Mobile Robotics

Path Planning

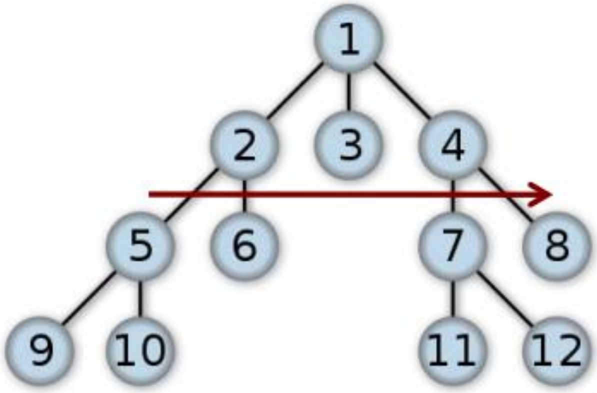
Programming Assignments and Projects

Metric Path Planning as Search

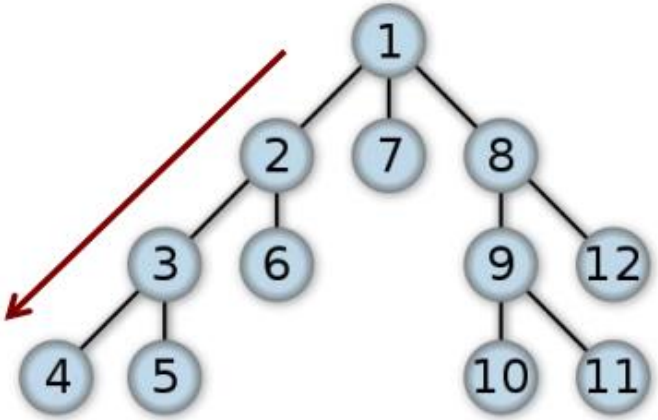
- In AI “search” means that the answer is in the search space, often just finding the path to the answer (goal)
- Types of AI search
  - Blind, brute-force, uninformed
    - Breadth-first (Wavefront)
    - Depth-first
  - Heuristic
    - Dijkstra
    - A*
- For Path planning
  - A* for relational graphs, regular girds
  - Breadth-first (Wavefront) for operating directly on regular grid

Uninformed Search

- Breadth-first (BF)
  - Complete
  - Optimal if action costs equal
  - Time and space: O(bd)



- Depth-first (DF)
  - Not complete in infinite spaces
  - Not optimal
  - Time: O(bm)
  - Space: O(bm) (can forget explored subtrees)



(b: branching factor, d: goal depth, m: max. tree depth)

A*: Minimize the Estimated Path Cost f(n)

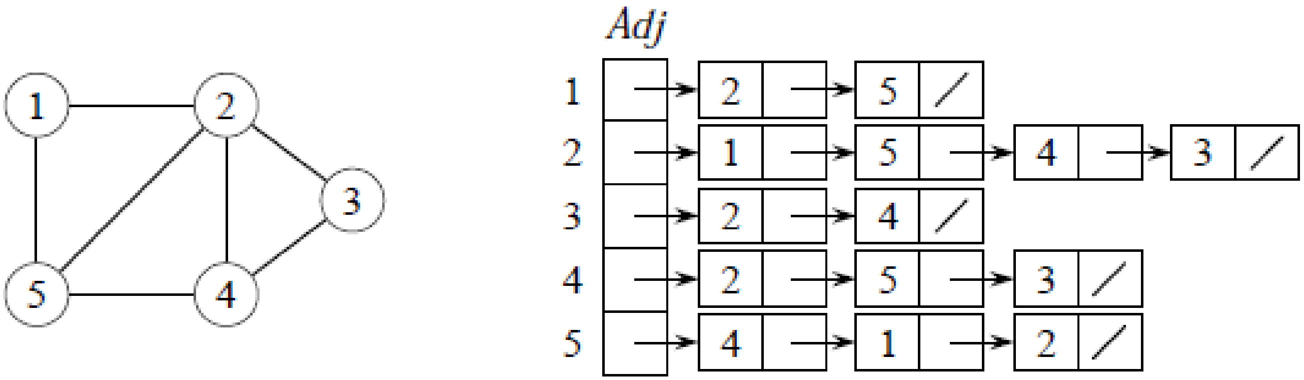
- g(n) = actual cost from the initial state to n.
- h(n) = estimated cost from n to the next goal.
- f(n) = g(n) + h(n), the estimated cost of the cheapest solution through n.
- Let h*(n) be the actual cost of the optimal path from n to the next goal.
- h is admissible if the following holds for all n:

$$h(n) \leq h^*(n)$$

- We require that for A* , h is admissible (the straight-line distance is admissible in the Euclidean Space)
- Note 1: when h(n) = 0 for all n, A* is Dijkstra's algorithm.
- Note 2: when all edges have the same cost, Dijkstra is BF search.

