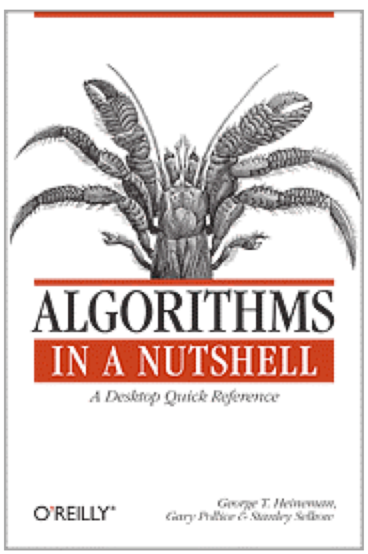
Algorithms in a Nutshell



Session 6

Graph Algorithms

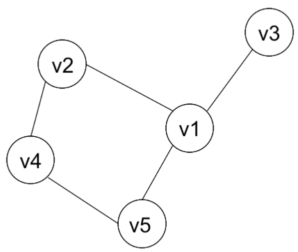
1:00 –1:50

Outline

- Overview
- Themes
  - Adjacency lists vs. adjacency matrix
  - Search strategy (breadth first vs. depth first)
  - Space vs. Time
- DIJKSTRA'S ALGORITHM
  - Implementations for sparse and dense graphs

Graphs

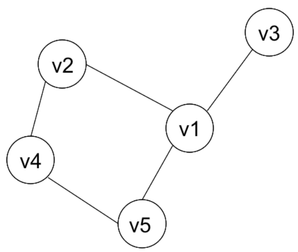
- Common data structure
  - Represents information relationships



Vertices:	v1, v2, v3, v4, v5
Edges:	(v1,v2), (v1,v3), (v1,v5), (v2,v4), (v4,v5)

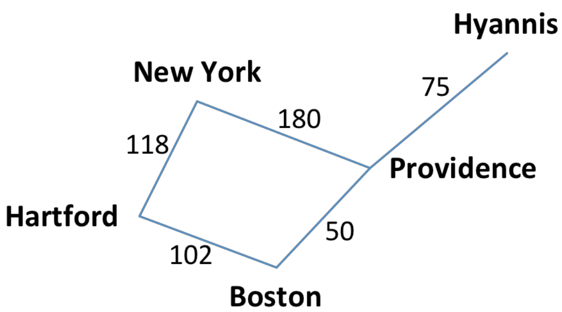
Graphs

- Common data structure
  - Represents information relationships



Vertices: v1, v2, v3, v4, v5

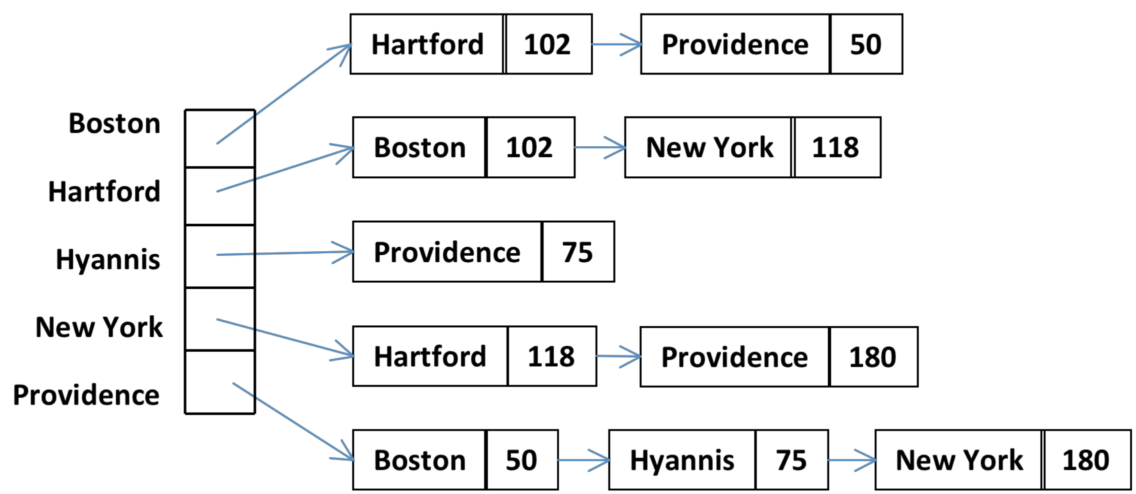
Edges: (v1,v2), (v1,v3), (v1,v5), (v2,v4), (v4,v5)



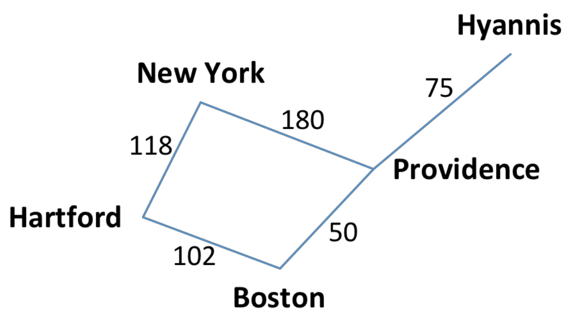
Graph Representation Options

	Boston	Hartford	Hyannis	New York	Providence
Boston	0	102	0	0	50
Hartford	102	0	0	118	0
Hyannis	0	0	0	0	75
New York	0	118	0	0	180
Providence	50	0	75	180	0

- Adjacency matrix
  - Two dimensional
  - Non-zero represents edge
  - Find edge by matrix[i][j] index
  - Space: $O(V^2)$
- Adjacency lists
  - Array of linked lists
  - Find edge requires search
  - Space: O(V+E)

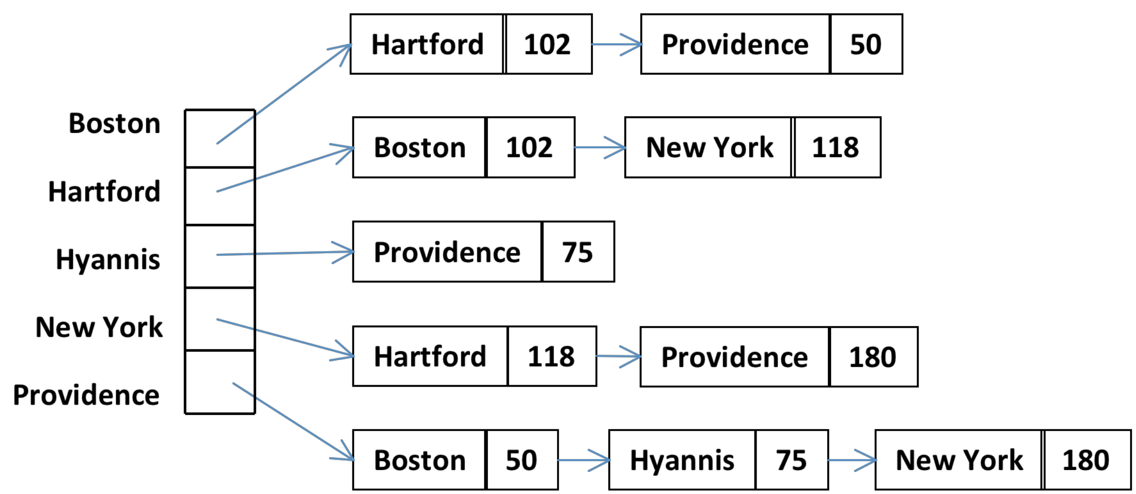


Graph Representation Options



	Boston	Hartford	Hyannis	New York	Providence
Boston	0	102	0	0	50
Hartford	102	0	0	118	0
Hyannis	0	0	0	0	75
New York	0	118	0	0	180
Providence	50	0	75	180	0

- Adjacency matrix
  - Two dimensional
  - Non-zero represents edge
  - Find edge by matrix[i][j] index
  - Space: $O(V^2)$
- Adjacency lists
  - Array of linked lists
  - Find edge requires search
  - Space: O(V+E)

