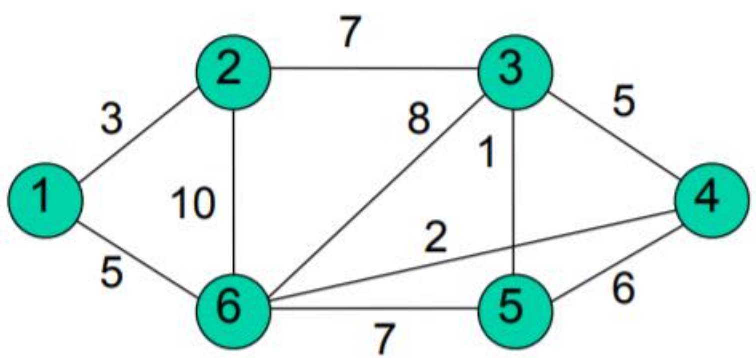
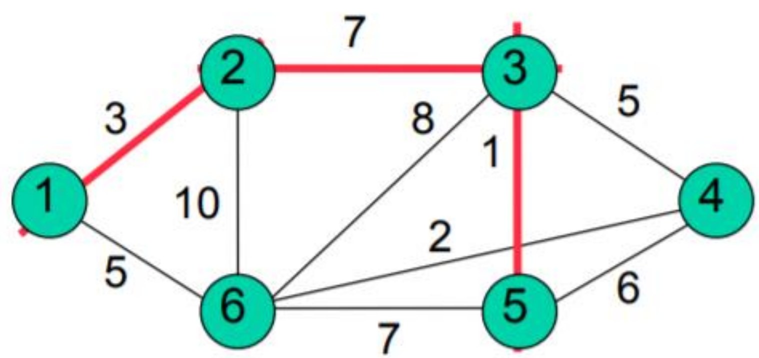
Graphs 2

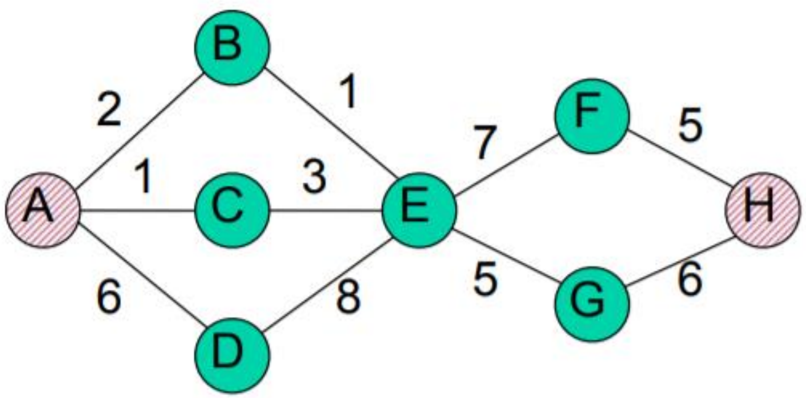
Shortest Paths

- Given a weighted connected graph G=(V, E), and a pair of vertices $v_s, v_d \in V$, what is the shortest path between v_s and v_d?
  - Path with the smallest sum of edge weights





Approach



A B E F H	15
A B E G H	14
A C E F H	16
A C E G H	15
A D E F H	26
A D E G H	25

Shortest Path (SP) from A to H: SP from A to E + SP from E to H

SP from A to H = $MIN_i(SP\ from\ A\ to\ v_i + SP\ from\ v_i\ to\ H)$

Dijkstra's Algorithm

```
Dijkstra(G = (V, E), source)
  S = {soruce}     # S is the set of explored nodes
  d (source) = 0   # d(v) is the shortest path from
                   # source to v
 while S != V
     Choose v ∈ V\S s.t. d(u) + |(u, v)| is minimized (u ∈ S)
     Add v to S, set d(v) = d(u) + |(u, v)|
```

Why Dijkstra's algorithm works

Suppose d (w) are the actual shortest path lengths for vertices w in S
Want to show that d (v) is the shortest path length for v

