CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

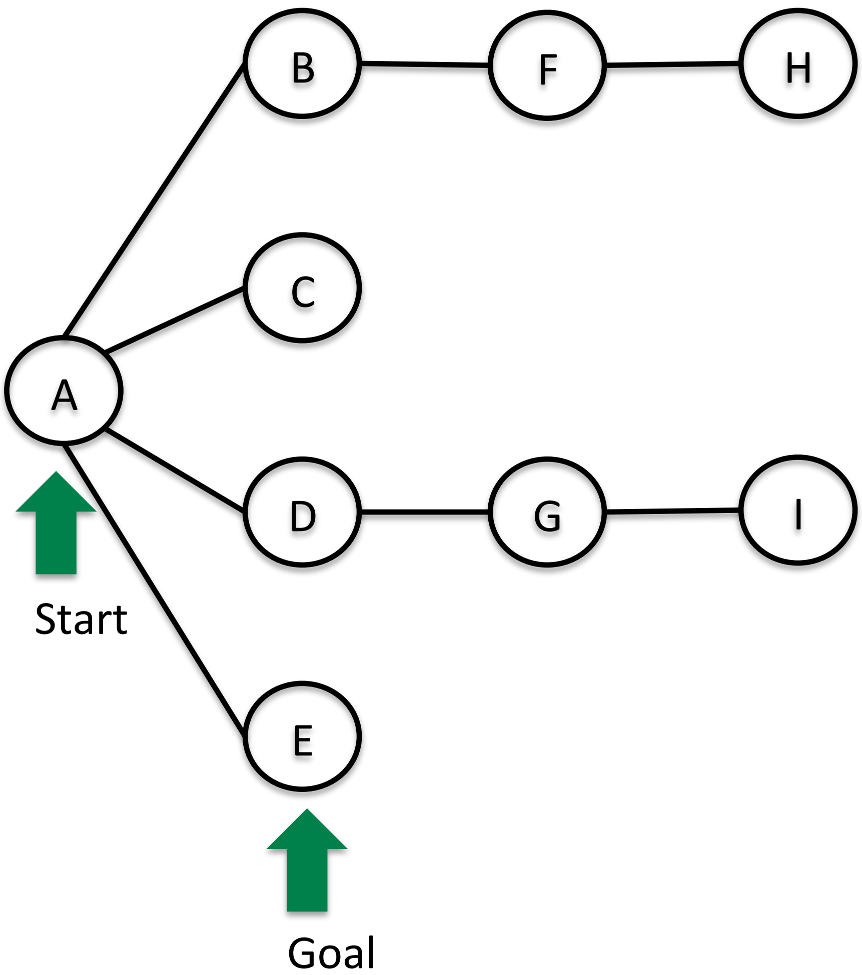
Day 17 – Shortest Path

Agenda

1. Shortest-path simulation
2. Dijkstra's algorithm
3. A* search
4. Implicit graphs

Previously we looked at finding the minimum number of steps between nodes

Breadth First Search



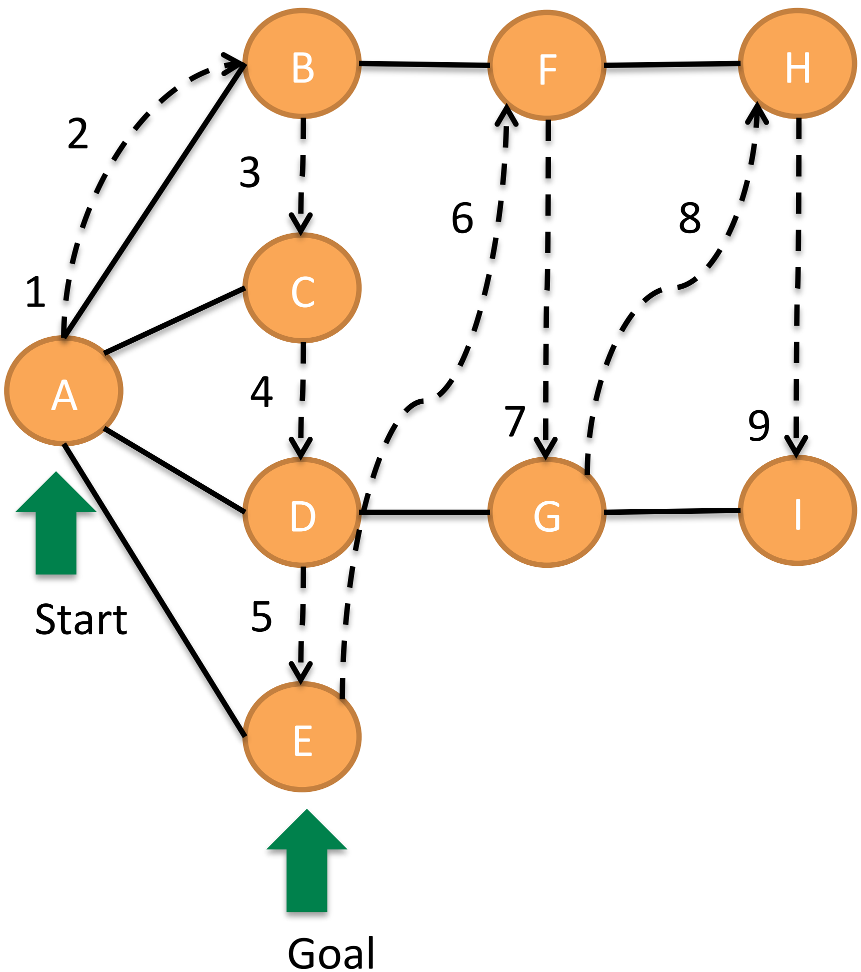
BFS is a good choice

Can find shortest number of steps between source and any other node

Could use BFS on a map to plan driving routes between cities

Previously we looked at finding the minimum number of steps between nodes

Breadth First Search



BFS is a good choice

Can find shortest number of steps between source and any other node

Could use BFS on a map to plan driving routes between cities

Search adjacent cities first

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

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

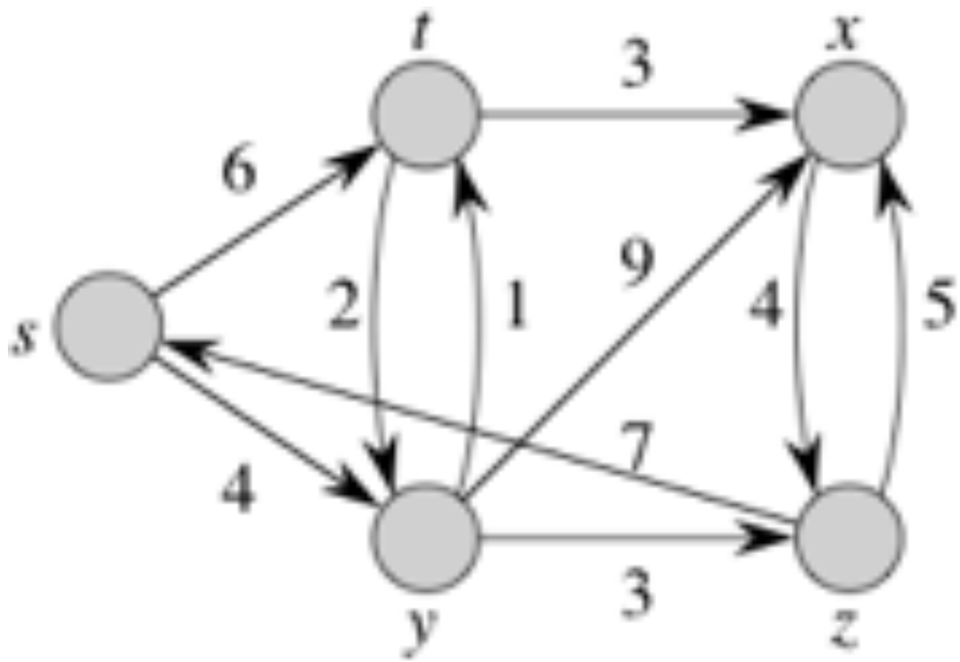
Note: our metric now is driving time, however total distance is longer!

