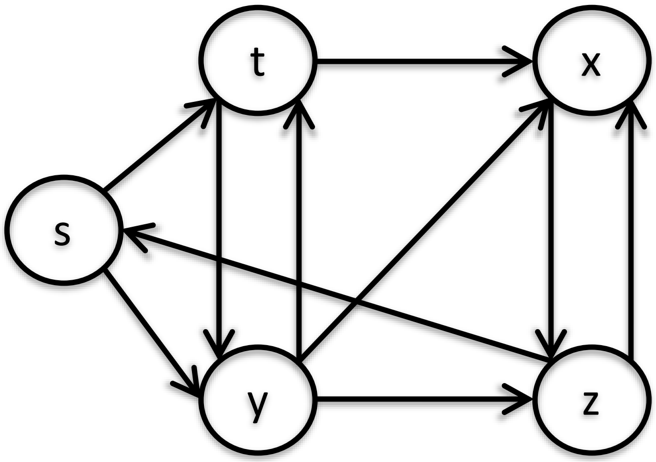
CS 10:
Problem solving via Object Oriented Programming

Shortest Path

Agenda

1. DFS and BFS on complex graph
2. Shortest-path simulation
3. Dijkstra’s algorithm
4. A* search
5. Implicit graphs

Last class we looked simple graphs, today we look at more complicated graphs

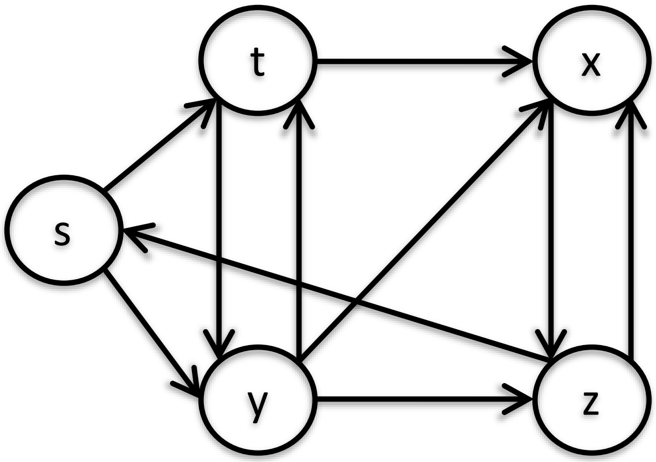


Graph with directed edges and several cycles

Depth First Search (DFS)

- Use a **Stack**
- Move forward until can't proceed farther
- Go back to last decision point and try another edge

DFS creates a tree of all reachable vertices

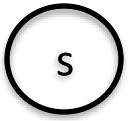


Graph with directed edges and several cycles

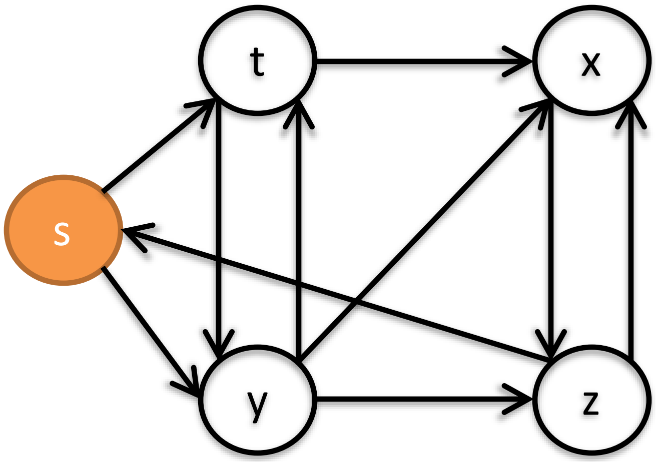
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack



DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

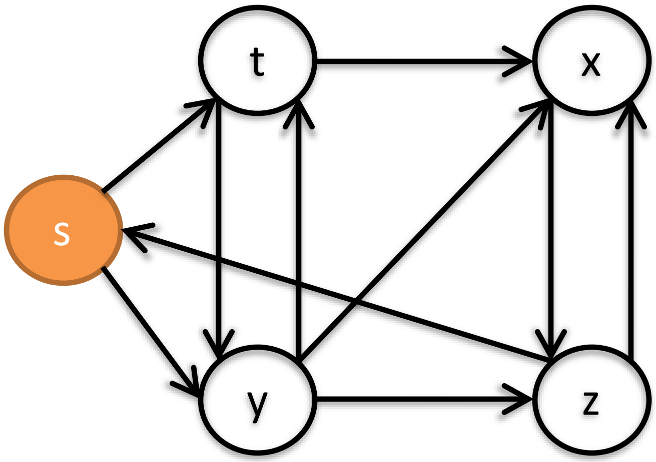
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
     u = stack.pop()
     if !u.visited
          u.visited = true
          (maybe do something while here)
          for v ∈ u.adjacent
                  if !v.visited
                    stack.push(v)
```

Stack

Pop -> s, mark visited

DFS creates a tree of all reachable vertices

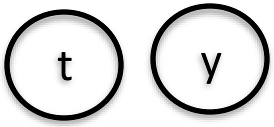


Graph with directed edges and several cycles

DFS algorithm

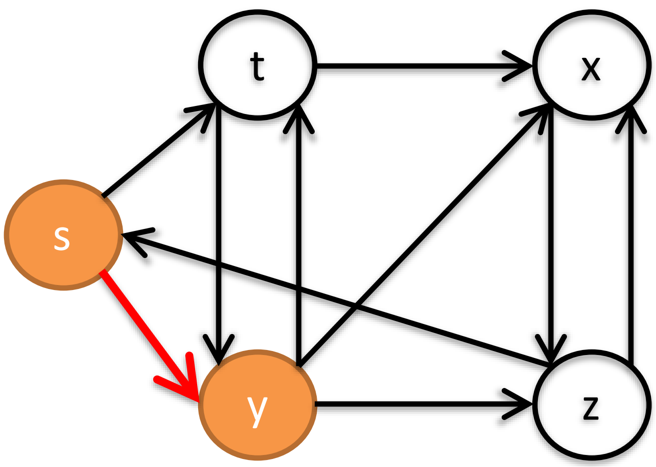
```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
            if !v.visited
              stack.push(v)
```

Stack



Push *s* unvisited neighbors

DFS creates a tree of all reachable vertices



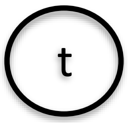
Graph with directed edges and several cycles

Red edges are discovery edges

DFS algorithm

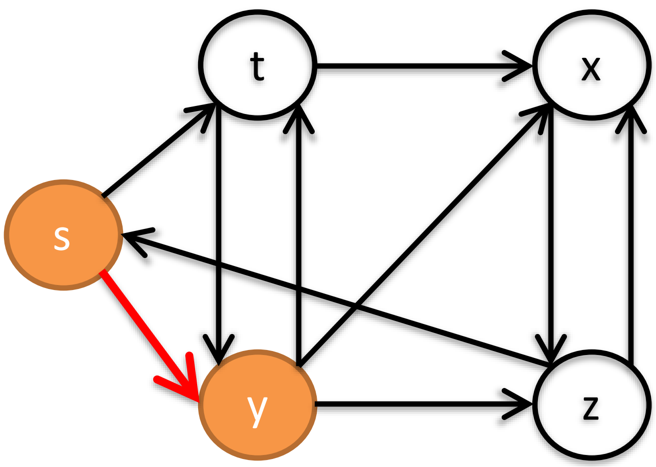
```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
     u = stack.pop()
     if !u.visited
          u.visited = true
          (maybe do something while here)
          for v ∈ u.adjacent
                  if !v.visited
                    stack.push(v)
```

Stack



Pop -> y, mark visited

DFS creates a tree of all reachable vertices



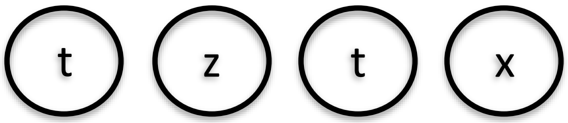
Graph with directed edges and several cycles

Note: t is in Stack twice because two edges lead to t from explored vertices

DFS algorithm

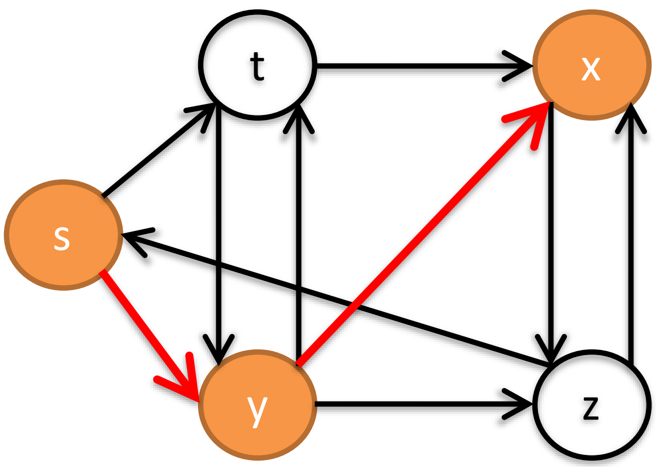
```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack



Push *y* unvisited neighbors

DFS creates a tree of all reachable vertices

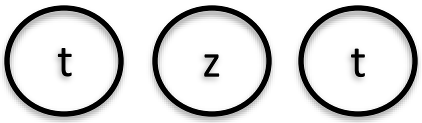


Graph with directed edges and several cycles

DFS algorithm

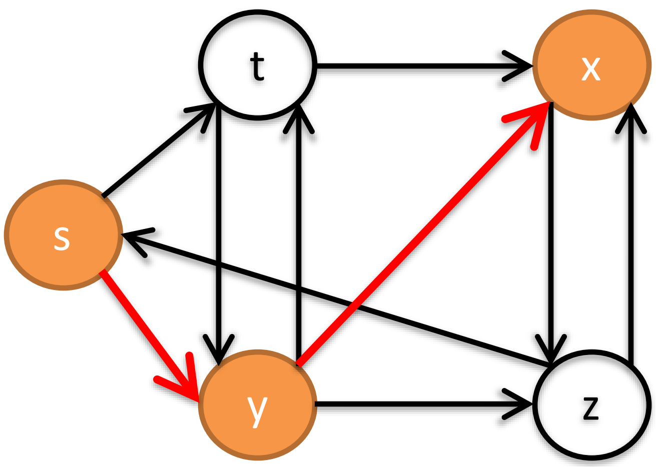
```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
      u = stack.pop()
      if !u.visited
            u.visited = true
            (maybe do something while here)
            for v ∈ u.adjacent
                  if !v.visited
                    stack.push(v)
```

Stack



Pop -> x, mark visited

DFS creates a tree of all reachable vertices

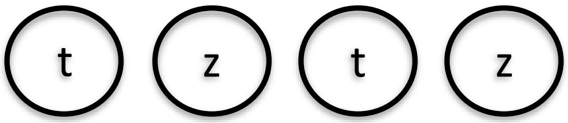


Graph with directed edges and several cycles

DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
            if !v.visited
              stack.push(v)
```

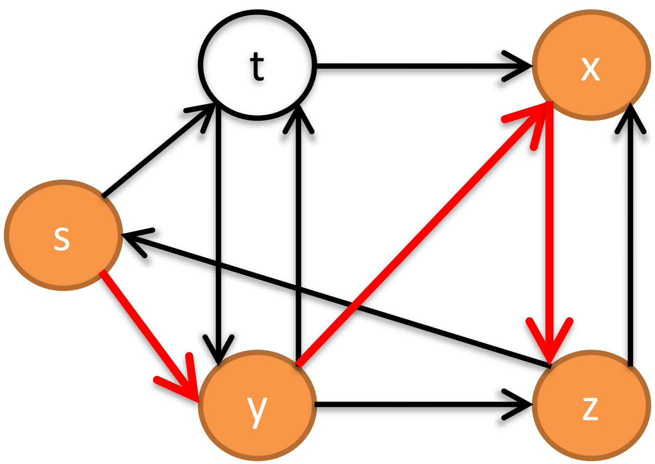
Stack



Push *x* unvisited neighbors

Note: z in Stack twice because two edges lead to z from explored vertices

DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

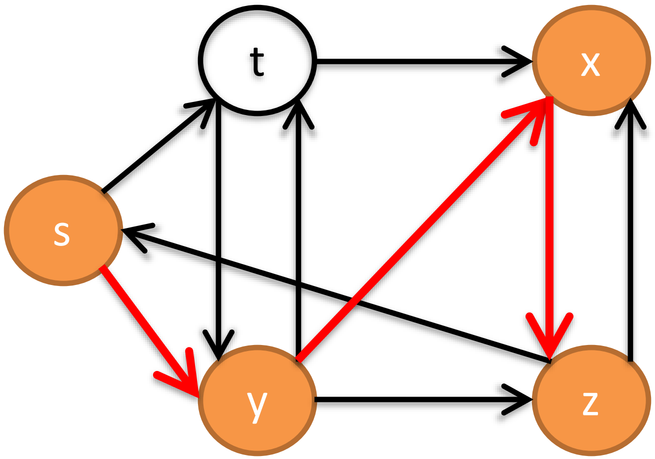
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
     u = stack.pop()
     if !u.visited
          u.visited = true
          (maybe do something while here)
          for v ∈ u.adjacent
                  if !v.visited
                    stack.push(v)
```

