Introduction to Cognitive Robotics

Module 3: Mobile Robots

Lecture 8: Finding a shortest path in a map; breadth-first search algorithm

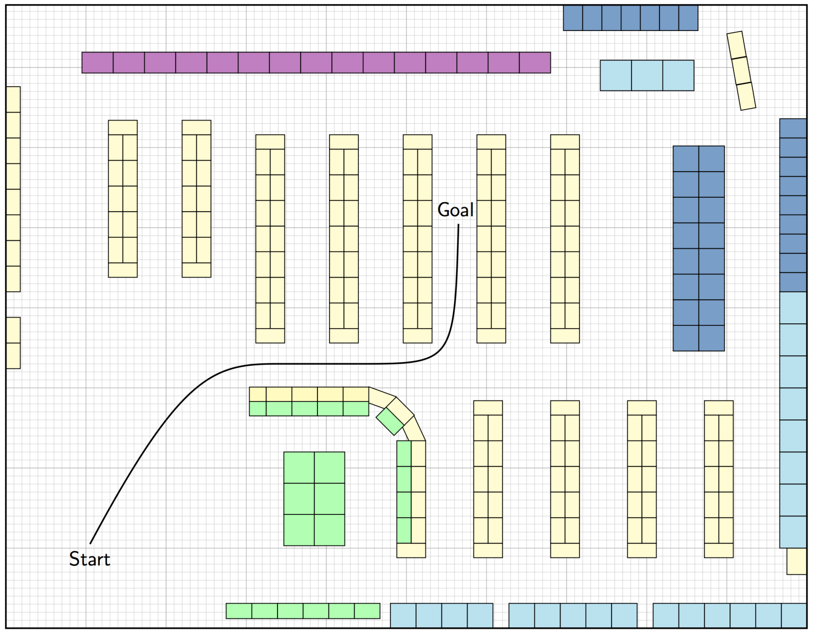
David Vernon
Carnegie Mellon University Africa

www.vernon.eu

Finding the Shortest Path in a Map

Environment map

- Assume we have a discrete map of the environment
  - It comprises an occupancy grid of n x m cells
  - Each cell in the map is either
    - free (and can be traversed by the robot)
    - occupied (by an obstacle)
- The goal is to find the shortest path
  - From a start position
  - To a goal position



Map recreated from the following papers:

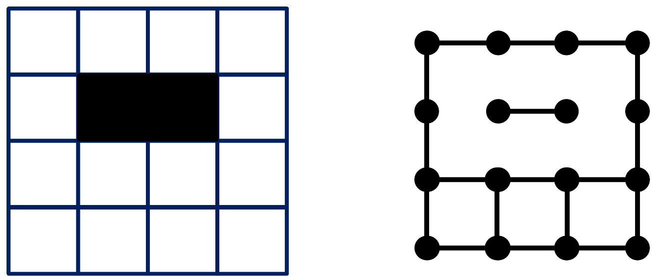
Joho, D., Senk, M., & Burgard, W. (2009). Learning wayfinding heuristics based on local information of object maps. Proceedings of the European Conference on Mobile Robots (ECMR) 2009, 117–122.

Kalff, C., & Strube, G. (2009). Background knowledge in human navigation: a study in a supermarket. Cognitive Processing, 10(2), 225-228.

Finding the Shortest Path in a Map

Environment map

– If we represent the map as a graph
  - Free cells are vertices in one or more connected components
  - Obstacle cells are vertices in one or more connected components
    – Not strictly necessary because the robot path is confined to the free space connected component(s)



– We can use graph traversal algorithms to find the shortest path connecting a start position and a goal position

Finding the Shortest Path in a Map

Environment map

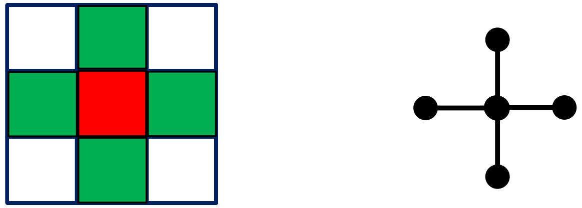
Vertices represent free space, i.e., navigable space

What about the edges? There are two possibilities

1. A vertex can be connected to four horizontal neighbour vertices: 4-connectivity

    - All edges represent the same distance, e.g. 1

Use an unweighted graph

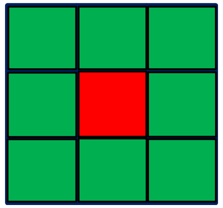


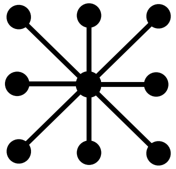
Finding the Shortest Path in a Map

Environment map

2. A vertex can be connected to all eight neighbour vertices: 8-connectivity

- Horizontal edges represent distance of 1
- Diagonal edges represent a distance of $\sqrt{2}$





Need to use a weighted graph:

- weight of 1 for horizontal and vertical edges
- weight $\sqrt{2}$ of for diagonal edges

