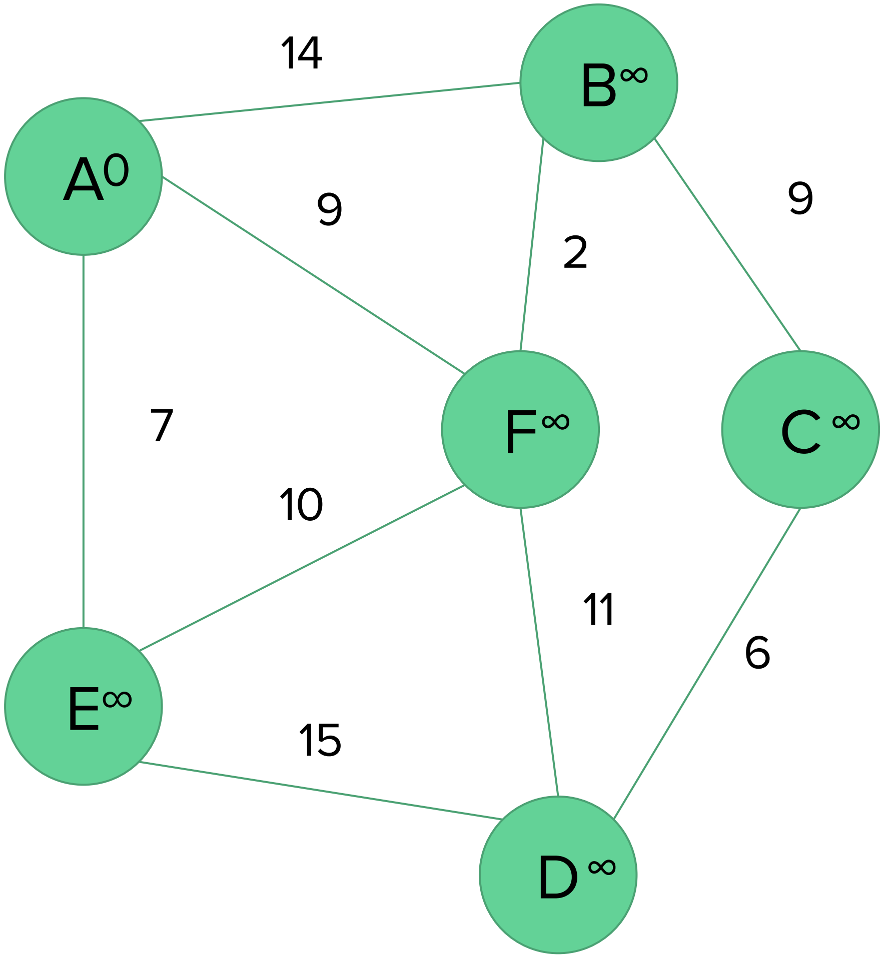
Then:

1. v has the minimum dist() of all the nodes in the queue
2. dist(*V*) is the length of the shortest path to *V* using nodes not in the queue

Therefore:

- The inductive case holds
- The invariants hold
    - including when the queue is empty
    - at which point we know the distance to every node
- The algorithm is correct

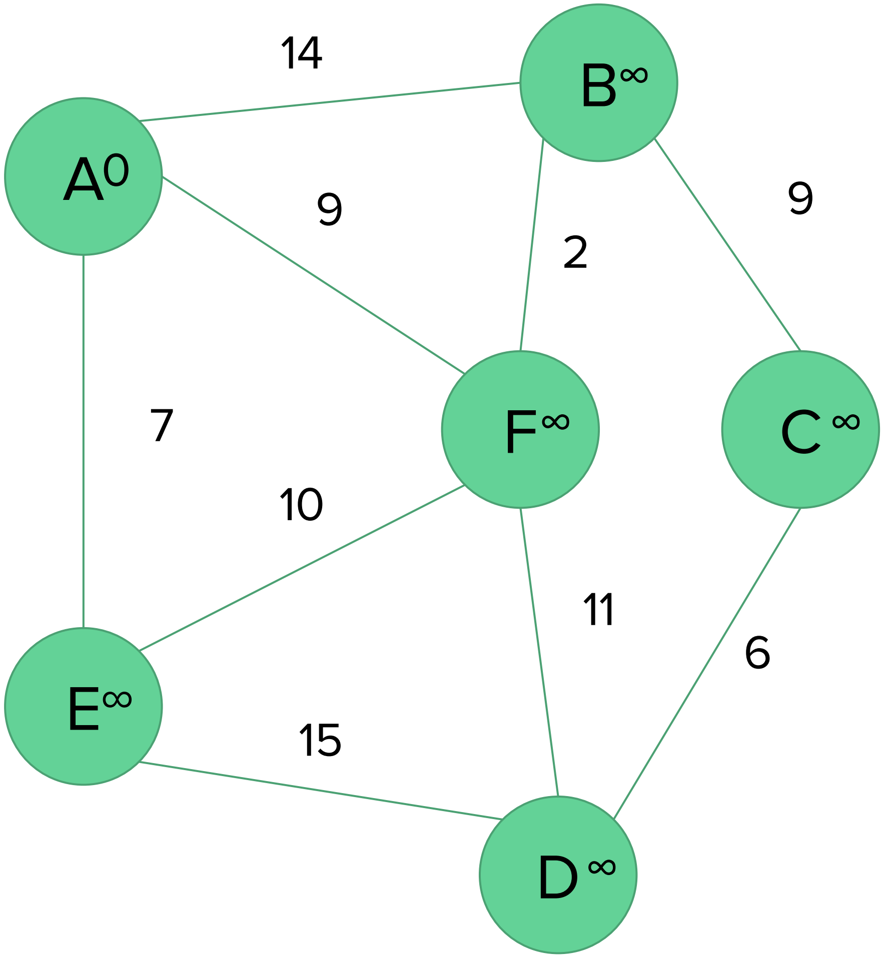
Running Dijkstra's



Initialize Priority Queue

Queue:

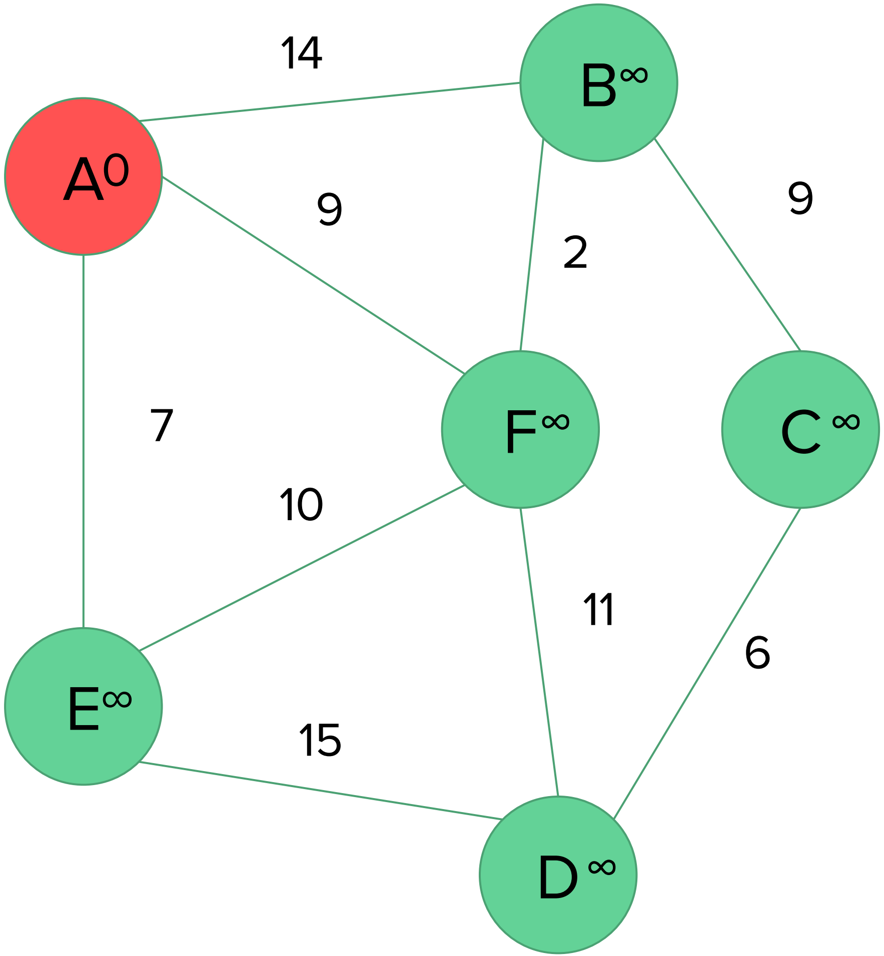
1. A (0)
2. B (∞)
3. C (∞)
4. D (∞)
5. E (∞)
6. F (∞)



Extract Min (A)

Queue:

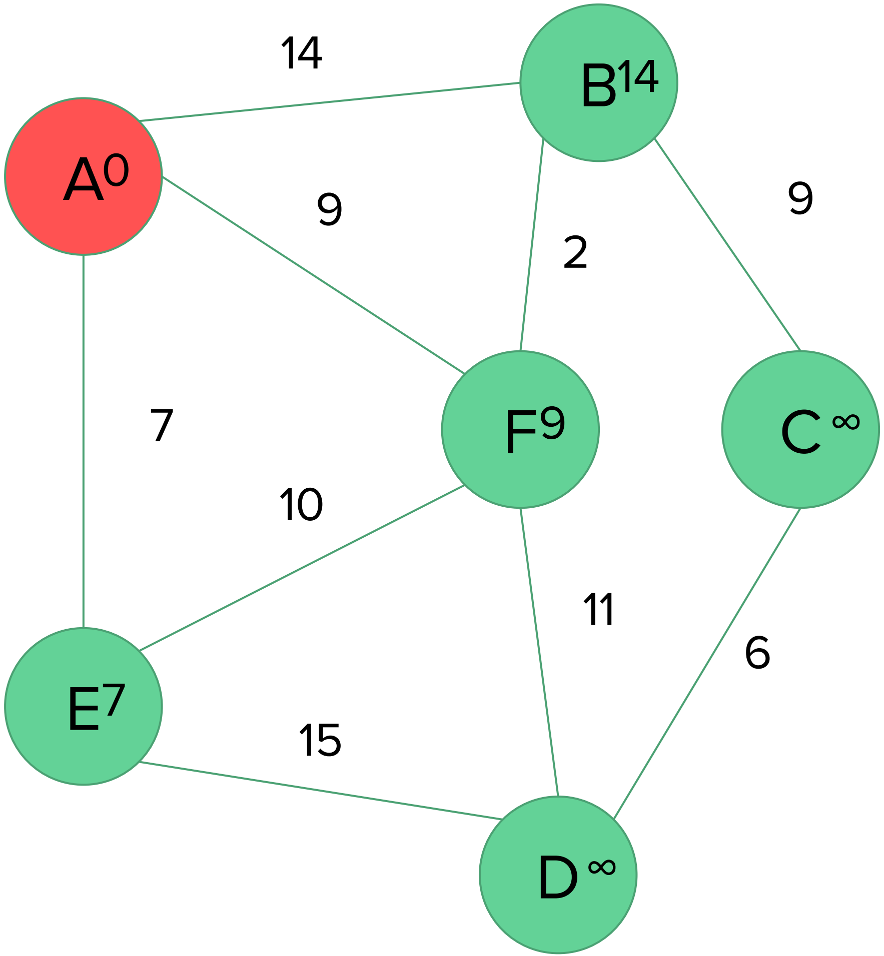
1. B (∞)
2. C (∞)
3. D (∞)
4. E (∞)
5. F (∞)



Update Neighbors

Queue:

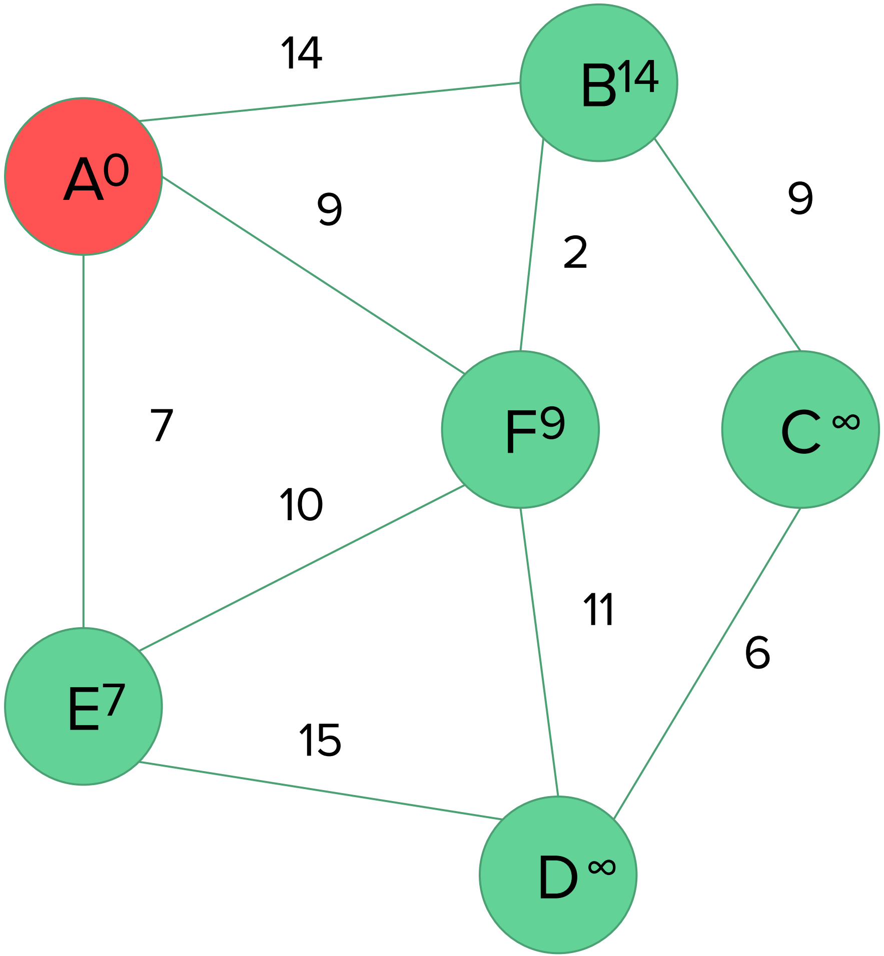
1. B (14)
2. C (∞)
3. D (∞)
4. E (7)
5. F (9)



Update Neighbors

Queue:

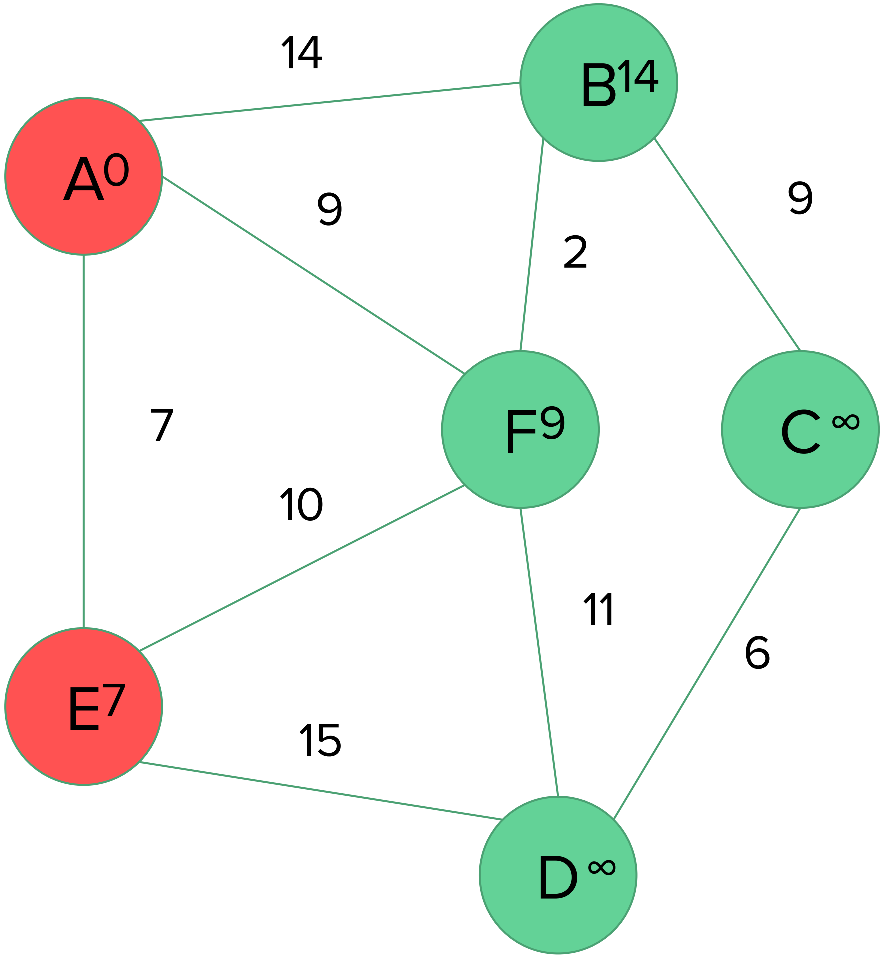
1. E (7)
2. F (9)
3. B (14)
4. C (∞)
5. D (∞)



Extract Min (E)

Queue:

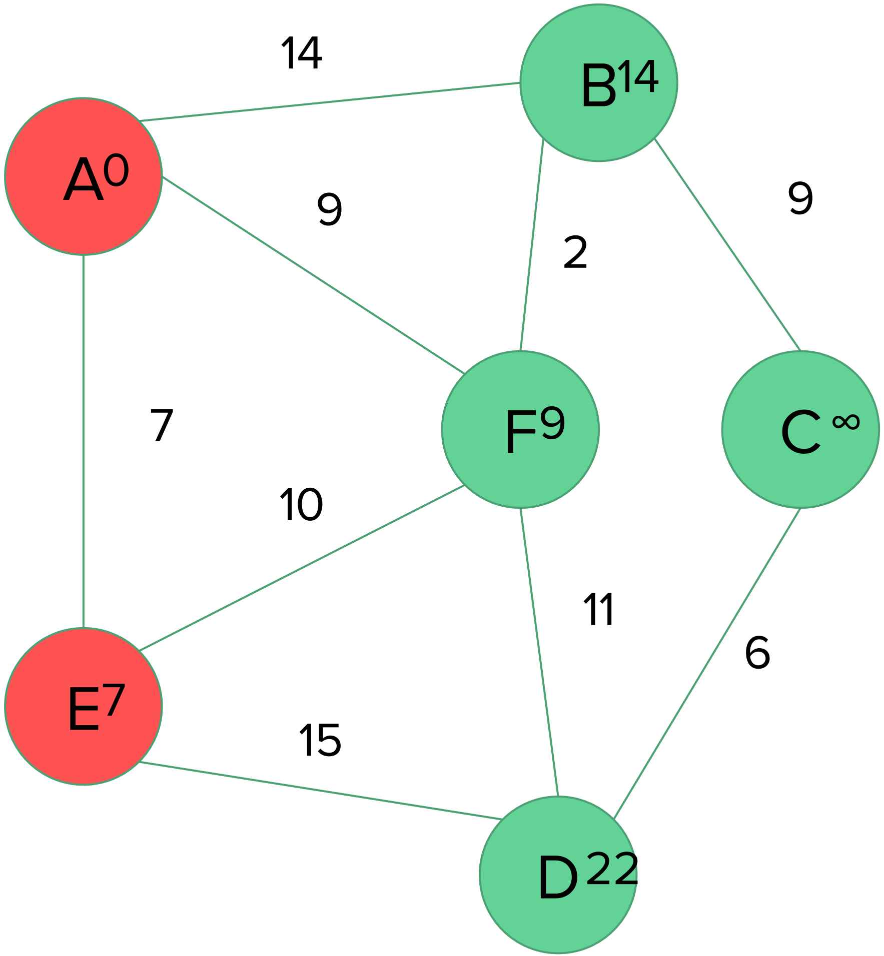
1. F (9)
2. B (14)
3. C (∞)
4. D (∞)