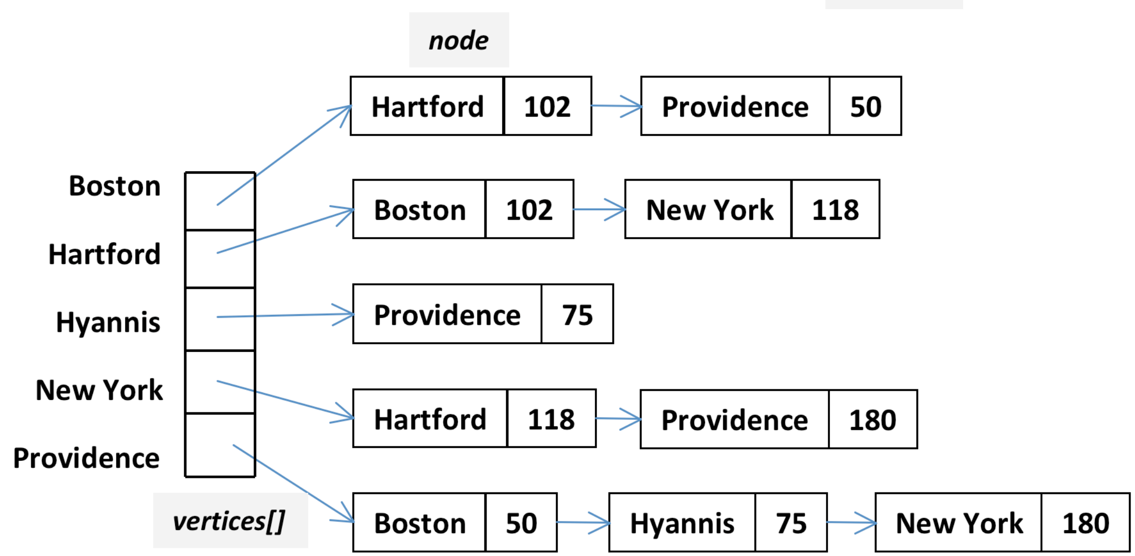


Does edge exist between "Boston" and "Providence"?

- Adjacency matrix
  - Use hash table to determine integer *i* associated with "Boston"
  - Use hash table to determine integer *j* associated with "Providence"
  - Edge exists if edge[*i*][*j*] > 0
- Adjacency lists
  - Use hash table to determine integer *i* associated with "Boston"
  - Search the linked list vertices[i] to see if a node exists whose name is "Providence"
  - Edge exists if node found

	Boston	Hartford	Hyannis	New York	Providence
Boston	0	102	0	0	50
Hartford	102	0	0	118	0
Hyannis	0	0	0	0	75
New York	0	118	0	0	180
Providence	50	0	75	180	0

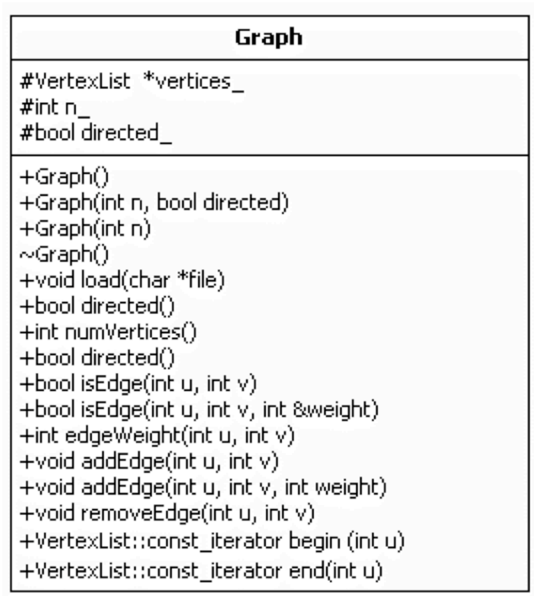
edge[][]



141

Normalized Graph Representation

- Assume all vertices are in the range [0, n)
  - Enables efficient edge lookup for adjacency matrix
- Assume all requests are normalized
  - Avoids hash table lookup



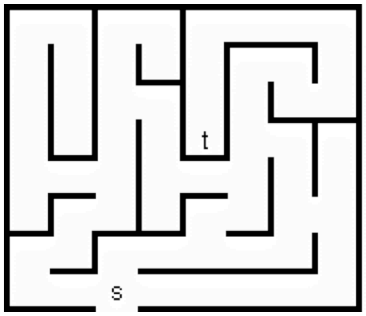

```
bool isEdge (int u, int v)
int edgeWeight (int u, int v)
void addEdge (int u, int v, int w)
```

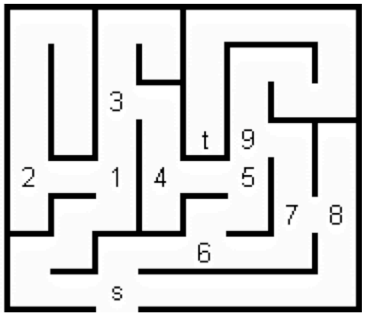
Common Graph Problems

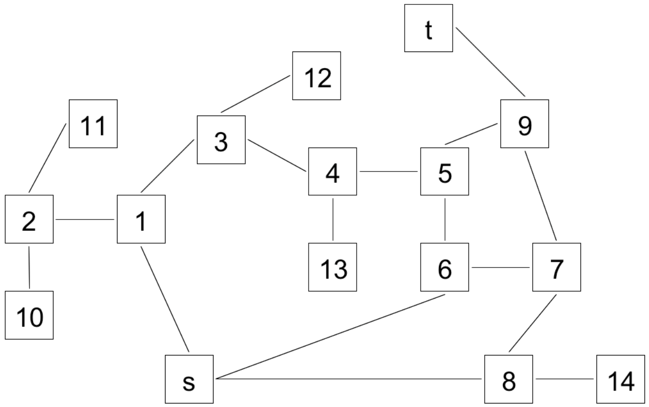
- Is there a path from V_0 to vertex V_1?
- What is shortest path from V_0 to vertex V_1?
  - In number of edges traversed
  - In accumulating edge weights
- What is the shortest path between any two vertices?
  - In accumulating edge weights

Maze Example

- Problem: Solve a rectangular maze
  - "Is there a path from S to T"
- Mapping a problem to a graph
  - Identify vertices and edges