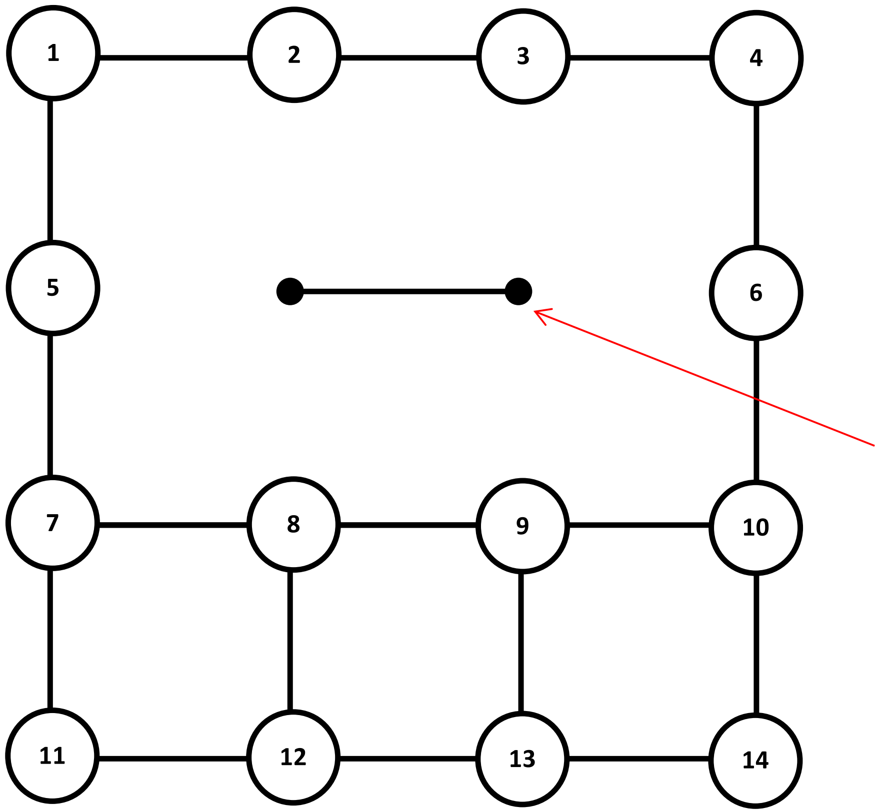
Breadth-First Search
BFS

A BFS from some start vertex

- finds the shortest path
- to all other vertices

in an unweighted graph

Breadth-First Search

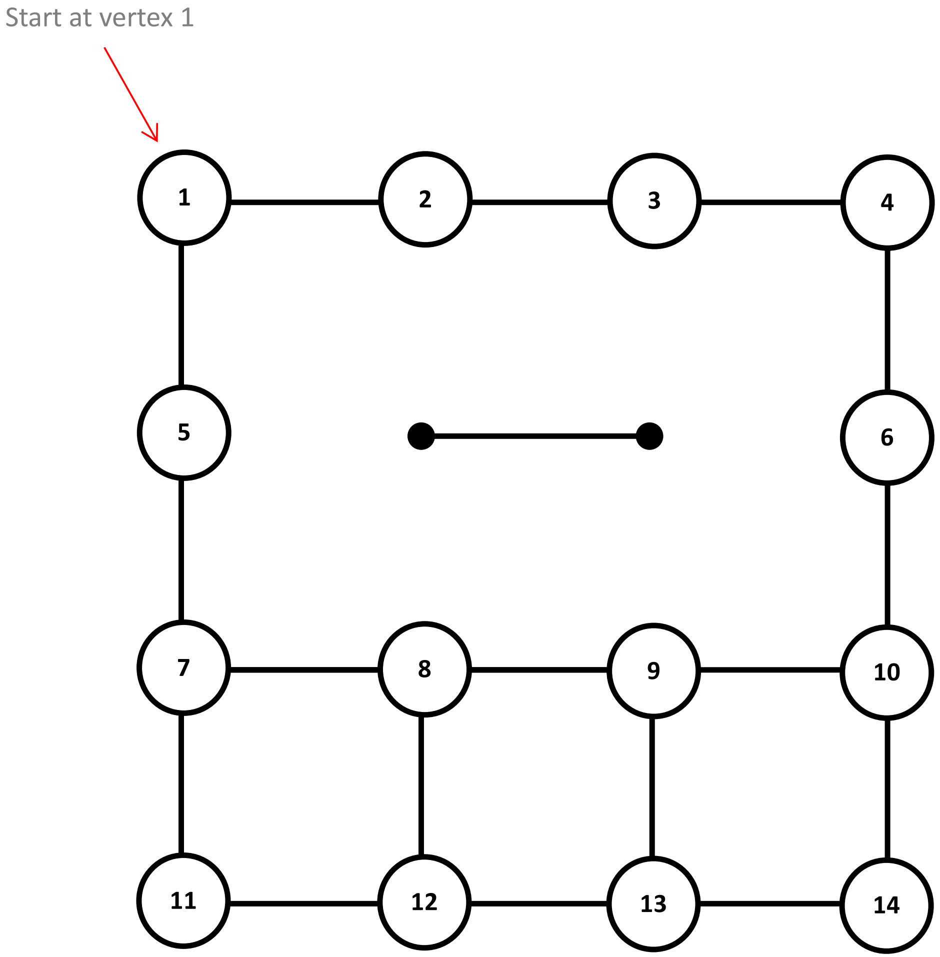


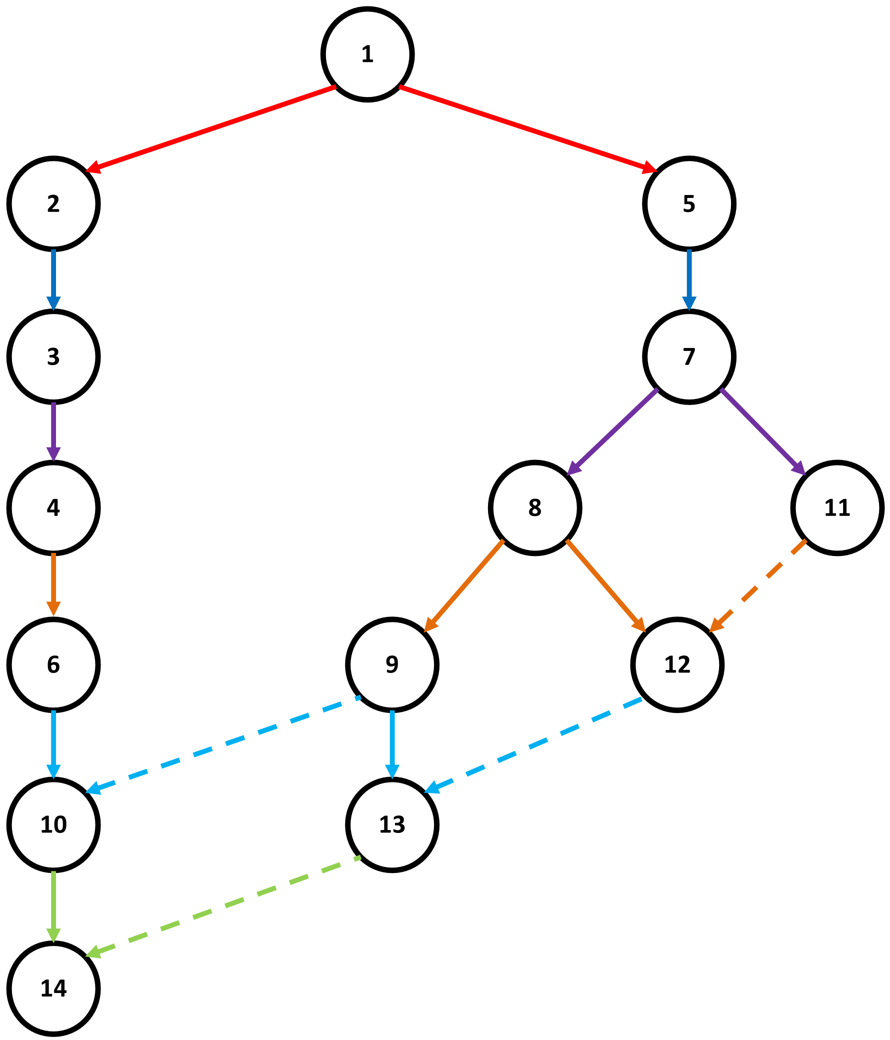
Alternatively, we could have included the object vertices and labelled the vertices 1 – 16.

In this case, the object vertices would have formed a separate component in the graph.

Thus, they would not be discoverable in a BFS from any free-space (i.e., navigable) vertex, e.g., vertex 1.