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



Start at node `s` (single source)

Find path with smallest sum of weights to all other nodes

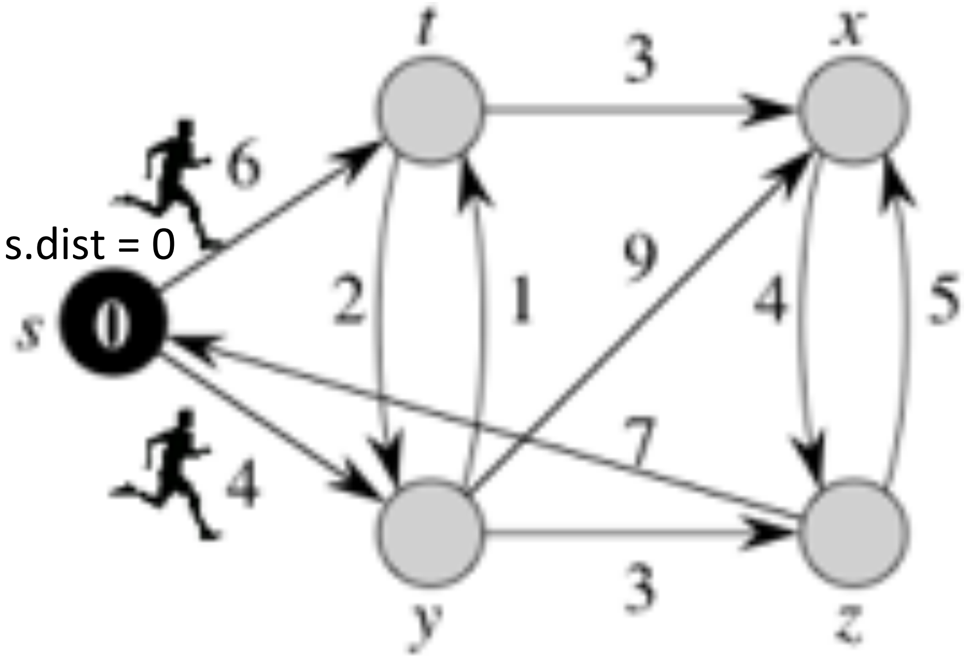
Store shortest path weights in `v.dist` instance variable

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



Simulation

`s.dist = 0`

Runners take edge weight minutes to arrive at adjacent nodes

When runners arrive at node:

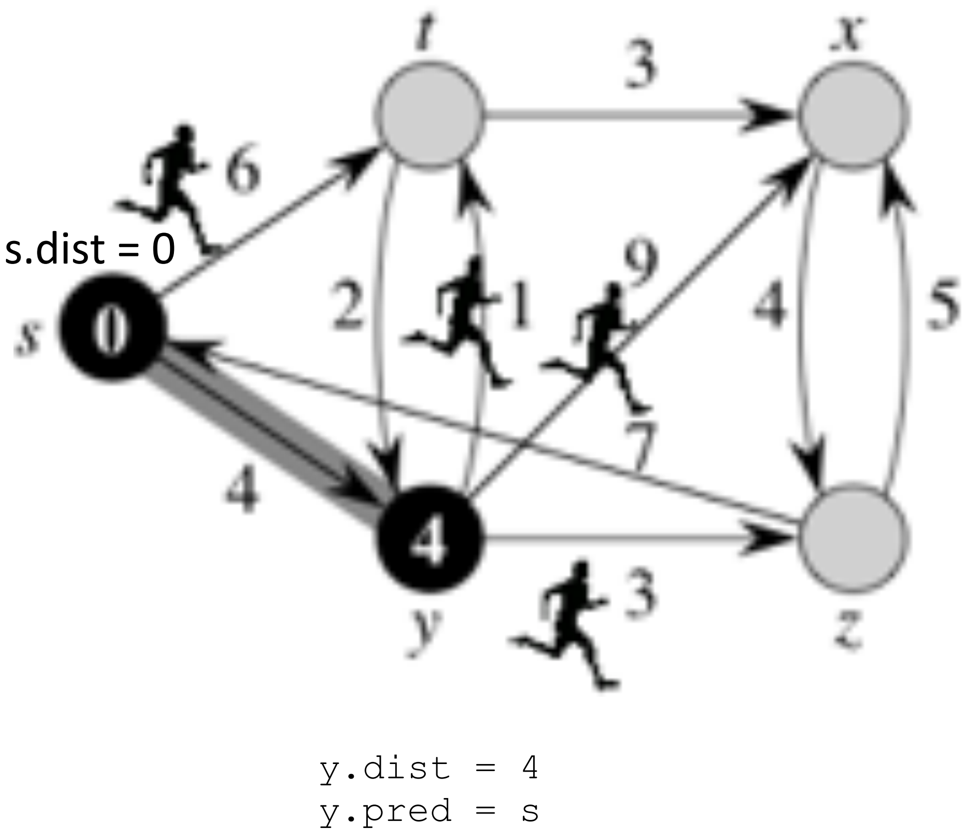
- Record arrival time in `v.dist`
- Record prior node in `v.pred`

Runners immediately leave for an adjacent node

Here runners leave for `y` and `t`

Imagine we send runners from the start to all adjacent nodes

Time 4



Runner arrives at `y` in 4 minutes

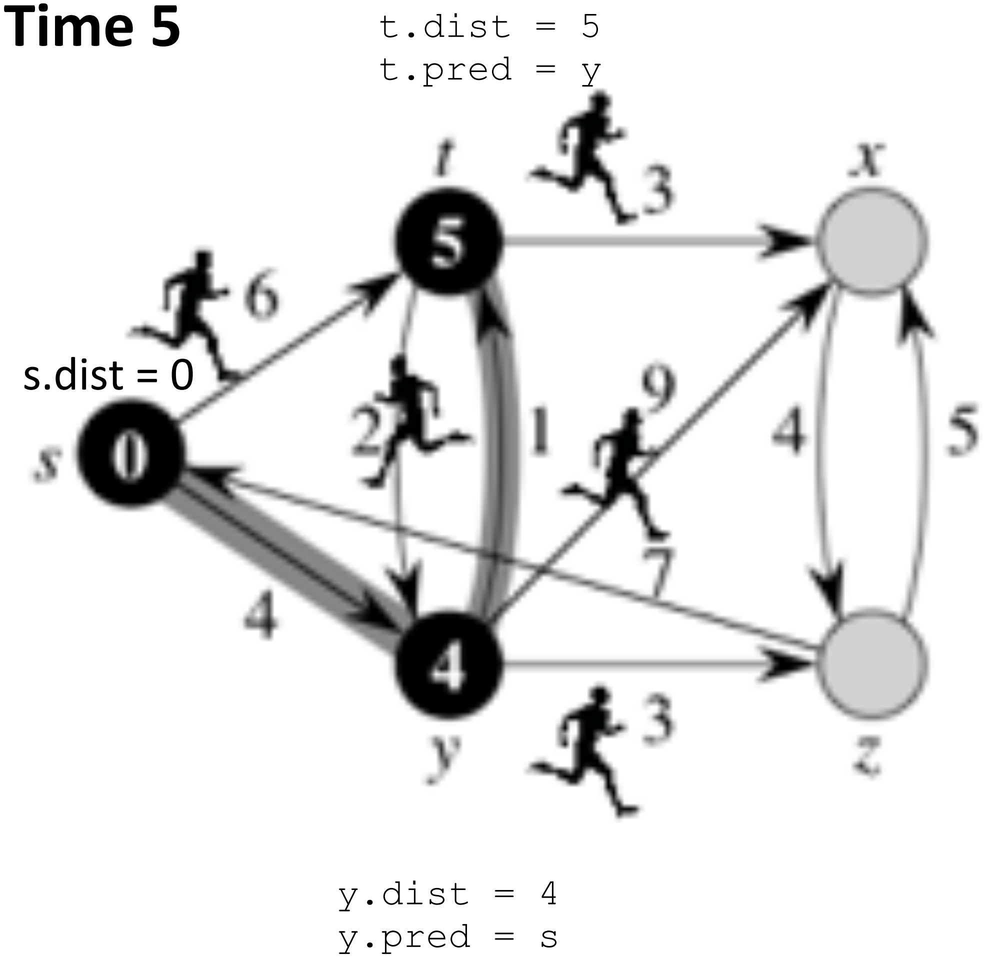
- Record `y.dist = 4`
- Record `y.pred = s`