Stack

Pop -> z, mark visited

DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

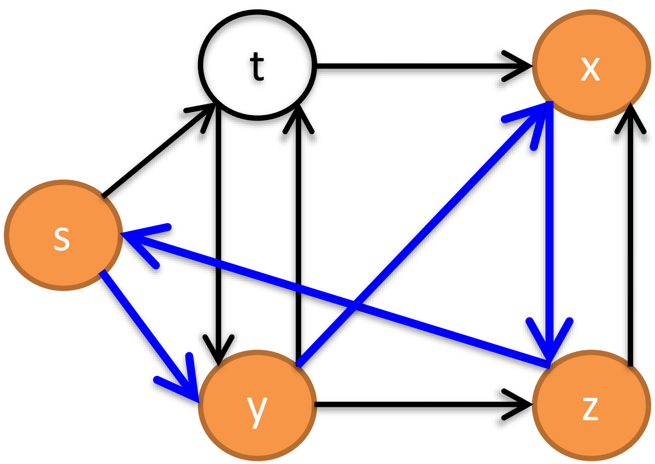
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack

All *z* neighbors (x,s) visited

DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

Found cycle!
s is an already visited neighbor

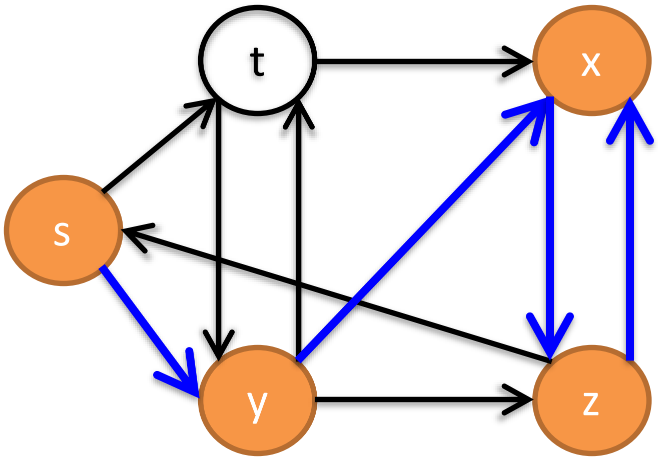
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack

All *z* neighbors (x,s) visited

DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

Found cycle!
s is an already visited neighbor (so is x)

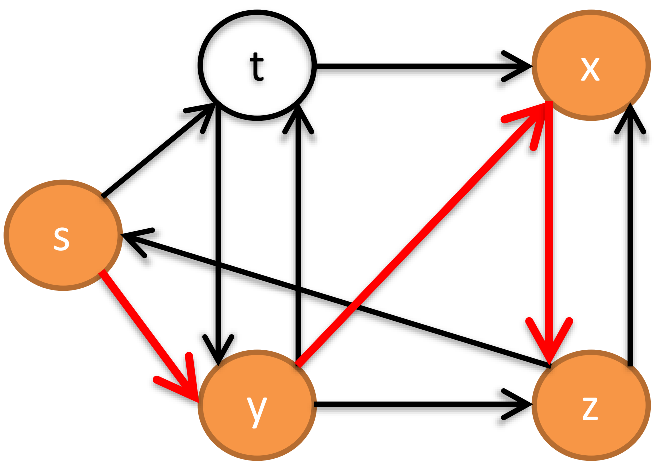
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
     u = stack.pop()
     if !u.visited
          u.visited = true
          (maybe do something while here)
          for v ∈ u.adjacent
                  if !v.visited
                    stack.push(v)
```

Stack

All *z* neighbors (x,s) visited

DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

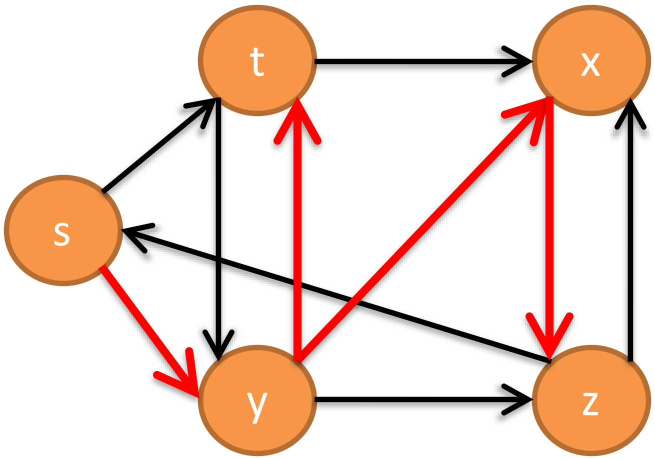
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack

All *z* neighbors (x,s) visited

DFS creates a tree of all reachable vertices

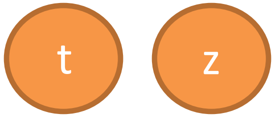


Graph with directed edges and several cycles

DFS algorithm

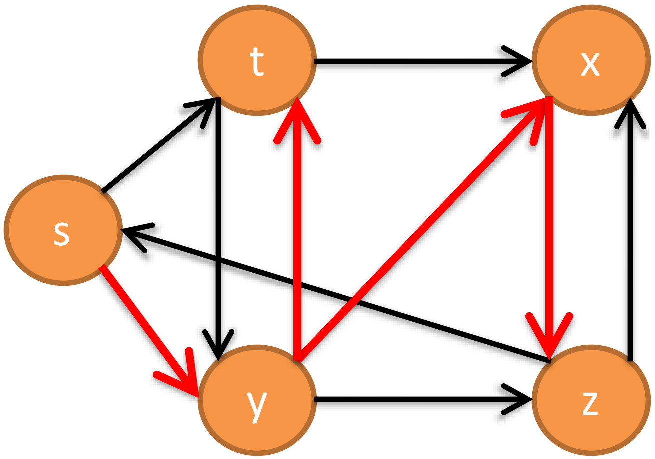
```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
     u = stack.pop()
     if !u.visited
          u.visited = true
          (maybe do something while here)
          for v ∈ u.adjacent
                  if !v.visited
                    stack.push(v)
```

Stack



Pop -> t, mark visited

DFS creates a tree of all reachable vertices

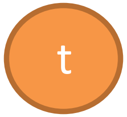


Graph with directed edges and several cycles

DFS algorithm

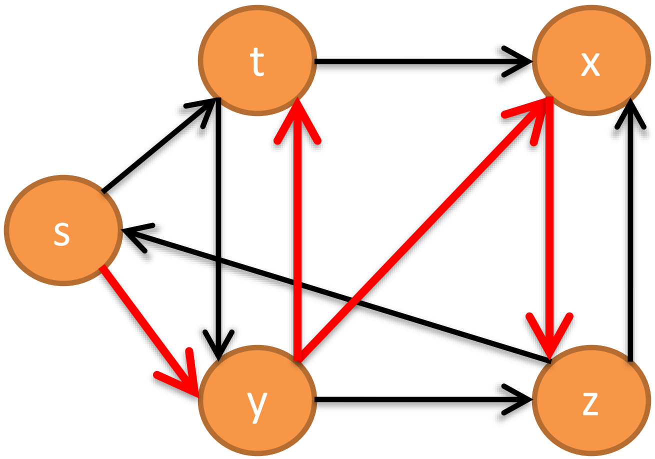
```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack



Pop -> z, skip, already visited

DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

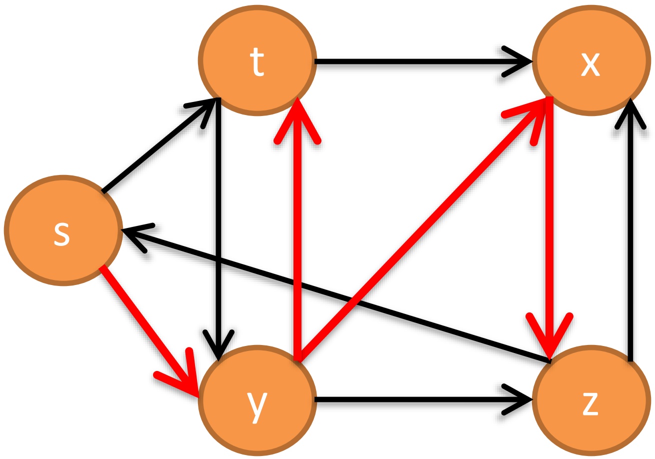
DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

Stack

Pop -> t, skip, already visited

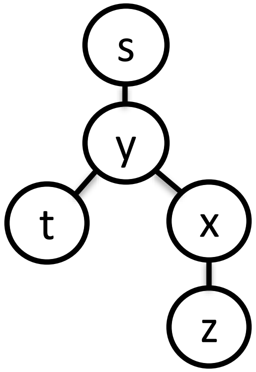
DFS creates a tree of all reachable vertices



Graph with directed edges and several cycles

DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

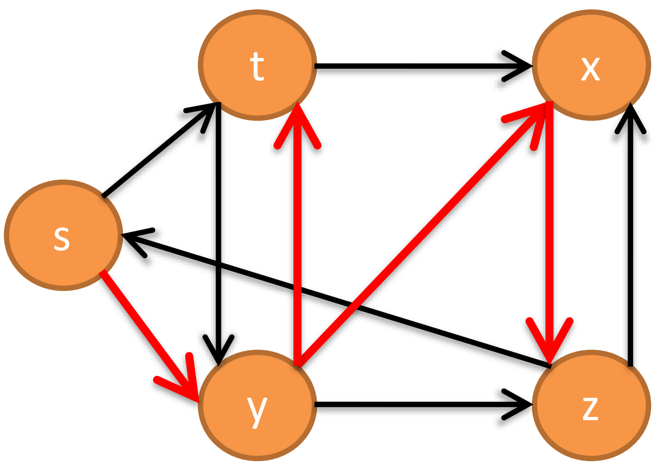


Stack

Done

- **Red lines indicate a tree (root and no cycles)**
- **Can traverse tree to find path from *s* to others**

DFS creates a tree of all reachable vertices



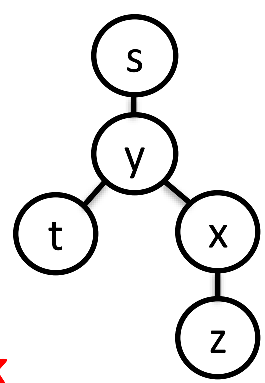
Graph with directed edges and several cycles

Could DFS have produced another tree?