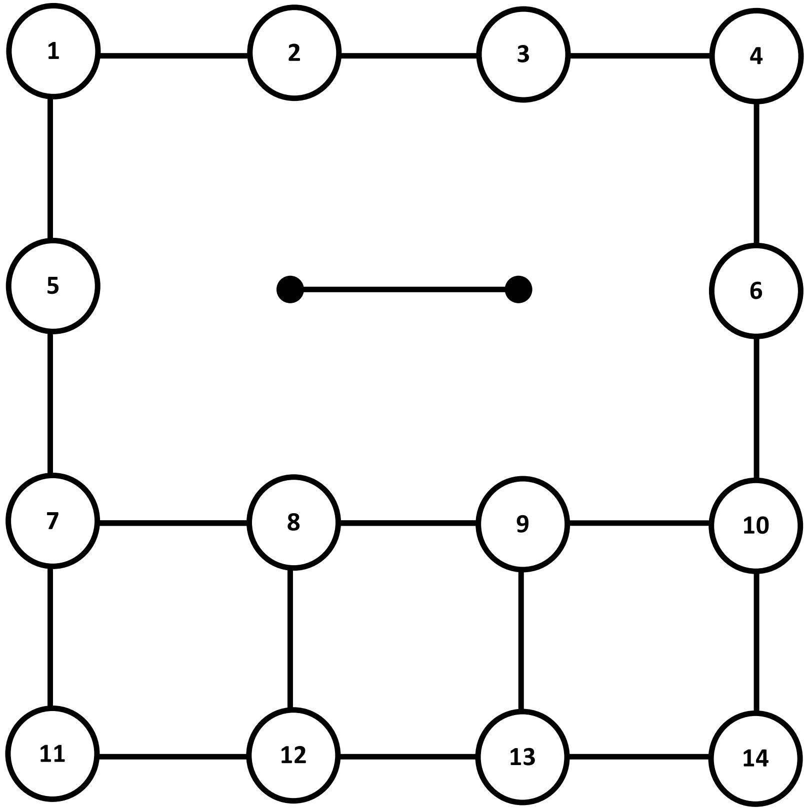
Breadth-First Search

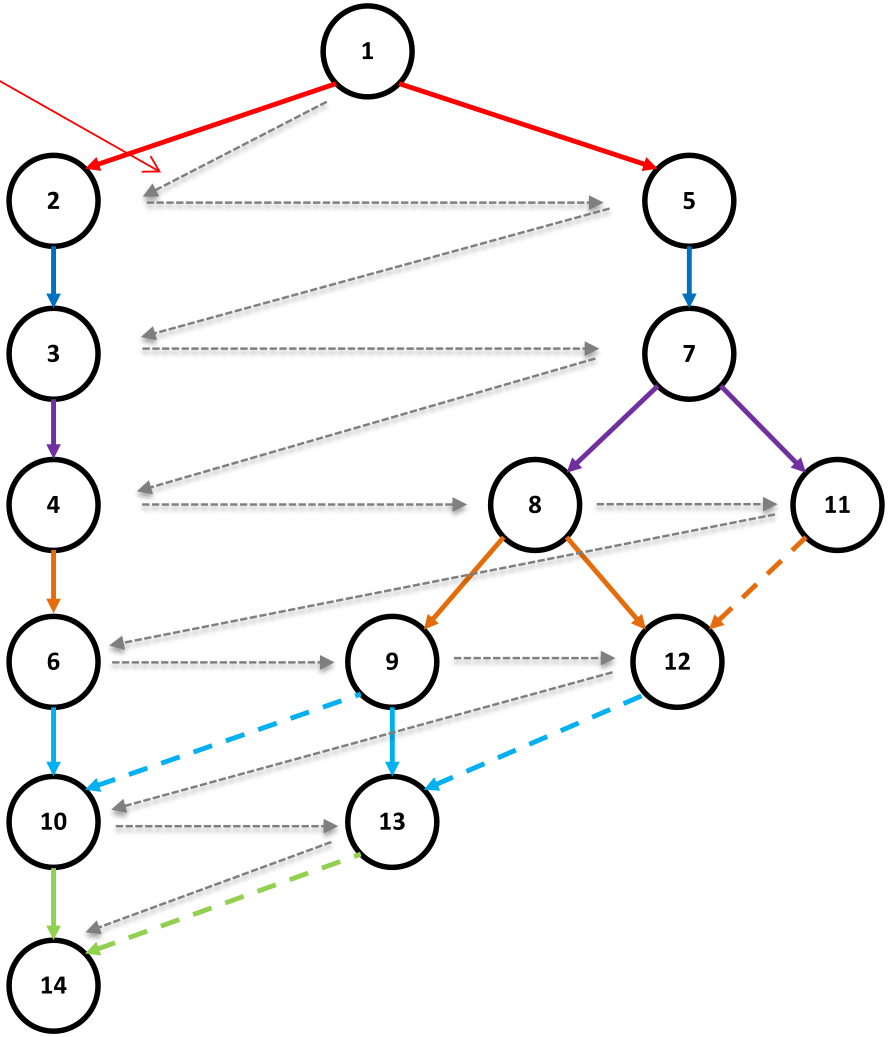




Breadth-First Search

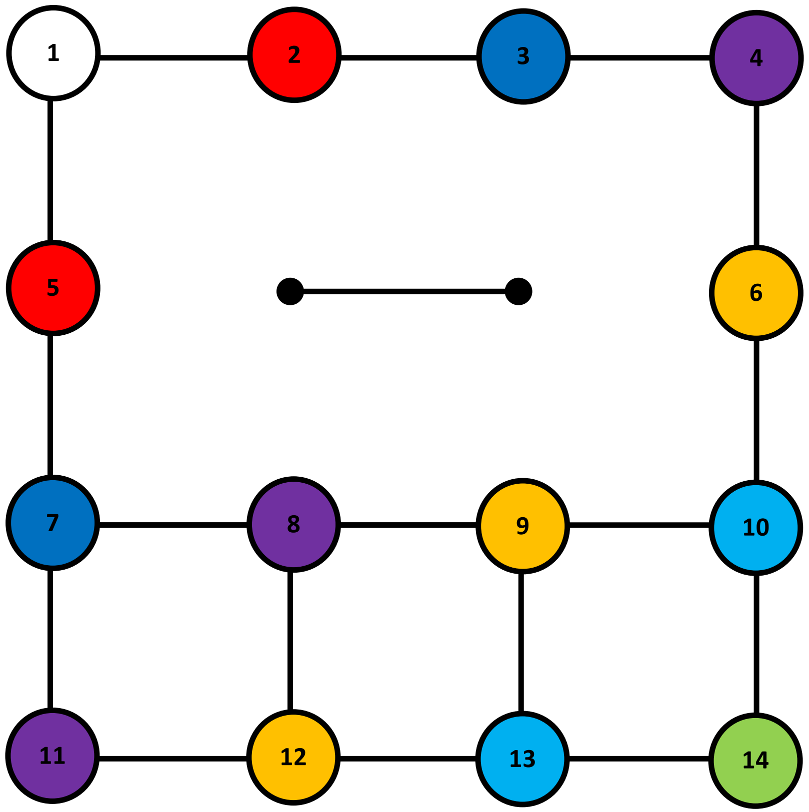
This ordering of vertices at each level in the traversal results from visiting the children vertices in ascending order of their label, i.e., 2 then 5, 8 then 11





Breadth-First Search

The traversal tree
defines the shortest path from the root
to every other node in the tree
(and, hence, from the start vertex
to every vertex in the graph)



The BFS visits vertices
closest to the start vertex first,

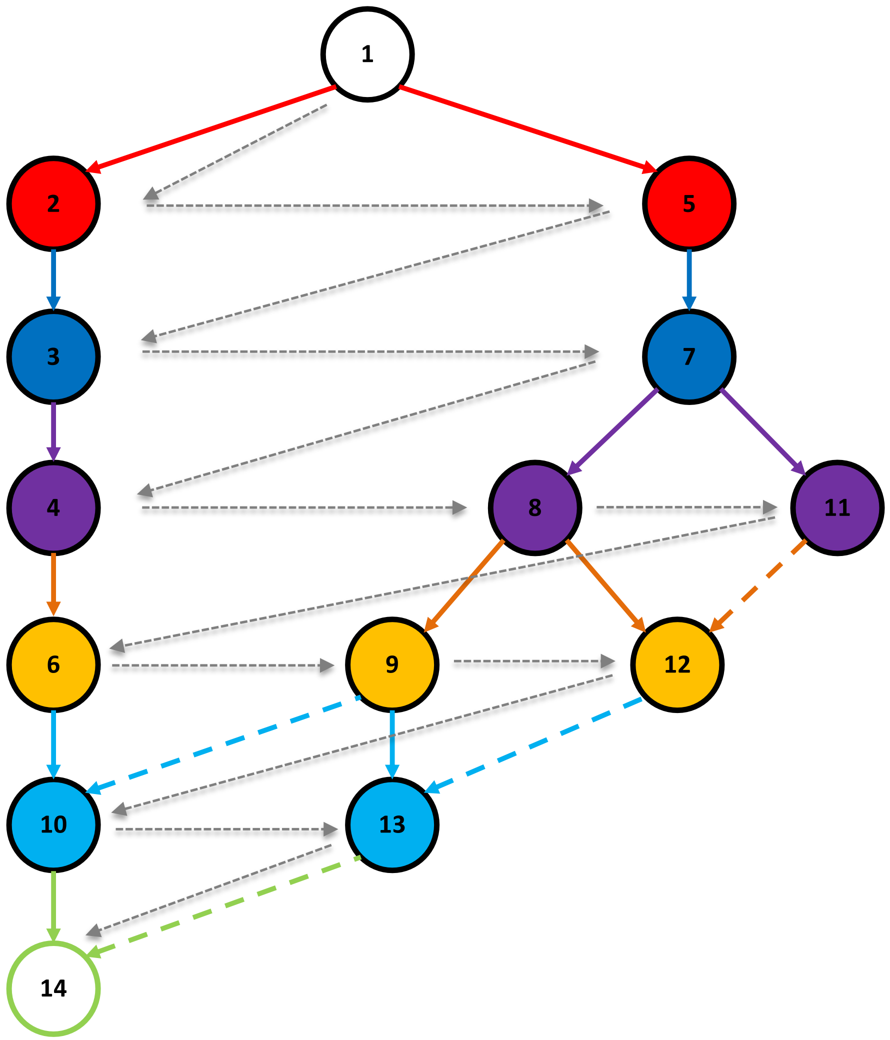
then the next closest,

then the next closest,

then the next closest,

then the next closest, ...