Graph representation

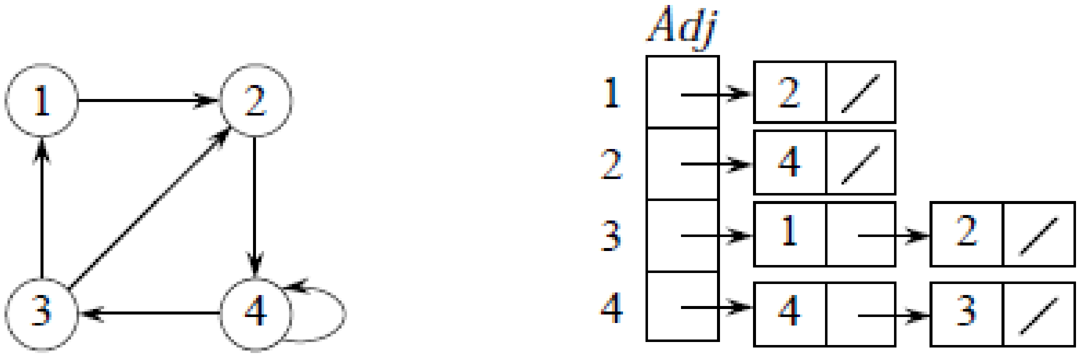
- **Adjacency lists:** Given a graph G=(V, E)
- Example: For an undirected graph:



- Space: $\Theta(V + E)$.
- Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.
- Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

Graph representation (Cont.)

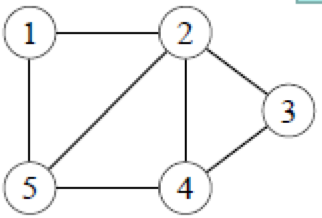
- **Adjacency lists:** Given a graph G=(V, E)
- Example: For a directed graph:



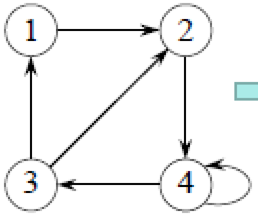
- Space: $\Theta(V + E)$.
- Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.
- Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

Graph representation (Cont.)

- **Adjacency matrix:** Given a graph G=(V, E)
- Examples:



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

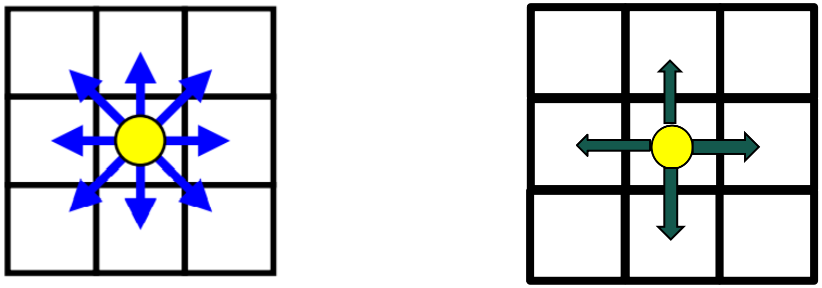


$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

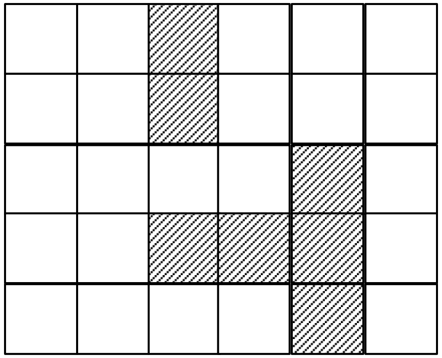
- Space: $\Theta(V^2)$.
- Time: to list all vertices adjacent to u: $\Theta(|V|)$.
- Time: to determine whether $(u, v) \in E$: $O(1)$.

Graph representation (Cont.)

- Occupancy Grid Map
  - Eight neighbors or four neighbors



  - We are going to use four neighbors in programming assignments PA4A and PA4B, and eight neighbors in programming project PP4
- Example and its representation
  - Cells with 1:
    - Occupied
  - Cells with 0:
    - Not Occupied



0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	0	1	0
0	0	1	1	1	0
0	0	0	0	1	0

Breadth-first search

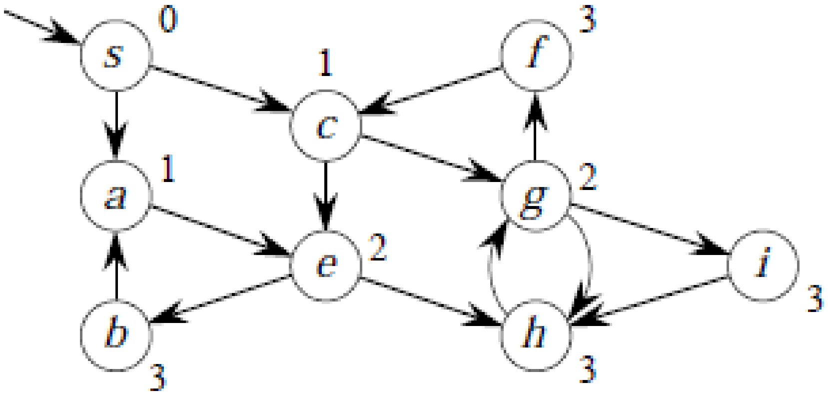
- Input: Graph $G = (V, E)$, either directed or undirected, and source vertex $s \in V$.
- Output: $v.d$ = distance (smallest # of edges) from s to v for all $v \in V$.
  - Idea: Send a wave out from s
  - First hits all vertices 1 edge from s.
  - From there, hits all vertices 2 edges from s.
  - Etc.
- Use FIFO queue Q to maintain wavefront.
  - $v \in Q$ if and only if wave has hit but has not come out of yet.