Yes, depends on the order vertices pushed onto Stack

DFS algorithm

```
stack.push(s) //start node
repeat until find goal vertex or stack
empty:
    u = stack.pop()
    if !u.visited
        u.visited = true
        (maybe do something while here)
        for v ∈ u.adjacent
              if !v.visited
                stack.push(v)
```

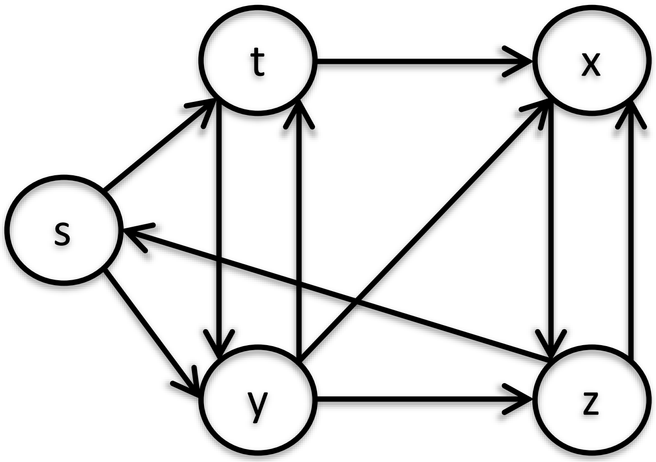


Stack

Done

- **Red lines indicate a tree (root and no cycles)**
- **Can traverse tree to find path from *s* to others**

BFS finds shortest path to all reachable vertices

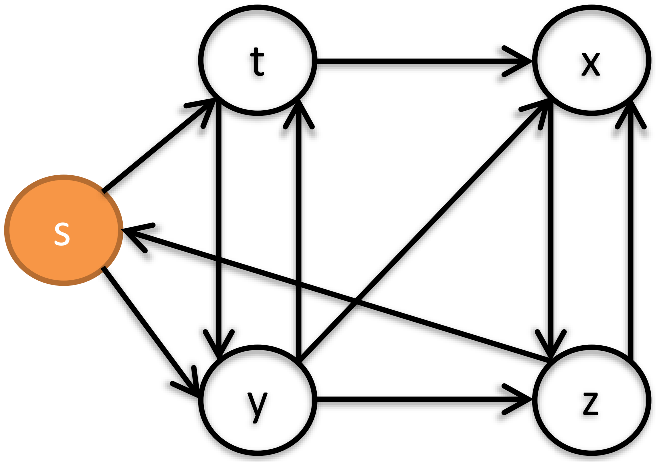


Graph with directed edges and several cycles

Breadth First Search (BFS)

- Use a **Queue**
- Ripple outward from start
- Finds *shortest* path to each node from start (DFS finds *a* path)

BFS finds shortest path to all reachable vertices



Graph with directed edges and several cycles

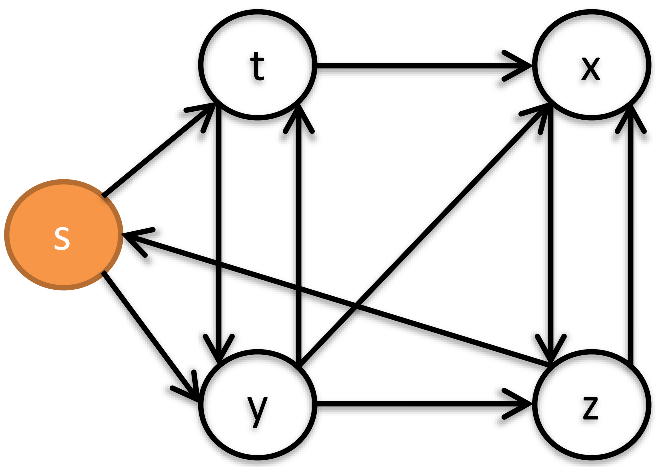
BFS algorithm

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
     u = dequeue()
     for v ∈ u.adjacent
          if !v.visited
               v.visited = true
               enqueue(v)
```

Queue

enqueue(s)

BFS finds shortest path to all reachable vertices



Graph with directed edges and several cycles

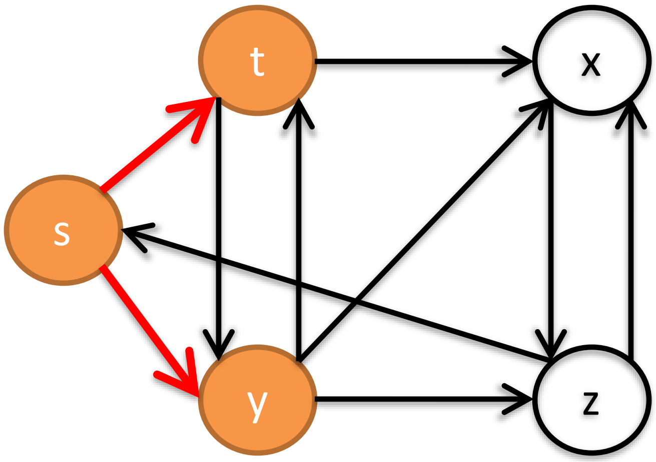
BFS algorithm

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue

dequeue -> s

BFS finds shortest path to all reachable vertices

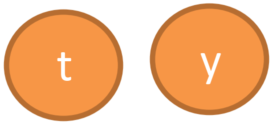


Graph with directed edges and several cycles

BFS algorithm

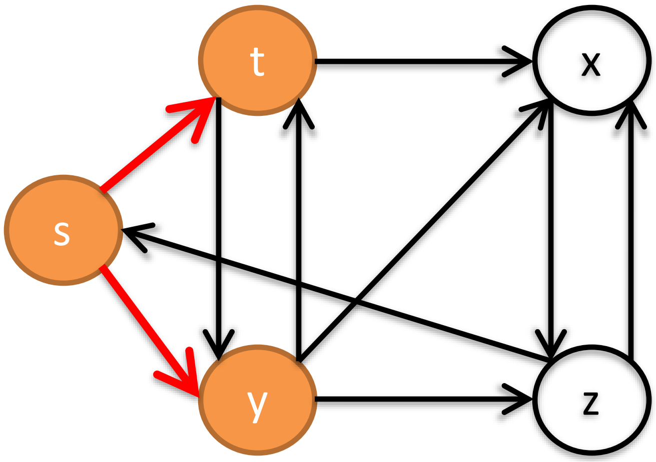
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue



***enqueue s* unvisited adjacent**

BFS finds shortest path to all reachable vertices

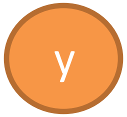


Graph with directed edges and several cycles

BFS algorithm

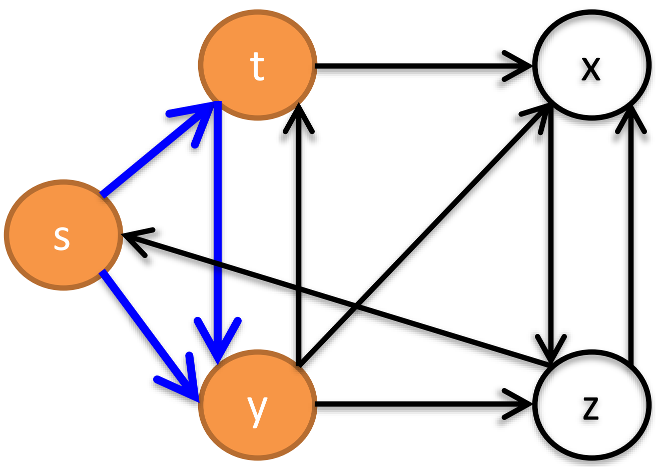
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue



dequeue -> t

BFS finds shortest path to all reachable vertices



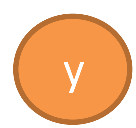
Graph with directed edges and several cycles

Adjacent vertex *y* is visited
Found cycle?
NO! Just another way to get to *y*
DFS easier for cycle detection

BFS algorithm

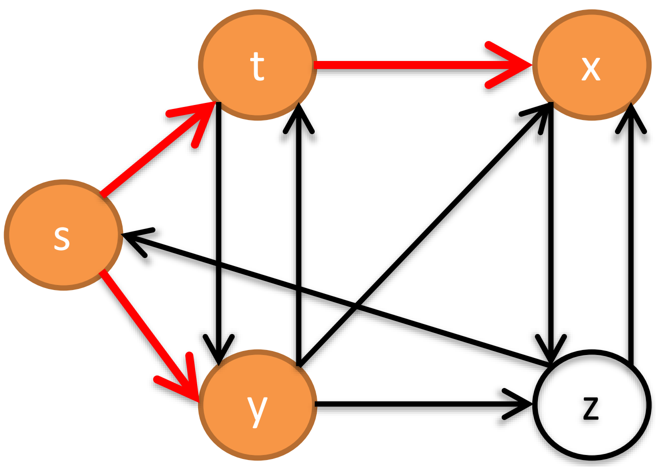
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue



***enqueue t* unvisited adjacent**

BFS finds shortest path to all reachable vertices

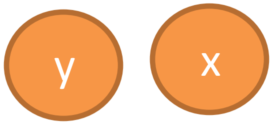


Graph with directed edges and several cycles

BFS algorithm

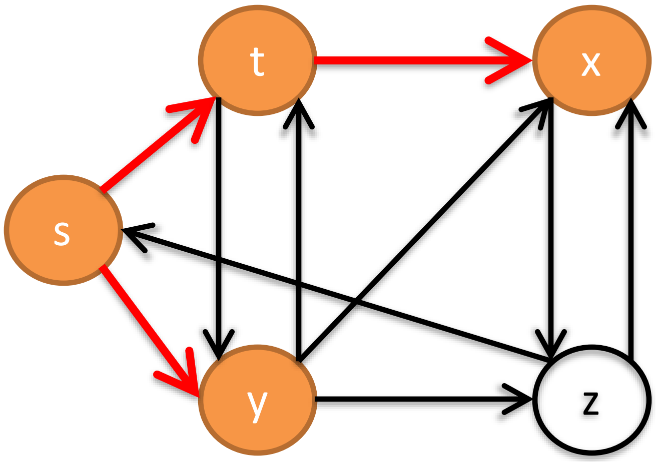
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue



***enqueue t* unvisited adjacent**

BFS finds shortest path to all reachable vertices

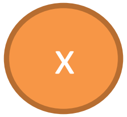


Graph with directed edges and several cycles

BFS algorithm

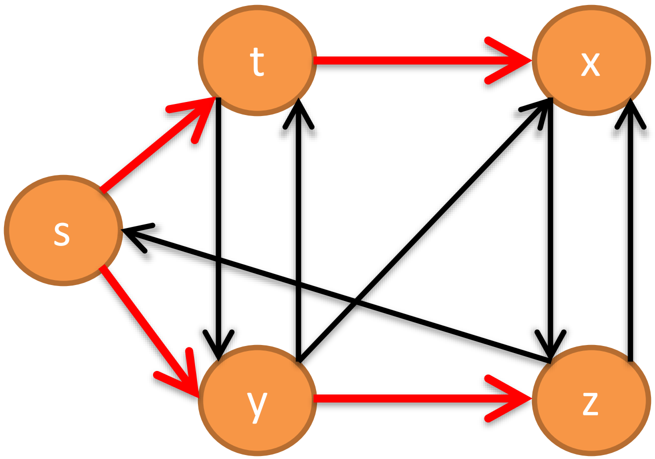
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue



dequeue -> y

BFS finds shortest path to all reachable vertices

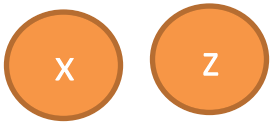


Graph with directed edges and several cycles

BFS algorithm

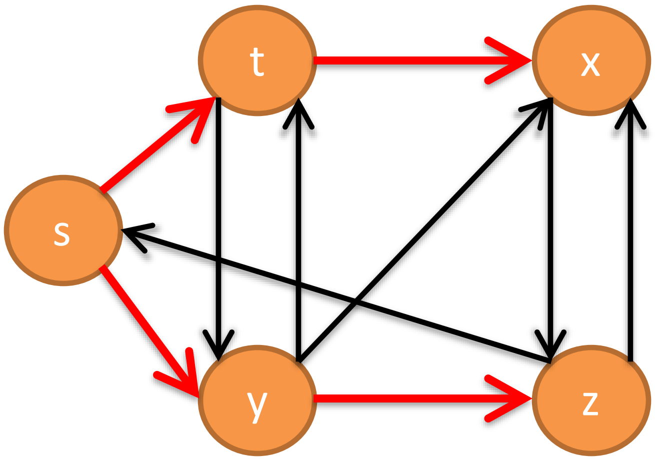
```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue



***enqueue y* unvisited adjacent**

BFS finds shortest path to all reachable vertices



Graph with directed edges and several cycles

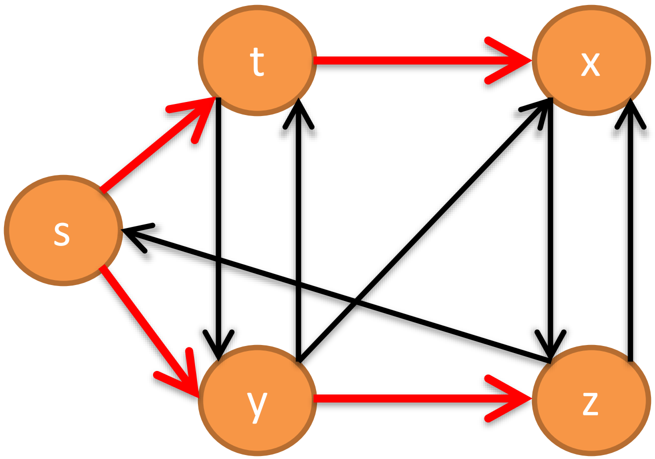
BFS algorithm

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue

dequeue -> x

BFS finds shortest path to all reachable vertices



Graph with directed edges and several cycles

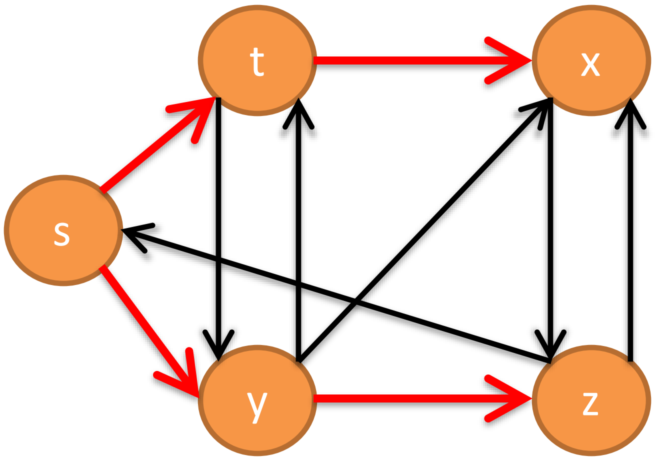
BFS algorithm

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue

dequeue -> z

BFS finds shortest path to all reachable vertices



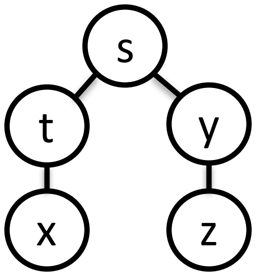
Graph with directed edges and several cycles

BFS algorithm

```
enqueue(s) //start node
s.visited = true
repeat until find goal vertex or
queue empty:
    u = dequeue()
    for v ∈ u.adjacent
        if !v.visited
            v.visited = true
            enqueue(v)
```

Queue

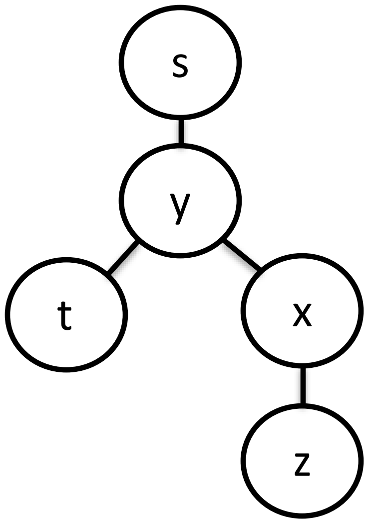
Done



- **Red lines indicate a tree (root and no cycles)**
- **Can traverse tree to find path from *s* to others**

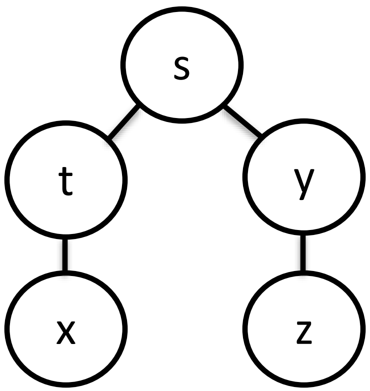
DFS and BFS can create different trees, both find path from start to other vertices

DFS



- Has path from start to all other reachable vertices
- No cycles
- Path s to z = 3 edges

BFS



- Has <u>shortest</u> path from start to all other reachable vertices
- No cycles
- Path s to z = 2 edges

Why do we care if path has cycles?

If cycles, could get caught in endless loop computing path from *s* to *end*

No cycles with tree

Agenda

1. DFS and BFS on complex graph
2. Shortest-path simulation
3. Dijkstra’s algorithm
4. A* search
5. Implicit graphs

BFS considers the number of steps, but not how long each step could take

Fastest driving route to Seattle from Hanover

Could try to take the most direct route

- Take local roads
- Try to keep on a line between Start and Goal

OR could try to take major highways:

- New York
- Chicago
- Seattle

Drive time estimates from travelmath.com

Now we consider the idea that not all steps are the same

Fastest driving route to Seattle from Hanover

BFS would choose the direct route (one leg)

Highway travel makes larger number of steps more attractive

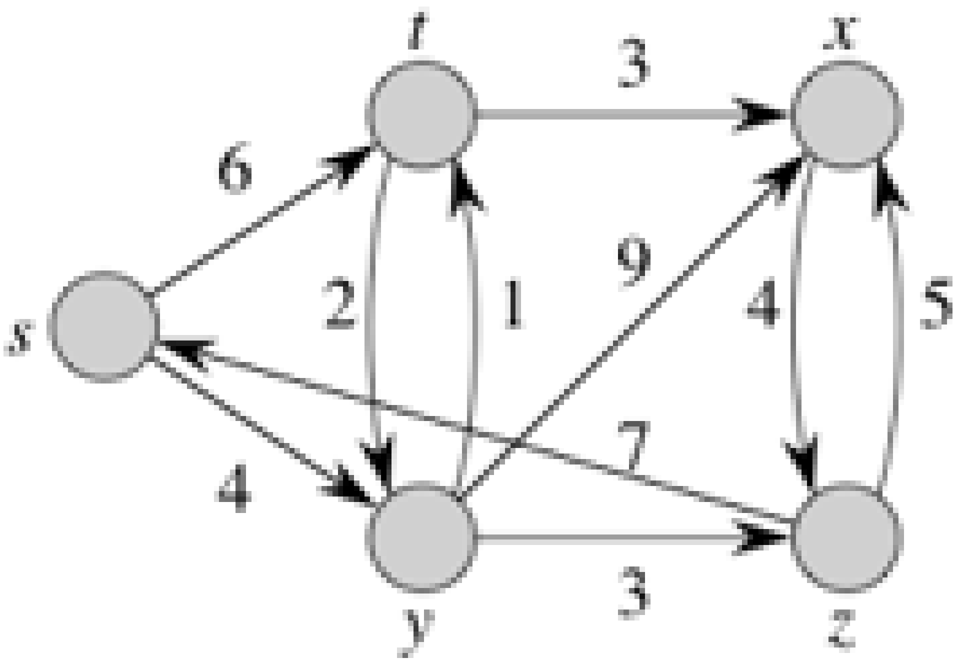
Note: our metric now is driving time, not number of edges, however total distance is longer!

Need a way to account for the idea that each step might have different "weight" (drive time here)

Drive time estimates from travelmath.com

With no negative edge weights, we can use Dijkstra's algorithm to find short paths

Goal: find shortest path to all nodes considering edge weights



Use weight as edge label (e.g., driving distance between nodes)

Start at node `s` (single source)

Find path with smallest sum of weights to all other nodes

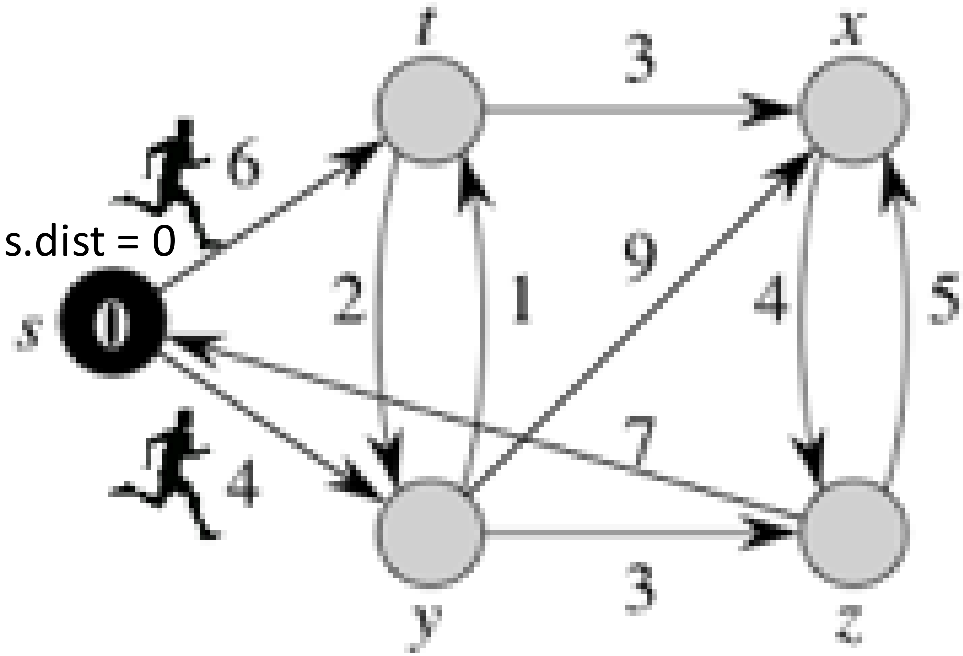
Store shortest path weights in `v.dist` instance variable

Keep back pointer to previous node in `v.pred`

Updated `v.dist` and `v.pred` if find shorter path later found

To get intuition, imagine sending runners from the start to all adjacent nodes

Time 0



Weights must be non-negative
Why?
Could end up arriving before you left!
If edge from *t* to *y* was -2, then could back up in time

Simulation

`s.dist = 0`

Runners take edge weight minutes to arrive at adjacent nodes

Runners arrive at node `v`:

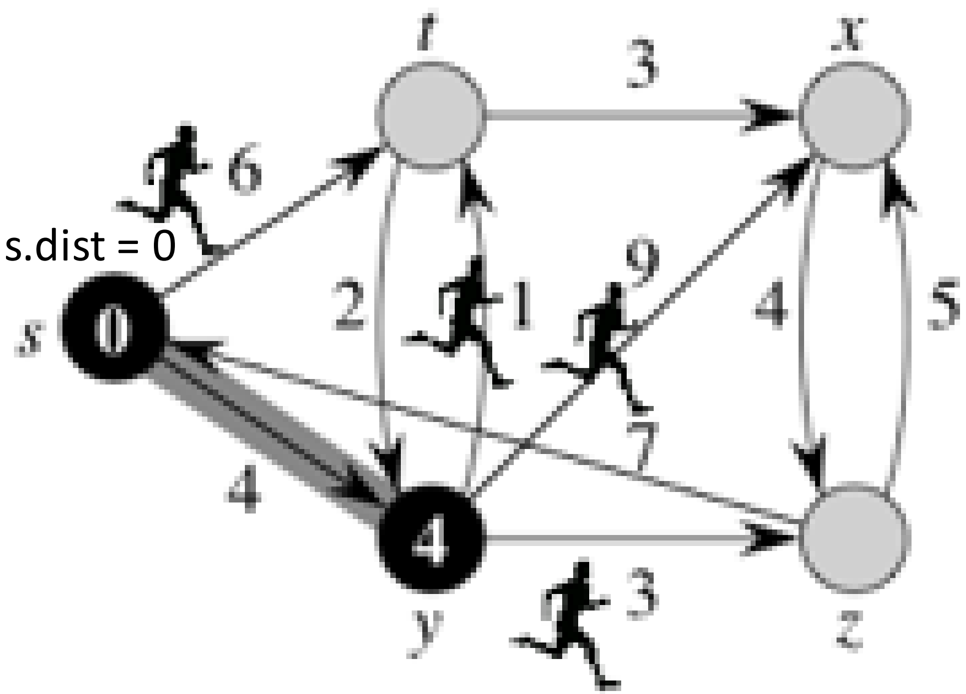
- Record arrival time in `v.dist`
- Record prior node in `v.pred`

