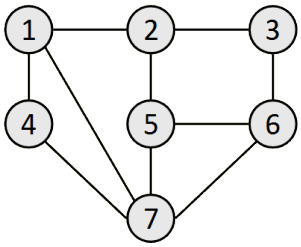
Graph

REPRESENTATION

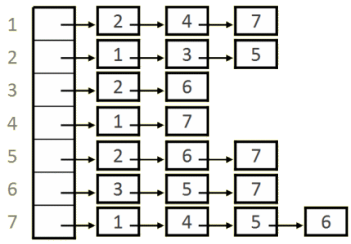
Consider the following graph:

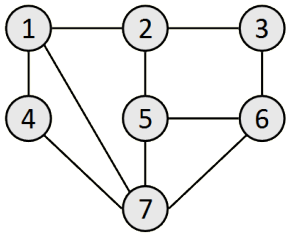


One way to implement a graph is using an **edge list**, where the only structure you store is a list (vector) of the edges. The information about the vertices is implicit inside the edges. Here is an edge list for the above graph:

(1, 2)	(1, 4)	(1, 7)	(2, 3)	(2, 5)	(3, 6)	(4, 7)	(5, 6)	(5, 7)	(6, 7)

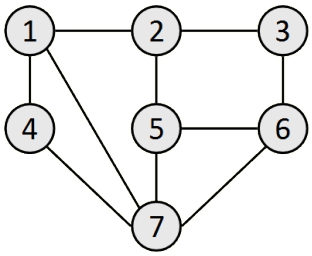
A second way to implement a graph is using an **adjacency list**, where the only structure you store is a list of the vertices, where each vertex contains a nested list of other vertices that are its neighbors. The information about the edges is implicit inside the vertices' neighbor lists. Here is an adjacency list for the above graph:





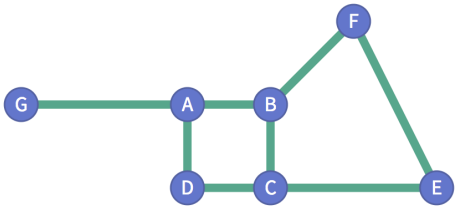
A third way to implement a graph is using an **adjacency matrix**, where the only structure you store is a two-dimensional grid where each row represents a start vertex and each column represents an end vertex. The grid cell [i, j] stores information about the edge, if any, from vertex i to vertex j. Here is an adjacency matrix for the above graph:

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



PROPTERTIES

A graph-based pathfinding algorithm needs to know ***what the locations are*** and also ***which locations are connected to which other ones***. You typically know a lot more than this, like the size and coordinates of the locations, but the algorithm doesn't actually know about these aspects. It only knows what the connections are.



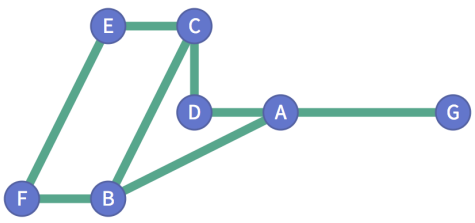
A mathematical graph is a set of ***nodes*** and ***edges***. The nodes (also called vertices or objects) are connected together by the edges (also called links or connections or arrows or arcs). For any graph we need to know two things:

1. **Set of nodes** in the graph
2. **Set of edges** from each node

What does the above graph look like?

1. **Set of nodes**: A B C D E F G.
2. **Set of edges** from each node:
    - **A**: A→B A→D A→G
    - **B**: B→A B→C B→F
    - **C**: C→B C→D C→E
    - **D**: D→C D→A
    - **E**: E→C E→F
    - **F**: F→B F→E
    - **G**: G→A

Note that the layout of the graph is not part of the graph. The graph in the above diagram and the graph shown below ***are the same graph***:



GRAPH CONCEPTS

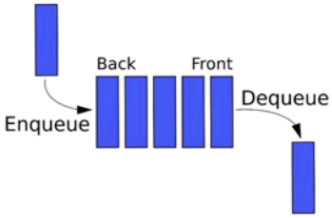
Graph Traversal Algorithms

- These algorithms specify an *order* to search through the nodes of a graph.
- We start at the source node and keep searching until we find the target node.
- The *frontier* contains nodes that we've seen but haven't explored yet.
- Each iteration, we take a node off the frontier, and add its neighbors to the frontier.