eventually visiting the vertices
that are furthest
from the start vertex

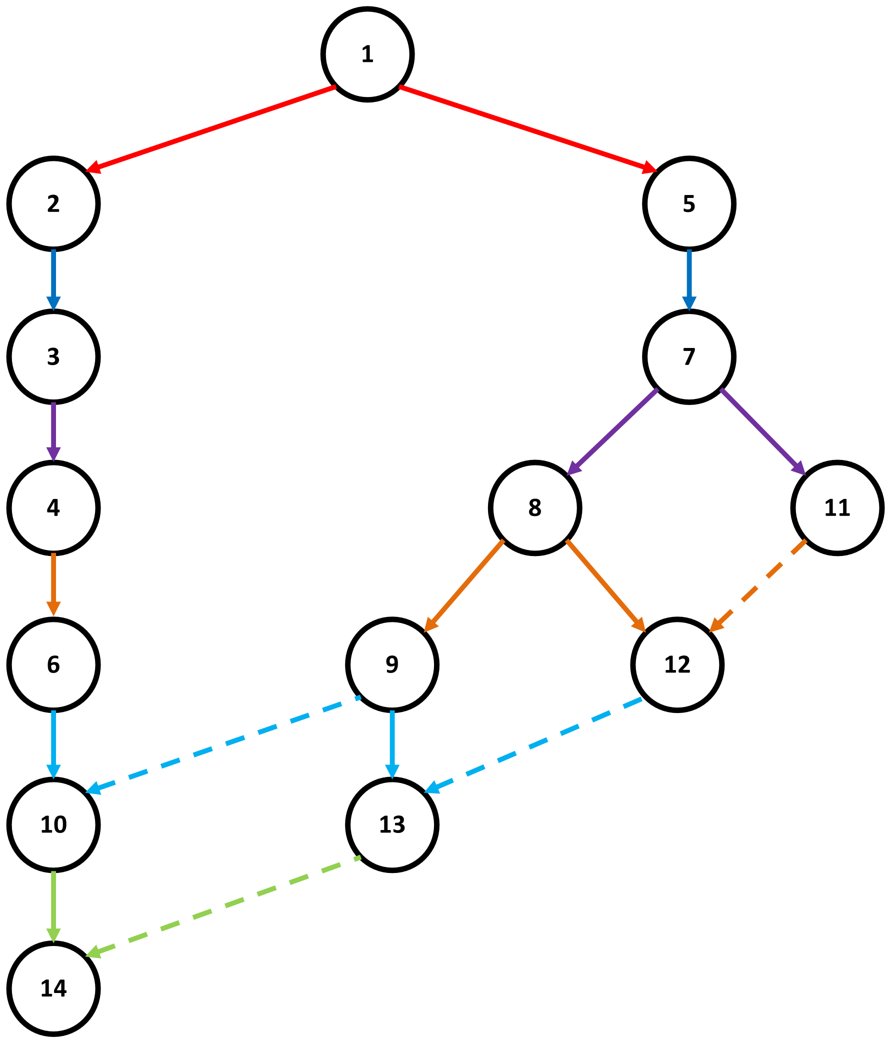


Breadth-First Search

Construct a parent array during the BFS

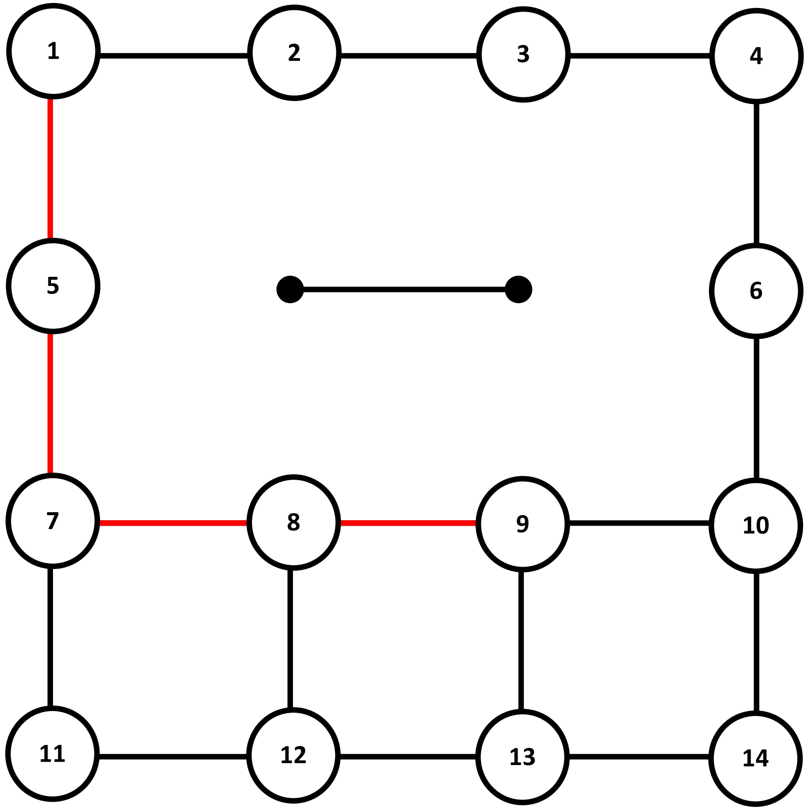
Parent	-1	1	2	3	1	4	5	7	8	6	7	8	9	10
Vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14

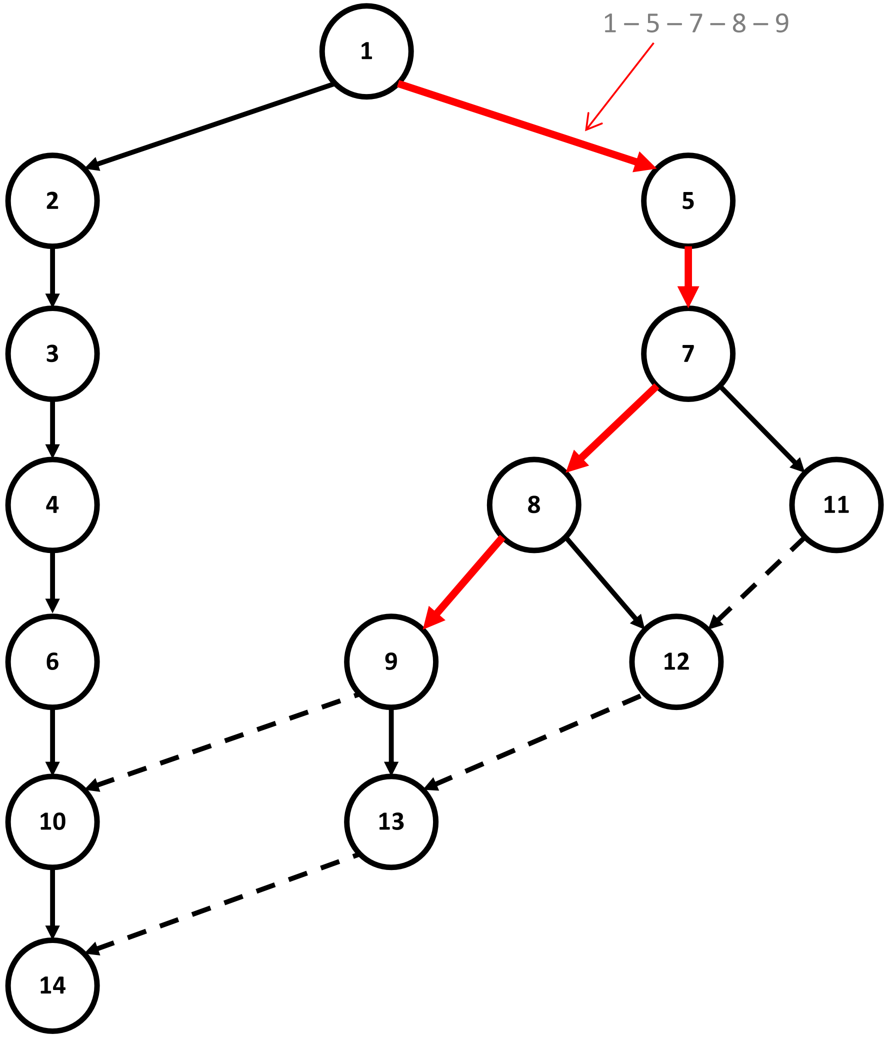
- This allows the shortest path from the start vertex to any other vertex to be determined
- Begin at the goal vertex and follow the parents back to the start vertex
- Reverse the order of vertices to specify the path from start vertex to goal vertex