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



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

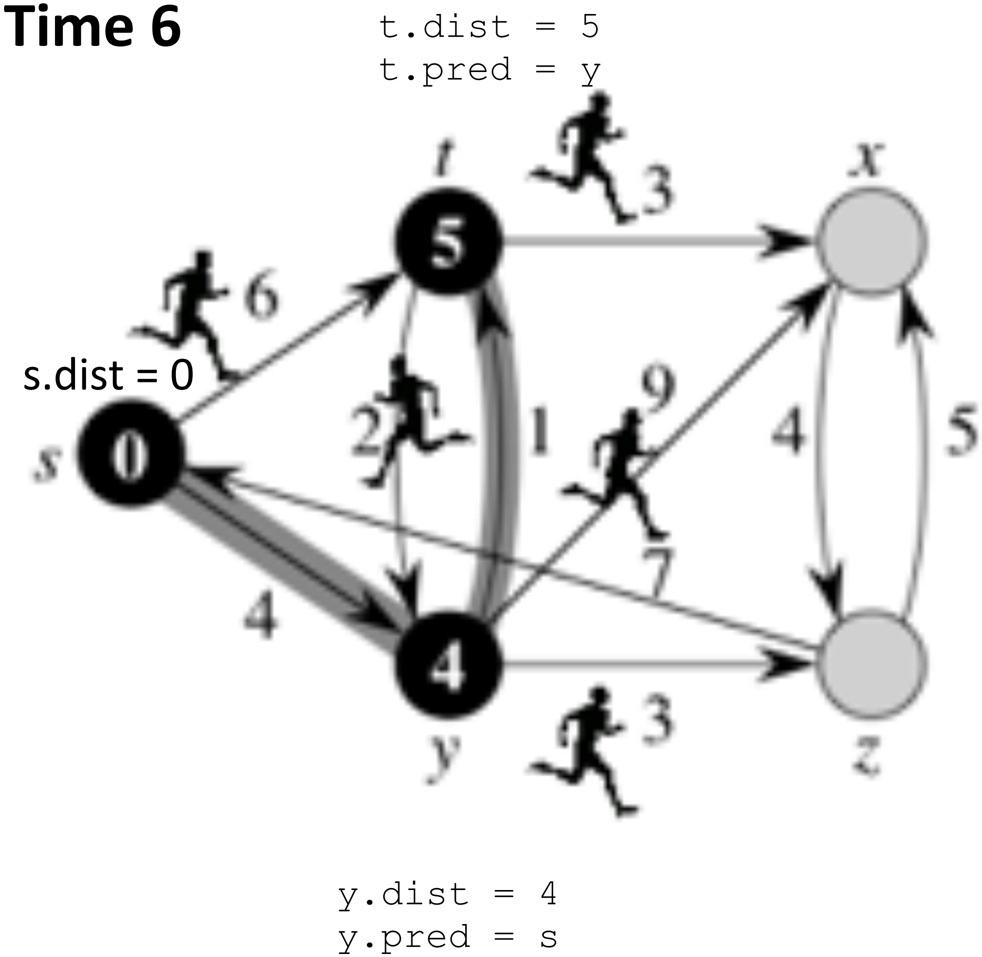
- `t.dist = 5`
- `t.pred = y`

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



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`

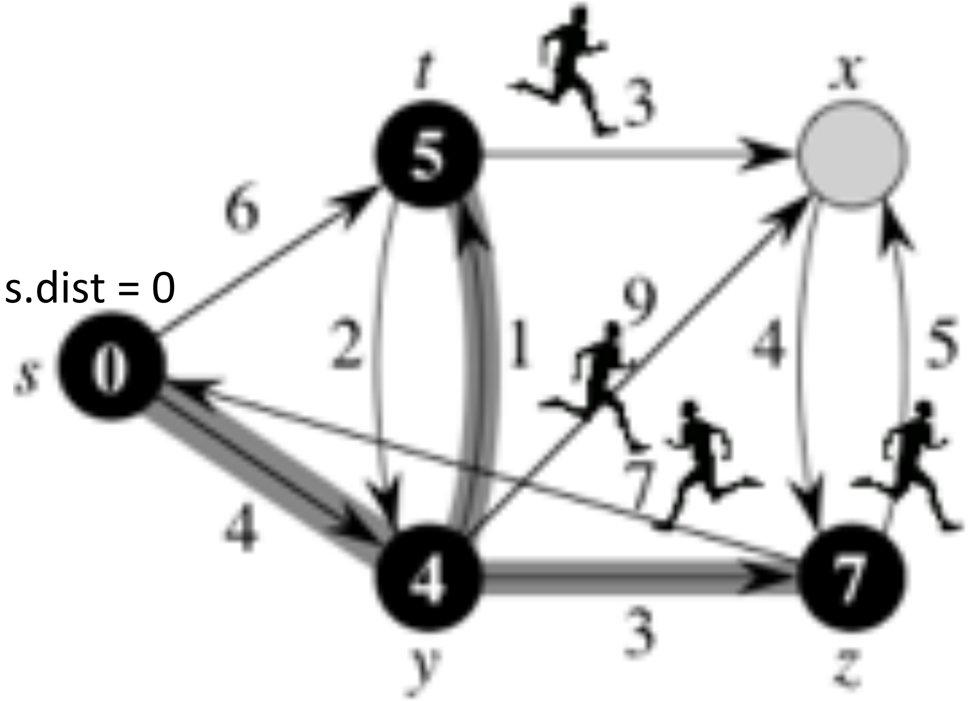
Do <u>not</u> update `t.dist` and `t.pred`

