Video Games Technologies

11498: MSc in Computer Science and Engineering
11156: MSc in Game Design and Development

Chap. 11 — Pathfinding

Pathfinding

Partially based on slides of the course:
CS134: Computer Game Design and Programming
Author: Soon Tee Teoh

Outline

…:

- Introduction
- Definitions: pathfinder, graph
- Graph traversal algorithms in games
- BFS, DFS, Dijkstra, and A*

Introduction

Why do we need pathfinding in games?

- In computer games, it is necessary for characters to make decisions about how to get from A to B.
- This is easy for people, but requires intelligence to solve.
- For example, in RTS games, you click the mouse on a location on the map and the entity must figure out the most efficient way to get there:

Introduction(*contd.*)

Well, this may involve:

- Going over mountains (slower?).
- Negotiating buildings.
- Negotiating bridges (or swimming?).
- Climbing ladders

and also:

- Opening doors.
- Moving other entities out of the way.
- In computer games, this is almost always achieved using the A* algorithm!

What is a pathfinder?

- It is a *graph traversal* algorithm.
- It allows us to find the *lowest cost* path through a graph.
- *Lowest* cost does not mean necessarily the following:
  - the shortest path
  - the path with the fewest nodes

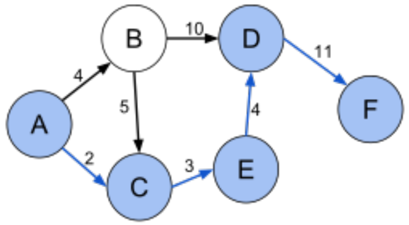
Google Maps is a big graph, whose e**dges** represent streets and ***vertices*** represent crossings.

But, *pathfinding does not work directly on geometry*. Instead, it is a simplified abstraction of movement possibilities in a graph.

What is a graph?

A graph G = (V, E)

- V = set of vertices, E = set of edges
- *Dense* graph: $|E| \approx |V|^2$; *Sparse* graph: $|E| \approx |V|$
- *Undirected graph:*
  - Edge (u,v) = Edge (v,u)
  - No self-loops
- *Directed* graph (or digraph):
  - Edge (u,v) goes from vertex u to vertex v, notated u→v
- A *weighted graph* associates weights with either the edges or the vertices



Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph

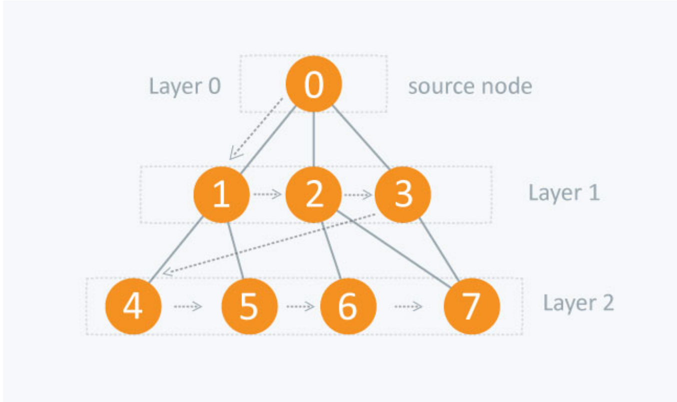
https://en.wikipedia.org/wiki/Shortest_path_problem

Graph traversal

Algorithms:

- Breath first search (BFS):
  - Least number of nodes
- Depth first search (DFS):
  - Exists!
- Dijkstra's algorithm:
  - Lowest cost path to all other nodes
- A*:
  - Lowest cost path to a destination

Breadth-first search



https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Graph traversal:

- First move horizontally and visit all the nodes of the current layer
- Move to the next layer

Remarks:

- The tree is not actually created.
- Instead, we use a queue (first-in-first-out policy) to mimic the behavior of the BFS on a tree.

A feasible BFS implementation:

- A FIFO-policy queue.
- Array of nodes; each node holds indexes of its neighbors in the array.
- Homologous array of booleans, indicating whether each node has been visited or not.