Update Neighbors

Queue:

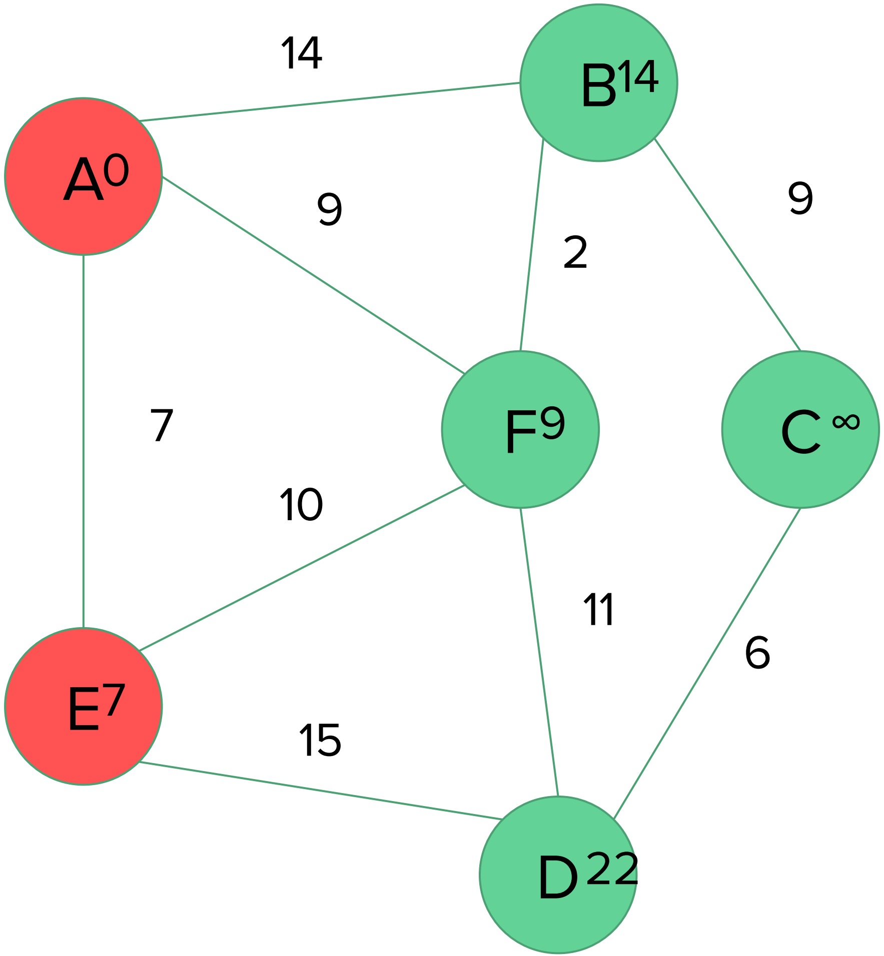
1. F (9)
2. B (14)
3. C (∞)
4. D (22)



Update Neighbors

Queue:

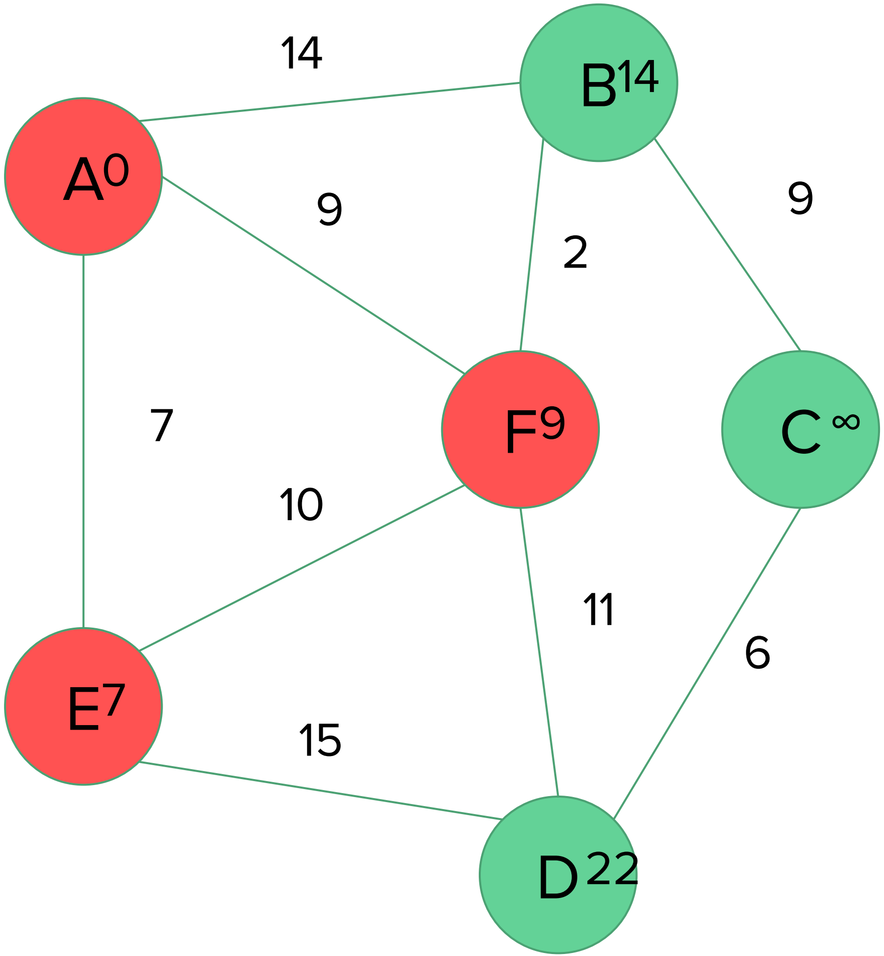
1. F (9)
2. B (14)
3. D (22)
4. C (∞)



Extract Min (F)

Queue:

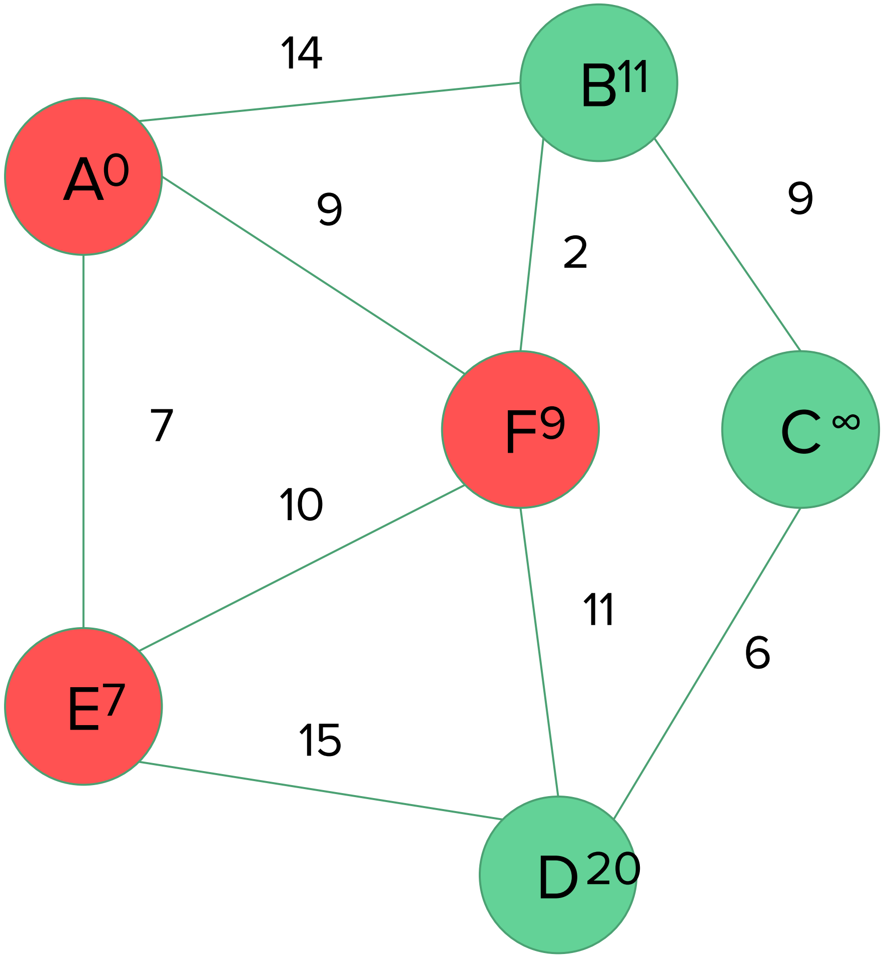
1. B (14)
2. D (22)
3. C (∞)



Update Neighbors

Queue:

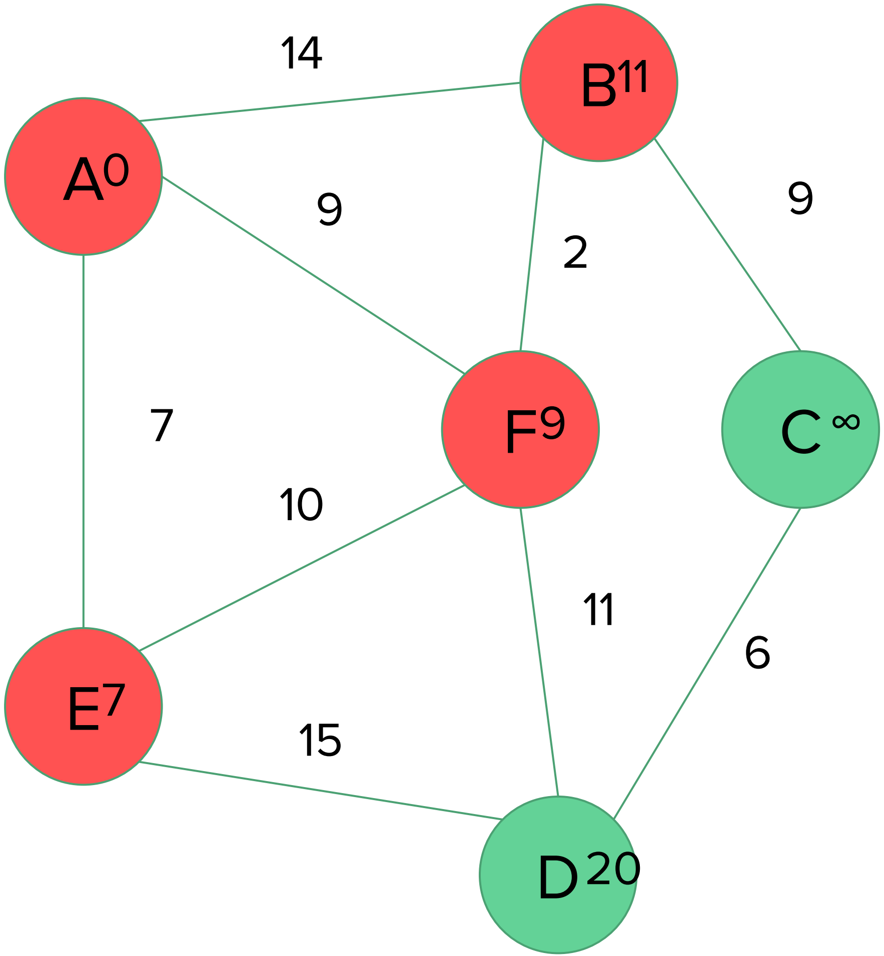
1. B (11)
2. D (20)
3. C (∞)



Extract Min (B)

Queue:

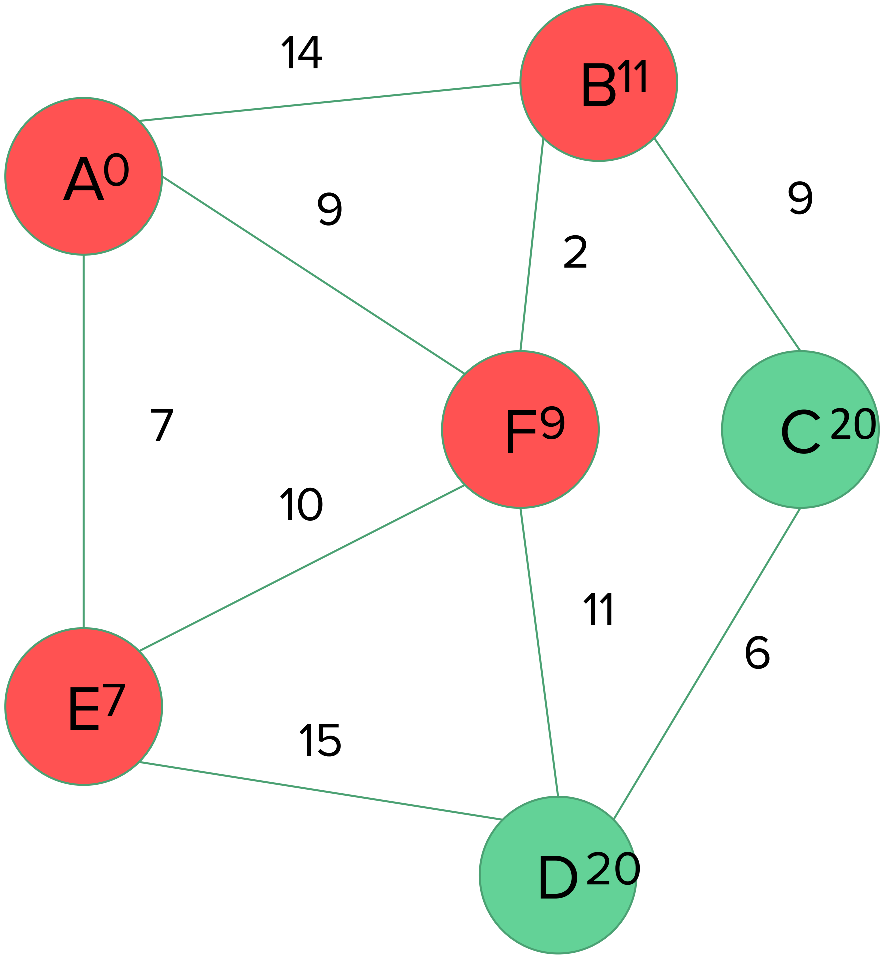
1. D (20)
2. C (∞)



Update Neighbors

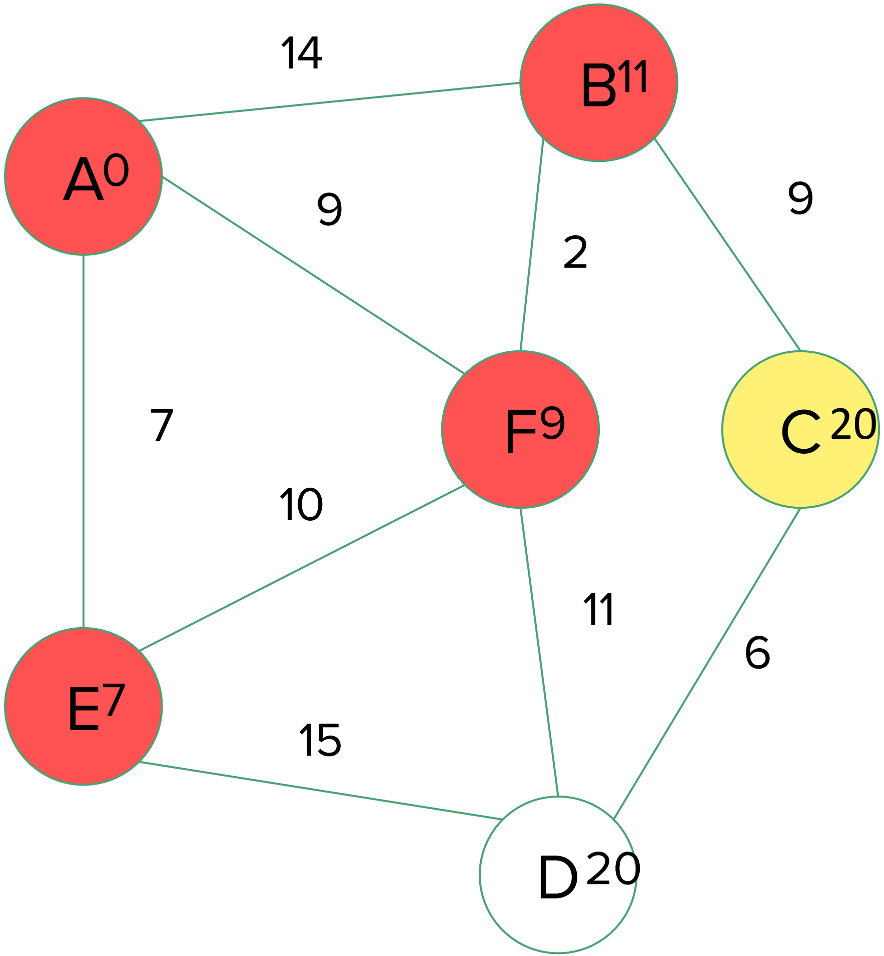
Queue:

1. D (20)
2. C (20)

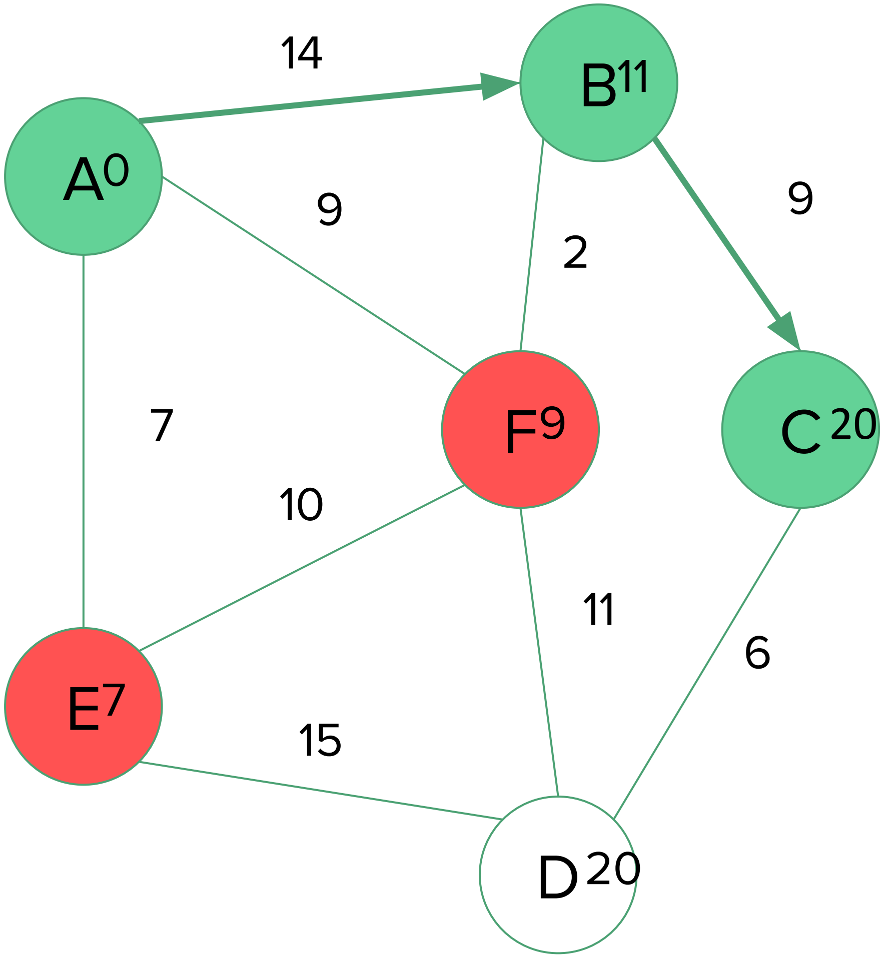


We Reached the Goal

So we can empty our queue, since we know that we have the shortest path.



Shortest Path



Runtime

```
Dijkstra(G, s, e):                                  Runs:
    PQ = new PriorityQueue() ________________       Once
    set all node costs to infinity___________       Once
    s.cost = 0 ______________________________       Once
    foreach node n in G:
        PQ.Insert(n, n.cost) ________________       O(V) times
    while not PQ.IsEmpty:
        u = PQ.ExtractMin() _________________       O(V) times
        if u == e: return;                          O(V)
        foreach neighbor in u.neighbors:
            w = edge(u, v).weight ___________       O(E) times*
            newcost = u.cost + w                    O(E)
            if newcost < v.cost:
                PQ.DecreaseKey(v, newcost)          O(E)
                v.cost = newcost                    O(E)
                v.predecessor = u                   O(E)
```

Runtime

Dijkstra(*G*, *s*, *e*):	Runs:
PQ = new PriorityQueue() ____	*Once*
set all node costs to infinity____	*Once*
s.cost = 0 ____	*Once*
foreach node n in G:	
PQ.Insert(n, n.cost) ____	*O(V) times*
while not PQ.IsEmpty:	
u = PQ.ExtractMin() ____	*O(V) times*
if u == e: return;	*O(V)*
foreach neighbor in u.neighbors:	
w = edge(u, v).weight ____	*O(E) times**
newcost = u.cost + w	*O(E)*
if newcost < v.cost:	
PQ.DecreaseKey(v, newcost)	*O(E)*
v.cost = newcost	*O(E)*
v.predecessor = u	*O(E)*

*Why not *O(EV)*?

Runtime

Dijkstra(*G, s, e*)**:**	Runs:	Execution Time:
PQ = new PriorityQueue()	*Once*	*O(1)*
set all node costs to infinity	*Once*	*O(V)*
s.cost = 0	*Once*	*O(1)*
foreach node n in G:		
PQ.Insert(n, n.cost)	*O(V) times*	*O(log V)*
while not PQ.IsEmpty:		
u = PQ.ExtractMin()	*O(V) times*	*O(log V)*
if u == e: return;	*O(V)*	*O(1)*
foreach neighbor in u.neighbors:		
w = edge(u, v).weight	*O(E) times*	*O(1)*
newcost = u.cost + w	*O(E)*	*O(1)*
if newcost < v.cost:		
PQ.DecreaseKey(v, newcost)	*O(E)*	*O(log V)*
v.cost = newcost	*O(E)*	*O(1)*
v.predecessor = u	*O(E)*	*O(1)*

Runtime

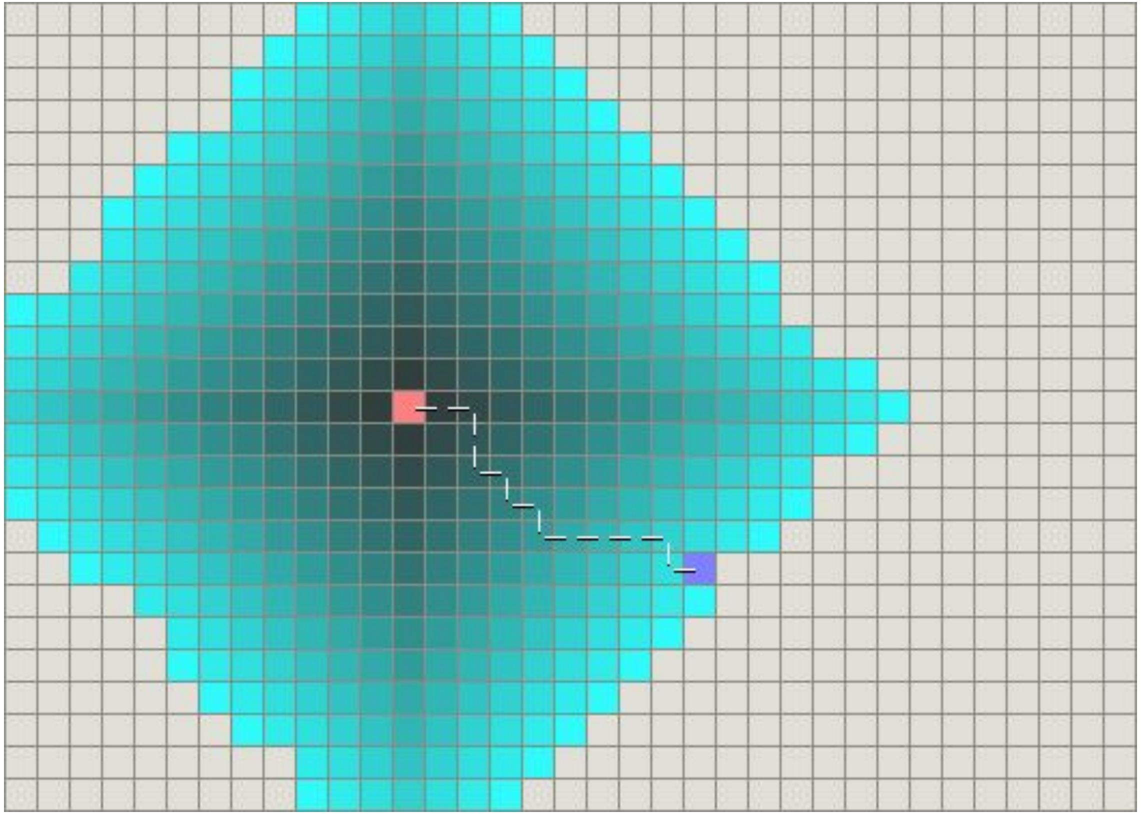
Dijkstra(*G*, *s*, *e*):	Runs:	Execution Time:	Total:
PQ = new PriorityQueue()	*Once*	*O(1)*	*O(1)*
set all node costs to infinity	*Once*	*O(V)*	*O(V)*
s.cost = 0	*Once*	*O(1)*	*O(1)*
foreach node n in G:			
PQ.Insert(n, n.cost)	*O(V) times*	*O(log V)*	*O(V log V)*
while not PQ.IsEmpty:			
u = PQ.ExtractMin()	*O(V) times*	*O(log V)*	*O(V log V)*
if u == e: return;	*O(V)*	*O(1)*	*O(V)*
foreach neighbor in u.neighbors:			
w = edge(u, v).weight	*O(E) times*	*O(1)*	*O(E)*
newcost = u.cost + w	*O(E)*	*O(1)*	*O(E)*
if newcost < v.cost:			
PQ.DecreaseKey(v, newcost)	*O(E)*	*O(log V)*	*O(E log V)*
v.cost = newcost	*O(E)*	*O(1)*	*O(E)*
v.predecessor = u	*O(E)*	*O(1)*	*O(E)*