Pseudocode of Breadth-first search

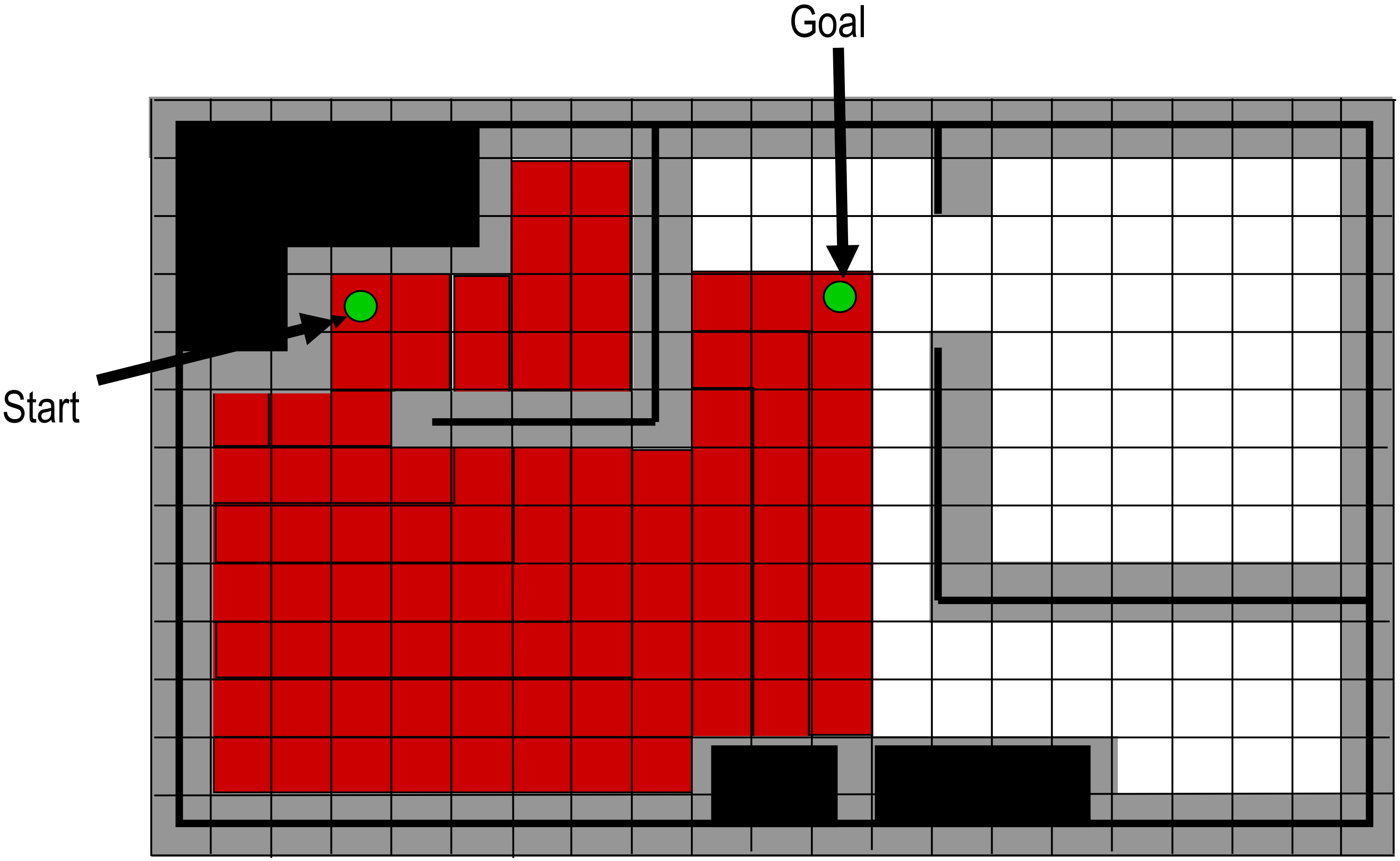
```
BFS(V, E, s)
    for each $u \in V - \{s\}$
        $u.d = \infty$
    $s.d = 0$
    $Q = \emptyset$
    Enqueue($Q$, $s$)
    while $Q \neq \emptyset$
        $u$ = Dequeue($Q$)
        for each $v \in G.Adj[u]$
            if $v.d = \infty$
                $v.d = u.d + 1$
                Enqueue($Q$, $v$)
```

- Time: O(V+E)
- Adjacency list is used.