Runners immediately leave for an adjacent node

Runners leave `s` for `y` and `t`

Imagine we send runners from the start to all adjacent nodes

Time 4



`y.dist = 4`
`y.pred = s`

Runner arrives at `y` in 4 minutes

- Record `y.dist = 4`
- Record `y.pred = s`

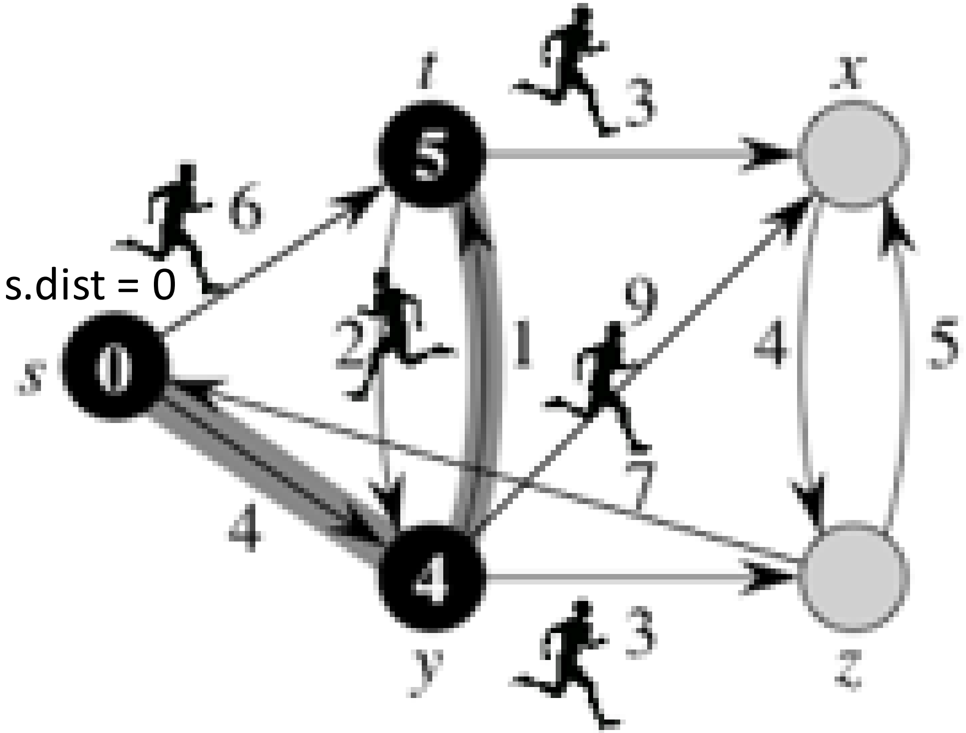
Runners leave `y` for adjacent nodes `t, x,` and `z`

Runner from `s` has not reached `t` yet

Imagine we send runners from the start to all adjacent nodes

Time 5

`t.dist = 5`
`t.pred = y`



`y.dist = 4`
`y.pred = s`

Runner from `y` arrives at `t` at time 5

- `t.dist = 5`
- `t.pred = y`

Runners from `s` still hasn't made it to `t`

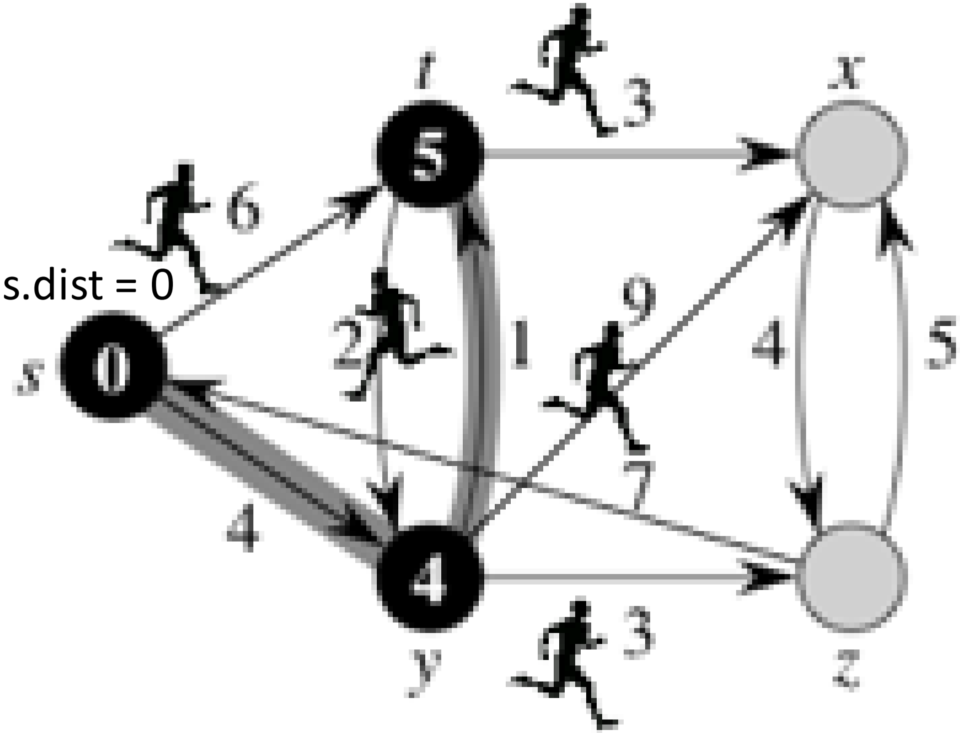
Runners leave `t` for adjacent nodes `x` and `y`

Imagine we send runners from the start to all adjacent nodes

Time 6

```
t.dist = 5
t.pred = y
```

s.dist = 0




```
y.dist = 4
y.pred = s
```

Runner from `s` arrives at `t` at time 6

Runner from `y` has already arrived, so best route is from `y`, not direct from `s`

Do <u>not</u> update `t.dist` and `t.pred`

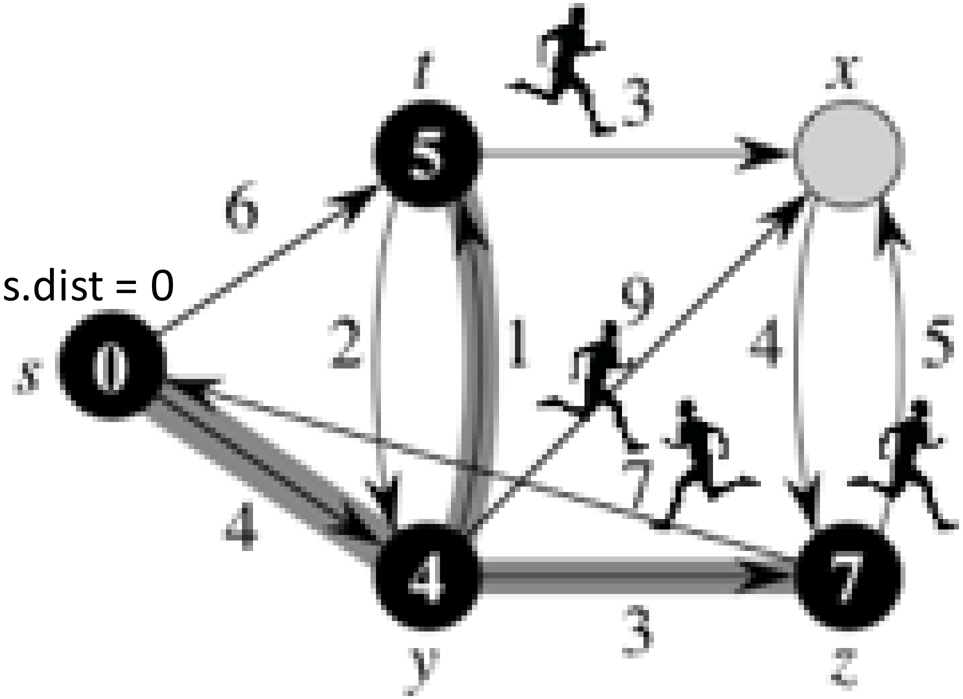
NOTE: BFS would have chosen the direct route to `t`

Imagine we send runners from the start to all adjacent nodes

Time 7

`t.dist = 5`
`t.pred = y`

s.dist = 0



`y.dist = 4`
`y.pred = s`

`z.dist = 7`
`z.pred = y`

Runner from `y` arrives at `z` at time 7

Record `z.dist = 7` and `z.pred = y`

Runners leave `z` for `s` and `x`

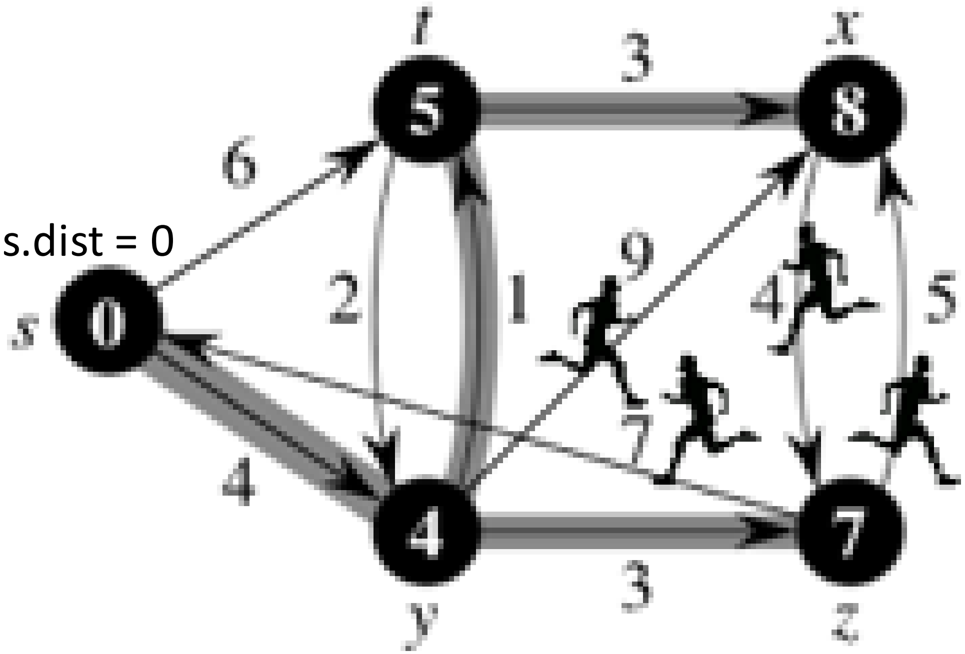
Imagine we send runners from the start to all adjacent nodes

Time 8

`t.dist = 5`
`t.pred = y`

`x.dist = 8`
`x.pred = t`

s.dist = 0



`y.dist = 4`
`y.pred = s`

`z.dist = 7`
`z.pred = y`

- **What ADT have we seen that works well for a simulation of this nature?**
- **PriorityQueue!**

Runner from `t` arrives at `x` at time 8

`x.dist = 8, x.pred = t`

All nodes explored

Now have shortest path from `s` to all other nodes

Shaded lines indicate best path to each node

Path forms a tree on graph

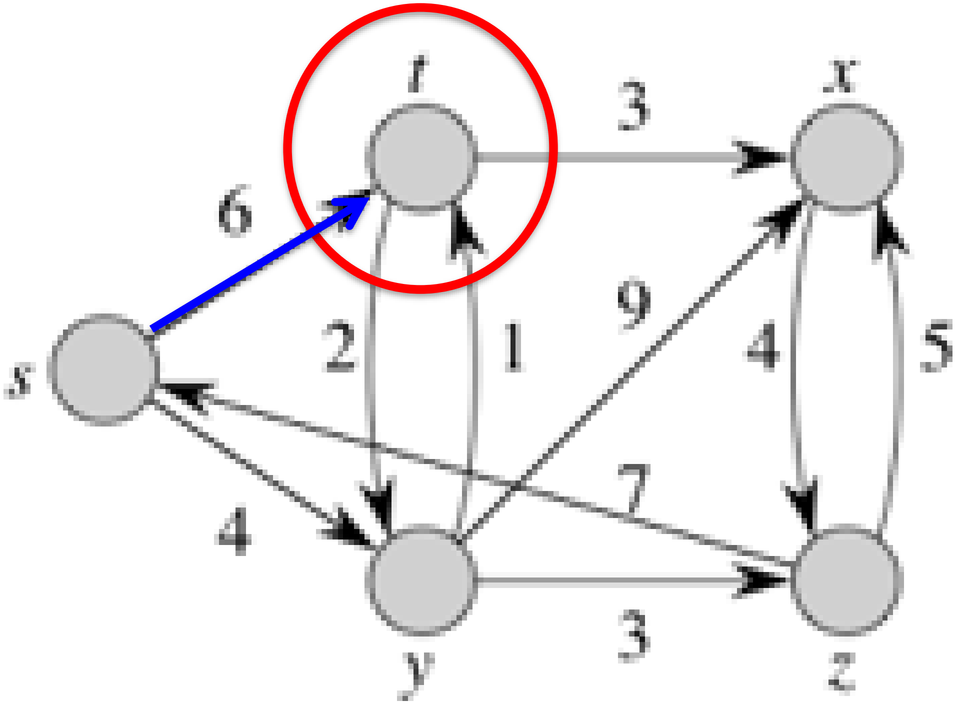
To find path from start to end: start at end and back track predecessors back to start