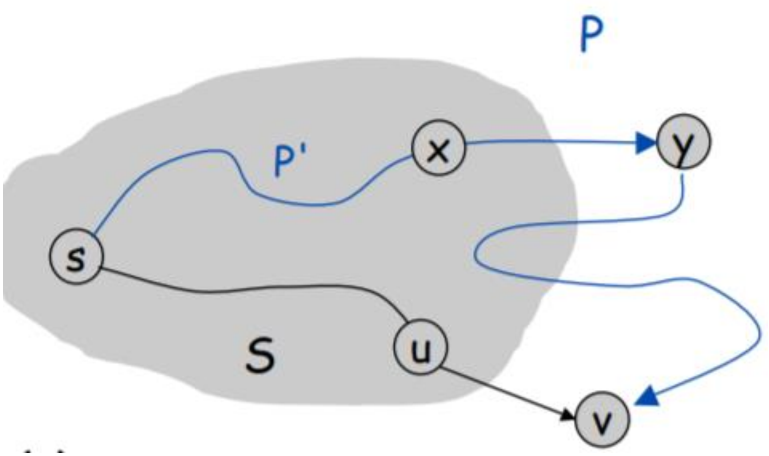
Consider the path s->x->y->v with y outside of S

s->x->y is already no shorter than s->u->v, or we would choose y instead of v

So a shortest path to v must be entirely within S

We chose the shortest path in S, so dist(v) is the length of the shortest path

```
Dijkstra(G = (V, E), source)
  S = {s}  # S is the set of explored nodes
  d (source) = 0   # d(v) is the shortest path from
                   # source to v using nodes in S
  while S != V
     Choose v ∈ V\S s.t. d(u) + |(u, v)| is minimized (u ∈ S)
     Add v to S, set d(v) = d(u) + |(u, v)|
```



Dijkstra’s Algorithm: complexity

- Depends on the implementation details
- Simplest implementation
  - To add one vertex to S, search through all possible additional vertices
  - $O(|V|^2)$
- Fancier implementation
  - Add potential v’s to a priority queue as S grows
  - $O\big((|E|)\log|V|\big)$
  - (skip this analysis)

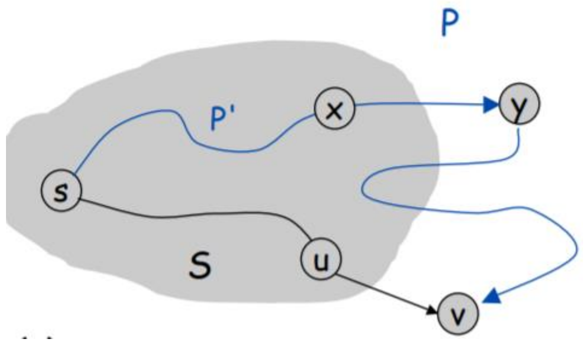
```
Dijkstra(G = (V, E), source)
  S = {source}      # S is the set of explored nodes
  d(source) = 0     # d(v) is the shortest path from
                    # source to v using nodes in S
  while S != V
     Choose v ∈ V\S s.t. d(u) + |(u, v)| is minimized (u ∈ S)
     Add v to S, set d(v) = d(u) + |(u, v)|
```

Dijkstra's Algorithm: Recovering the path

```
Dijkstra(G = (V, E), source)
  S = {source}       # S is the set of explored nodes
  d (source) = 0     # d(v) is the shortest path from
                     # source to v
 while S != V
     Choose v ∈ V\S s.t. d(u) + |(u, v)| is minimized (u ∈ S)
     Add v to S, set d(v) = d(u) + |(u, v)|
     prev(v) = u
```