Breadth-First Search

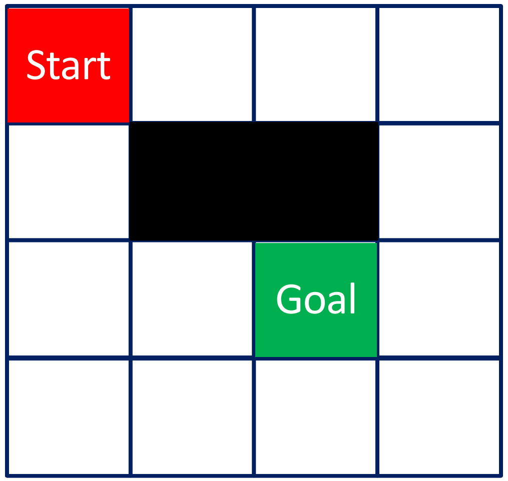
Shortest path from
vertex 1 to vertex 9:
1 – 5 – 7 – 8 – 9

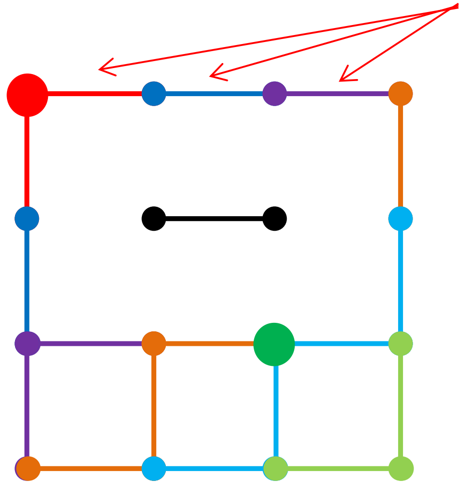




Breadth-First Search

BFS from the start position



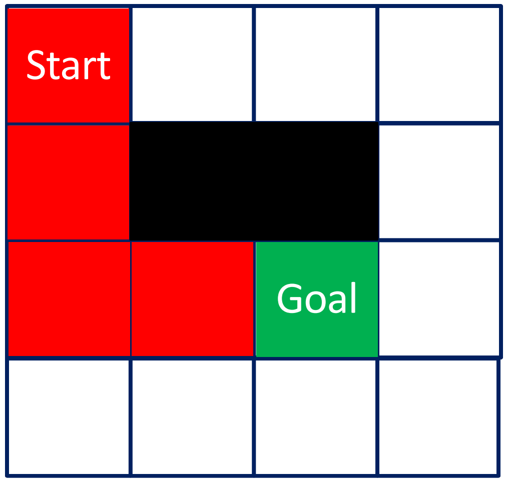


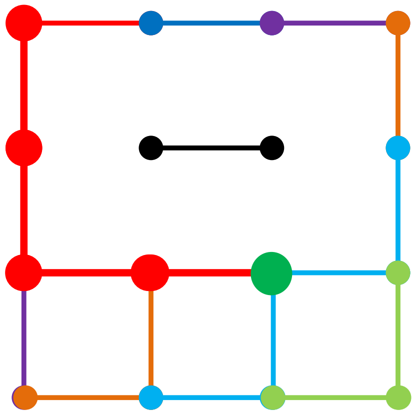
The colours indicate
the depth in the BFS traversal

Breadth-First Search

Use the parent array to reconstruct the shortest path

- Begin at the goal vertex and follow the parents back to the start vertex
- Reverse the order of vertices to specify the part from start vertex to goal vertex

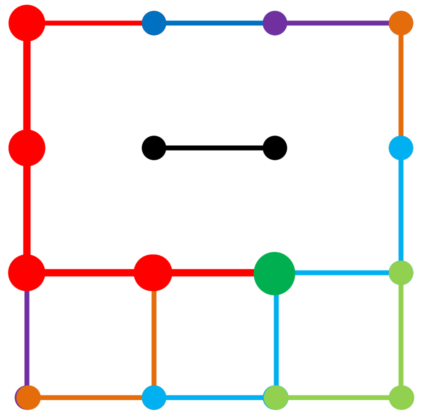




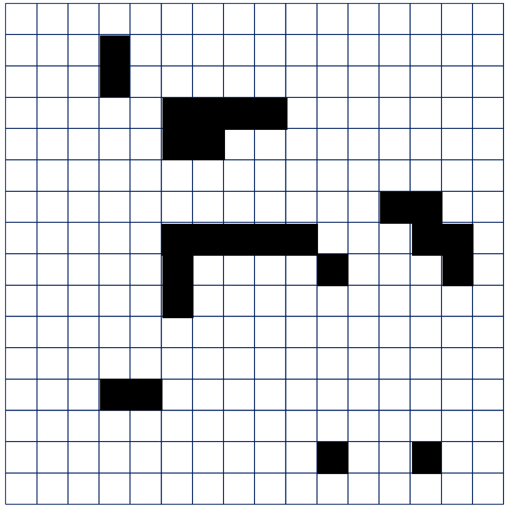
Breadth-First Search

If required, mark the path from the robot start position to the goal position on the occupancy grid (value = 2)

2	0	0	0
2	1	1	0
2	2	2	0
0	0	0	0

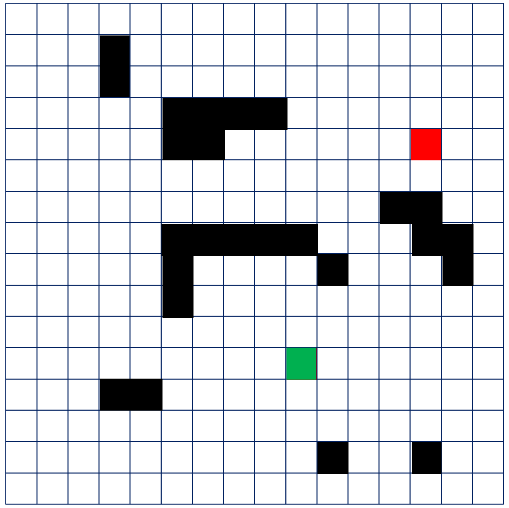


Breadth-First Search



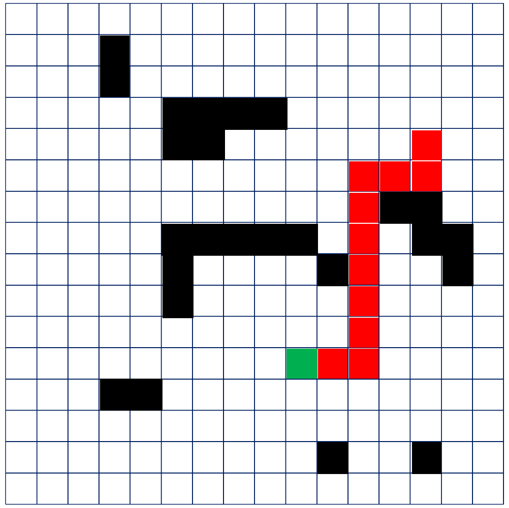
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0
0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	1	0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

Breadth-First Search



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

Breadth-First Search



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	0	0	0	2	2	2	0	0
0	0	0	0	0	0	0	0	0	0	0	2	1	1	0	0
0	0	0	0	0	1	1	1	1	1	0	2	0	1	1	0
0	0	0	0	0	1	0	0	0	0	1	2	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0	0	2	2	2	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Traversing a Graph

- Visit every vertex and edge in a systematic way

- Key idea: mark each vertex when we first visit it &
  keep track of what we have not yet completely explored

- Each vertex will exist in one of three states
  1. Undiscovered – the vertex is in its initial untouched state
  2. Discovered – the vertex has been found, but we have not yet processed all its edges
  3. Processed – the vertex after we have visited all its edges