Breadth-first search: pseudocode

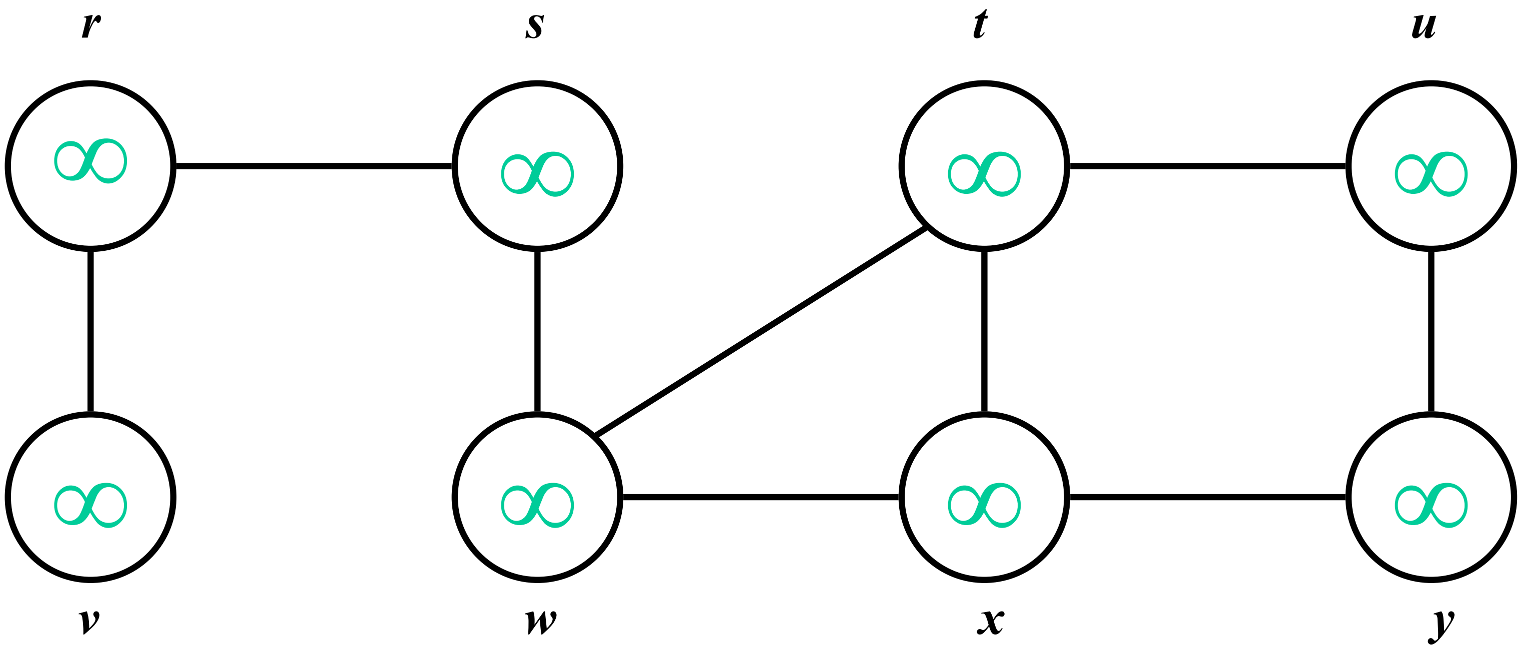
white node = unvisited node
grey node = visited node
black color = closed node (removed from the queue)

```
BFS(G, s) {
    initialize vertices;
    Q = {s};     // Q is a queue (duh); initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∊ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;       What does v->d represent?
                v->p = u;              What does v->p represent?
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

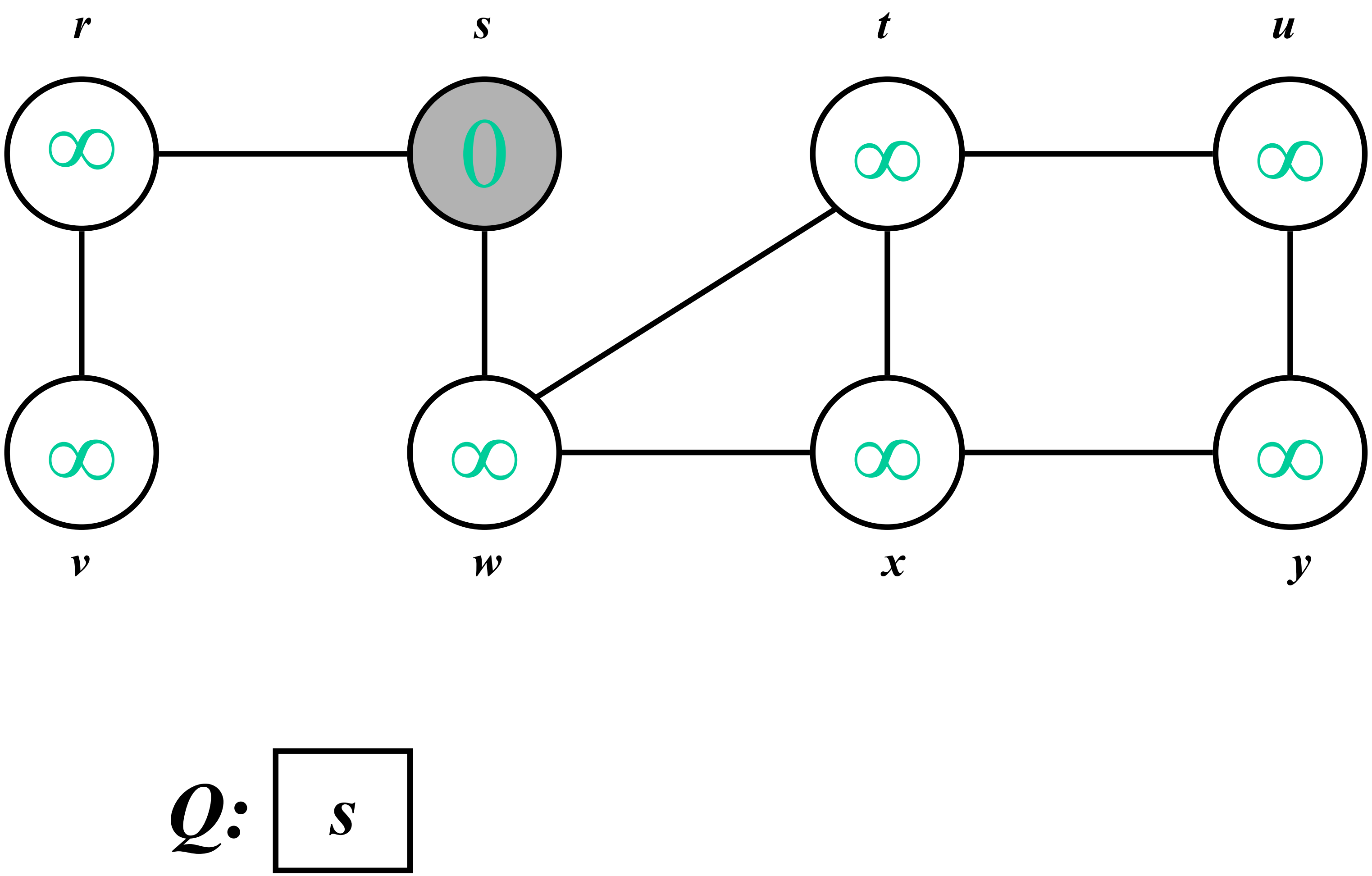
ATTENTION:

- To prevent the appearance of loops, only unvisited nodes are processed, i.e., the cost function d (e.g., degree or tree level) is computed.
- The field p denotes the node's parent.