Agenda

1. DFS and BFS on complex graph
2. Shortest-path simulation
3. Dijkstra’s algorithm
4. A* search
5. Implicit graphs

Dijkstra's algorithms works similarly but doesn't rely on waiting for runners

Dijkstra's algorithm



Overview

Start at `s`

Process all out edges at the same time

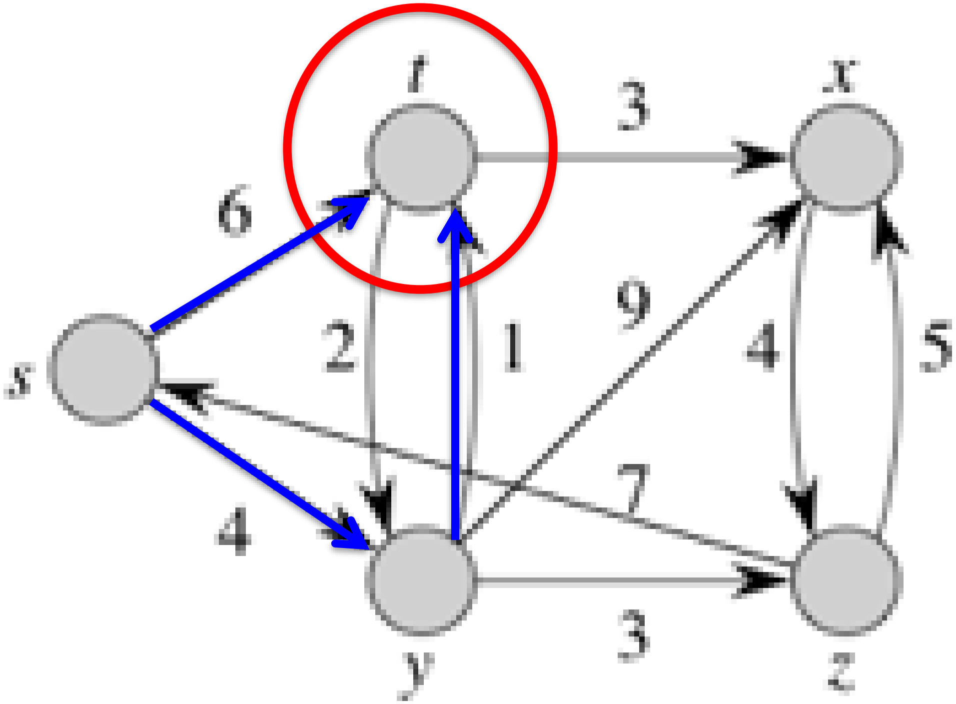
Compare distance to adjacent nodes with best so far

If current path < best, update best distance and predecessor node

Example: one hop from `s` set `t.dist = 6, t.pred = s`

Dijkstra’s algorithms works similarly but doesn’t rely on waiting for runners

Dijkstra’s algorithm



Overview

Start at `s`

Process all out edges at the same time

Compare distance to adjacent nodes with best so far

If current path < best, update best distance and predecessor node

Example: one hop from `s` set `t.dist = 6, t.pred = s`, then update `t.dist = 5, t.pred = y` on second hop

Dijkstra uses a Min Priority Queue with `dist` values as keys to get closest vertex

Dijkstra's algorithm starting from s

```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

Set up Min Priority Queue

Initialize `dist` and `pred`

Use `dist` as key for Min Priority Queue (initially infinite)

Initialize `s` distance to zero

While not all nodes have been explored

Get closest node based on distance (initially `s`)

Examine adjacent and relax

Dijkstra defines a relax method to update best path if needed

Dijkstra's relax method

```
void relax(u, v) {
  if (u.dist + w(u,v) < v.dist) {
    v.dist = u.dist + w(u,v);
    v.pred = u;
  }
}
```

Currently at vertex `u`, considering distance to vertex `v`

Check if distance to `u` + distance from `u` to `v` < best distance to `v` so far

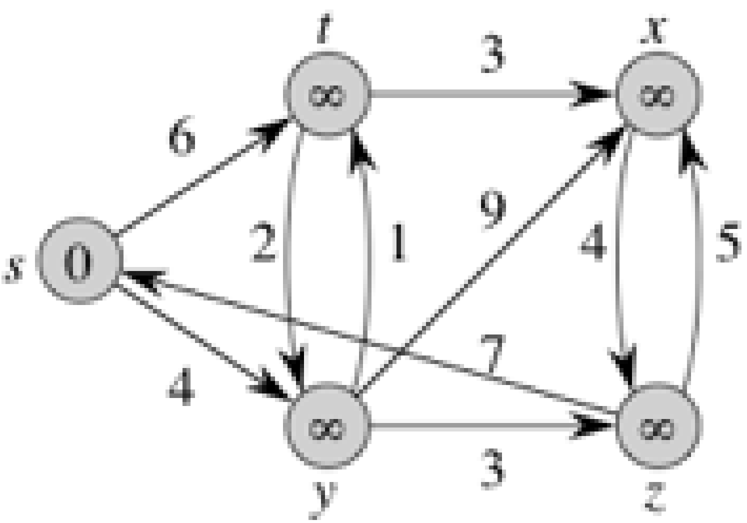
Distance from `u` to `v` is `w(u,v)`

If shorter total distance to `v` than previous, then update:

```
v.dist = u.dist + w(u,v)
v.pred = u
```

Example

Dijkstra's algorithm




```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

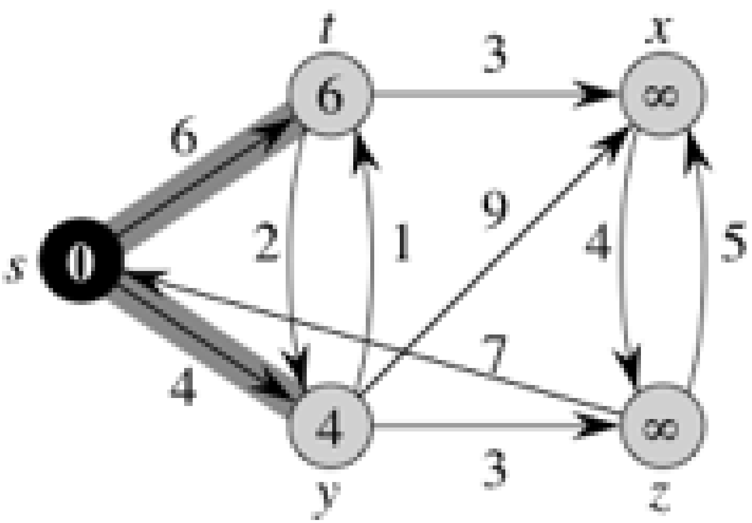
All nodes have distance `Infinity`, except Start with distance 0

Distances shown in center of vertices

extractMin() from Min Priority Queue first selects `s` (`dist =0`)

Example

Dijkstra's algorithm




```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

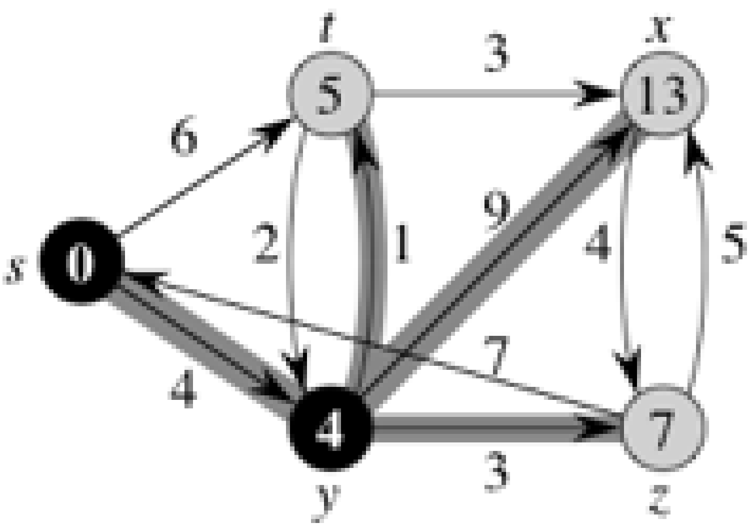
Loop over all adjacent nodes `v`

If distance less than smallest so far, then relax

That is the case here, so update `dist` and `pred` on `t` and `y`

Example

Dijkstra's algorithm




```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

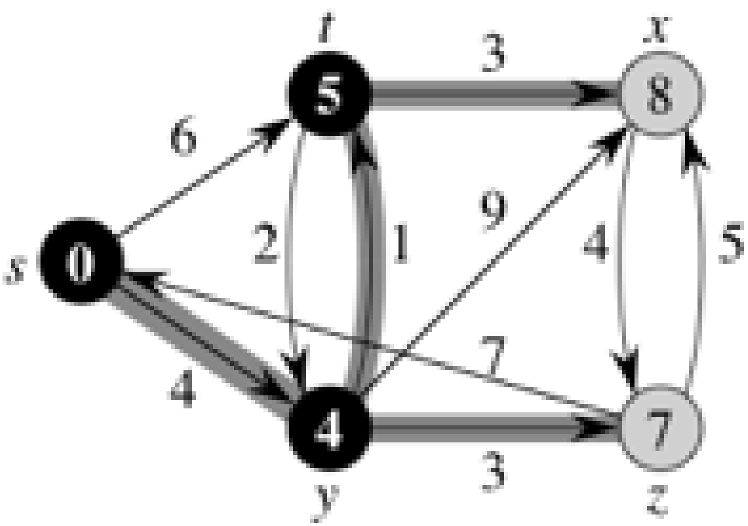
extractMin() now picks `y` (`dist=4`)

Look at adjacent `t`, `x`, and `z`

Relax each of them

Example

Dijkstra's algorithm




```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

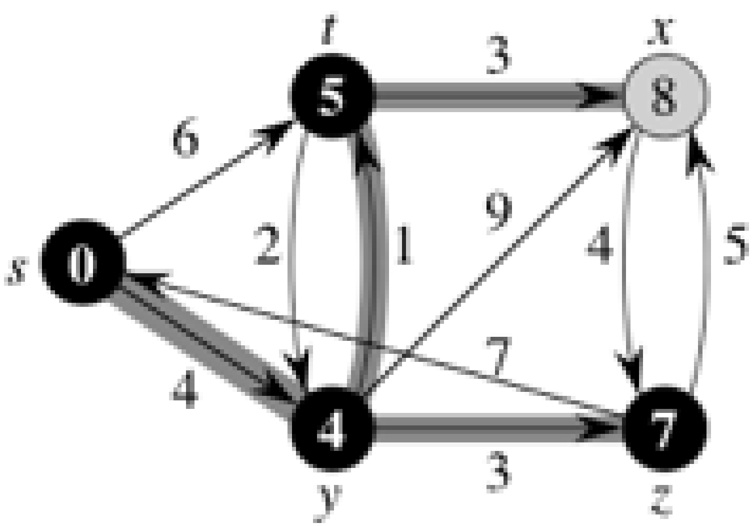
extractMin() now picks `t` (`dist =5`)

Look at adjacent `x` and `y`

Relax `x`, but not `y`

Example

Dijkstra's algorithm




```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

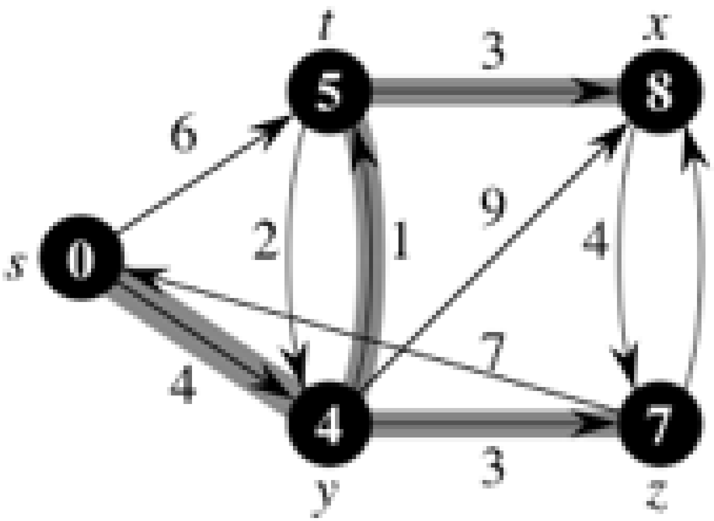
extractMin() now picks `z` (`dist = 7`)