- Example:



Example of Wavefront Planning



Bread-First: How Many Steps

S		▨			
		▨			
				▨	
		▨	▨	▨	
				▨	G

Minimum distance from source

0	1	▨	7	8	9
1	2	▨	6	7	8
2	3	4	5	▨	9
3	4	▨	▨	▨	10
4	5	6	7	▨	11

Ordering of existing the queue
(checking or expanding)

0	2	▨	15	17	19
1	4	▨	13	16	18
3	6	9	11	▨	20
5	8	▨	▨	▨	21
7	10	12	14	▨	22

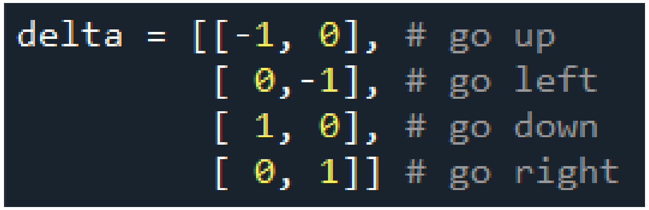
Order to check neighbors: up, left, down, right

Pseudocode of Breadth-first search: Path Planning

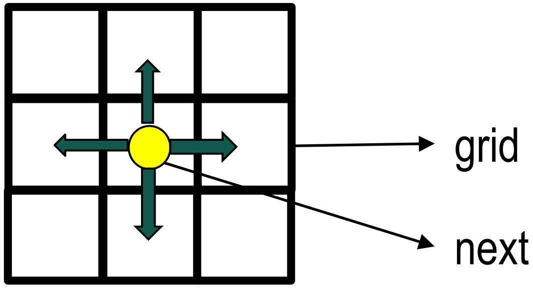
```
BFS(V, E, s, g)
    for each $u \in V - \{s\}$
        $u.d = \infty$
    $s.d = 0$
    $Q = \emptyset$
    Enqueue($Q, s$)
    step = 0
    while $Q \neq \emptyset$
        $u$ = Dequeue($Q$)
        $u.c = \text{step}$
        step = step + 1
        if (u != g)
            for each $v \in G.Adj[u]$
                if $v.d = \infty$
                    $v.d = u.d + 1$
                    Enqueue($Q, v$)
        else
            break
```

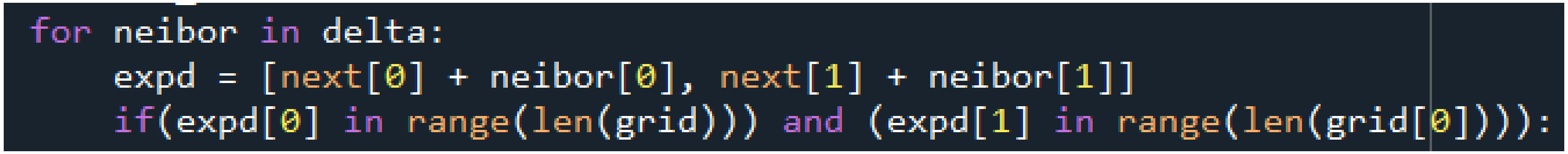
Occupancy Grid Map: Find Neighbors

- Order to check neighbors: up, left, down, right



```python
delta = [[-1, 0], # go up
         [ 0,-1], # go left
         [ 1, 0], # go down
         [ 0, 1]] # go right
```





```python
for neibor in delta:
    expd = [next[0] + neibor[0], next[1] + neibor[1]]
    if(expd[0] in range(len(grid))) and (expd[1] in range(len(grid[0]))):
```

Programming Assignment: PA4A

Using the breadth-first algorithm, define a function, search() that returns two tables:

- Value table that keeps track the minimum value from source to each cell, and
- Expand table that keeps track of which step each node was expanded.

S		▨			
		▨	G		
				▨	
		▨	▨	▨	
				▨	

Minimum distance from source

0	1	▨			
1	2	▨	6		
2	3	4	5	▨	
3	4	▨	▨	▨	
4	5	6	7	▨	

Ordering of existing the queue (checking or expanding)

0	2	▨			
1	4	▨	13		
3	6	9	11	▨	
5	8	▨	▨	▨	
7	10	12		▨	

Order to check neighbors: up, left, down, right

Programming Assignment: PA4B

Modify the search function so that it returns an additional table

- Action table that shows the shortest path as follows:

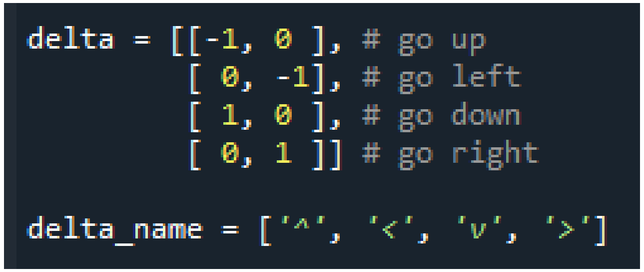
‘>’	‘v’	‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘v’	‘ ‘	‘*’	‘ ‘	‘ ‘
‘ ‘	‘>’	‘>’	‘^’	‘ ‘	‘ ‘
‘ ’	‘ ’	‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘