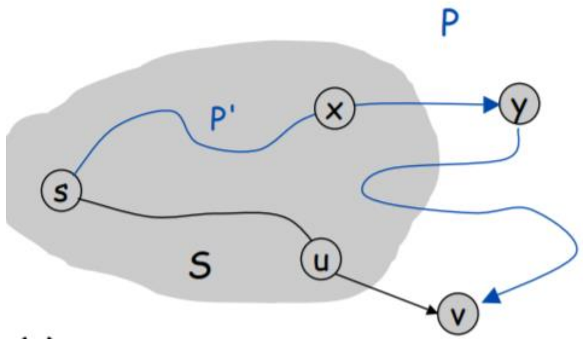
Dijkstra's Algorithm: Efficient Implementation

```
Dijkstra(G = (V, E), source)
  S = {source}     # S is the set of explored nodes
  d(source) = 0    # d(v) is the shortest path from
                   # source to v using nodes in S
  while S != V
     Choose v ∈ V\S s.t. d(u) + |(u, v)| is minimized (u ∈ S)
     Add v to S, set d(v) = d(u) + |(u, v)|
```



- Idea
  - Maintain the distances from S to neighbours of S
  - When we add a vertex v to S, only need to compute the distances of neighbours of v to S
  - Maintain a priority queue with the closest neighbour of S at the top

```
Dijkstra(G = (V, E), source)
  S = {source}     # S is the set of explored nodes
  d(source) = 0    # d(v) is the shortest path from
                   # source to v using nodes in S
  while S != V
     Choose v ∈ V\S s.t. d(u) + |(u, v)| is minimized (u ∈ S)
     Add v to S, set d(v) = d(u) + |(u, v)|
```




```
Dijkstra(G = (V, E), source)
  S = {}     # S is the set of explored nodes
  pq = (0, source)
  while pq is not empty
     if cur_node in S
        continue
     cud_dist, cur_node = pq.pop()
     d(cur_node) = cur_dist
     add cur_node to S
     for each neighbour v of cur_node
        pq.push((cud_dist + |(cur_node, v|), v)
```

Complexity

```
Dijkstra(G = (V, E), source)
  S = {}     # S is the set of explored nodes
  pq = (0, source)
  while pq is not empty
     if cur_node in S
        continue
     cur_dist, cur_node = pq.pop()
     d(cur_node) = cur_dist
     add cur_node to S
     for each neighbour v of cur_node
        pq.push((cud_dist + |(cur_node, v|), v)
```

- Pop and push into pq: O(log(|V|)
- Upper bound on the number of times a node is pushed: 2|E|
- Total O(|E|log(|V|))

Greedy Best-first search

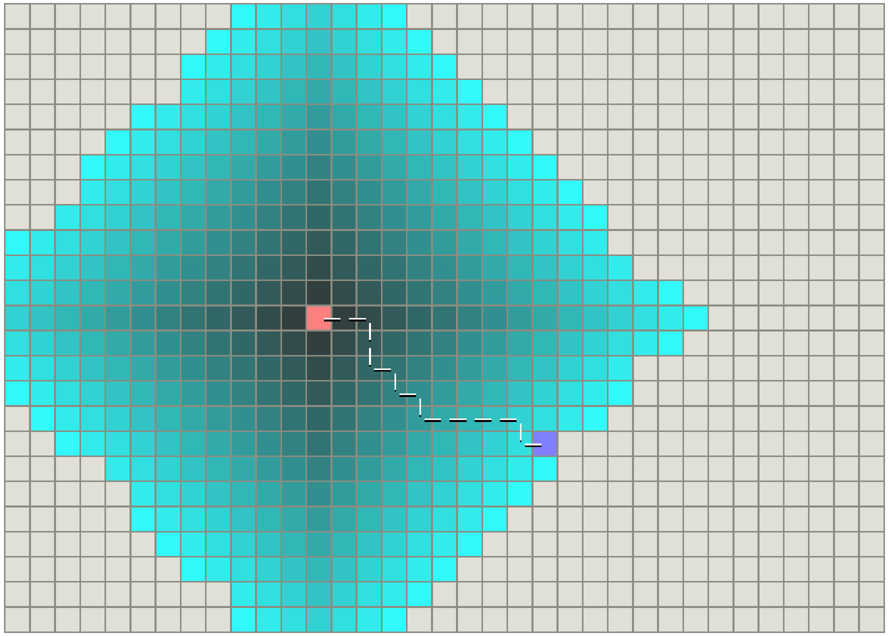
- Goal: find the shortest path from a source node to a destination node
- Could run Dijkstra's algorithm, and stop once we add the destination node to S
  - Could be wasteful
- We sometimes have some estimate of how far a node is from the destination
  - Want to use that