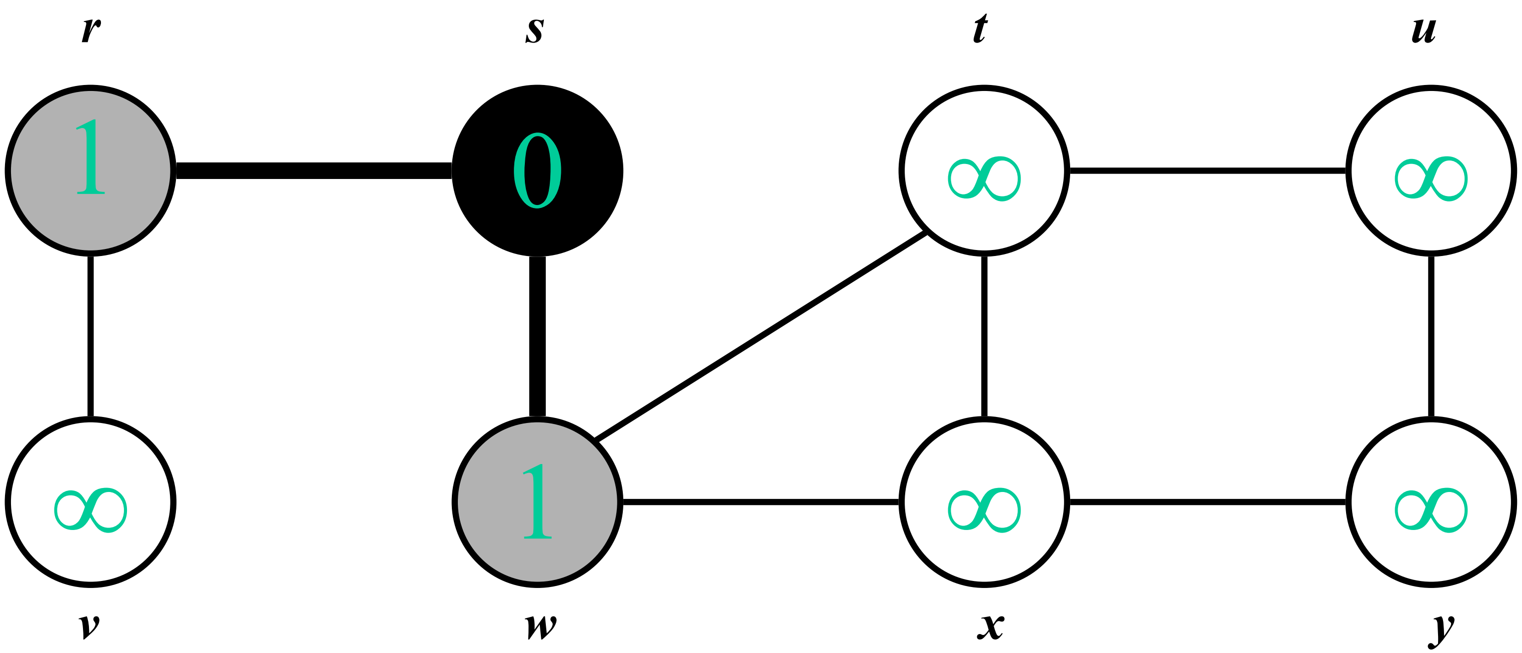
Breadth-first search: *example*



Breadth-first search: *example*



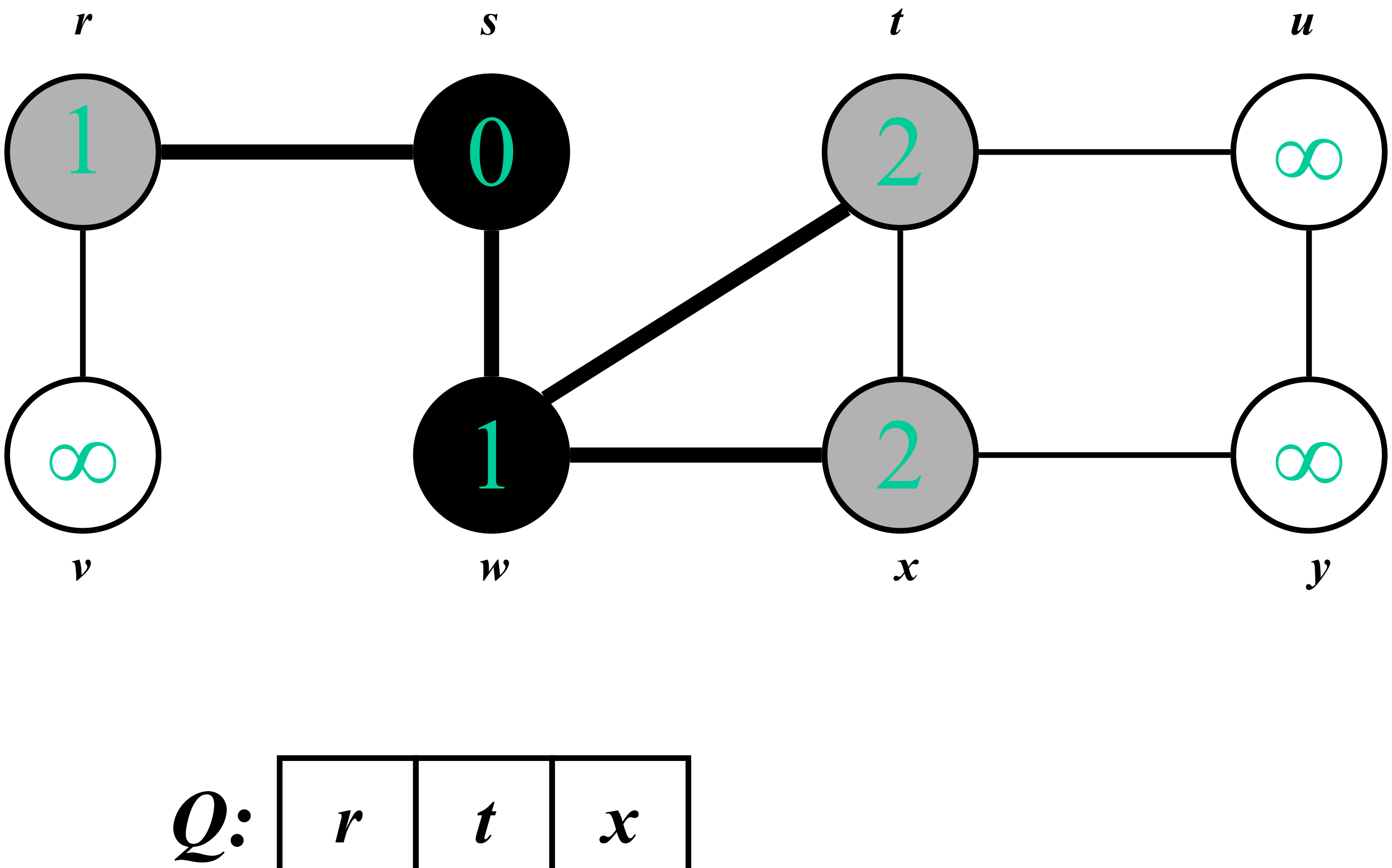
Breadth-first search: *example*



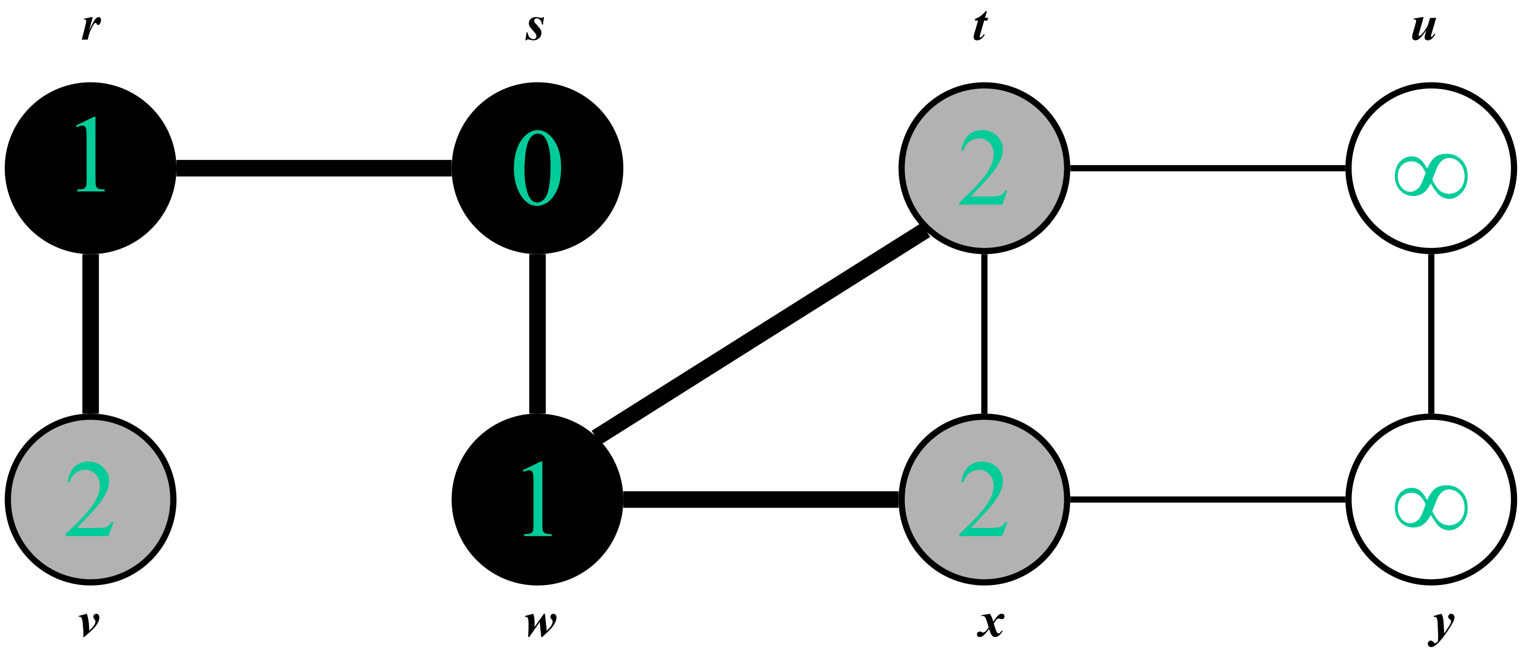
Q:

w	*r*

Breadth-first search: *example*



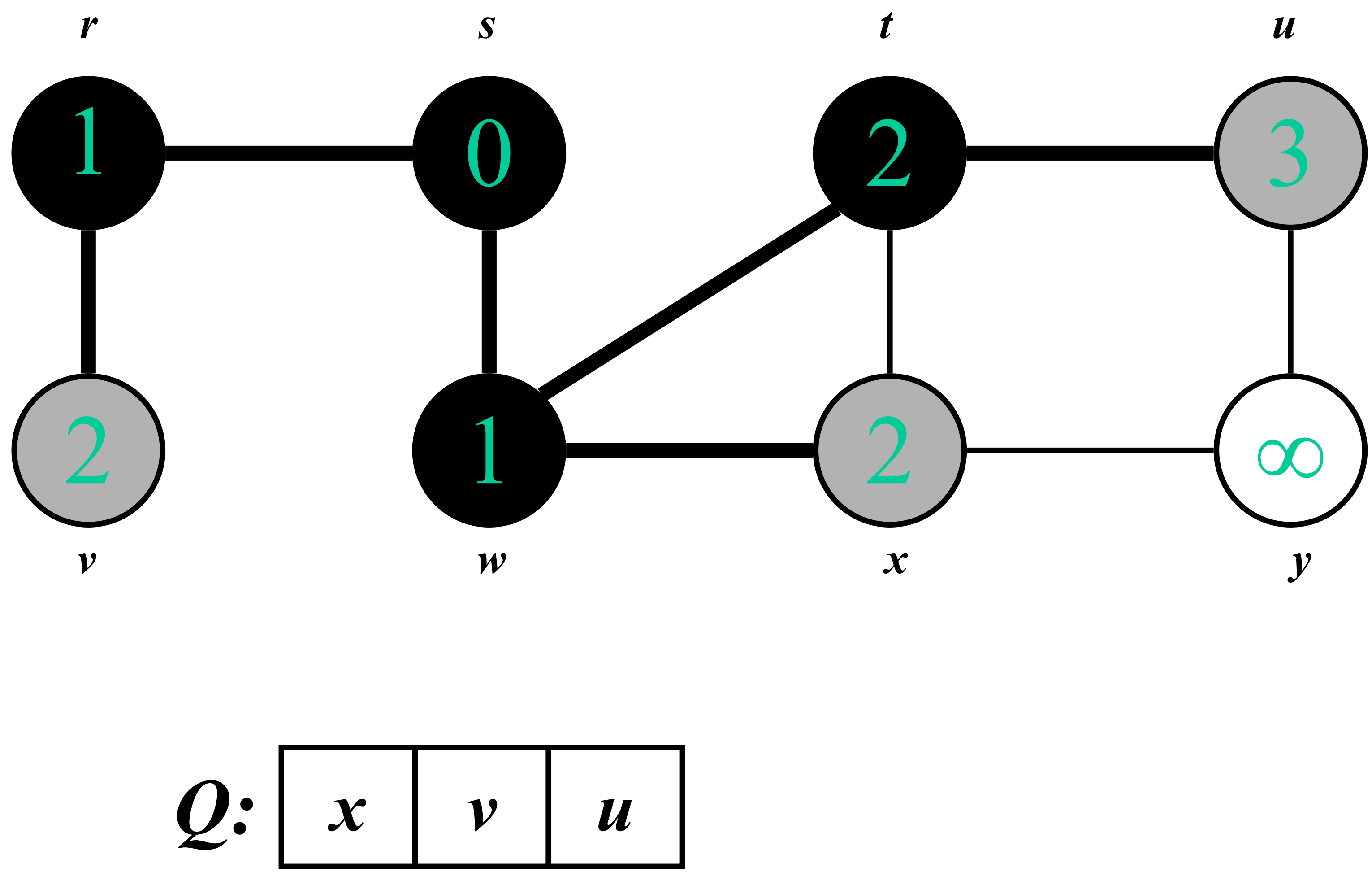
Breadth-first search: *example*



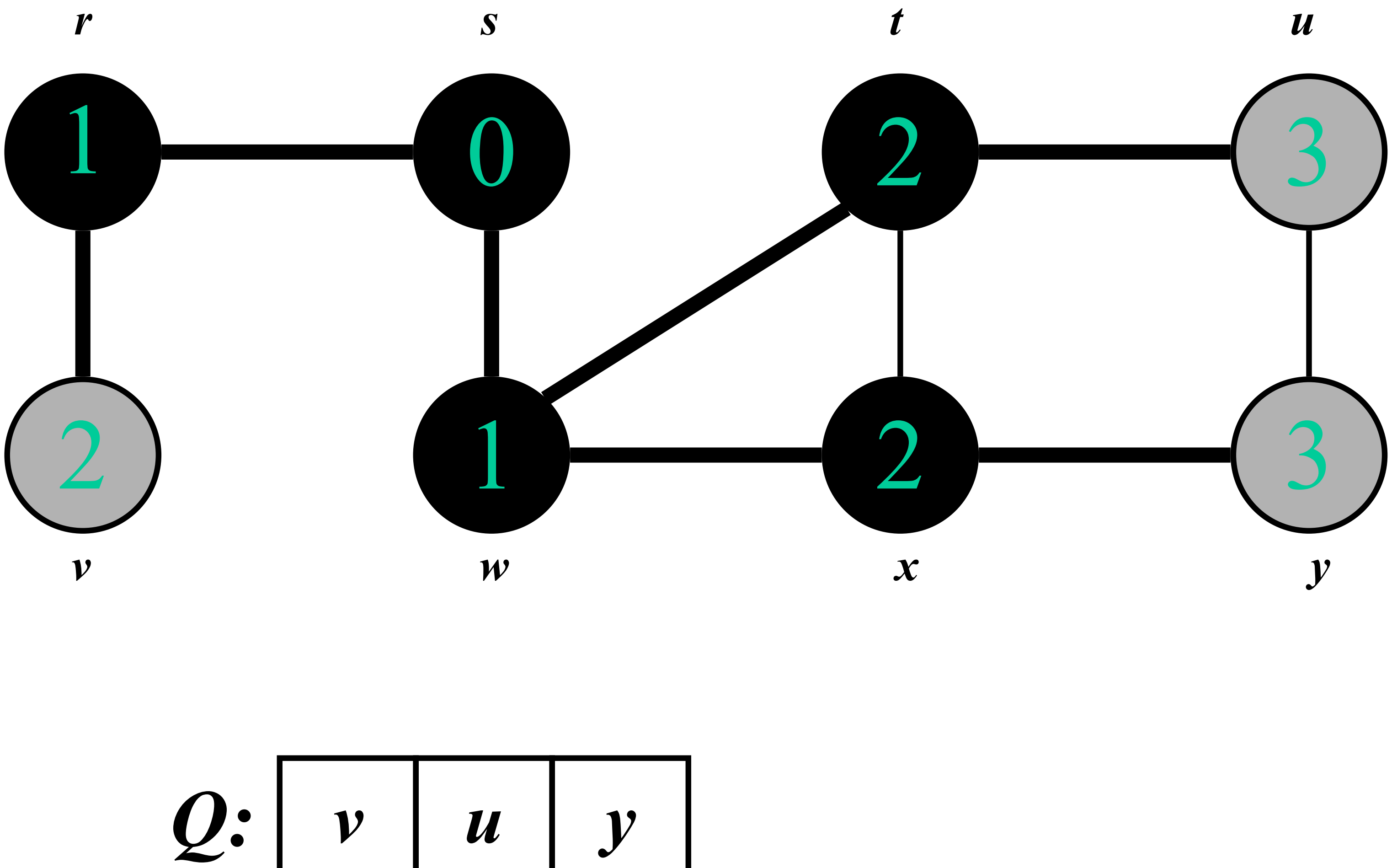
Q:

t	*x*	*v*

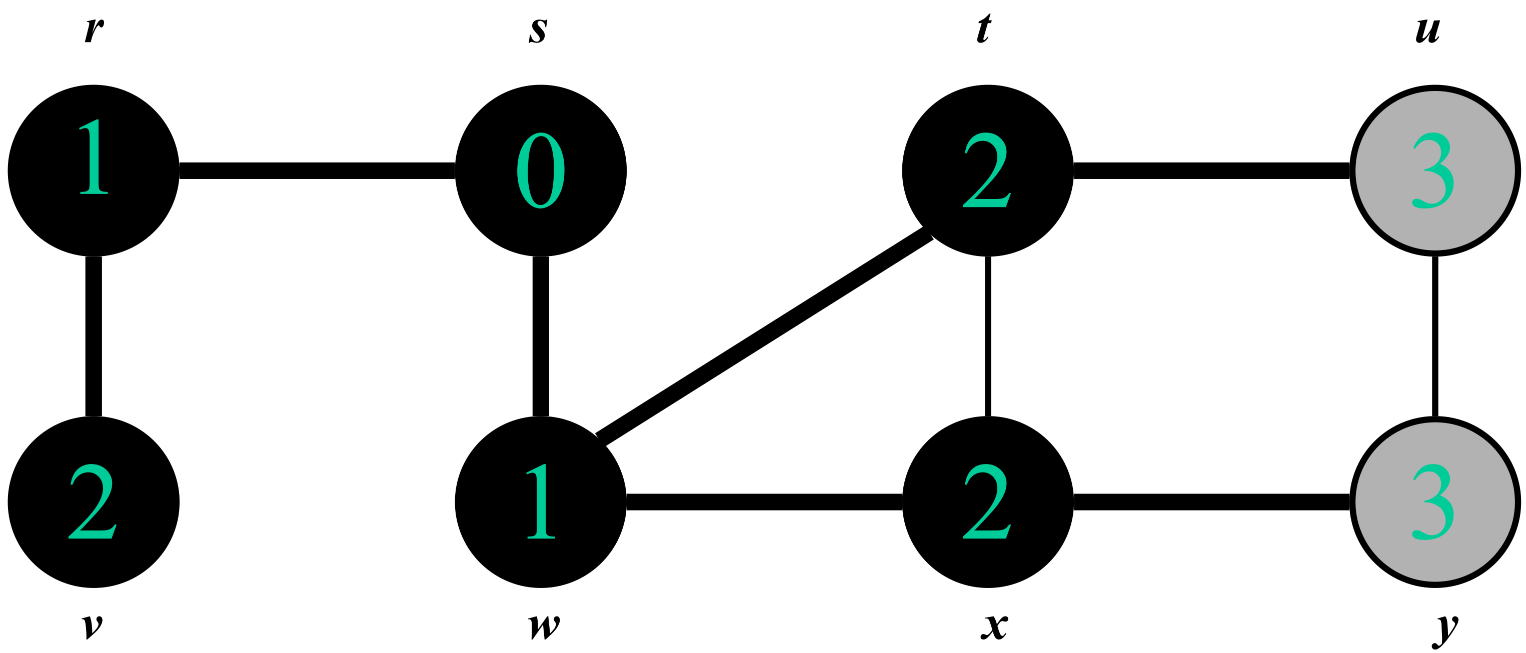
Breadth-first search: *example*



Breadth-first search: *example*

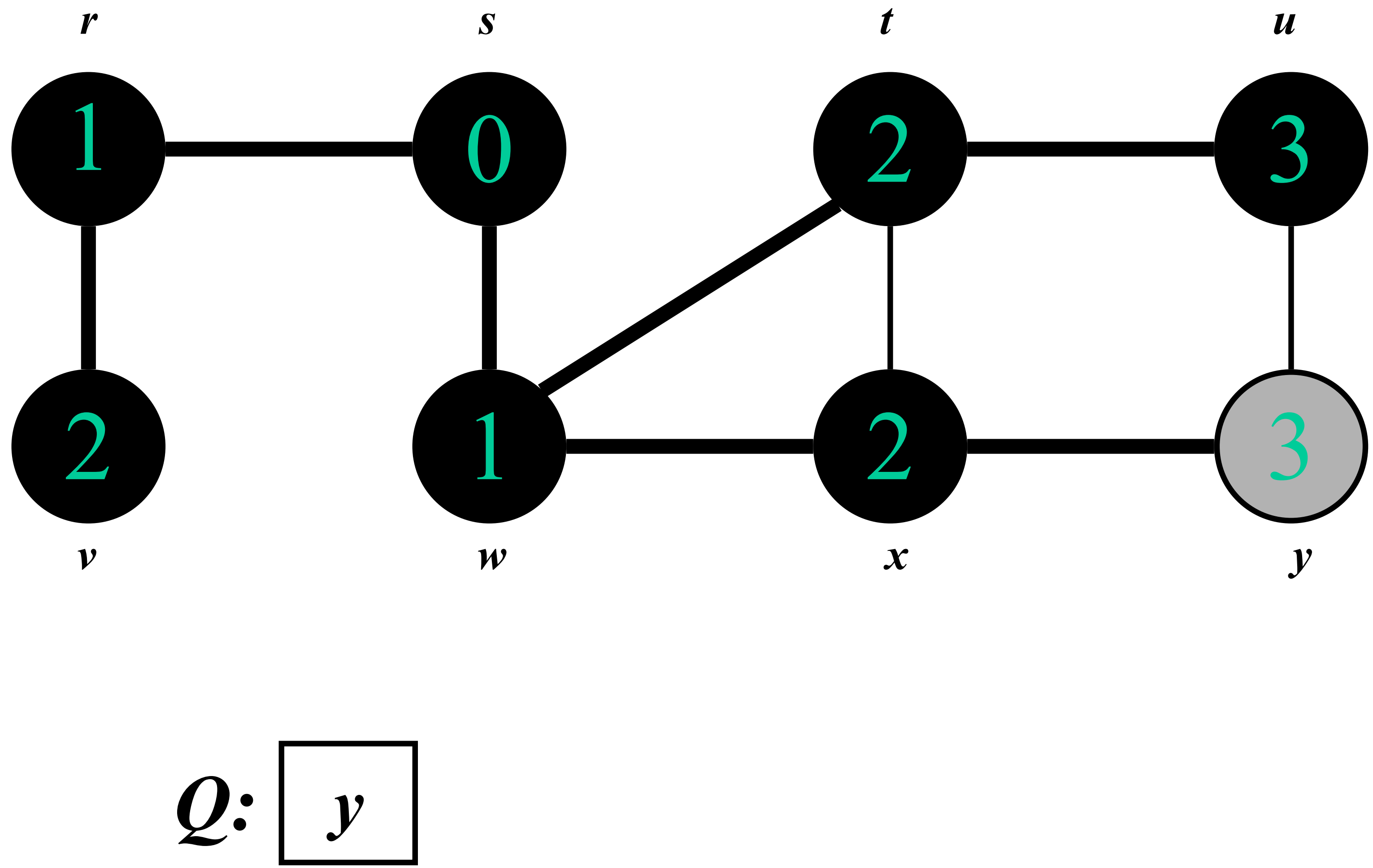


Breadth-first search: *example*

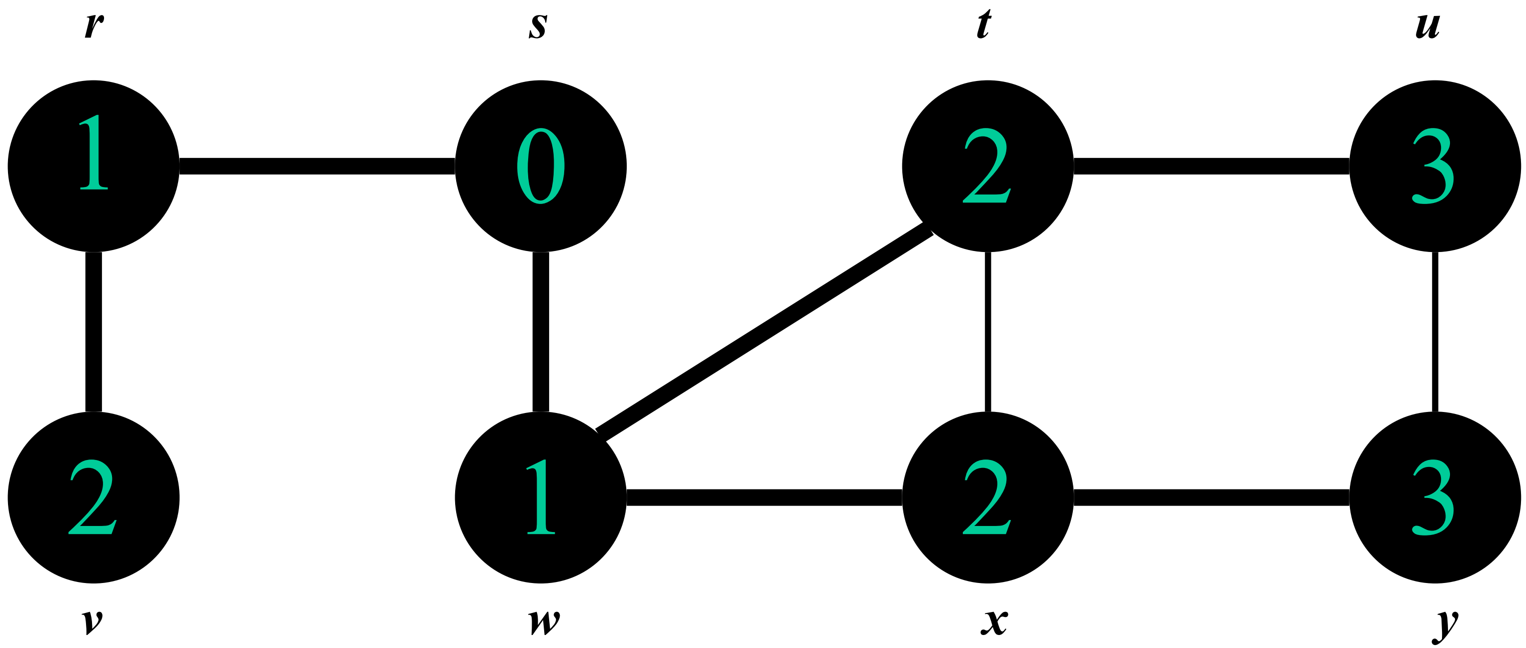


Q: | *u* | *y* |

Breadth-first search: *example*



Breadth-first search: *example*



Q: Ø

Breadth-first search: *pseudocode again*