How to mark a proper symbol

- up → ‘v’,
- left → ‘>’
- down → ‘^’
- right → ‘<‘

Algorithm (Note: next is goal intially)

```
mark action(next) with ‘*’
while (next != start)
    for each $v \in G.Adj[u]$
        if value($v$) == values[next] - 1
            mark action($v$) with a proper symbol.
            next = $v$
            break
```



```
delta = [[-1, 0 ], # go up
         [ 0, -1], # go left
         [ 1, 0 ], # go down
         [ 0, 1 ]] # go right

delta_name = ['^', '<', 'v', '>']
```

delta_name[(a+2)%4]

$0 \le a \le 3$

Shortest paths

- Input:
  - Directed graph $G = (V, E)$
  - Weight function $w : E \to R$
- Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

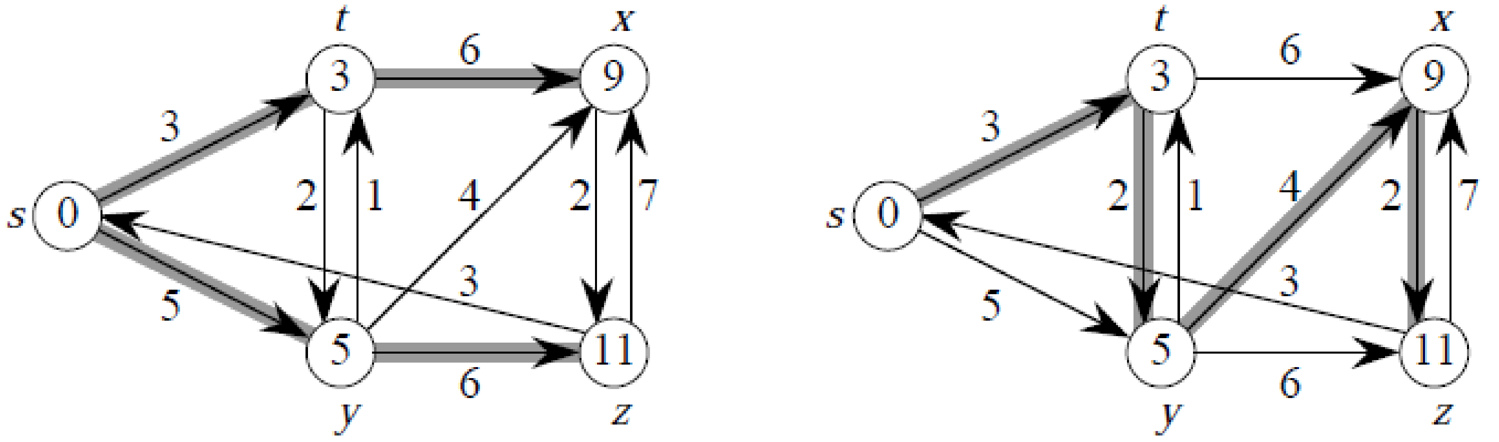
- Shortest-path weight u to v:

$$\delta(u, v) = \begin{cases} \min\left\{ w(p) : u \overset{p}{\leadsto} v \right\} & \text{if there exists a path } u \leadsto v\ , \\ \infty & \text{otherwise .} \end{cases}$$

- Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example

- Shortest paths from s



- This example shows that the shortest path might not be unique.
- It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

Variants

- **Single-source:** Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- **Single-destination**: Find shortest paths to a given destination vertex.
- **Single-pair**: Find shortest path from u to v. No way known that's better in worst case than solving single-source.
- **All-pairs**: Find shortest path from u to v for all $u, v \in V$.

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- v.*d* = $\delta(s, v)$*.*
  - Initially, v.*d* = ∞*.*
  - Reduces as algorithms progress. But always maintain v.*d* $\geq \delta(s, v)$.
  - Call v.*d a* ***shortest-path estimate.***
- v.π = predecessor of v on a shortest path from s.
  - If no predecessor, v.π = NIL.
  - π induces a tree—***shortest-path tree.***
  - We won't prove properties of π in lecture.

Dijkstra's algorithm

- No negative-weight *edges.*
- Essentially a weighted version of breadth-first search.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights (*v*.*d*).
- Have two sets of vertices:
  - S = vertices whose final shortest-path weights are determined,
  - Q = priority queue = V - S.

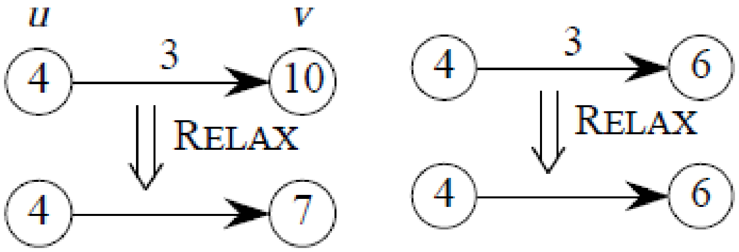
Initialization and Relaxing

- INIT-SINGLE-SOURCE.