Traversing a Graph

- Keep a record of all the vertices discovered but not yet completely processed

- Begin with a starting vertex

You have to decide where to start
or be told where to start

- Explore each vertex
  - Evaluate each edge leaving it
  - If the edge goes to an undiscovered vertex
    - Mark it discovered
    - Add it to the list of work to do
  - If the edge goes to a processed vertex, ignore it
  - If the edge goes to a discovered unprocessed vertex, ignore it

Traversing a Graph

- There are two primary graph traversal algorithms
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- The difference is the order in which they explore vertices

Traversing a Graph

The order depends completely on the container data structure used to store the discovered but not processed vertices

- BFS uses a **queue**
  - By storing the vertices in a FIFO queue, we explore the **oldest** unexplored vertices first
  - Thus explorations radiate out slowly from the starting vertex

This is the key attribute for computing shortest paths

- DFS uses a **stack**
  - By storing the vertices in a LIFO stack, we explore the vertices by diving down a path, visiting a new neighbour if one is available, and backing up only when we are surrounded by (i.e., connected by edges to) previously discovered vertices
  - Thus explorations quickly wander away from out starting vertex

Breadth-First Search

- Assign a direction to each edge,
  from discoverer vertex u to discovered vertex v

- Since each node has exactly one parent, except for the root (i.e., start vertex), this defines a tree on the vertices of the graph

- This tree defines the shortest path from the root to every other node in the tree

Breadth-First Search

BFS(G, s)

for each vertex $u \in V[G] - \{s\}$ do

    $state[u]$ = "undiscovered"

    $p[u] = nil$, i.e. no parent is in the BFS tree

$state[s]$ = "discovered"

$p[s] = nil$

$Q = \{s\}$

while $Q \neq \emptyset$ do

    u = dequeue$[Q]$

    process vertex u as desired

    for each $v \in Adj[u]$ do

        process edge (u, v) as desired

        if $state[v]$ = "undiscovered" then

            $state[v]$ = "discovered"

            $p[v] = u$

            enqueue$[Q, v]$

    $state[u]$ = "processed"

S. Skiena, The Algorithm Design Manual, Springer 2010

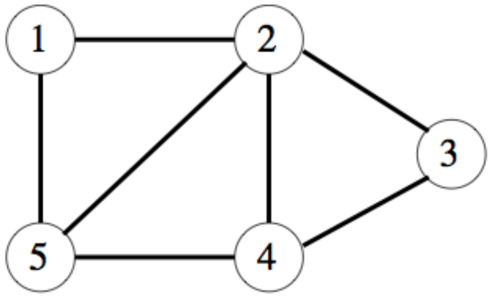
Graphs

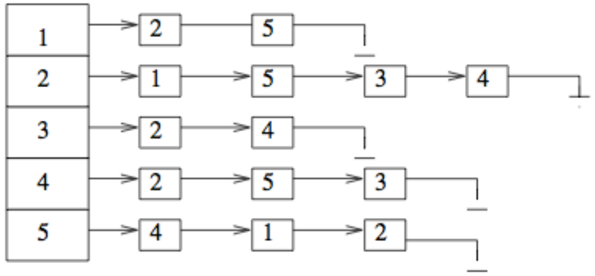
Assuming a graph $G = (V, E)$ with n vertices and m edges, there are two basic choices for data structures

- Adjacency Matrix: an $n \times n$ matrix M, where element $M[i, j] = 1$ if (i, j) is an edge of G, and 0 if it isn't (or, alternatively $M[i, j] = w$, the weight of the edge)

- Adjacency List: a linked list that stores the neighbours that are adjacent to each vertex

	1	2	3	4	5
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





Graphs

```
/* Adjacency list representation of a graph of degree MAXV          */
/*                                                                   */
/* Directed edge (x, y) is represented by edgenode y in x's         */
/* adjacency list. Vertices are numbered 1 .. MAXV                   */

#define MAXV 1000  /* maximum number of vertices */

typedef struct {
   int y;                       /* adjacent vertex number          */
   int weight;                  /* edge weight, if any             */
   struct edgenode *next;       /* next edge in list               */
} edgenode;

typedef struct {
        edgenode *edges[MAXV+1]; /* adjacency info: list of edges   */
        int degree[MAXV+1];     /* number of edges for each vertex */
        int nvertices;          /* number of vertices in graph     */
        int nedges;             /* number of edges in graph        */
        bool directed;          /* is the graph directed?          */
} graph;
```

S. Skiena, The Algorithm Design Manual, Springer 2010

Graphs

```
/* Initialize graph from data in a file                                    */

initialize_graph(graph *g, bool directed){

   int i;                             /* counter */

   g->nvertices = 0;
   g->nedges = 0;
   g->directed = directed;

   for (i=1; i<=MAXV; i++)
      g->degree[i] = 0;

   for (i=1; i<=MAXV; i++)
      g->edges[i] = NULL;
}
```

S. Skiena, The Algorithm Design Manual, Springer 2010

directed	nvertices	nedges
false	0	0

	degree	edges
0		
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	
9	0	
10	0	
11	0	
12	0	
13	0	
14	0	

Graphs

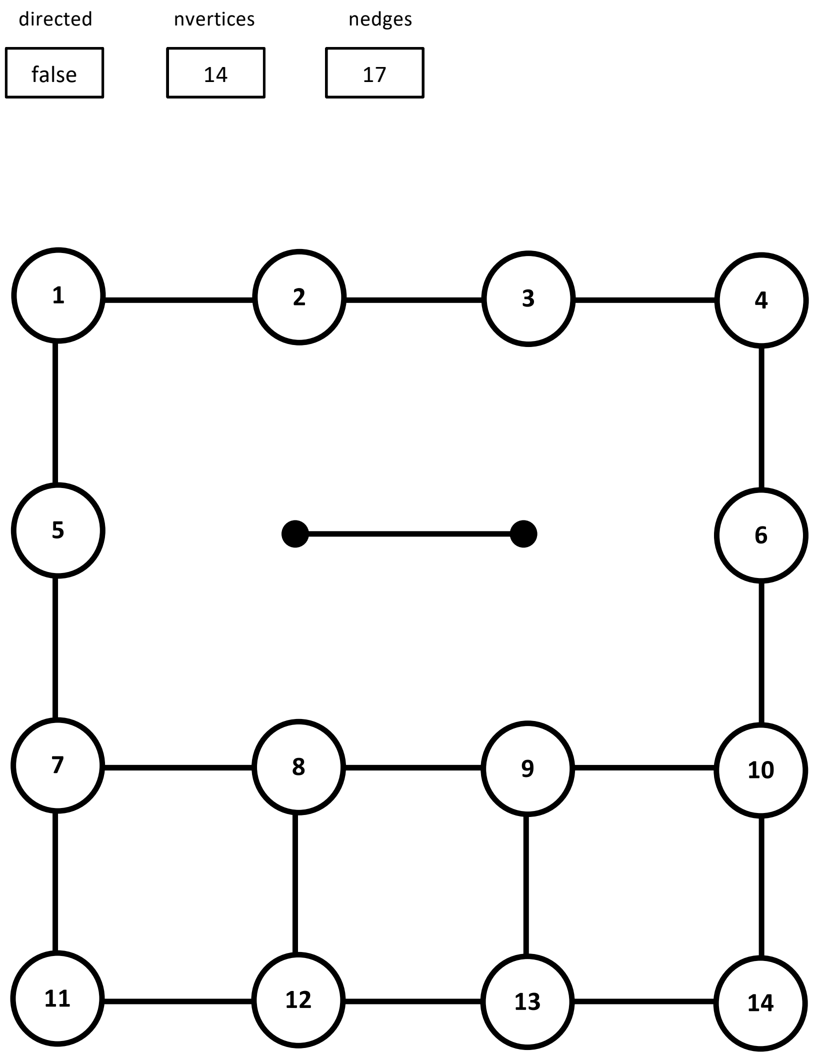
```
/* build graph from data */

read_graph(graph *g, bool directed) {

   int i;    /* counter                  */
   int m;    /* number of edges          */
   int x, y; /* vertices in edge (x,y)   */

   initialize_graph(g, directed);

   scanf("%d %d",&(g->nvertices),&m);

   for (i=1; i<=m; i++) {
      scanf("%d %d",&x,&y);
      insert_edge(g,x,y,directed);
   }
}
```