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`



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

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

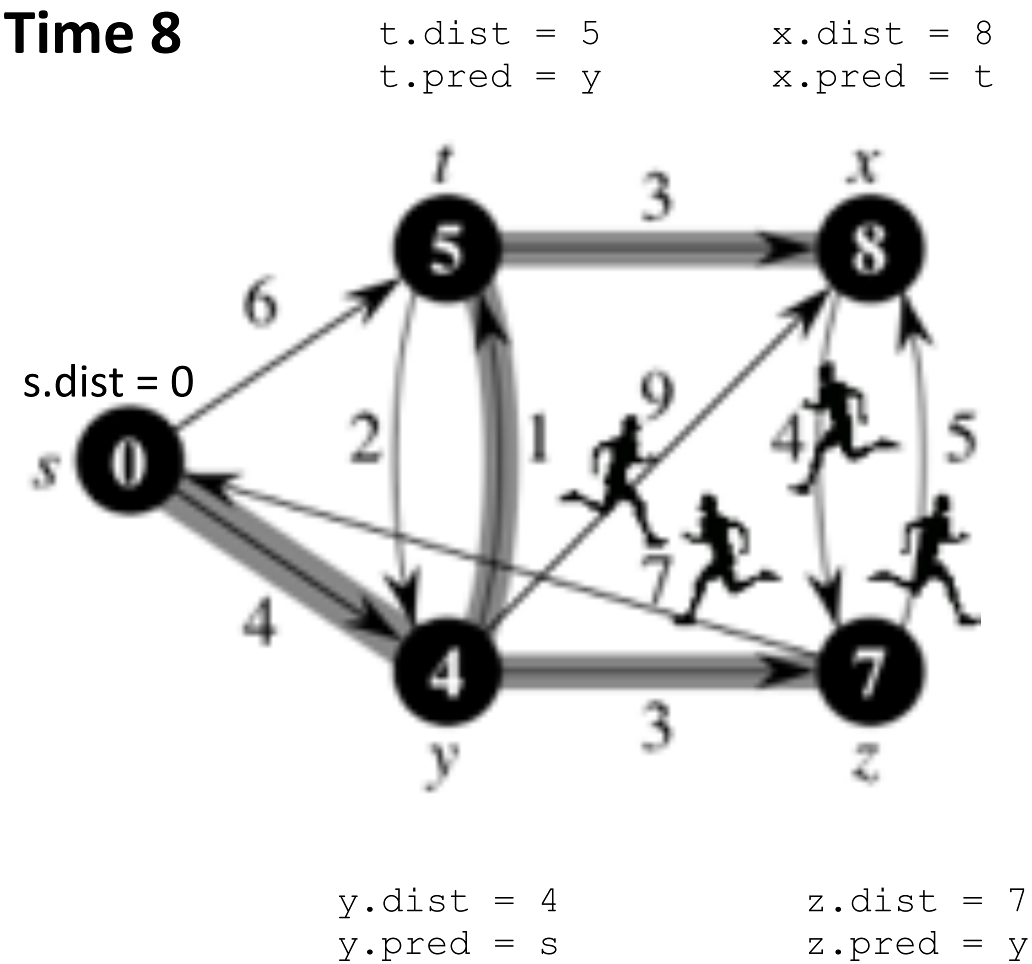
Runners leave `z` for `s` and `x`

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

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

Imagine we send runners from the start to all adjacent nodes

Time 8



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

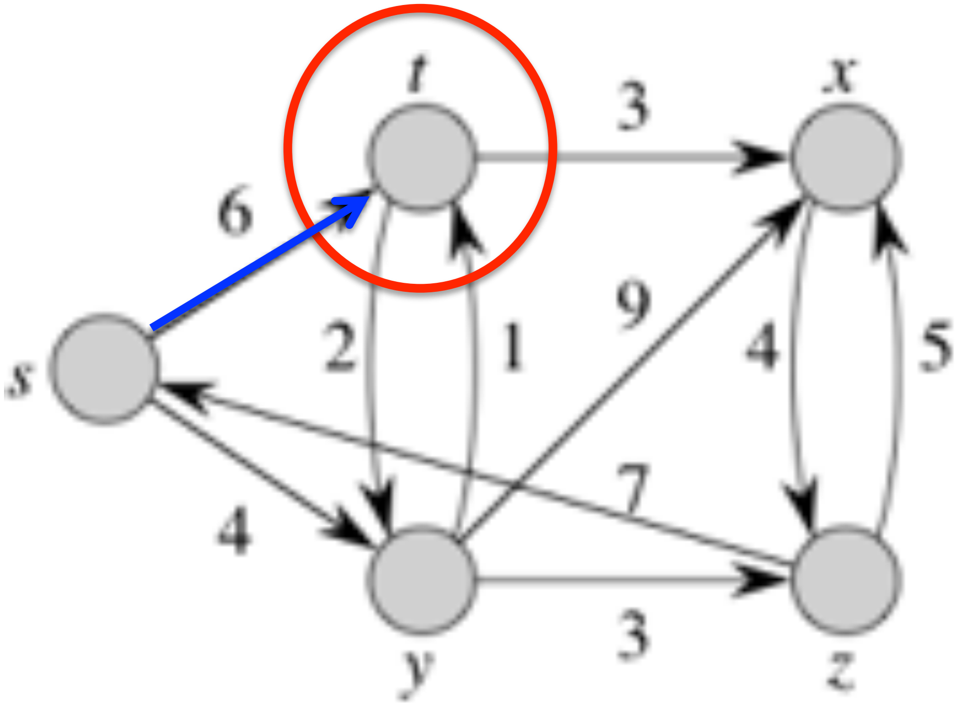
Path forms a tree on graph

Agenda

1. Shortest-path simulation
2. Dijkstra's algorithm
3. A* search
4. 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

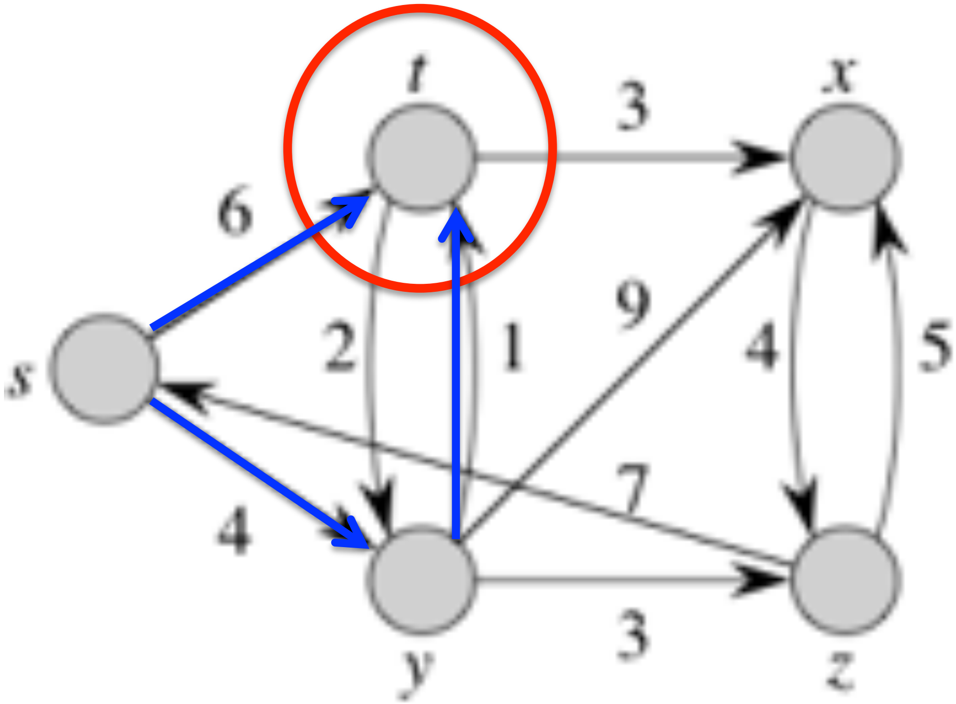
Compare distance to adjacent nodes with best so far