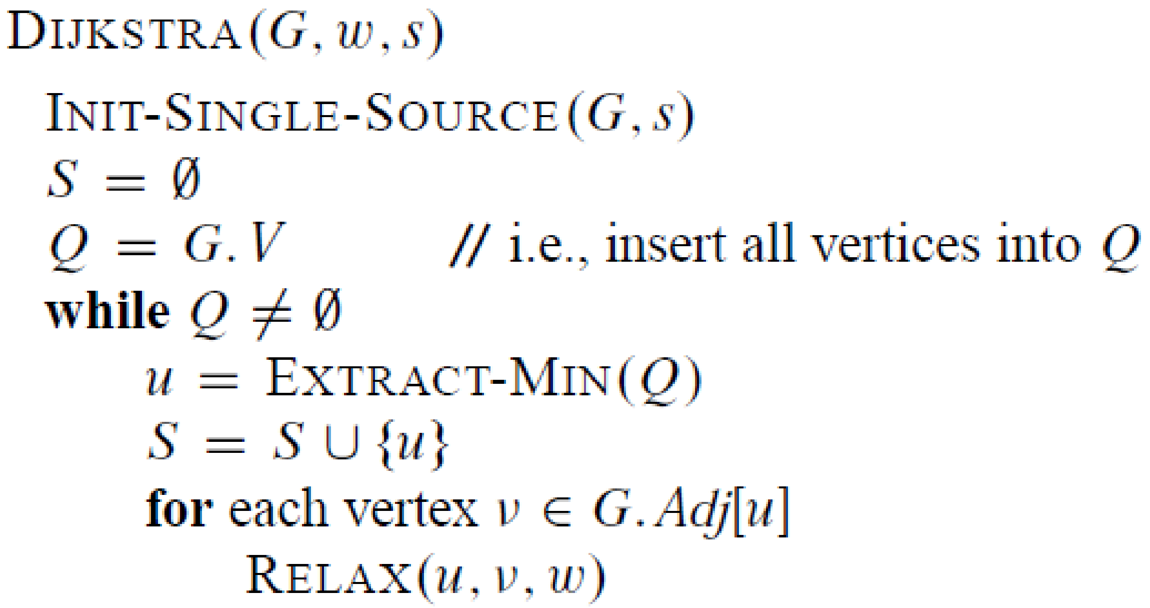
```
INIT-SINGLE-SOURCE$(G, s)$
  for each $v \in G.V$
      $v.d = \infty$
      $v.\pi = \text{NIL}$
  $s.d = 0$
```

- Relaxing an edge (u, v)

```
RELAX$(u, v, w)$
  if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
```



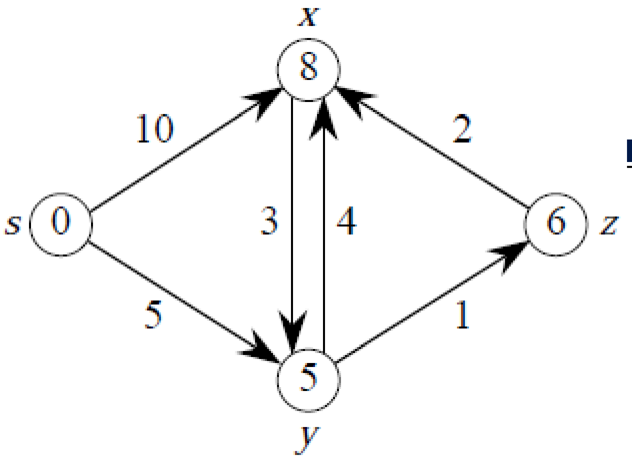
Dijkstra’s algorithm (Cont.)