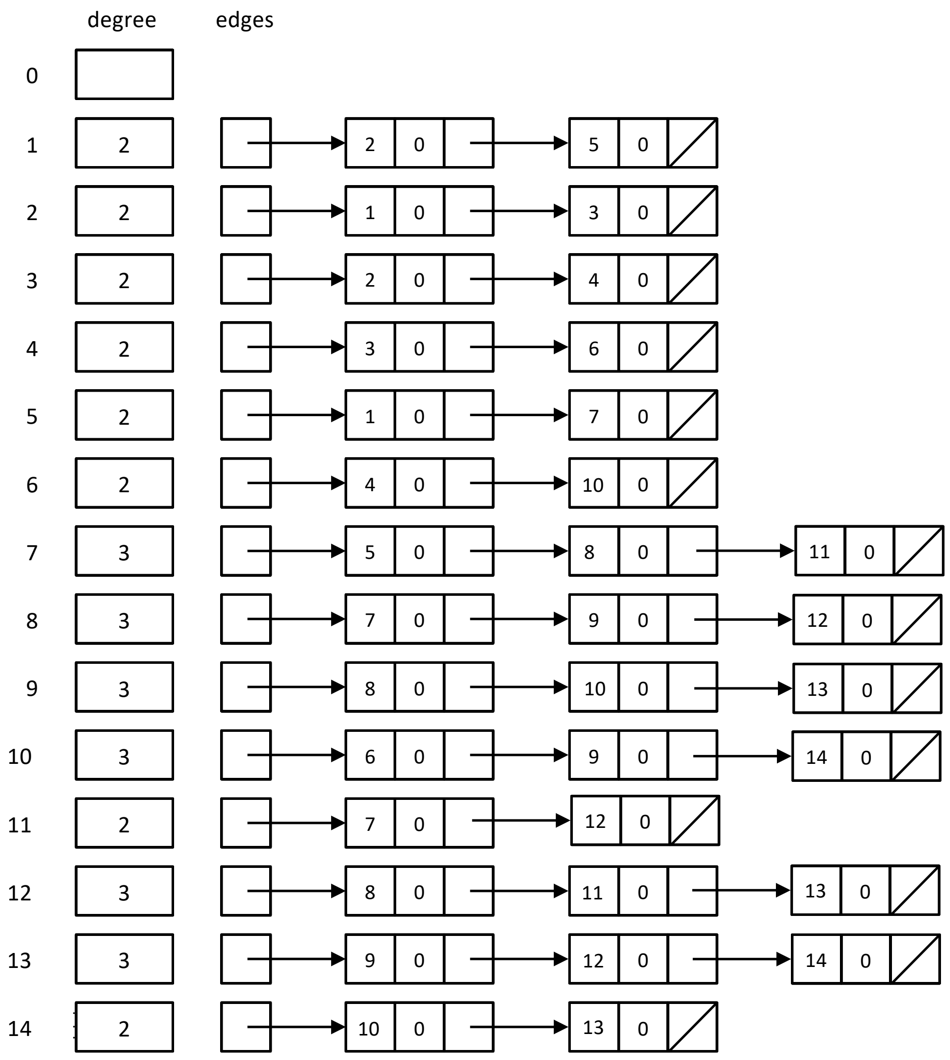
S. Skiena, The Algorithm Design Manual, Springer 2010

Graphs

```
/* Initialize graph from data in a file                          */

insert_edge(graph *g, int x, int y, bool directed) {

   edgenode *p;                  /* temporary pointer */

   p = malloc(sizeof(edgenode)); /* allocate edgenode storage     */

   p->weight = 0;
   p->y = y;
   p->next = g->edges[x];        /* edge node points to the       */
                                 /* existing edge list            */
   g->edges[x] = p;              /* insert at head of list        */
   g->degree[x] ++;

   if (directed == false)        /* NB: if undirected add         */
      insert_edge(g,y,x,true);   /* the reverse edge recursively  */
   else                          /* but directed TRUE so we do it */
      g->nedges ++;              /* only once                     */
}
```

S. Skiena, The Algorithm Design Manual, Springer 2010


directed
nvertices
nedges
false
14
17
1
2
3
4
5
6
7
8
9
10
11
12
13
14


degree
edges
0
1 2 2 0 5 0
2 2 1 0 3 0
3 2 2 0 4 0
4 2 3 0 6 0
5 2 1 0 7 0
6 2 4 0 10 0
7 3 5 0 8 0 11 0
8 3 7 0 9 0 12 0
9 3 8 0 10 0 13 0
10 3 6 0 9 0 14 0
11 2 7 0 12 0
12 3 8 0 11 0 13 0
13 3 9 0 12 0 14 0
14 2 10 0 13 0

Graphs

```
/* Print a graph                                                        */

print_graph(graph *g) {

   int i;                                  /* counter           */
   edgenode *p;                            /* temporary pointer */

   for (i=1; i<=g->nvertices; i++) {
      printf("%d: ",i);
      p = g->edges[i];
      while (p != NULL) {
         printf(" %d",p->y);
         p = p->next;
      }
      printf("\n");
   }
}
```

S. Skiena, The Algorithm Design Manual, Springer 2010

Breadth-First Search

```
/* Breadth-First Search                                          */

bool processed[MAXV+1];   /* which vertices have been processed  */
bool discovered[MAXV+1];  /* which vertices have been found      */
int parent[MAXV+1];       /* discovery relation                  */


/* Each vertex is initialized as undiscovered:                   */

initialize_search(graph *g){

   int i;                            /* counter */

   for (i=1; i<=g->nvertices; i++) {
      processed[i] = discovered[i] = false;
      parent[i] = -1;
   }
}
```

BFS(G, s)
- for each vertex $u \in V[G] - \{s\}$ do
  - $state[u]$ = "undiscovered"
  - $p[u] = nil$, i.e. no parent is in the BFS tree
- $state[s]$ = "discovered"
- $p[s] = nil$
- $Q = \{s\}$
- while $Q \neq \emptyset$ do
  - $u = \text{dequeue}[Q]$
  - process vertex u as desired
  - for each $v \in Adj[u]$ do
    - process edge (u, v) as desired
    - if $state[v]$ = "undiscovered" then
      - $state[v]$ = "discovered"
      - $p[v] = u$
      - enqueue$[Q, v]$
  - $state[u]$ = "processed"

S. Skiena, The Algorithm Design Manual, Springer 2010

directed	nvertices	nedges
false	14	17

	processed	discovered	parent	degree	edges
0					
1	false	false	-1	2	→ 2 0 → 5 0
2	false	false	-1	2	→ 1 0 → 3 0
3	false	false	-1	2	→ 2 0 → 4 0
4	false	false	-1	2	→ 3 0 → 6 0
5	false	false	-1	2	→ 1 0 → 7 0
6	false	false	-1	2	→ 4 0 → 10 0
7	false	false	-1	3	→ 5 0 → 8 0 → 11 0
8	false	false	-1	3	→ 7 0 → 9 0 → 12 0
9	false	false	-1	3	→ 8 0 → 10 0 → 13 0
10	false	false	-1	3	→ 6 0 → 9 0 → 14 0
11	false	false	-1	2	→ 7 0 → 12 0
12	false	false	-1	3	→ 8 0 → 11 0 → 13 0
13	false	false	-1	3	→ 9 0 → 12 0 → 14 0
14	false	false	-1	2	→ 10 0 → 13 0

Breadth-First Search

```
/* Once a vertex is discovered, it is placed in a queue.           */
/* Since we process these vertices in first-in, first-out order,   */
/* the oldest vertices are expanded first, which are exactly those */
/* closest to the root                                             */

bfs(graph *g, int start)
{
   queue q;                    /* queue of vertices to visit */
   int v;                      /* current vertex             */
   int y;                      /* successor vertex           */
   edgenode *p;                /* temporary pointer          */

   init_queue(&q);
   enqueue(&q,start);
   discovered[start] = true;
```