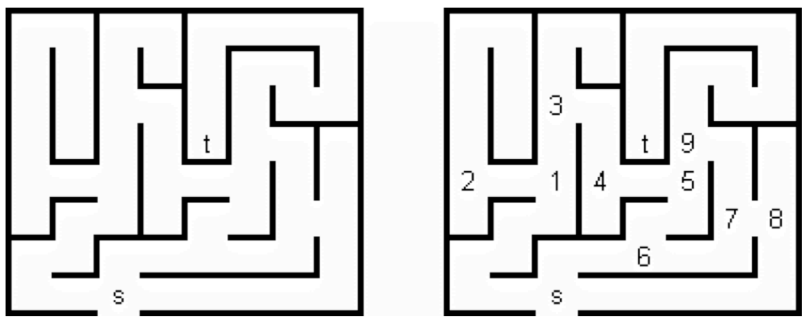


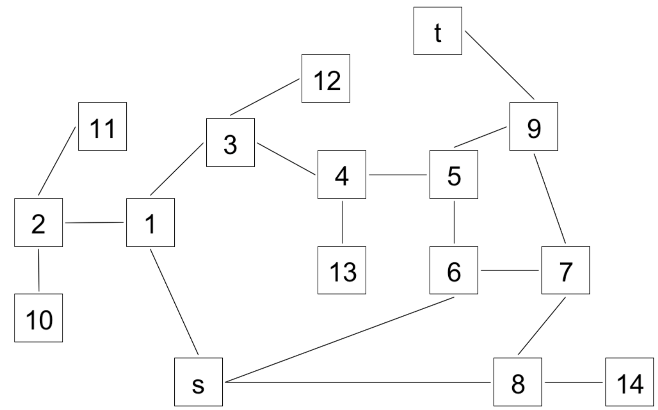


Vertex represents maze decision point

Edge represents path in maze between decision points

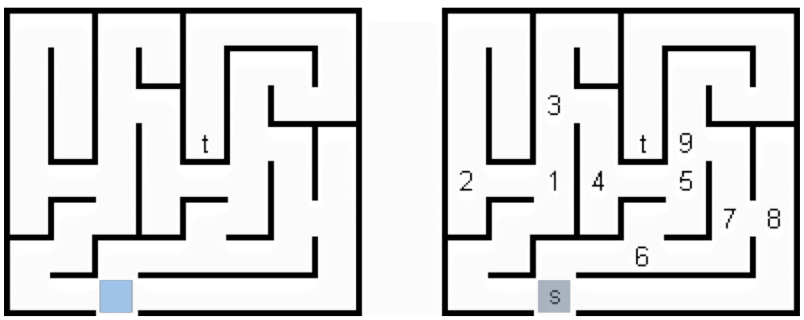
Maze Search

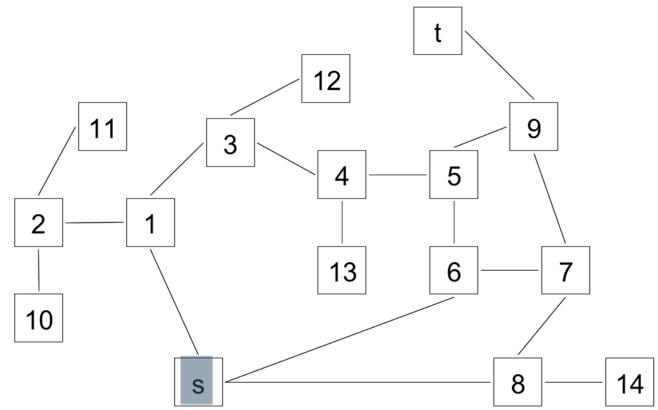




- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options
- To implement
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision

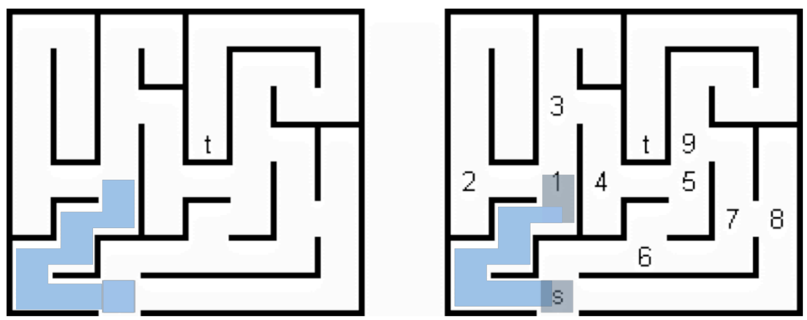
Maze Search

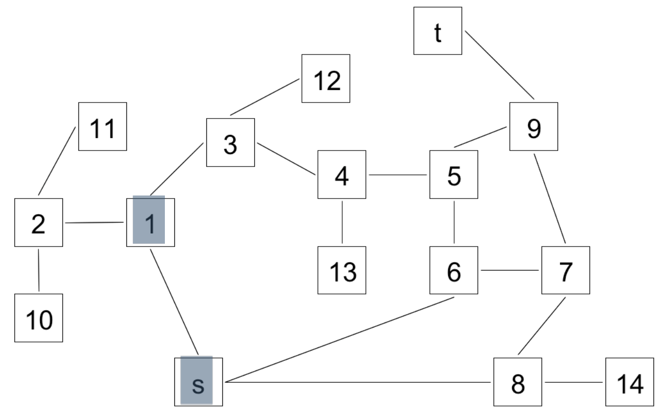




- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options
- To implement
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision

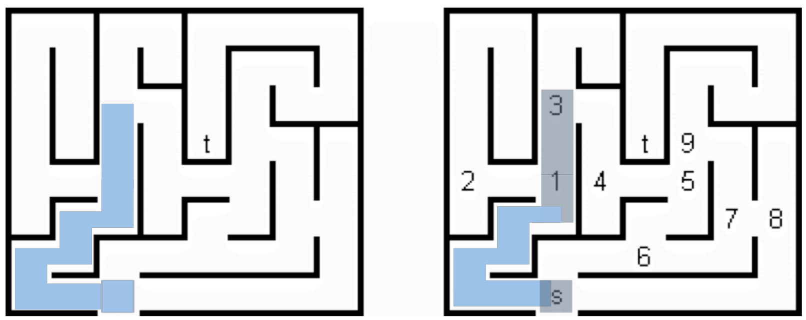
Maze Search

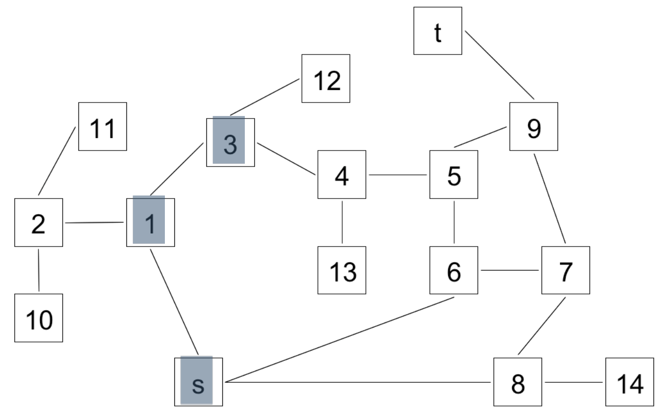




- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options
- To implement
  - Must keep track of "active search horizon"
  - Must be able to backtrack to revisit earlier decision

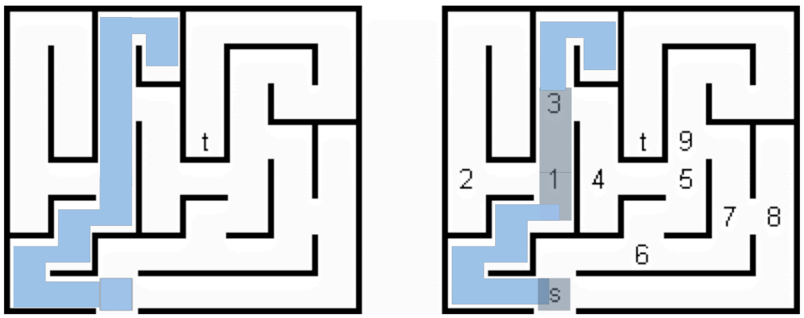
Maze Search

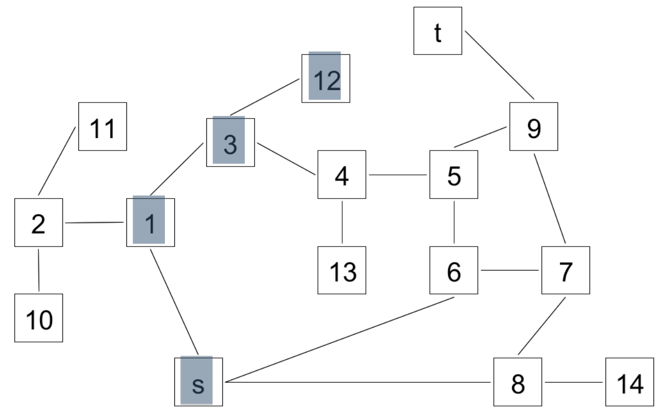




- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options
- To implement
  - Must keep track of “active search horizon”
  - Must be able to backtrack to revisit earlier decision

Maze Search





- Depth-First Strategy
  - Assume solution is always one step away
  - Never visit the same vertex twice – avoids infinite loops
  - Backtrack to earlier decision when you run out of options
- To implement
  - Must keep track of "active search horizon"
  - Must be able to backtrack to revisit earlier decision

Depth-first search of a graph

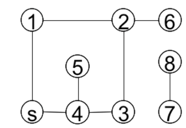
- Backtracking using recursion
  - Each invocation of **dfs_visit** visits a new vertex
  - Only called on White vertices
- Record search progress by *coloring* vertices after they have been visited
  - White = Unvisited
  - Gray = Visited but haven't visited all neighbors
  - Black = Visited and have visited all neighbors
- To record search path, use `pred` array to store path
- If graph is disconnected
  - Lines 5-7 completes search

DEPTH-FIRST SEARCH

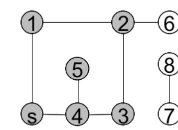
Graph · Recursion · Array · Backtracking

Best	Average	Worst
O(V+*E*)	O(V+*E*)	O(V+*E*)

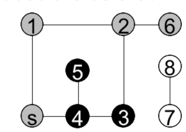
```
depthFirstSearch (G, s)
1. foreach v∈V do
2.    pred[v] = – 1
3.    color[v] = White
4. dfs_visit (s)

5. foreach v∈V do
6.    if (color[v] = White) then
7.       dfs_visit (v)
end

dfs_visit (u)
1. color[u] = Gray

2. for each neighbor v of u
3.    if (color[v] = White) then
4.       pred[v] = u
5.       dfs_visit (v)
6. color[u] = Black
end
```



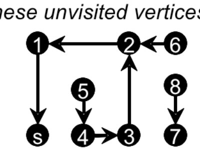
`dfs_visit` recursively visits the vertices (1--5) marking each one Gray until it finds one with no White neighbor vertex (i.e., 5)



As each `dfs_visit` completes, unvisited vertices initially passed over are explored (i.e., 6 was a White neighbor of 2). Completed vertices are colored Black.