Look at adjacent `x` and `s`

Do not relax `x` or `s`

Example

Dijkstra's algorithm




```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;
    v.pred = null;
    queue.enqueue(v);
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

extractMin() now picks `x` (`dist = 8`)

Look at adjacent `z`

Do not relax `z`

Done!

Run-time complexity is O(n log n + m log n)

Dijkstra's algorithm

- Assume n nodes and m edges
- Add and remove each vertex once in Priority Queue
- Relax each edge (and perhaps reduce key) once
- O(n*(insert time + extractMin) + m*(reduceKey))
- If using heap-based Priority Queue, then each queue operation takes O(log n)
- Total = O(2n log n + m log n) = O(n log n + m log n)

- Can implement with a Fibonacci heap with $O(n^2)$
- Take CS31 to find out how!

Agenda

1. DFS and BFS on complex graph
2. Shortest-path simulation
3. Dijkstra’s algorithm
4. A* search
5. Implicit graphs

Dijkstra's algorithm can find shortest path but what about a huge graph?

Consider a GPS device that finds path from current location to destination

How does it find path quickly?

Roads from Hanover can lead up to Alaska or down to Argentina!

Does the "little" GPS computer consider all those roads?

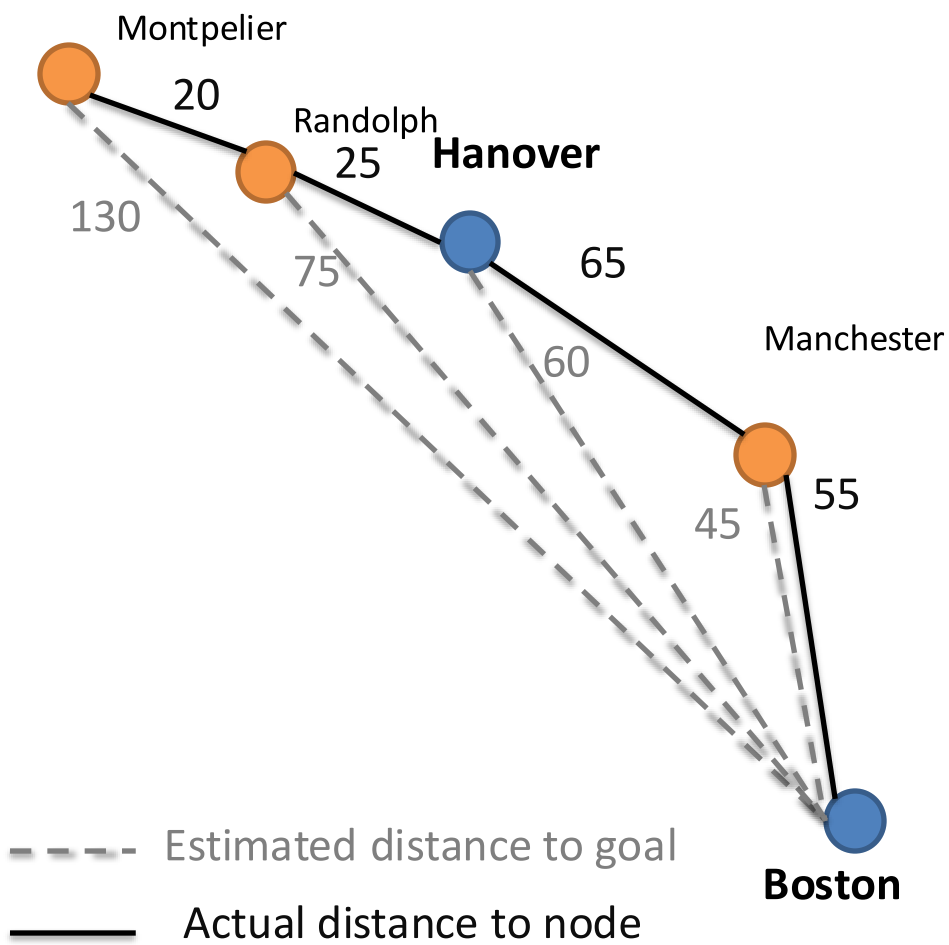
NO! It uses variant of Dijkstra called A* to rule out paths that will clearly be longer than best path discovered so far

A* is able to "stop early", without considering every possible path

Interview example

A* can help find the best path between two nodes faster than Dijkstra

A* algorithm from Hanover to Boston



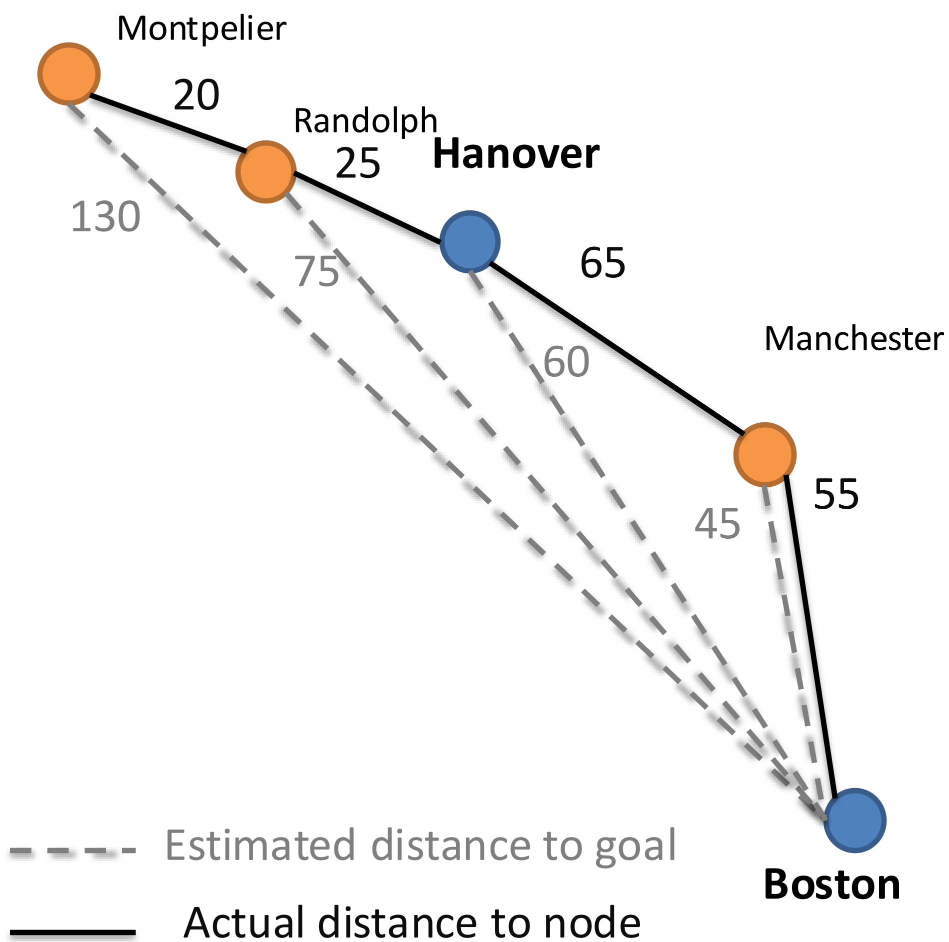
Estimate distance to goal (maybe use Euclidean distance) from each node

Estimate must be ≤ actual distance (admissible)

Distances non-negative (distance monotonically increasing; driving further cannot make trip shorter!)

A* can help find the best path between two nodes faster than Dijkstra

A* algorithm from Hanover to Boston

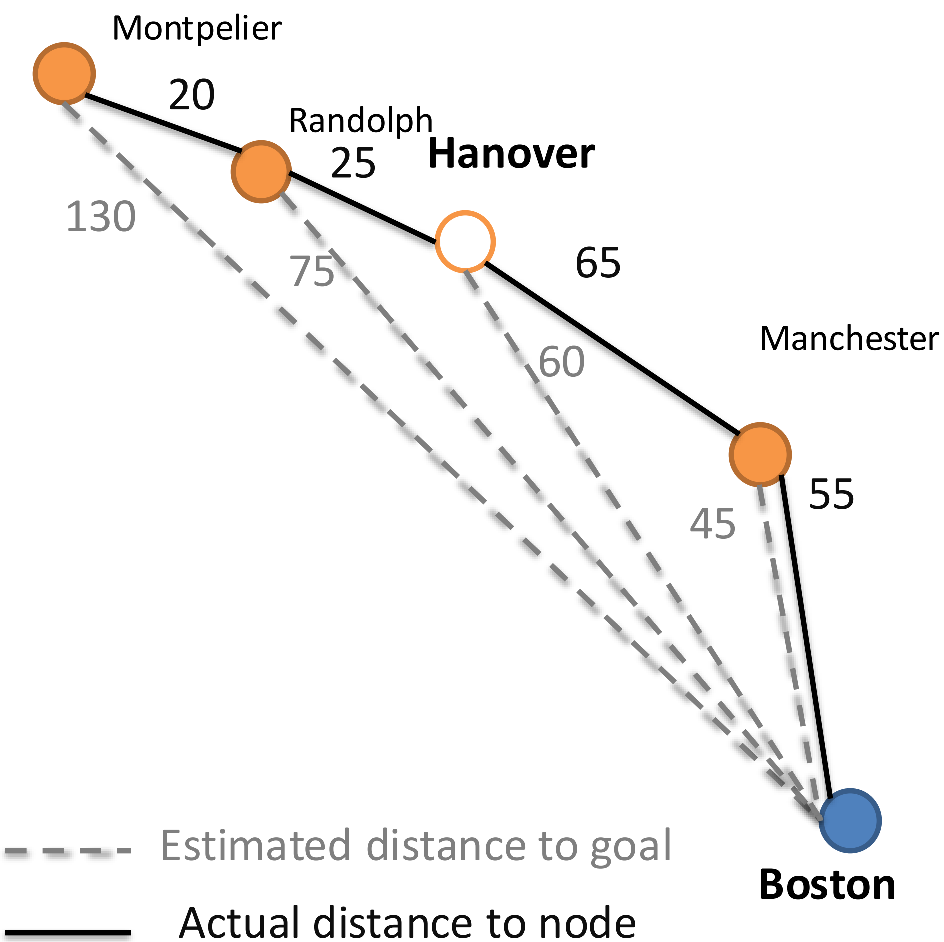


Keep Priority Queue using distance so far + estimate for each node ("open set")

Keep "closed set" where we know we already found the best route