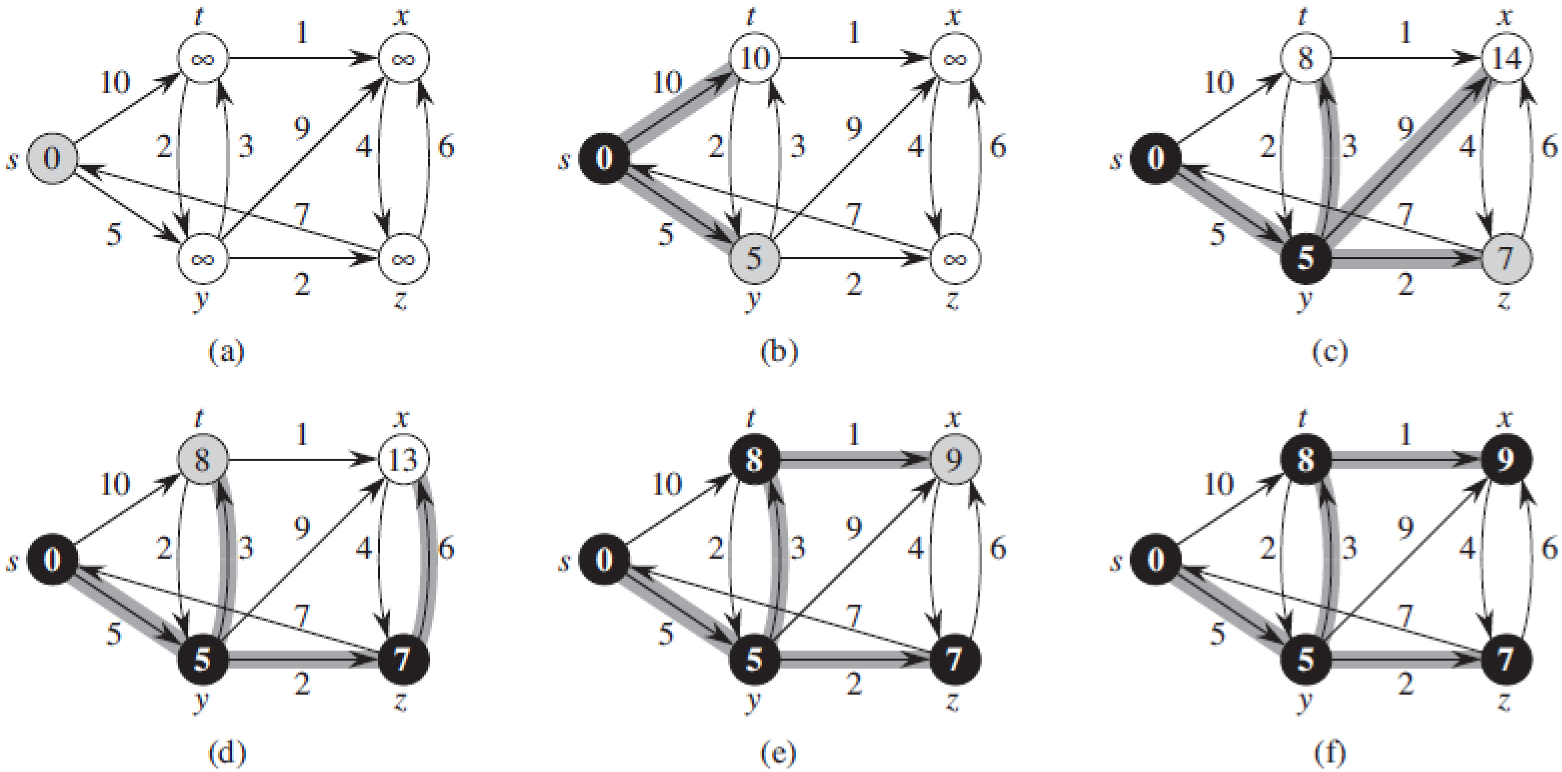
```
DIJKSTRA$(G, w, s)$
  INIT-SINGLE-SOURCE$(G, s)$
  $S = \emptyset$
  $Q = G.V$          // i.e., insert all vertices into $Q$
  while $Q \neq \emptyset$
      $u$ = EXTRACT-MIN$(Q)$
      $S = S \cup \{u\}$
      for each vertex $v \in G.Adj[u]$
          RELAX$(u, v, w)$
```

- Dijkstra’s algorithm can be viewed as greedy, since it always chooses the “lightest” (“closest”?) vertex in V-S to add to S.

Examples



- Order of adding to S: s, y, z, x.



(a) (b) (c)

(d) (e) (f)

Pseudocode of Dijkstra’s algorithm

```
DIJKSTRA$(G, w, s)$
  INIT-SINGLE-SOURCE$(G, s)$
  $S = \emptyset$
  $Q = G.V$          // i.e., insert all vertices into $Q$
  while $Q \neq \emptyset$
      $u =$ EXTRACT-MIN$(Q)$
      $S = S \cup \{u\}$
      for each vertex $v \in G.Adj[u]$
          RELAX$(u, v, w)$
```

```
INIT-SINGLE-SOURCE$(G, s)$
  for each $v \in G.V$
      $v.d = \infty$
      $v.\pi =$ NIL
  $s.d = 0$
```

```
RELAX$(u, v, w)$
  if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
```

```
Dijkstra(V, E, w, s)
  for each $u \in V$
      $u.d = \infty$
      $u.\pi = nil$
  $s.d = 0$
  $S = \emptyset$
  $Q = V$
  while $Q \neq \emptyset$
     $u$ = Extract-Min($Q$) //based on $u.d$
     $S = S \cup \{u\}$
     for each $v \in G.Adj[u]$
        if $v.d >= u.d + w(u, v)$
           $v.d = u.d + w(u, v)$
           $v.\pi = u$
```