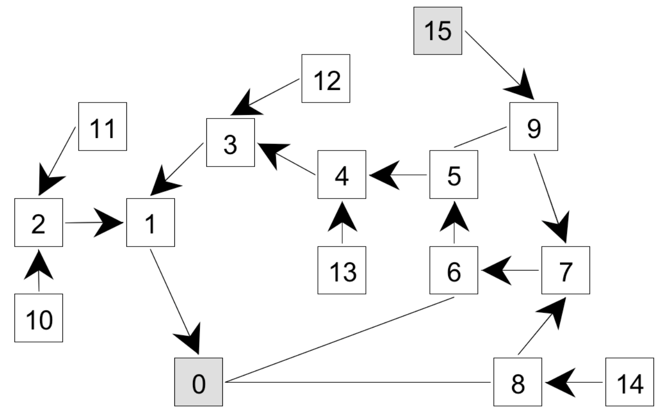
If the graph is unconnected then some vertex will be colored White. Continue to explore these unvisited vertices.



pred[] information records depth-first forest discovered, shown as arrows

Code Check

- Code check
  - Debug figure6_10.exe
  - Review code
- Breakpoint in dfs_visit
  - Note stack trace when u=15




```
dfs_visit
dfs_visit
dfs_visit
dfs_visit
dfs_visit
dfs_visit
dfs_visit
dfs_visit
dfs_visit
dfs_search
main
```

```
pred[] results
0:  -1
1:  0
2:  1
3:  1
4:  3
5:  4
6:  5
7:  6
8:  7
9:  7
10: 2
11: 2
12: 3
13: 4
14: 8
15: 9
```

Implementation Details

- Keep track of "active search horizon"
  - The recursion stack of dfs_visit invocations
- Backtrack to revisit earlier decision

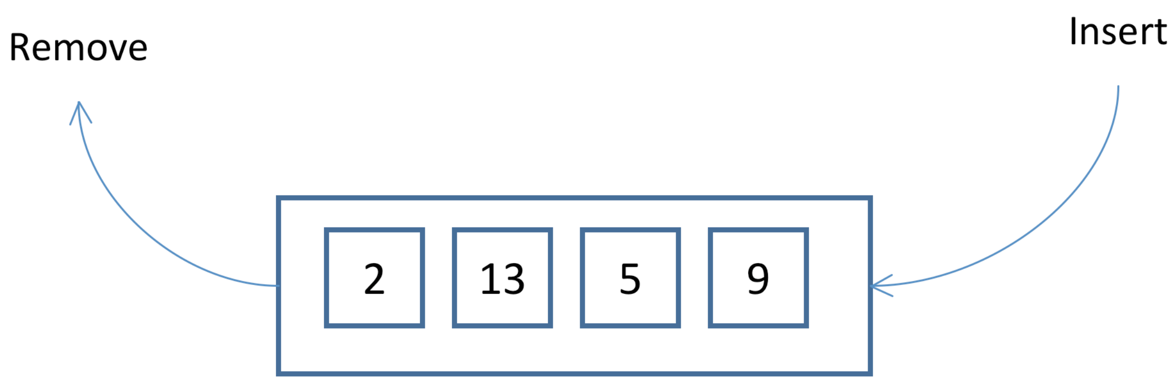
```
void dfs_visit (Graph const &graph, int u, vector<int> &pred, vector<vertexColor>
&color) {
  color[ u] = Gray;

  // process all neighbors of u.
  for (VertexList::const_iterator ci = graph.begin(u); ci != graph.end(u); ++ci) {
    int v = ci->first;

    // Explore unvisited vertices immediately and record pred[ ] . Once
    // recursive call ends, backtrack to adjacent vertices.
    if (color[ v] == White) {
      pred[ v] = u;
      dfs_visit (graph, v, pred, color);
    }
  }

  color[ u] = Black;  // our neighbors are complete; now so are we.
}
```

Breadth-First Search Strategy

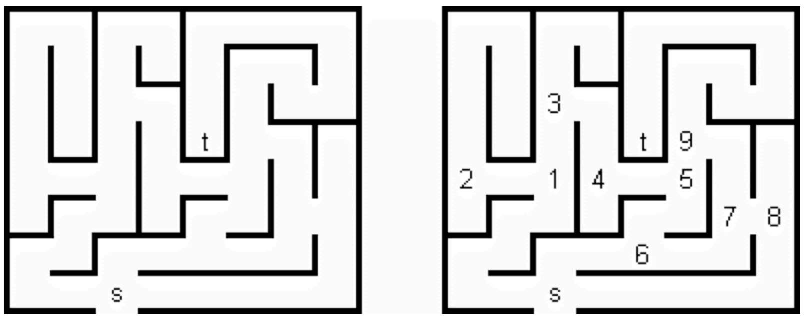
- Systematic exploration of graph
  – Visit all vertices that are *k* edges away from initial vertex before visiting vertices *k+1* edges away
- Only visit unmarked vertices
  – Use same coloring scheme as DEPTH-FIRST SEARCH
- Maintain "active search horizon"
  – Use <u>queue</u> to store to-be-visited vertices