A* can help find the best path between two nodes faster than Dijkstra

Step 1: Start at Hanover, add to Open set



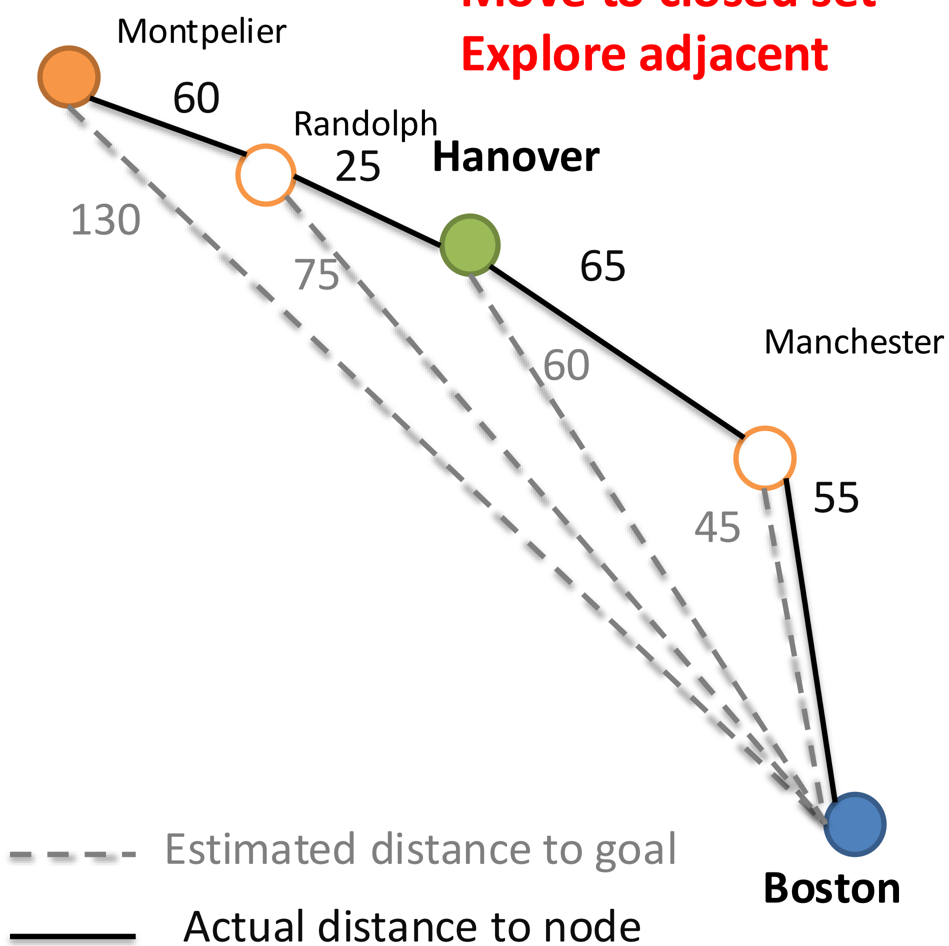
Open set (Priority Queue)

Hanover 0 + 60 = 60

Closed set

A* can help find the best path between two nodes faster than Dijkstra

Step 2: extractMin from Open set and explore adjacent



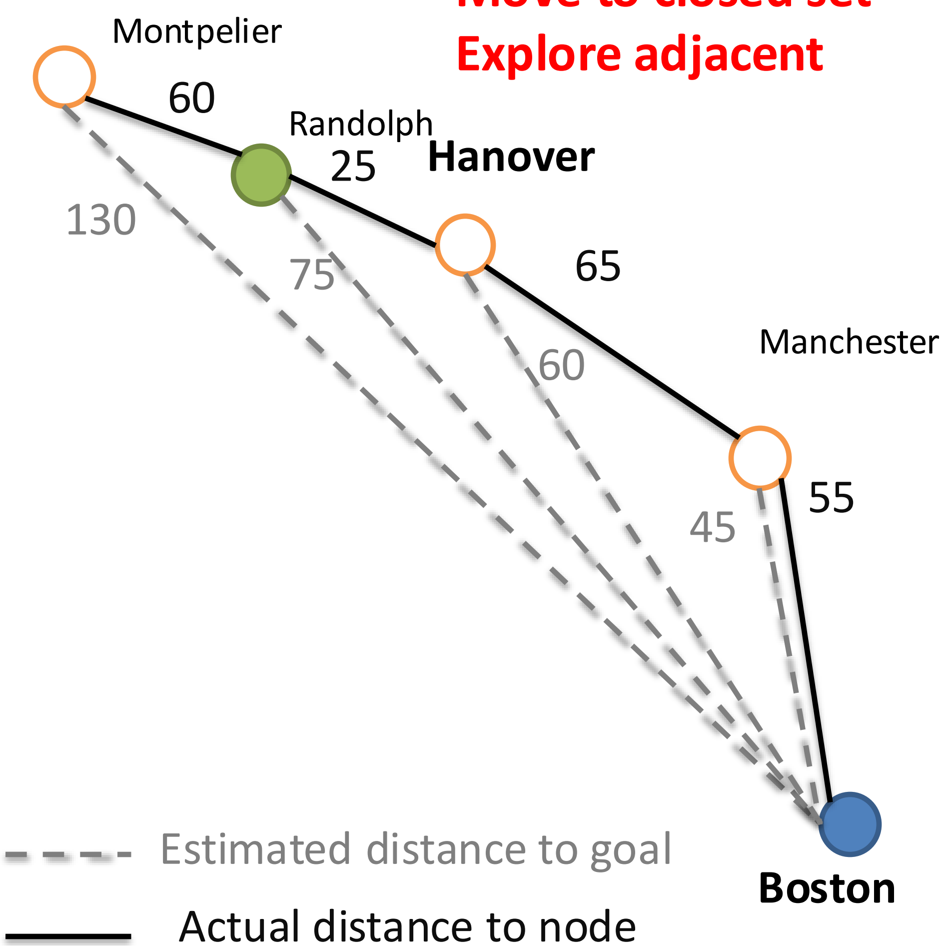
Extract Hanover
Move to closed set
Explore adjacent

Open set (Priority Queue)
Randolph 25 + 75 = 100
Manchester = 65 + 45 = 110

Closed set
Hanover 0 + 60 = 60

A* can help find the best path between two nodes faster than Dijkstra

Step 3: extractMin from Open set and explore adjacent



Extract Randolph
Move to closed set
Explore adjacent

Open set (Priority Queue)

Manchester = 65 + 45 = 110

Montpelier = 25 + 60 + 130 = 215

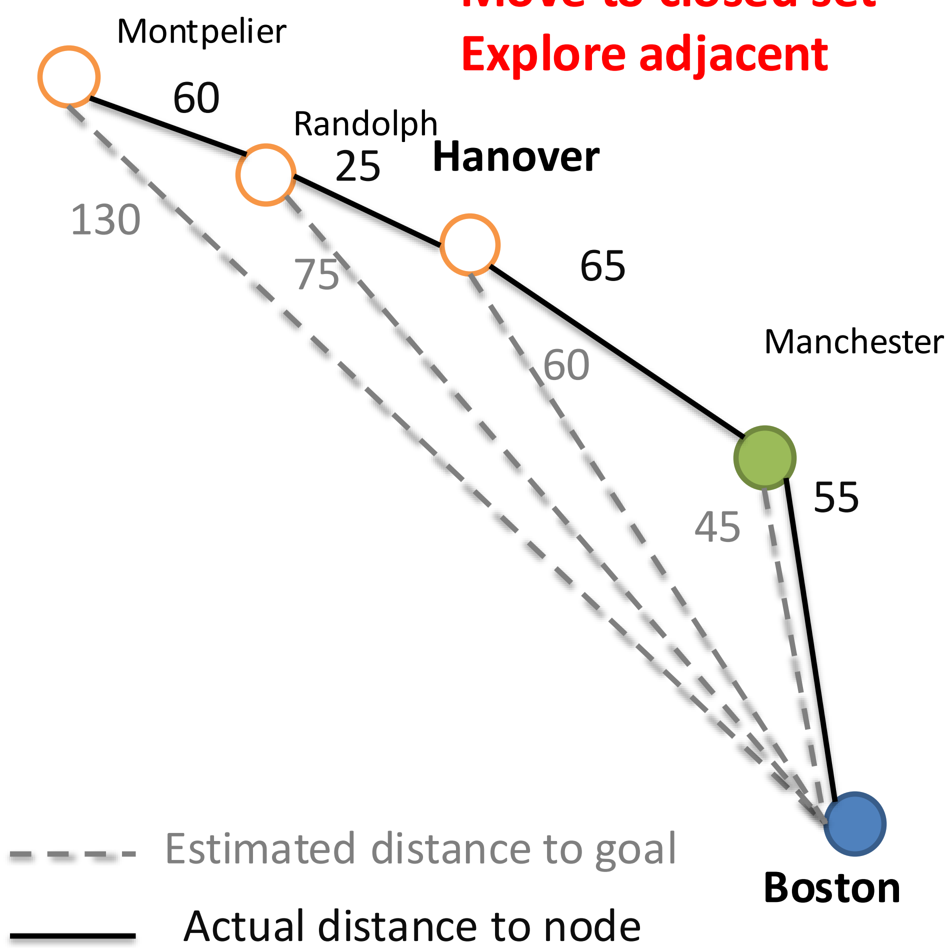
Closed set

Hanover 0 + 60 = 60

Randolph 25 + 75 = 100

A* can help find the best path between two nodes faster than Dijkstra

Step 4: extractMin from Open set and explore adjacent



Extract Manchester
Move to closed set
Explore adjacent

Open set (Priority Queue)

Boston = 65 + 55 + 0 = 120

Montpelier = 25 + 60 + 130 = 215

Closed set

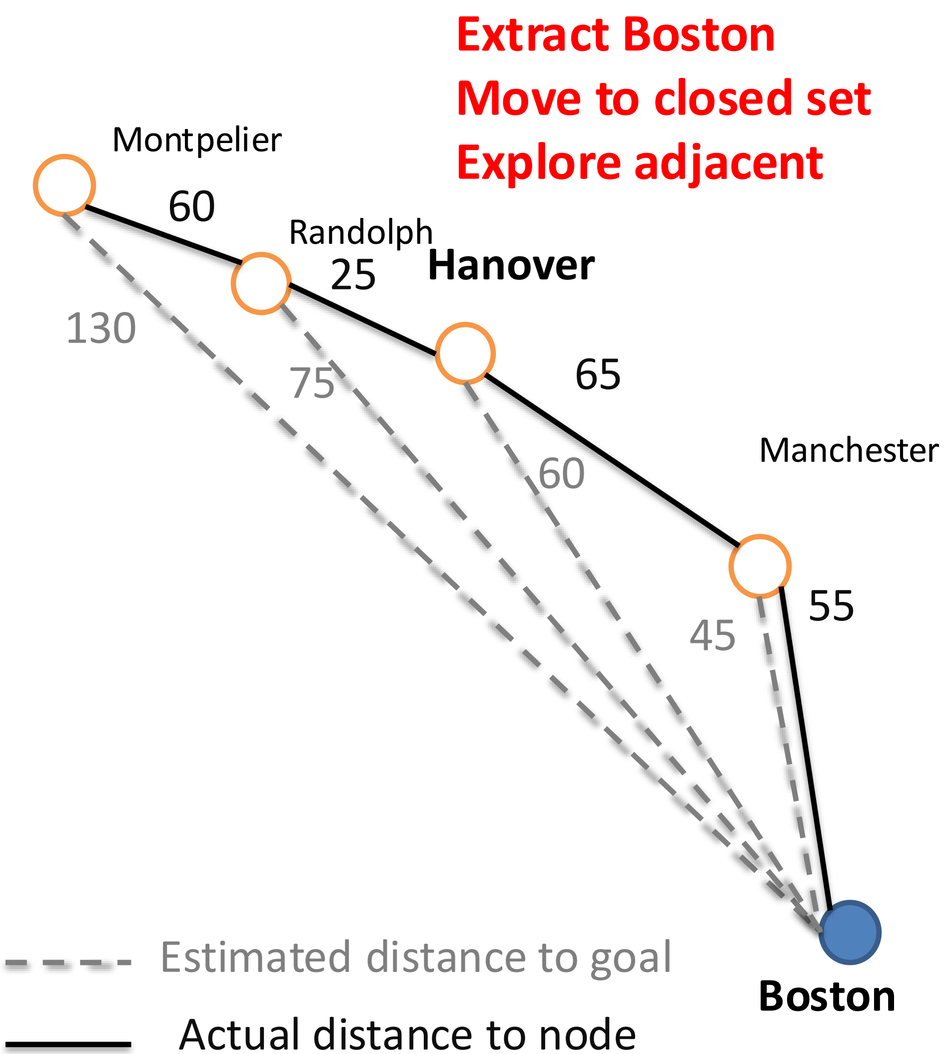
Hanover 0 + 60 = 60

Randolph 25 + 75 = 100

Manchester = 65 + 45 = 110

A* can help find the best path between two nodes faster than Dijkstra

Step 5: extractMin from Open set and explore adjacent



Open set (Priority Queue)

Montpelier = 25 + 60 + 130 = 215

Closed set

Hanover 0 + 60 = 60

Randolph 25 + 75 = 100

Manchester = 65 + 45 = 110

Boston = 65 + 55 + 0 = 120

Found goal at distance of 120

- Still check nodes in open set with estimate less than this route (120)
- No need to check other routes
- Montpelier can't be closer, a straight line would be greater than best path so far

Agenda

1. DFS and BFS on complex graph
2. Shortest-path simulation
3. Dijkstra's algorithm
4. A* search
5. Implicit graphs

Demo: Model maze intersections as vertices and run DFS/BFS/A*

MazeSolver.java

- Run
- Load map 5
- Try with:
  - Stack == DFS
  - Queue = BFS
  - A*