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

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

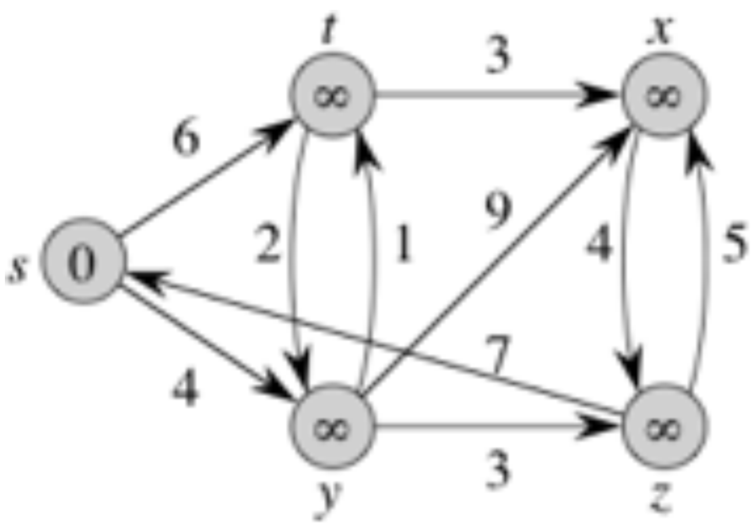
Distance from `u` to `v` is `w(u,v)`

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);
  }
}
```

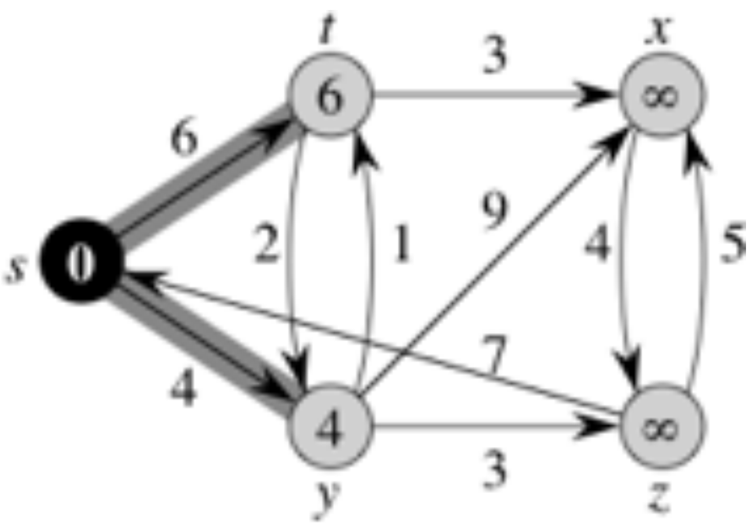
All nodes have distance `Infinity`, except Start with distance 0

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);
  }
}
```

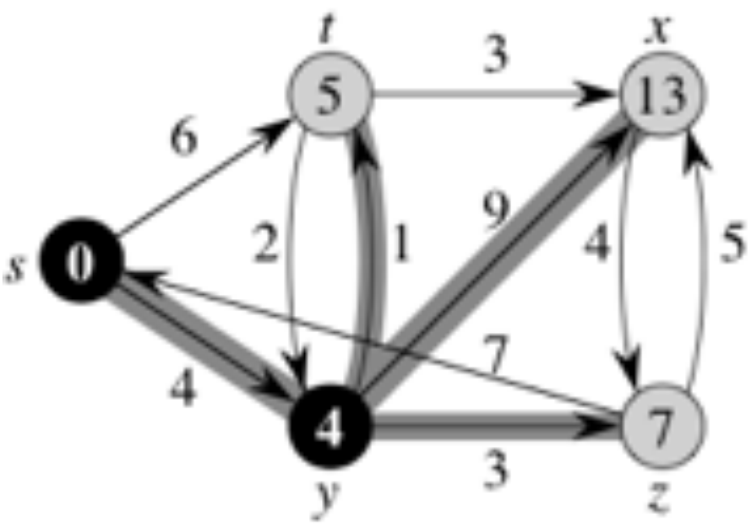
Loop over all adjacent nodes `v`

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



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

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

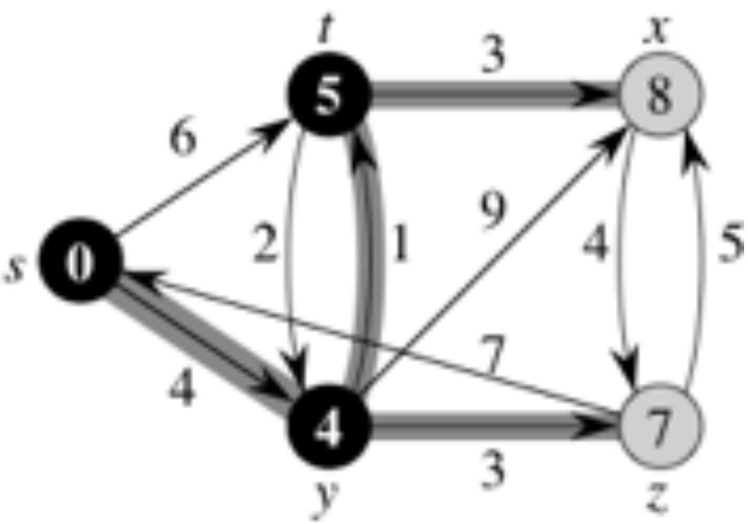
Relax each of them

```
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);
  }
}
```

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);
  }
}
```

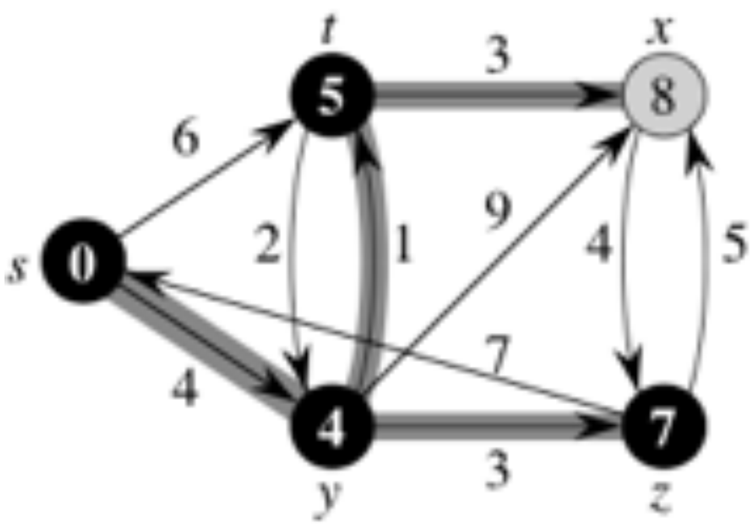
extractMin() now picks `t` (`dist =5`)

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);
  }
}
```

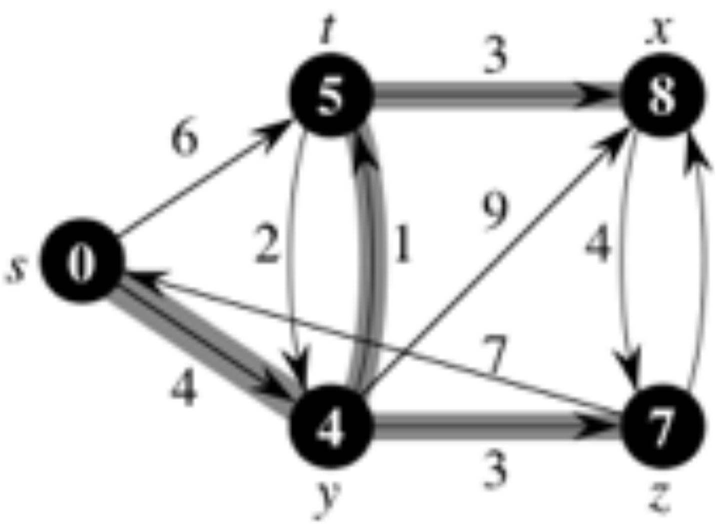
extractMin() now picks `z` (`dist = 7`)