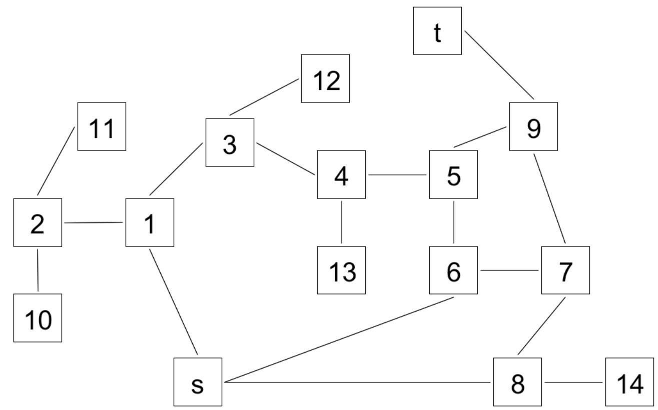
Queue Data structure

- Insert elements to the end
- Remove elements from the front



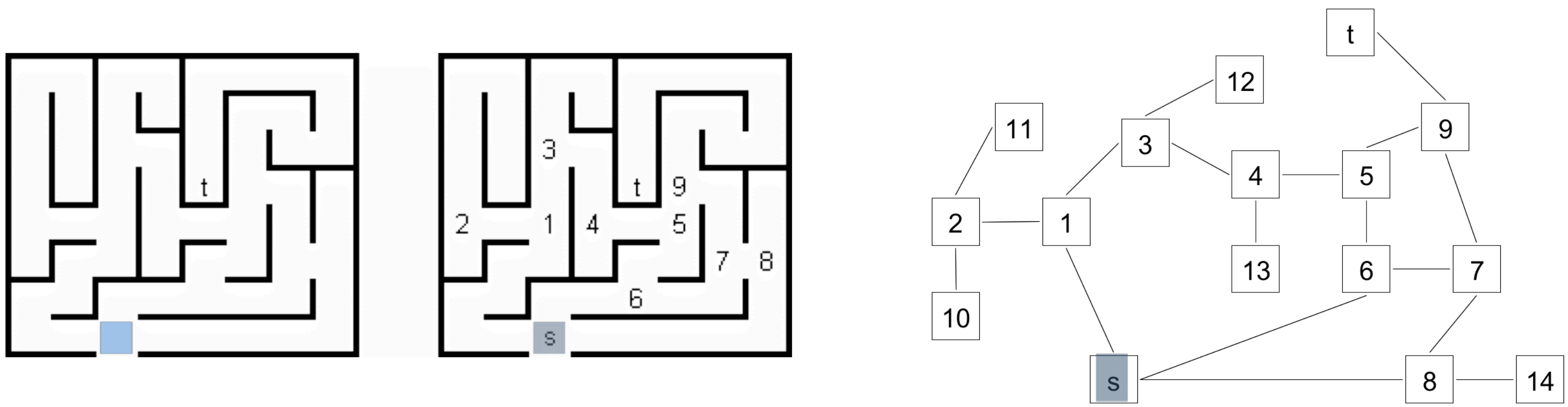
Maze Search





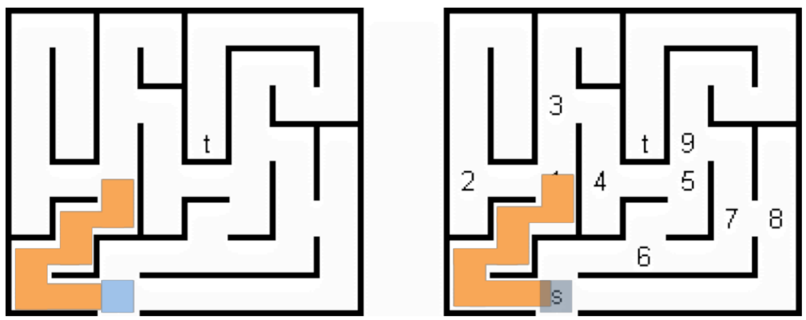
- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

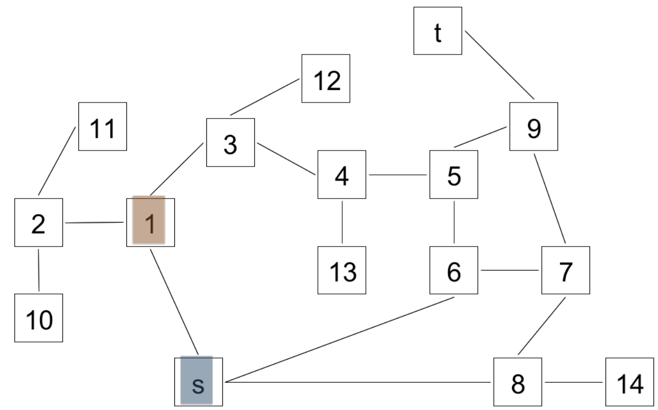
Maze Search



- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

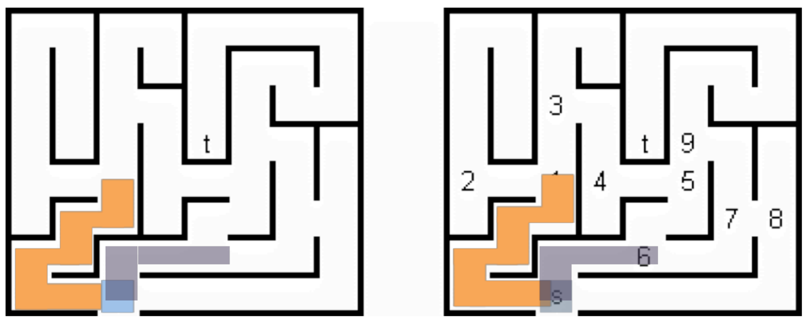
Maze Search

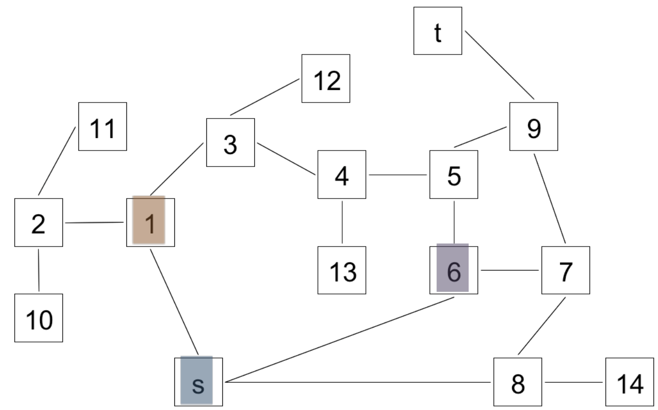




- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

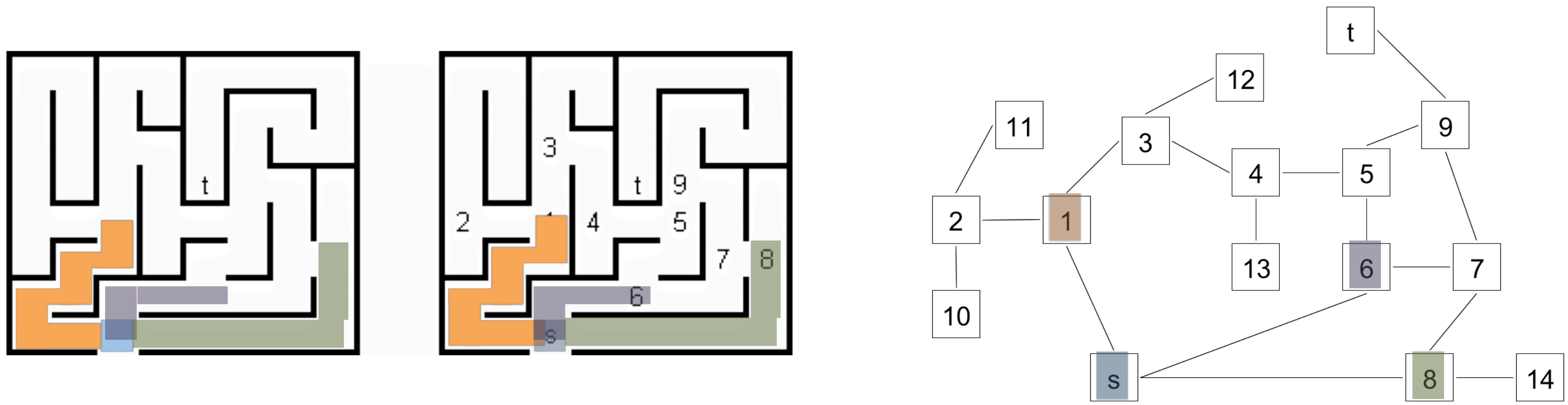
Maze Search





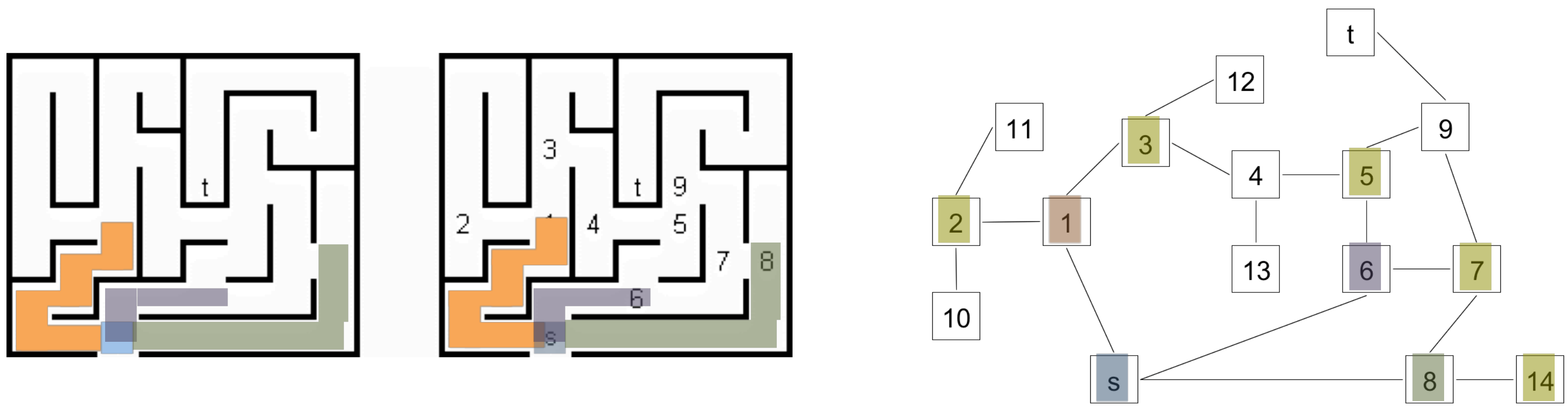
- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