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once (Why?)

So v = every vertex that appears in some other vert's adjacency list

What will be the running time?

Total running time: O(V+E)

What will be the storage cost in addition to storing the graph?

Total space used: O(max(degree(v))) = O(E)

Breadth-first search: properties

BFS calculates the *shortest-path distance* to the source node

- Shortest-path distance δ(s,v) = minimum number of edges from s to v, or ∞ if v not reachable from s

BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G

- Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Representing a graph G(V,E): *review*

As a *network* of edges E connecting vertices V

As an *adjacency matrix* represents the graph as a *n* x *n* matrix A:

– $A[i, j] = 1$ if edge $(i, j) \in E$ (or weight of edge)
  $= 0$ if edge $(i, j) \notin E$

– Storage requirements: $O(V^2)$

  - A dense representation

– But, can be very efficient for small graphs

  - Especially if store just one bit/edge
  - Undirected graph: only need one diagonal of matrix

BFS in games: *grid-based pathfinding*

- There is no need to construct a graph or adjacency matrix because a grid-based map is already represented by a *n*x*m* matrix, so adjacency relationships between a node and its 8-neighbor nodes are well defined.
- We only need the queue Q.
- The path between a source (start) node a sink (end) node can be easily reconstructed since we set the parent node for each node added to Q.

Deepth First Search (DFS)

Intuition:

- Depth-first search works much like people try to get out from a maze:
  - First, we follow a path until we hit a dead end or reach the end of the maze.
  - Second, if a given path doesn't work, we backtrack and take an alternative path from a past junction, and try that path.
  - This means that we replace "neighbor" by "child".

Comparison with BFS:

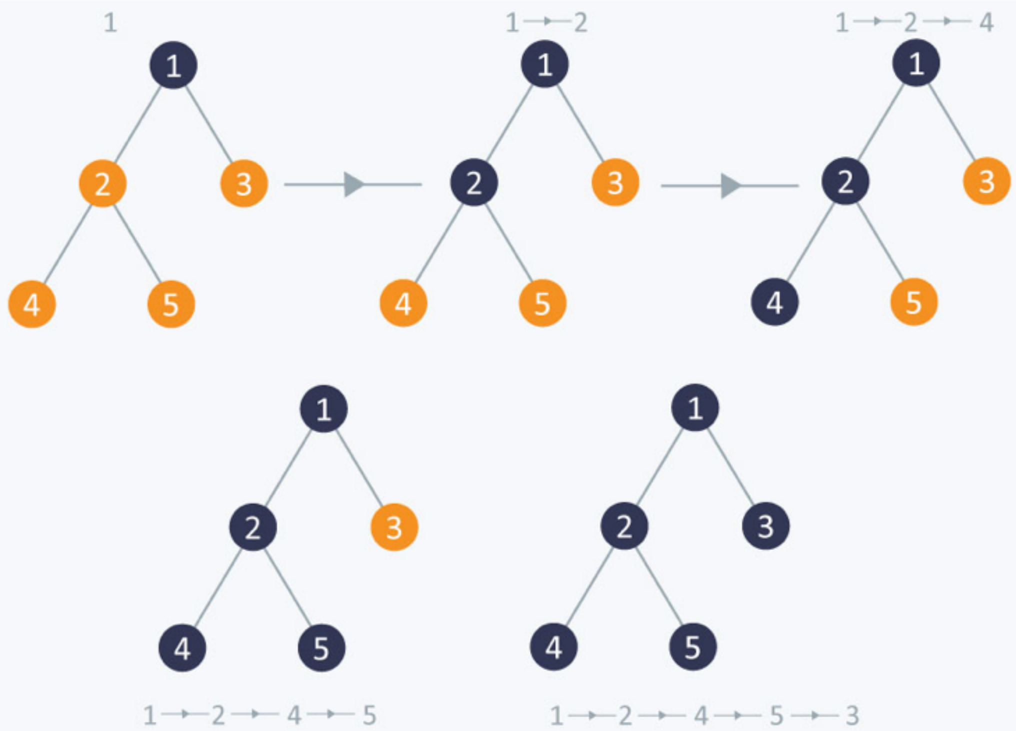
- Just like in BFS, to prevent infinite loops, we only want to visit each vertex once.
- As in BFS, we use flags to keep track of the vertices that have already been visited.
- Also, just like in BFS, we can use this search to build a spanning tree. The difference is that the tree is built in depth rather that in breadth.

Recursive DFS

https://hackernoon.com/graphs-in-cs-and-its-traversal-algorithms-cfee5533f74e