Look at adjacent `x` and `s`

Do not relax `x` or `s`

Example

Dijkstra's algorithm 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);
  }
}
```

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

- 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(n log n + m log n)

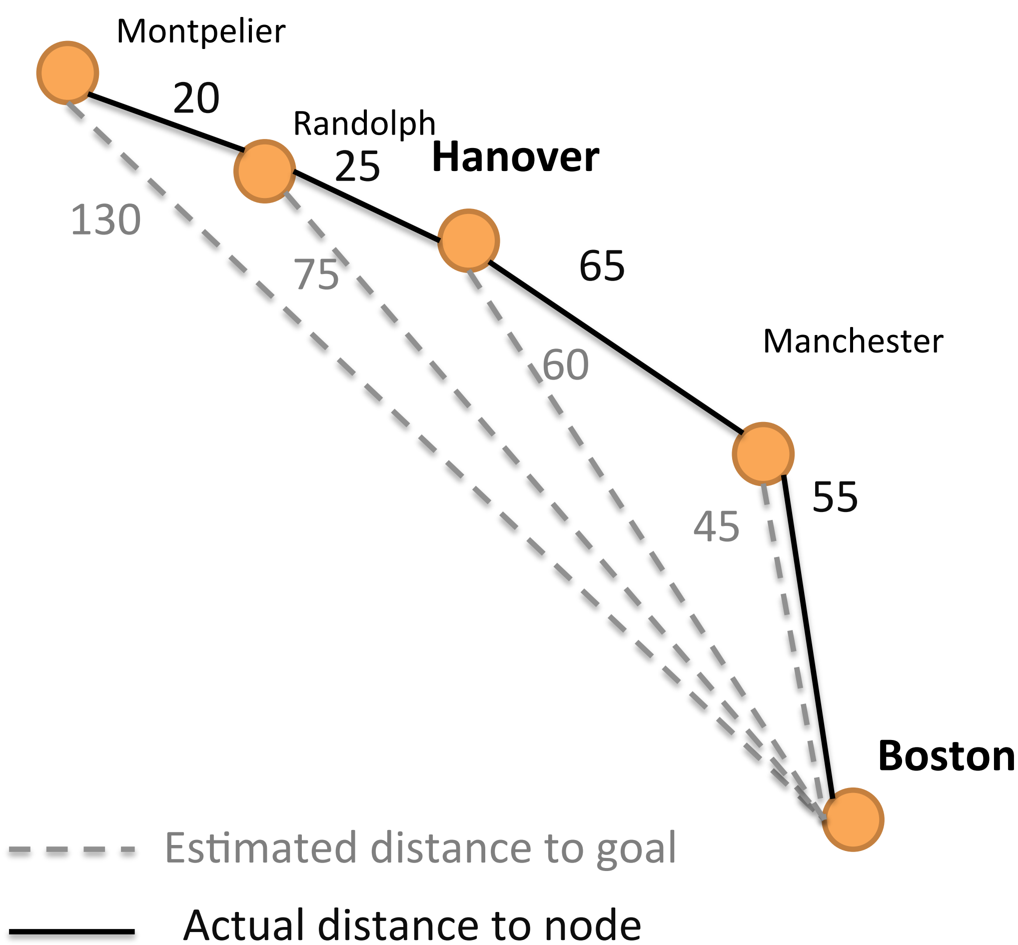
- Can implement with a Fibonacci heap with O(n^2)
- Take CS31 to find out how!

Agenda

1. Shortest-path simulation
2. Dijkstra’s algorithm
3. A* search
4. Implicit graphs

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

A* algorithm from Hanover to Boston



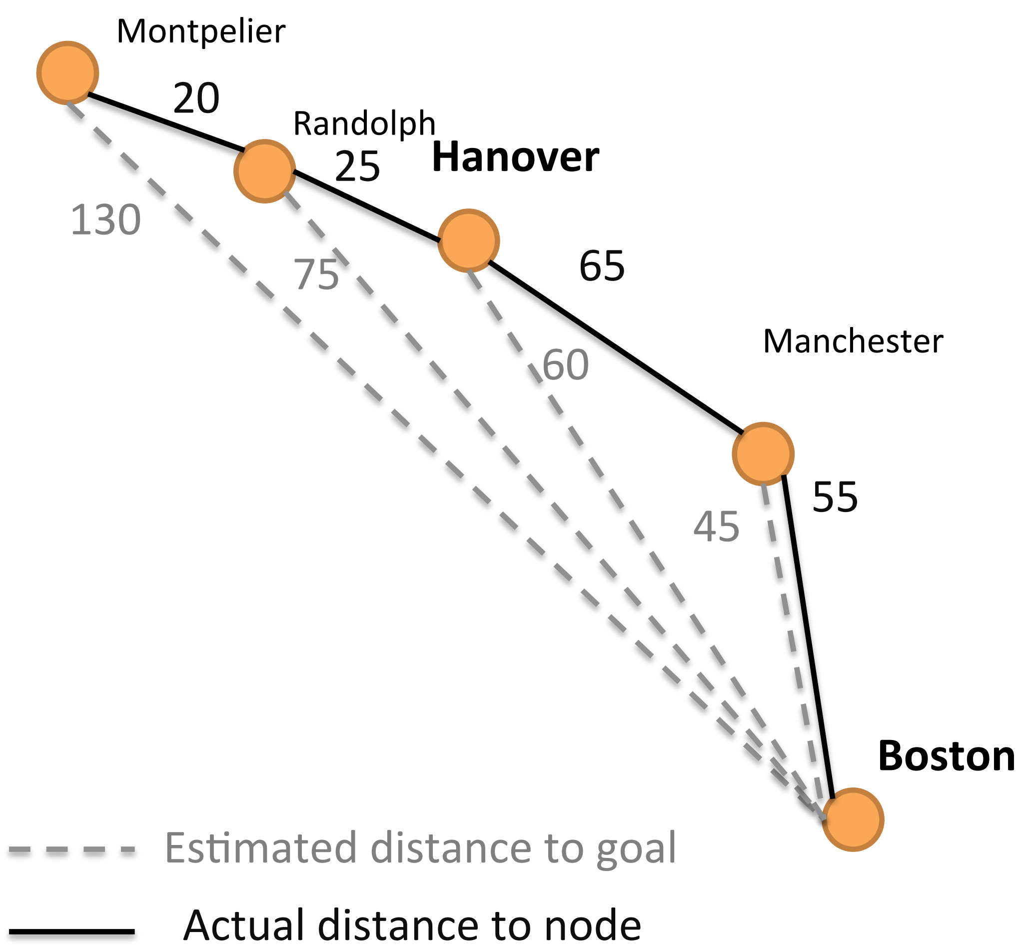
Estimate distance to goal (maybe use Euclidean distance)

Estimate must be ≤ actual distance (admissible)

Distances non-negative (distance monotone increasing)

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