Maze Search



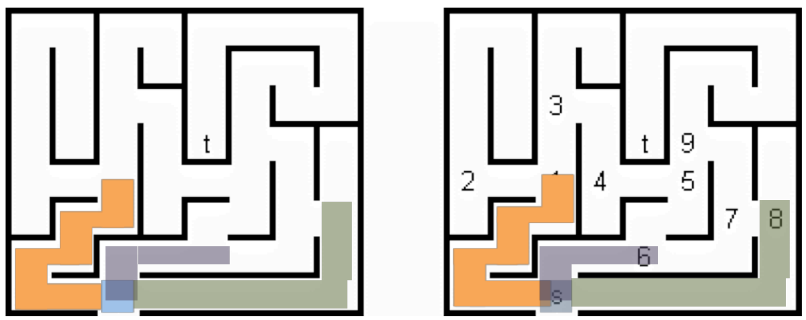
- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

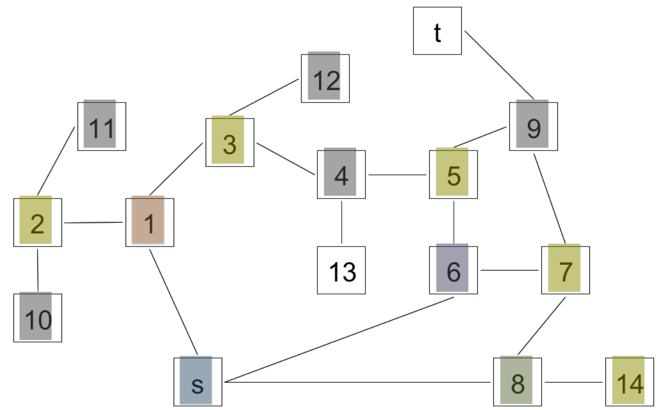
Maze Search



- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of "active search horizon"

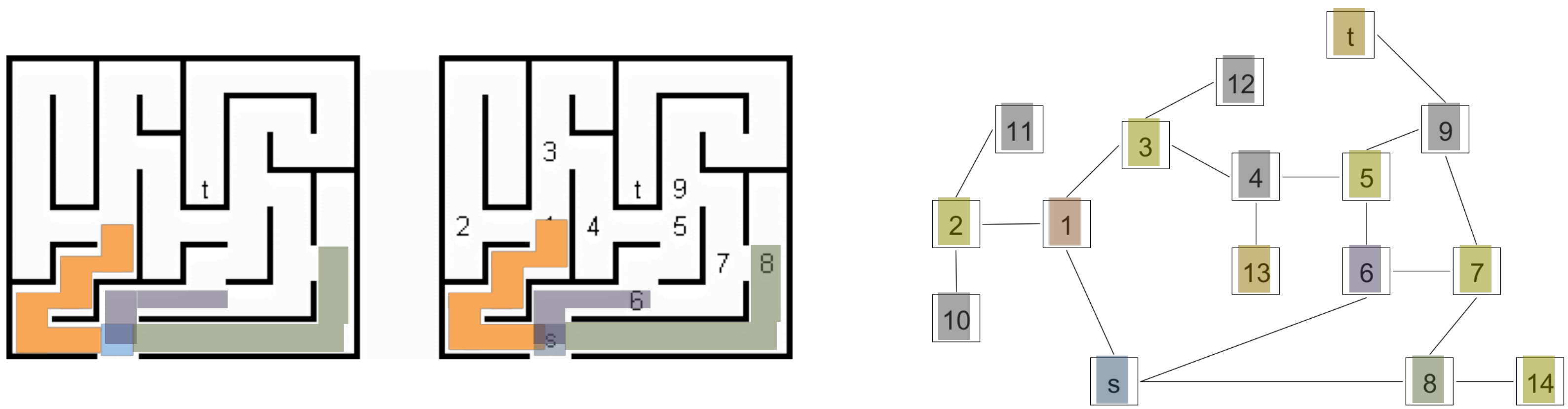
Maze Search





- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

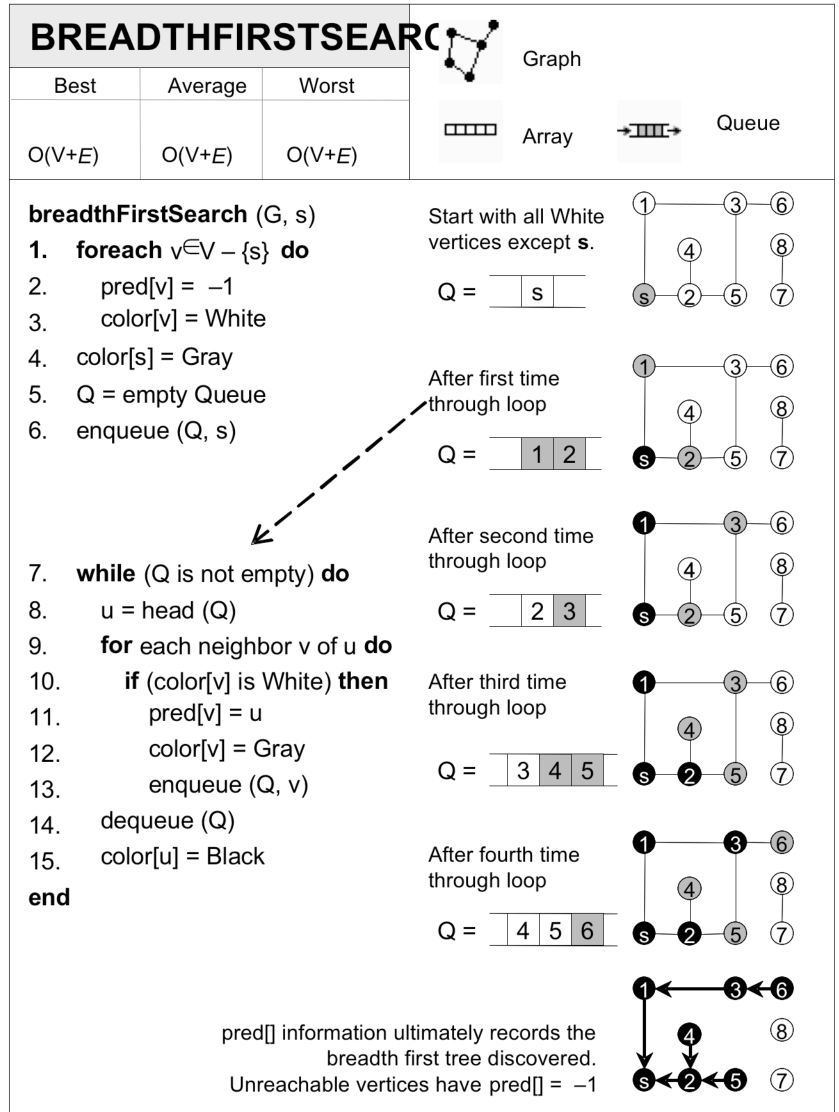
Maze Search



- Breadth-First Strategy
  - Visit vertices *k* edges away before visiting those *k+1* edges away
  - Never visit the same vertex twice – avoids infinite loops
- To implement
  - Use queue to keep track of “active search horizon”

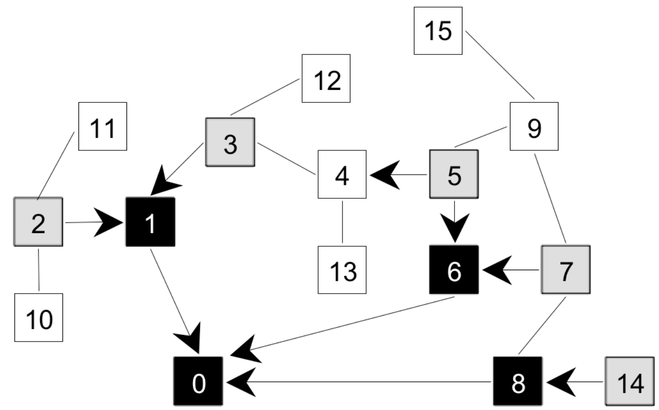
Breadth-first search of a graph

- Systematic Exploration of graph
  - Will find shortest path from *s* to every node in graph
  - Will leave unreachable vertices unvisited
- Non-recursive