BFS and DFS Logic:

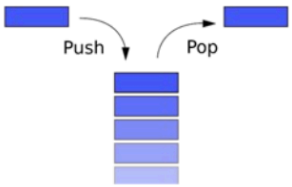
Breadth First Search vs. Depth First Search

BFS uses "first in first out".



This is a queue.

DFS uses "last in first out".



This is a stack.

Greedy Best First Search:

Greedy Best First Algorithm

- Recall: BFS and DFS pick the next node off the frontier based on which was "first in" or "last in".
- Greedy Best First picks the "best" node according to some rule of thumb, called a *heuristic*.

Definition: A *heuristic* is an approximate measure of how close you are to the target.

A heuristic guides you in the right direction.

A* :

A* Search

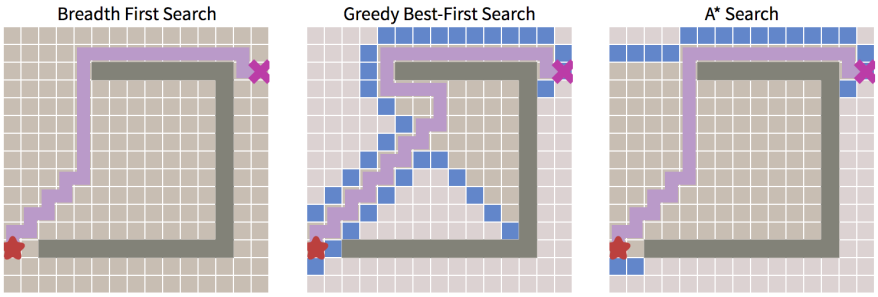
- A* Search combines the strengths of Breadth First Search and Greedy Best First.
- Like BFS, it finds the shortest path, and like Greedy Best First, it's fast.
- Each iteration, A* chooses the node on the frontier which minimizes:

steps from source + approximate steps to target

Like BFS, looks at nodes close to source first (thoroughness)

Like Greedy Best First, uses heuristic to prioritize nodes closer to target (speed)

Compare the algorithms:

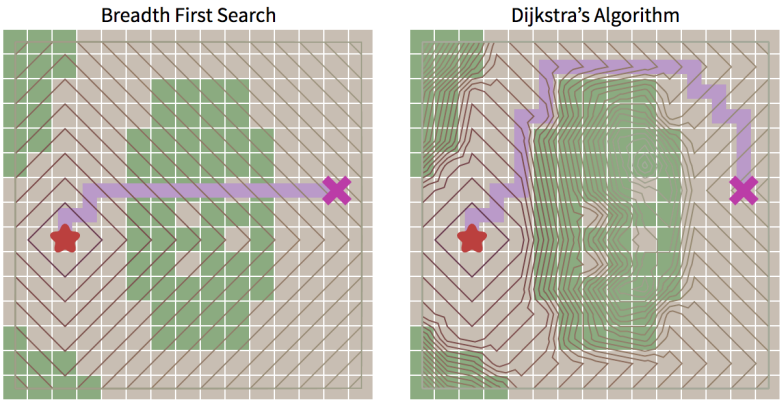


WEIGHTED GRAPH

Dijkstra's Algorithm:

Dijkstra's algorithm

- Like BFS for weighted graphs.
  - If all costs are equal, Dijkstra = BFS!
- Explores nodes in increasing order of *cost* from source.



Weighted A* :

Weighted A*

Regular A* priority function:

steps from source + approximate steps to target

Weighted A* priority function:

cost from source + approximate cost to target

SUMMARY

Recap

Search algorithms for unweighted and weighted graphs

Breadth First Search	First in first out, optimal but slow
Depth First Search	Last in first out, not optimal and meandering
Greedy Best First	Goes for the target, fast but easily tricked
A* Search	"Best of both worlds": optimal and fast

Dijkstra	Explores in increasing order of cost, optimal but slow
Weighted A*	Optimal and fast