Greedy Best-first search

- h(node): an estimate for how far the node is from the destination
  - A “heuristic function”

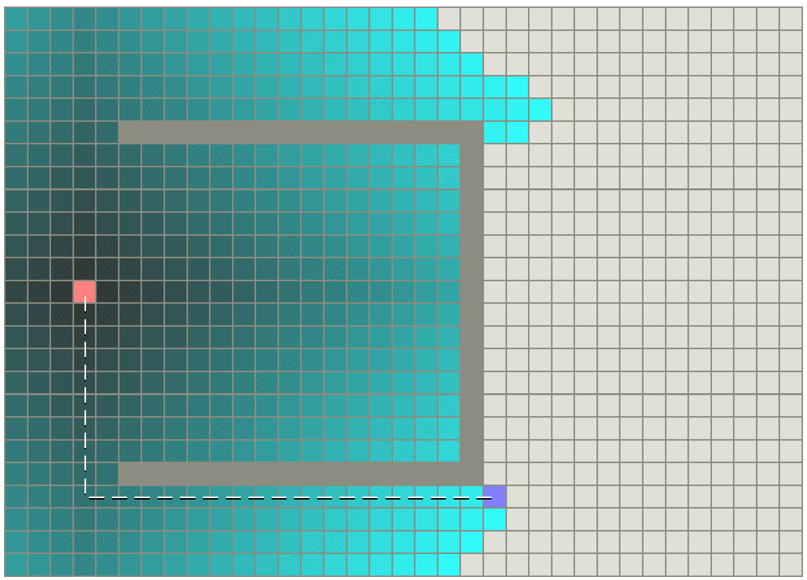
```
Greedy-Best-First(G = (V, E), source, dest)
  S = {}     # S is the set of explored nodes
  v = source
  while v is not dest
     select v from the neighbourhood of S with the smallest h(v)
     add v to S
```

- Not guaranteed to find the shortest path
- Will work well if h(node) is a good estimate

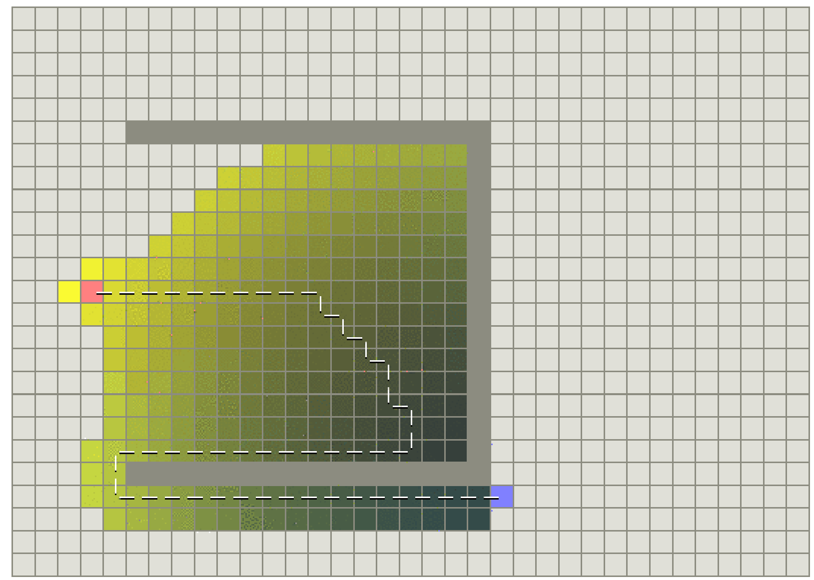


- All cells are connected vertically and horizontally
- Pink is source, purple is destination
- Shortest path shown
- h(node): Manhattan distance from node to dest
  - Distance between (x_1, y_1) and (x_2, y_2) is $|x_2 - x_1| + |y_2 - y_1|$
  - The distance you need to go on a grid if you're only allowed to go along the x and y axes
  - Like in a city with a grid layout