Code Check

- Code check
  - Debug figure6_12.exe
  - Review code
- Breakpoint in bfs_visit
  - Stop when *u* = 2
  - Colored vertices as shown
  - *pred*[] info as shown



v	dist[]	pred[]
0	0	-1
1	1	0
2	2	1
3	2	1
4	INF	-1
5	2	6
6	1	0
7	2	6
8	1	0
9	INF	-1
10	INF	-1
11	INF	-1
12	INF	-1
13	INF	-1
14	2	8
15	INF	-1

Implementation Details

- Keep track of "active search horizon"
  - Queue holds vertices to be visited
  - Only add "Gray" nodes to Queue

```
void bfs_search (Graph const &graph, int s,             /* in */
                 vector<int> &dist, vector<int> &pred){  /* out */
  // initialize dist and pred. Begin at s
  // and mark as Gray since we haven't yet visited its neighbors.
  const int n = graph.numVertices();
  pred.assign(n, -1);
  dist.assign(n, numeric_limits<int>::max());
  vector<vertexColor> color (n, White);

  dist[ s] = 0;
  color[ s] = Gray;

  queue<int> q;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();

    // Explore neighbors of u to expand the search horizon
    for (VertexList::const_iterator ci = graph.begin(u);
         ci != graph.end(u); ++ci) {
      int v = ci->first;
      if (color[ v] == White) {
        dist[ v] = dist[ u]+1;
        pred[ v] = u;
        color[ v] = Gray;
        q.push(v);
      }
    }

    q.pop();
    color[ u] = Black;
  }
}
```

Space vs. Time

- Depth-First and Breadth-First both iterate over the edges for a vertex
  - Adjacency List via Iterator
  - Adjacency Matrix via double-loop
- Costs change if **sparse** or **dense** graph

```
// Explore neighbors of u to expand
// search horizon
for (VertexList::const_iterator ci = graph.begin(u);
     ci != graph.end(u); ++ci) {

  int v = ci->first;

  ...
}
```

```
// Explore neighbors of u to expand
// search horizon
for (int v = 0; v < n; v++) {
  if (graph.edge[ u][ v] == 0) { continue; }

  ...
}
```

Searching

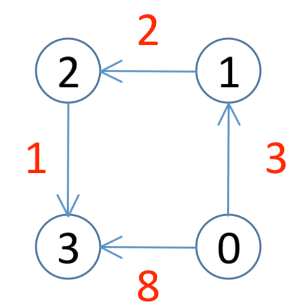
- Breadth-First and Depth-First can determine whether path exists between two vertices
  - What if you wanted to consider edge weights?
  - That is, find shortest path between v_0 and v_1?
    - Breadth-first finds path with smallest number of edges
- Single-Source Shortest Path
  - Edges are now directed and have weights
  - DIJKSTRA'S ALGORITHM (1959)

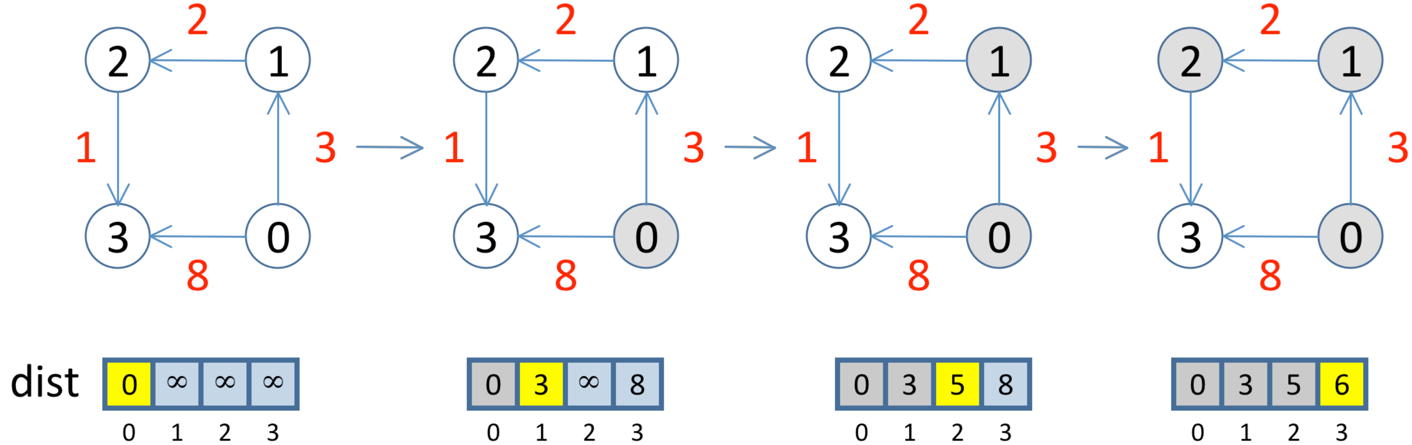
Searching with purpose

- Breadth-First and Depth-First are blind searches
  - BFS ignores context as it systematically executes
  - DFS selects a direction at random
- Goal: find shortest distance using edge weights
  - How do we avoid generating all possible paths?
- Employ Greedy Strategy
  - Find shortest distance from v_0 to all vertices
  - Computing for all makes problem easier to solve!

Single-Source Shortest Path

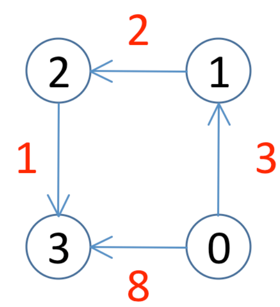
- Goal: find shortest distance from 0 to 3
- Key idea
  - Compute running "shortest distance" from source to all vertices
  - Expand marked region by adding the vertex with smallest distance (marked in yellow below)

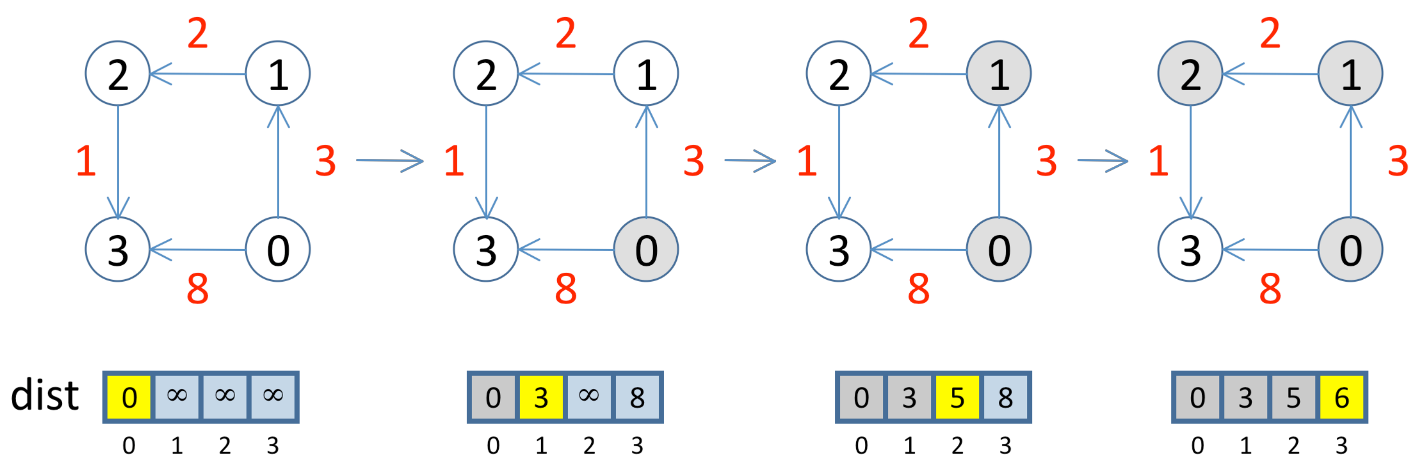




Single-Source Shortest Path

- Goal: find shortest distance from 0 to 3
- Key idea
  - Compute running "shortest distance" from source to all vertices
  - Expand marked region by adding the vertex with smallest distance (marked in yellow below)





How can we efficiently locate the vertex with smallest distance?