```
recursive_dfs(graph G, vertex v)
{
    visit(v);
    for each neighbor w of v
        if w is unvisited
        {
            dfs(w);
            add edge vw to tree T
        }
}
```

Iterative DFS



https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

```
iterative_dfs(graph G, vertex s)
{
    let S be stack
    S.push(s) //Inserting s in stack
    mark s as visited
    while ( S is not empty )
        //Pop a vertex from stack to visit next
        v = S.top()
        S.pop()
        //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in G
            if w is not visited
                S.push(w)
                mark w as visited
}
```

Edsger Wybe Dijkstra

"The question of whether computers can think is rather like the question of whether submarines can swim"

A* pathfinding

Problem:

- Given a non-player character (NPC), we intend to find the best path from the current NPC location to a destination. Obstacles may exist in between.

A*:

- We will find the best path (as long as heuristic *underestimates* the true cost)
  - Efficient
- However, it is overkill if line-of-sight exists
- Works in a tiled map
  - Or somehow, must model space with finite nodes

A*

```
Astar(graph G, vertex start_node, vertex goal_node)
{
    Add start_node to open_list

    while (not_empty(open_list))
    {
        current_node := node from open_list with lowest cost
        if (current_node==goal_node)
            path complete
        else
        {
            move current_node to closed_list
            for each node N adjacent to current_node {
                if ((N is not in open_list) && (N is not in closed_list))
                {
                    move N to open_list
                    assign cost to N
                }
        }
    }
}
```

Tracing the path in A*

```
Astar(graph G, vertex start_node, vertex goal_node)
{
    Add start_node to open_list