Pseudocode of Dijkstra’s and A* algorithms

```
Dijkstra(V, E, w, s, g)
    for each $u \in V$
        $u.d = \infty$
        $u.\pi = nil$
    $s.d = 0$
    $S = \emptyset$
    $Q = V$
    while $Q \neq \emptyset$
        $u$ = Extract-Min($Q$) //based on $u.d$
        if ($u = g$)
            return “success”
        $S = S \cup \{u\}$
        for each $v \in G.Adj[u]$
            if $v.d >= u.d + w(u, v)$
                $v.d = u.d + w(u, v)$
                $v.\pi = u$
return “fail”
```

```
aStar(V, E, w, s, g, h)
    for each $u \in V$
        $u.f = \infty$ and $u.d = \infty$
        $u.\pi = nil$
    $s.f = 0 \; and \; s.d = 0$
    $S = \emptyset$
    $Q = V$
    while $Q \neq \emptyset$
        $u$ = Extract-Min($Q$) //based on $u.f$
        if ($u = g$)
            return “success”
        $S = S \cup \{u\}$
        for each $v \in G.Adj[u]$
            if $v.d >= u.d + w(u, v)$
                $v.d = u.d + w(u, v)$
                $v.\pi = u$
                $v.f = v.d + h(v)$
return “fail”
```

BFS A* algorithm

```
BFS(V, E, s, g )
    for each $u \in V - \{s\}$
        $u.d = \infty$
    $s.d = 0$
    $Q = \emptyset$
    Enqueue($Q$, $s$)    // FIFO Queue
    step = 0
    while $Q \neq \emptyset$
        $u$ = Dequeue($Q$) //based on $u.d$
        $u.c = \text{step}$
        step = step + 1
        if (u != g)
            for each $v \in G.Adj[u]$
                if $v.d = \infty$
                    $v.d = u.d + 1$
                    Enqueue($Q$, $v$)
        else
            break
```

```
BFSaStar (V, E, s, g, h)
    for each $u \in V - \{s\}$
        $u.d = \infty$
    $s.d = 0$
    $s.f = s.d + s.h$
    $Q = \emptyset$
    Enqueue($Q$, $s$) // Priority Queue
    step = 0
    while $Q \neq \emptyset$
        $u$ = Dequeue($Q$) //based on $u.f$
        $u.c = \text{step}$
        step = step + 1
        if (u != g)
            for each $v \in G.Adj[u]$
                if $v.d = \infty$
                    $v.d = u.d + 1$
                    $s.f = s.d + s.h$
                    Enqueue($Q$, $v$)
        else
            break
```

How Many Steps: Value and Expand

S		▨			
		▨	G		
				▨	
		▨	▨	▨	
				▨	

BF: Value

0	1	▨			
1	2	▨	6		
2	3	4	5	▨	
3	4	▨	▨	▨	
4	5	6	7	▨	

BF: Expand

0	2	▨			
1	4	▨	13		
3	6	9	11	▨	
5	8	▨	▨	▨	
7	10	12		▨	

A*: h-values

4	3	▨ 2	1	2	3
3	2	▨ 1	0	1	2
4	3	2	1	▨ 2	3
5	4	▨ 3	▨ 2	▨ 3	4
6	5	4	3	▨ 4	5

A*: g-values

0	1	▨			
1	2	▨	6		
2	3	4	5	▨	
3	4	▨	▨	▨	
				▨	

A*: Expand

0	1	▨			
2	3	▨	8		
4	5	6	7	▨	
		▨	▨	▨	
				▨	

Programming Assignment: PA4C

Using BFS A* algorithm, define a function, search() that returns two tables:

- Value table that keeps track the minimum value from source to each cell, and
- Expand table that keeps track of which step each node was expanded.

S		▨			
		▨	G		
				▨	
		▨	▨	▨	
				▨	

A*: g-values

0	1	▨			
1	2	▨	6		
2	3	4	5	▨	
3	4	▨	▨	▨	
				▨	

A*: Expand

0	1	▨			
2	3	▨	8		
4	5	6	7	▨	
		▨	▨	▨	
				▨	

Programming Assignment: PA4D

Modify the search function so that it returns an additional table

- Action table that shows the shortest path as follows:

‘>’	‘v’	‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘v’	‘ ‘	‘*’	‘ ‘	‘ ‘
‘ ‘	‘>’	‘>’	‘^’	‘ ‘	‘ ‘
‘ ’	‘ ’	‘ ‘	‘ ‘	‘ ‘	‘ ‘
‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘

S		▨			
		▨	G		
				▨	
		▨	▨	▨	
				▨	

A*: g-values

0	1	▨			
1	2	▨	6		
2	3	4	5	▨	
3	4	▨	▨	▨	
				▨	

Programming Project: PP4

- Implement Dijkstra Algorithm and A* Algorithm
- The map is the occupancy grid using eight-neighbor connection. Each cell has a probability of occupancy.

0	0.2	0.8	0	0	0
0	0.2	0.8	0	0	0
0	0.2	0	0	0.8	0
0	0	0.2	0.8	0.8	0
0	0	0	0.2	0.8	0

- A skeleton code is given. You only need to provide implementations for the following three functions and the update (relax) step of the Dijkstra algorithm.
  1. get_neighborhood: This function returns a vector of the neighbors of a given cell, considering the boundaries of the map.
  2. get_edge_cost: This function calculates the cost of moving from a given cell to one of its neighbors. Note that if the occupancy probability of the neighbor is greater than or equal to 0.5, then the cost is infinity. Otherwise, the cost is the distance between the two cells plus 2 times the occupancy probability of the neighbor.
  3. get_heuristic: This function calculates the distance of a given cell to the goal cell.