A* algorithm from Hanover to Boston

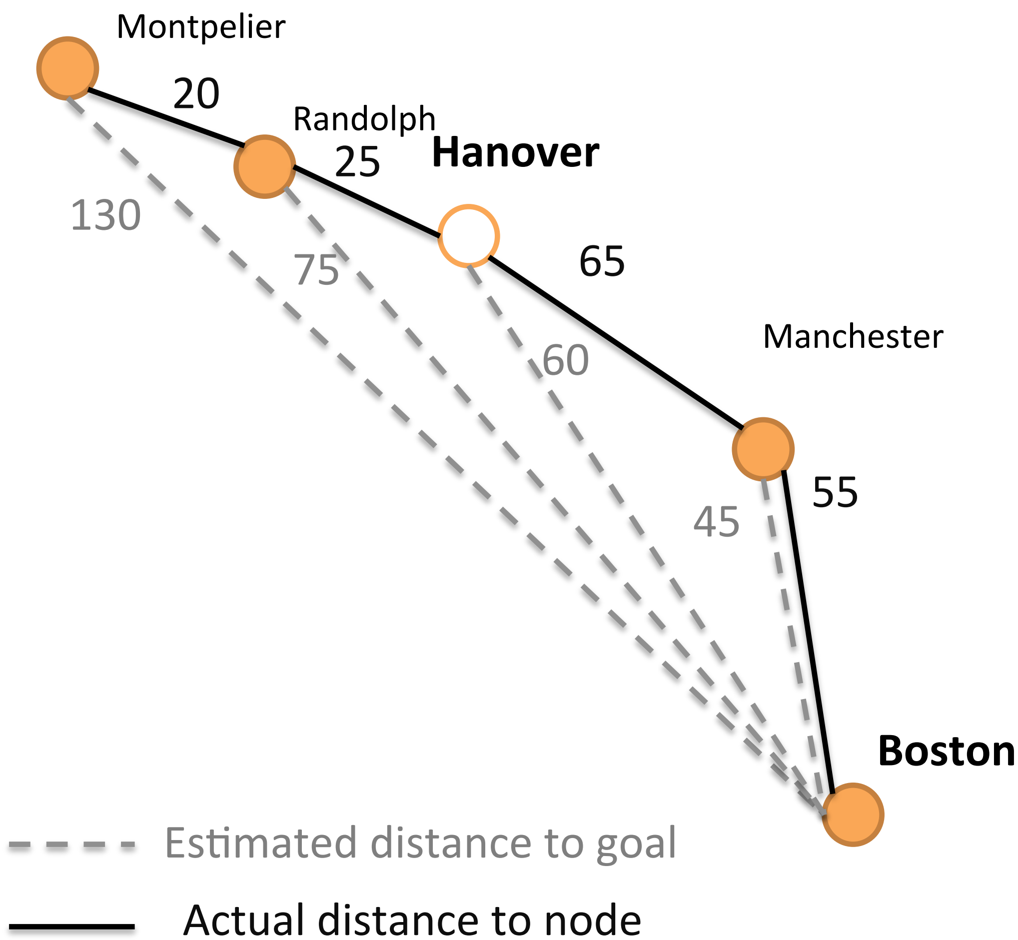


Keep Priority Queue using distance so far + estimate for each node (“open set”)

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



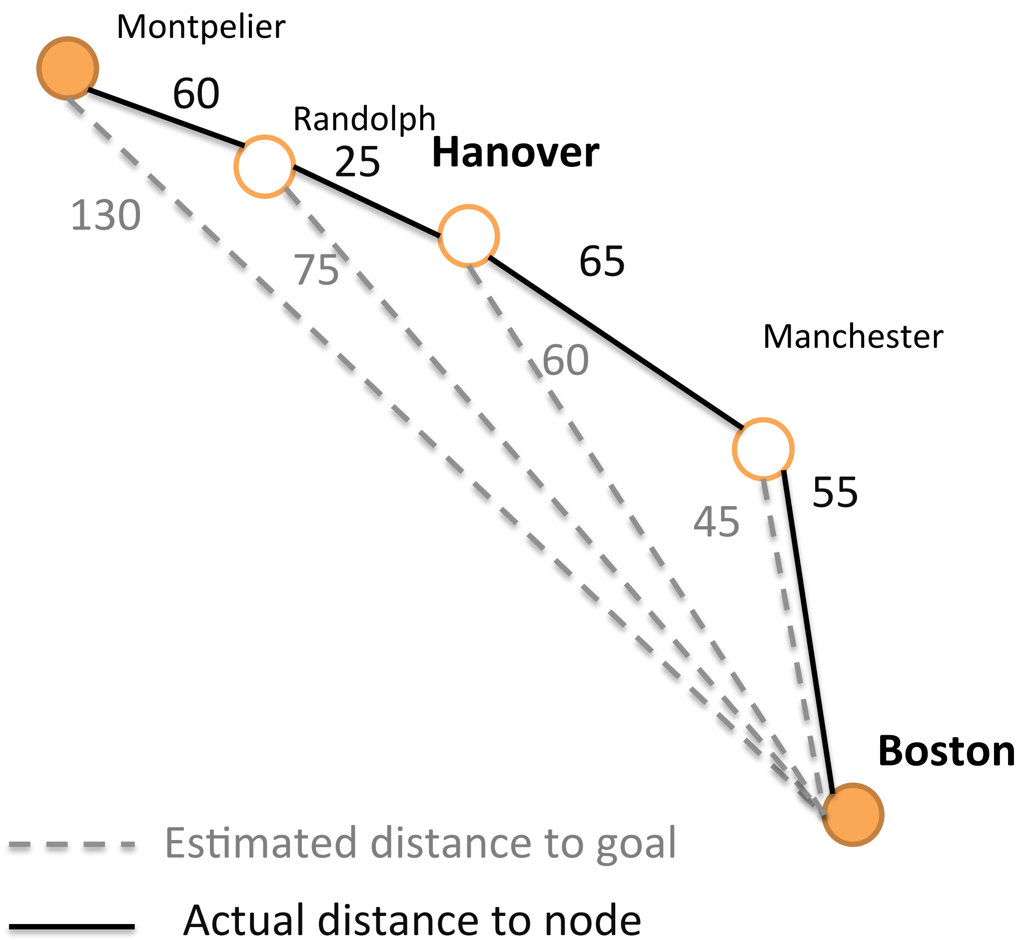
Open set (Priority Queue)

Hanover 0 + 60 = 60

Closed set

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

Step 2: select min from Open set and explore adjacent



Open set (Priority Queue)

Randolph 25 + 75 = 100

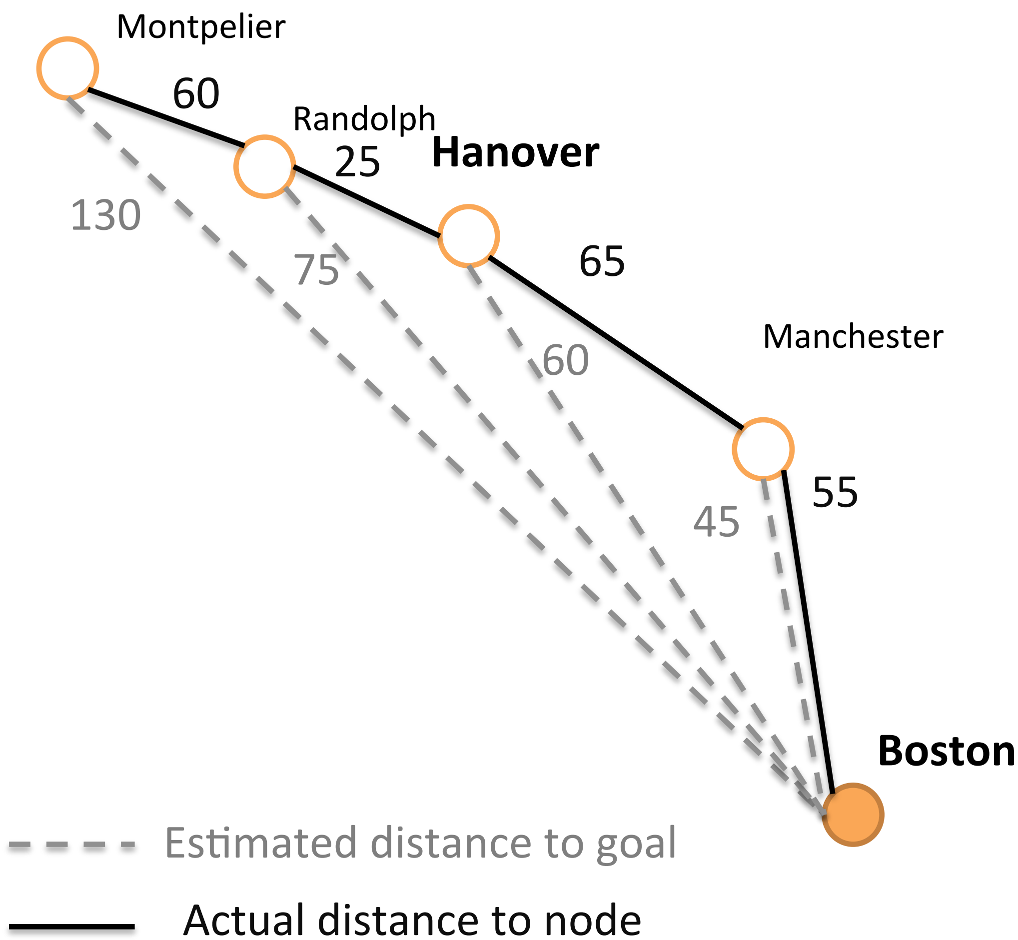
Manchester = 65 + 45 = 110

Closed set

Hanover 0 + 60 = 60

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

Step 3: select min from Open set and explore adjacent



Open set (Priority Queue)

Manchester = 65 + 45 = 110

Montpelier = 25 + 60 + 130 = 215

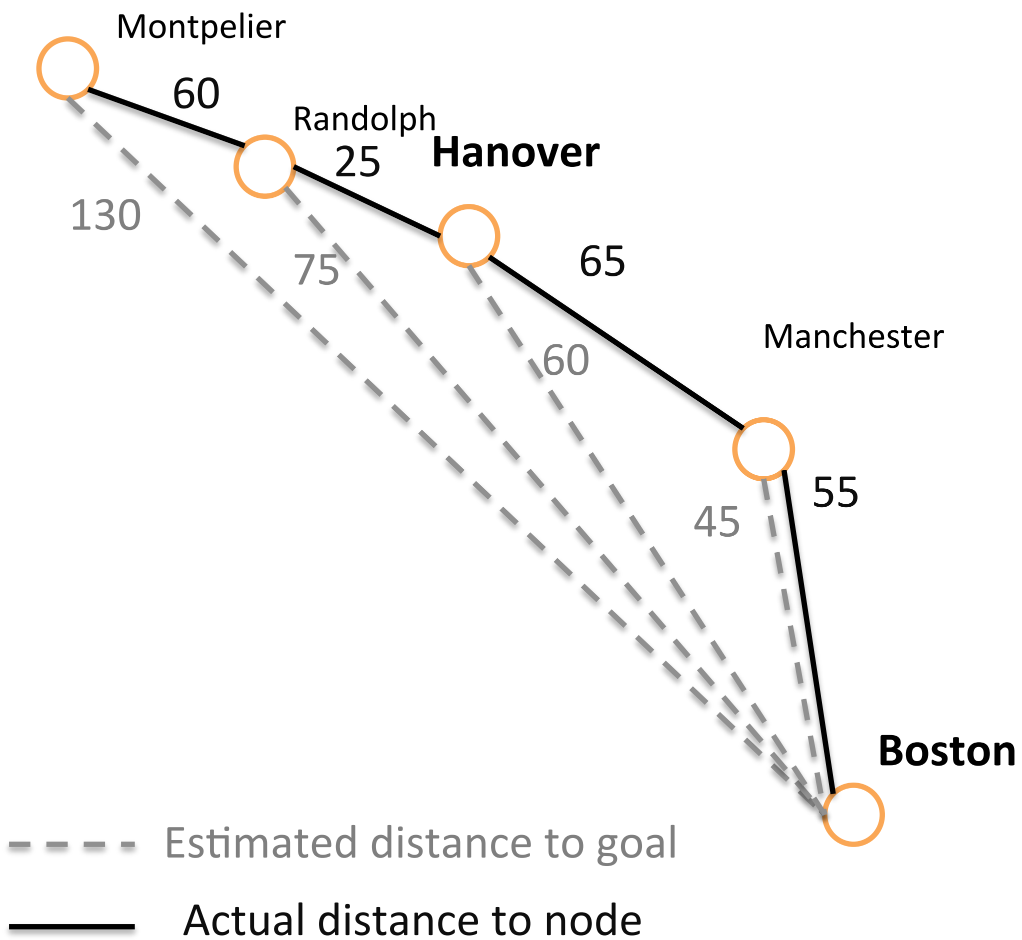