    while (not_empty(open_list))
    {
        current_node := node from open_list with lowest cost
        if (current_node==goal_node)
            path complete
        else
        {
            move current_node to closed_list
            for each node n adjacent to current_node {
                if ((n is not in open_list) && (n is not in closed_list))
                {
                    move N to open_list
                    assign cost to n
                }
        }
    }
}
```

At the end, when the path is found, follow the parent pointers to trace out the path.

When a node n is added to the open_list, keep a pointer to its parent_node. Here the parent node is current_node.

Cost function at location n:

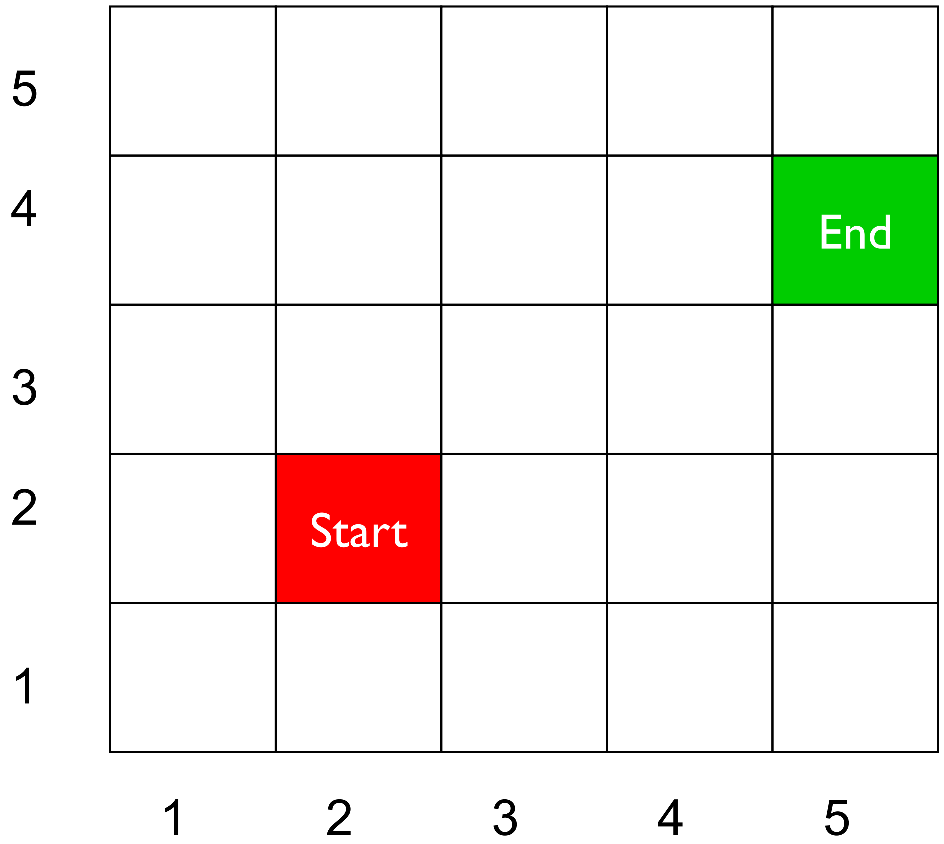
$$f(n) = g(n) + h(n)$$

$g(n)$: distance from the start point to point n

$h(n)$: estimated distance from point n to the goal point

$f(n)$: current estimated cost for point n

A* example



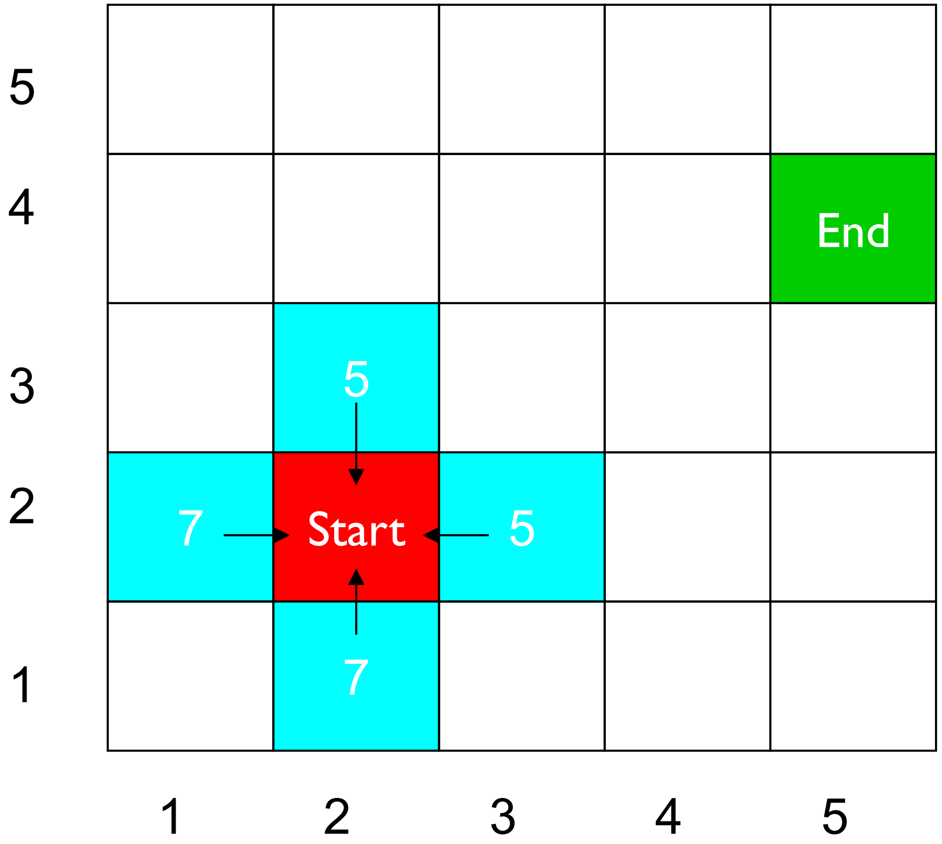
Iteration 1:

Open List:

Node: (2,2)
Cost: 5
Distance from start: 0

Closed List: Empty

A* example



Iteration 2:

***Open List*:**

Node: (1,2)
Cost: 7
Distance from start: 1

Node: (2,1)
Cost: 7
Distance from start: 1

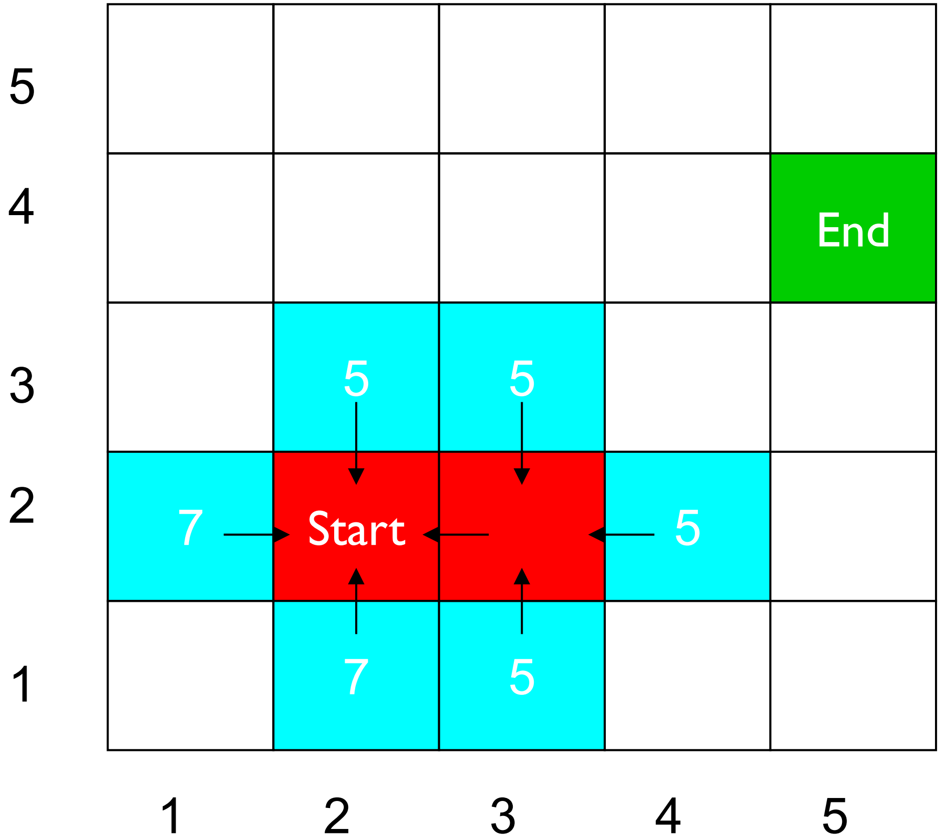
Node: (3,2)
Cost: 5
Distance from start: 1

Node: (2,3)
Cost: 5
Distance from start: 1

***Closed List*:**

Node: (2,2)

A* example



Iteration 3:

Open List:

Node: (1,2) Cost: 7 Distance from start: 1
Node: (2,1) Cost: 7 Distance from start: 1
Node: (2,3) Cost: 5 Distance from start: 1
Node: (3,3) Cost: 5 Distance from start: 2
Node: (4,2) Cost: 5 Distance from start: 2
Node: (3,1) Cost: 7 Distance from start: 2

Closed List:

Node: (2,2)	Node: (3,2)

A*: additional notes

- ***Dead ends***. If open_list is empty before the goal node is found, it means that there is a dead end. There does not exist a path between the starting point and the destination node.
- ***Terrain cos***t. Simple A* algorithm does not consider terrain cost. If different tiles are made of different terrain, some of which are harder to cross, we can assign a cost to each tile according to its terrain. Instead of adding 1 to the distance from start, add the cost.
- ***Influence cost***. Every time the character gets shot at in a tile, add the influence cost of the tile, to make it most costly to go that way.

Summary:

…:

- Introduction
- Definitions: pathfinder, graph
- Graph traversal algorithms in games
- BFS, DFS, ~~Dijkstra~~, and A*

https://movingai.com/benchmarks/

https://qiao.github.io/PathFinding.js/visual/