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



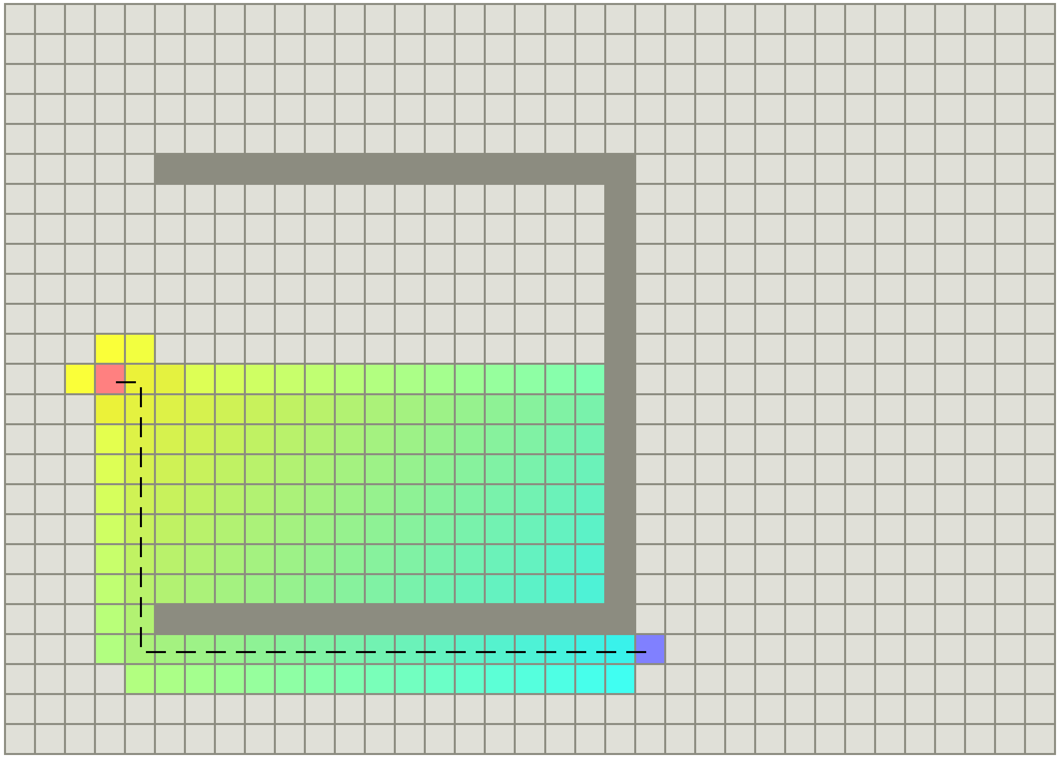
Shortest path



Greedy path
(N.B., should actually go to the extreme right first)

A* Algorithm

```
A*(G = (V, E), source, dest)
  S = {}     # S is the set of explored nodes
  pq = (h(source), 0, source)
  while pq is not empty
     if cur_node in S
        continue
     cur_node, cur_priority, cur_dist = pq.pop()
     d(cur_node) = cur_dist
     add cur_node to S
     for each neighbour v of cur_node
        dist = cur_dist + |(cur_node, v)|
        pq.push(h(v)+dist, dist, v)
```



A* and Dijkstra's Algorithm

- Dijkstra: priority is the current estimate for the shortest path length from source
- A*: priority is the current estimate for the shortest path length from source + an estimate for the path length to destination
- When h(n) is always 0, A* is just Dijkstra
- Theorem (stated without proof): if h(node) never overestimates the distance to destination (terminology: h is *admissible*), A* finds the shortest path
  - A* is not guaranteed to find the shortest path otherwise

A* properties

- h(node) is *admissible* if it never overestimates the distance from node to destination
- *h*