Closed set

Hanover 0 + 60 = 60

Randolph 25 + 75 = 100

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

Step 4: select min from Open set and explore adjacent



Open set (Priority Queue)

Boston = 65 + 55 = 120

Montpelier = 25 + 60 + 130 = 215

Closed set

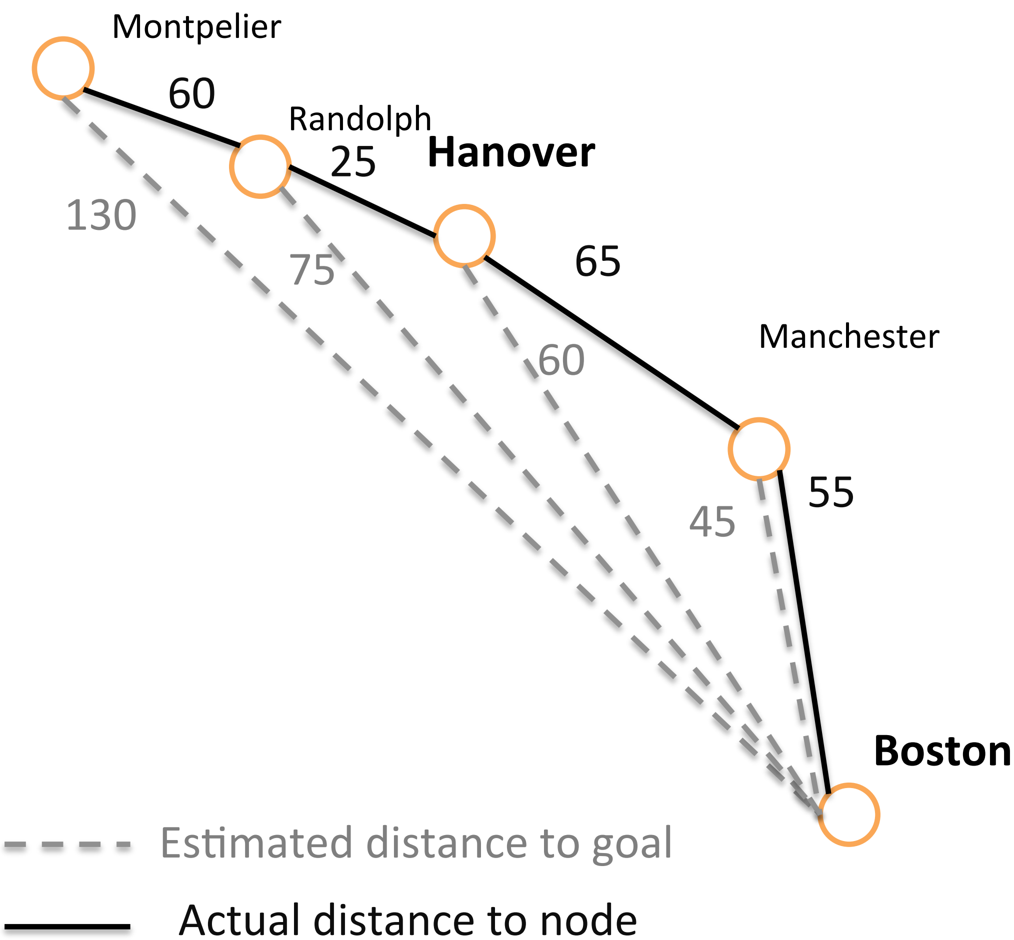
Hanover 0 + 60 = 60

Randolph 25 + 75 = 100

Manchester = 65 + 45 = 110

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

Step 5: select min 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 = 120

Found goal!

No need to check Montpelier – it can't be closer because a straight line would still be greater than best path so far

Agenda

1. Shortest-path simulation
2. Dijkstra's algorithm
3. A* search
4. Implicit graphs

Demo

MazeSolver.java

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