BFS(G, s)
for each vertex $u \in V[G] - \{s\}$ do
  $state[u]$ = "undiscovered"
  $p[u] = nil$, i.e. no parent is in the BFS tree
$state[s]$ = "discovered"
$p[s] = nil$
$Q = \{s\}$
while $Q \neq \emptyset$ do
  u = dequeue$[Q]$
  process vertex u as desired
  for each $v \in Adj[u]$ do
    process edge (u, v) as desired
    if $state[v]$ = "undiscovered" then
      $state[v]$ = "discovered"
      $p[v] = u$
      enqueue$[Q, v]$
  $state[u]$ = "processed"

S. Skiena, The Algorithm Design Manual, Springer 2010

Breadth-First Search

```
  while (empty_queue(&q) == FALSE) {
     v = dequeue(&q);
     process_vertex_early(v);
     processed[v] = TRUE;
     p = g->edges[v];

     while (p != NULL) {
        y = p->y;
        if ((processed[y] == FALSE) || g->directed)
           process_edge(v,y);
        if (discovered[y] == FALSE) {
           enqueue(&q,y);
           discovered[y] = TRUE;
           parent[y] = v;
        }
        p = p->next;
     }
     process_vertex_late(v);
  }
}
```

BFS(G, s)
- for each vertex $u \in V[G] - \{s\}$ do
  - $state[u]$ = "undiscovered"
  - $p[u] = nil$, i.e. no parent is in the BFS tree
- $state[s]$ = "discovered"
- $p[s] = nil$
- $Q = \{s\}$
- while $Q \neq \emptyset$ do
  - $u = \text{dequeue}[Q]$
  - process vertex u as desired
  - for each $v \in Adj[u]$ do
    - process edge (u, v) as desired
    - if $state[v]$ = "undiscovered" then
      - $state[v]$ = "discovered"
      - $p[v] = u$
      - enqueue$[Q, v]$
  - $state[u]$ = "processed"

S. Skiena, The Algorithm Design Manual, Springer 2010

Breadth-First Search

```
/* The exact behaviour of bfs depends on the functions              */
/*    process vertex early()                                        */
/*    process vertex late()                                         */
/*    process edge()                                                */
/* These functions allow us to customize what the traversal does   */
/* as it makes its official visit to each edge and each vertex.    */
/* Here, e.g., we will do all of vertex processing on entry        */
/* (to print each vertex and edge exactly once)                     */
/* so process vertex late() returns without action                 */

process_vertex_late(int v) {
}

process_vertex_early(int v){
   printf("processed vertex %d\n",v);
}

process_edge(int x, int y) {
   printf("processed edge (%d,%d)\n",x,y);
}
```

S. Skiena, The Algorithm Design Manual, Springer 2010

Breadth-First Search

```
/* this version just counts the number of edges                    */

process_edge(int x, int y) {
   nedges = nedges + 1;
}
```

S. Skiena, The Algorithm Design Manual, Springer 2010

Breadth-First Search

Finding Paths

- The `parent` array in `bfs()` is necessary to find the shortest paths through a graph
- The vertex that discovered vertex `i` is defined as `parent[i]`

Breadth-First Search

Finding Paths

- Every vertex is discovered during the course of a traversal so every node has a parent (except the root)
- The parent relation defines a tree of discovery with the initial search node as the root of the tree
- Because vertices are discovered in order of increasing distance from the root, this tree has a very important property
  - The unique tree path from the root to each node uses the smallest number of edges (and intermediate nodes) possible on any path from the root to that vertex
  - This is why the BFS can be used to find shortest paths in an **unweighted** graph

Breadth-First Search

Finding Paths

- To reconstruct a path we follow the chain of ancestors from the destination node x to the root
- Note we have to work backwards (we only know the parents)
- We find the path from the target vertex to the root and
  - Either store it and explicitly reverse it using a stack
  - Or construct the path recursively (in which case the stack is implicit)

Breadth-First Search

```
bool find_path(int start, int end, int parents[]) {

   bool is_path;

   if (end == -1) {
      is_path = false; // some vertex on the path back from the end
                       // has no parent (not counting start)
   }
   else if ((start == end)) {
       printf("\n%d",start); // or store start in a path DS
       is_path = true;       // we have reached the start vertex
   }
   else {
      is_path = find_path(start,parents[end],parents);
      printf(" %d",end);     // or store end in a path data structure
   }
   return(is_path);
}
```

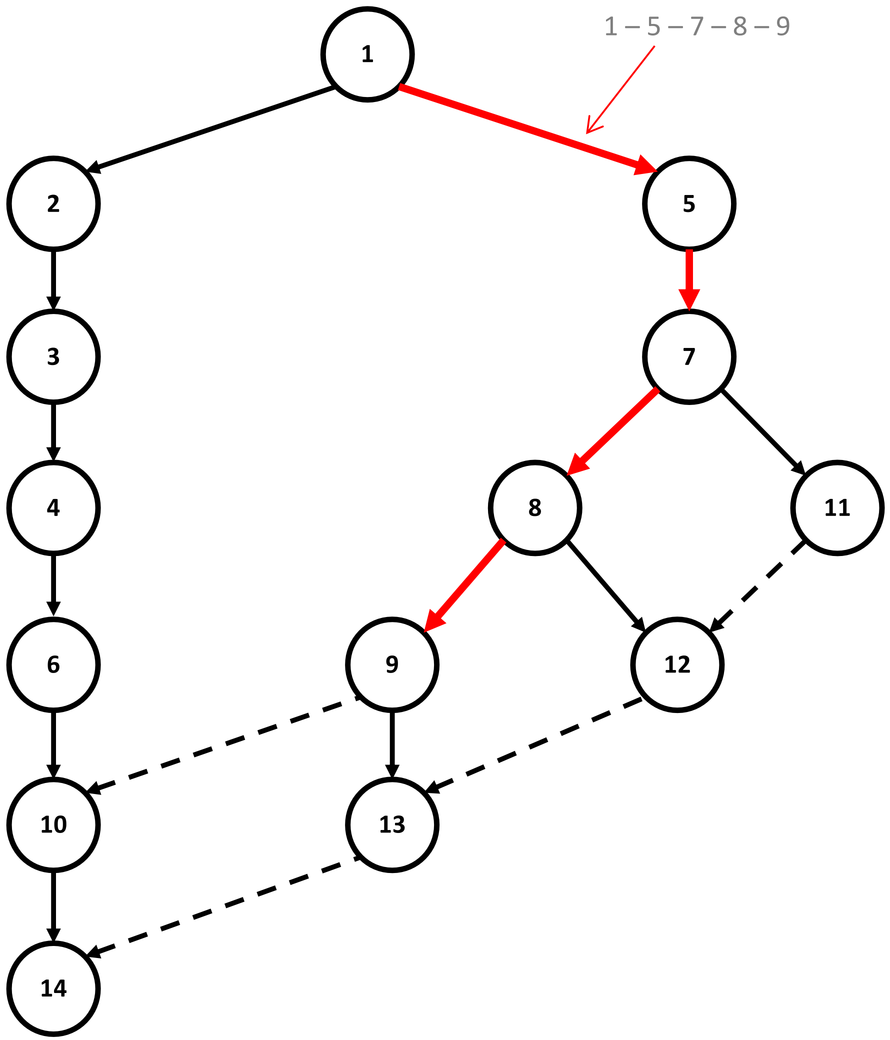
Recursive call

Breadth-First Search

Parent	-1	1	2	3	1	4	5	7	8	6	7	8	9	10
Vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
find_path(1,9,parent)

-> find_path(1,8,parent)

    -> find_path(1,7,parent)

        -> find_path(1,5,parent)

            -> find_path(1,1,parent)

                -> printf(1)
               printf(5)
           printf(7)
       printf(8)
   printf(9)
```

Shortest path from vertex 1 to vertex 9: 1 – 5 – 7 – 8 – 9



Breadth-First Search

This ordering of vertices at each level in the traversal results from visiting the children vertices in ascending order of their label, i.e. 2 then 5, 8 then 11. This means we always favour travelling is one direction.

Choose randomly if you would prefer a more zig-zag path (the path length remains the same)