Runtime

$$O(1) + O(V) + O(E) + O(V \log V) + O(E \log V)$$
$$= O(1) + O(V + E) + O((E + V) \log V)$$
$$= O(V + E) + O((E + V) \log V)$$
$$= O((V + E)(1 + \log V))$$
$$= O((V + E) \log V)$$

N.B.: you can *technically* get better runtime [$O(E + V \log V)$] if you use something called a Fibonacci heap, but it's so laughably theoretical that this might as well be the best that we've got.

Example(ish): Searching in a 4-Connected Grid



Source: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Dijkstra’s Does a lot of Useless Searching

- It’s a blind search algorithm
  - it exhaustively explores nodes in increasing order of distance
  - even if they’re in the opposite direction of the goal! D:
- Is there some way of biasing the search towards more productive nodes?
  - search the more promising nodes first

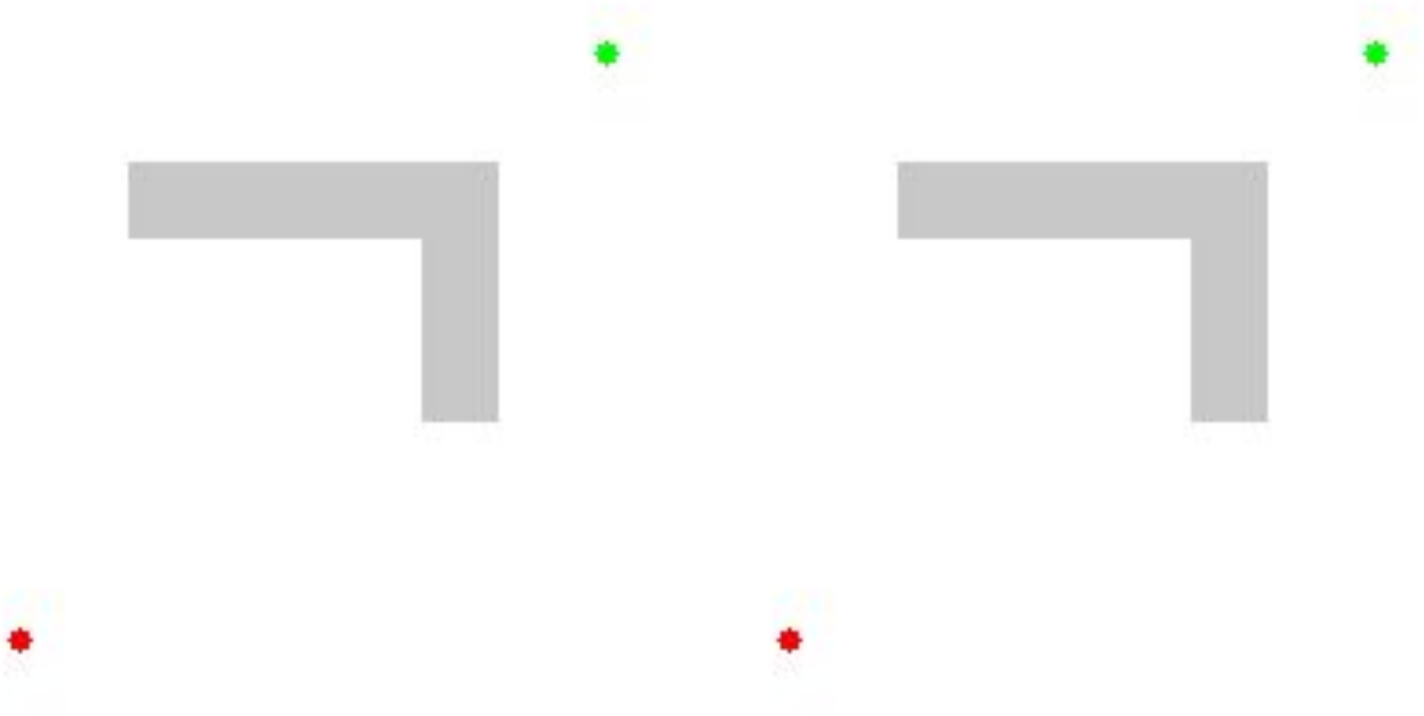
Heuristic Search

- Dijkstra's sorts the queue by distance from the start node
  - and so searches in order of increasing distance from the start
- Suppose you had some estimate **h(*v*)** of *v*'s distance from the goal
  - we call that a heuristic
- Now we sort the priority queue by estimated total distance
  - (known) distance from start + estimated distance to the goal
  - id est: dist(*V*) + h(*V*)

A*

- Almost identical to Dijkstra's, but uses dist(V) + h(V) to sort the priority queue
- Guaranteed to find the optimal path provided that **h(*V*)** never overestimates the distance of V to the goal
    - referred to as an optimistic algorithm
    - the proof is basically the same as for Dijkstra's
    - in fact, you may notice that Dijkstra's algorithm is just A* where the heuristic functions is constant
- The standard algorithm for computing paths in space
    - e.g., Google Maps, robotics, game AI, etc.
    - there are others, but this is bread and butter
- Most common heuristic is **h(*V*)** = || position(V) - position(*goal*) ||

A* is Way More Efficient (if you've got a heuristic)



Reading

- I can’t (and don’t want to) assign y’all homework, but this was in Ian’s slide deck so here is a relevant reading that I assume that he wants you to read:
    - CLR chapter 24.3 (Dijkstra’s Algorithm)