Use a Priority Queue!

Priority Queue data structure

- Add element with associated numeric priority
  - Lower priority numbers imply greater priority
- Retrieve element with lowest priority

If these are the only operations you need, then you can use an ordinary Binary Heap for efficient implementation. However, we also need:

- Decrease priority of existing element
  - How to avoid O(q) search for element within PQ?

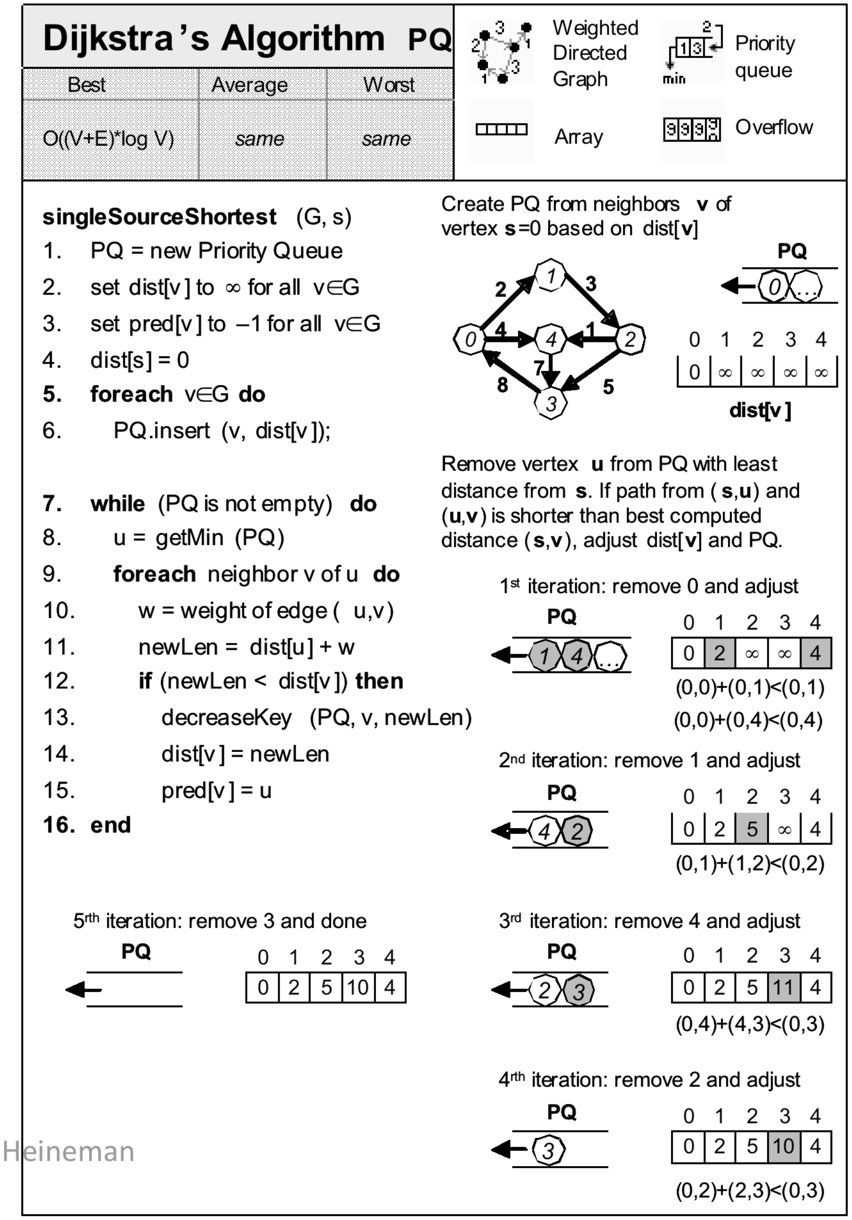
Binary Heap with extra space

- We can use binary heap as PQ here because
  - We know maximum size will be *n*
- *decreaseKey* operation can be done in O(log q)
  - Store additional space, only O(n)

```
class BinaryHeap {
 public:
  BinaryHeap (int);
  ~BinaryHeap ();

  bool isEmpty() { return (_n == 0); }
  int smallest();
  void insert (int, int);
  void decreaseKey (int, int);

 private:
  int          _n;          // number of elements in binary heap
  ELEMENT_PTR  _elements;   // values in the heap
  int          *_pos;       // pos[ i] is index into elements for ith value
};
```

DIJKSTRA'S ALGORITHM

- Initialization
  - Construct PQ with *n* vertices
- Core step
  - Extract vertex *u* with smallest distance
  - If distance (*s*,*u*) + (*u*,*v*) ≤ (*s*,*v*) for a neighboring *v* of *u*, then reduce `dist`[*v*] and its location in PQ
- How to reproduce actual shortest path?
  - Follow `pred`[] reference which is computed by the algorithm



Dijkstra's Algorithm PQ

Weighted Directed Graph · Priority queue · Array · Overflow

Best	Average	Worst
O((V+E)*log V)	*same*	*same*

```
singleSourceShortest (G, s)
  PQ = new Priority Queue
  set dist[v] to ∞ for all v∈G
  set pred[v] to –1 for all v∈G
  dist[s] = 0
  foreach v∈G do
    PQ.insert (v, dist[v]);

  while (PQ is not empty) do
    u = getMin (PQ)
    foreach neighbor v of u do
      w = weight of edge ( u,v)
      newLen = dist[u] + w
      if (newLen < dist[v]) then
        decreaseKey (PQ, v, newLen)
        dist[v] = newLen
        pred[v] = u
end
```

Code Check

```
void singleSourceShortest(Graph const &g, int s,                  /* in */
                          vector<int> &dist, vector<int> &pred) { /* out */
  // initialize dist[] and pred[] arrays. Start with vertex s by setting
  // dist[] to 0. Priority Queue PQ contains all v in G.
  const int n = g.numVertices();
  pred.assign(n, -1);
  dist.assign(n, numeric_limits<int>::max());
  dist[s] = 0;
  BinaryHeap pq(n);
  for (int u = 0; u < n; u++) { pq.insert (u, dist[u]); }

  // find vertex in ever shrinking set, V-S, whose dist[] is smallest.
  // Recompute potential new paths to update all shortest paths
  while (!pq.isEmpty()) {
    int u = pq.smallest();

    // For neighbors of u, see if newLen (best path from s->u + weight
    // of edge u->v) is better than best path from s->v. If so, update
    // in dist[v] and re-adjust binary heap accordingly. Compute in
    // long to avoid overflow error.
    for (VertexList::const_iterator ci = g.begin(u); ci != g.end(u); ++ci) {
      int v = ci->first;
      long newLen = dist[u];
      newLen += ci->second;
      if (newLen < dist[v]) {
        pq.decreaseKey (v, newLen);
        dist[v] = newLen;
        pred[v] = u;
      }
    }
  }
}
```

Summary

- Rich family of graph algorithms
  - BFS and DFS provide search strategies
  - Greedy Algorithms (PRIM's Minimum Spanning Tree)
  - Dynamic Programming
- Algorithm designer Robert Tarjan said
  - "with the right data structure most quadratic problems can be solved in O(n log n)" (paraphrased)

Performance Comparison

- Compare the following performance families
  - $O((V+E)*\log V)$ DIJKSTRA'S ALGORITHM
  - $O(V^2 + E)$ DIJKSTRA'S ALGORITHM DG

Graph Type	**O ((V+E)*log V)**	**Comparison**	**$O(V^2+E)$**	**Example**
Sparse: E is O(V)	O (V log V)	Is smaller than	$O(V^2)$	4096 **Vertices** 6000 **Edges** (.03%)
Break-Even: E is $O(V^2/\log V)$	$O(V^2 + V*\log V) = O(V^2)$	Is equivalent to	$O (V^2+V^2/\log V) = O(V^2)$	4096 **Vertices** 1,398,101 **Edges** (8%)
Dense: E is $O(V^2)$	$O (V^2 \log V)$	Is larger than	$O(V^2)$	4096 **Vertices** 4,193,280 **Edges** (25%)