INTRODUCTION TO AI - SEARCH

Philip Mortimer

“**Machine learning is the science of getting computers to learn without being explicitly programmed**” – Sebastian Thrun

COURSE OVERVIEW

- Understanding of the many different types of AI systems.
- Discuss "hard-coded" AI (i.e., heuristic and algorithmic systems).
- Some programming (no prior knowledge needed) for some interesting practical examples.
- Discuss a wide range of AI concepts with references to real world applications (e.g., solving a maze, playing chess or driving a car).
- Introduction to machine learning – the cutting-edge field of AI that is at the forefront of research.

WHY STUDY AI?

- Can be applied to a wide range of problems.
- AI can solve problems that are impossible to “hard-code”.
- For example, AI can perform facial recognition or convert human speech to text.
- Due to widespread availability of datasets and programming libraries, anyone with sufficient expertise is able to make a very powerful AI model.

WHAT IS AI?

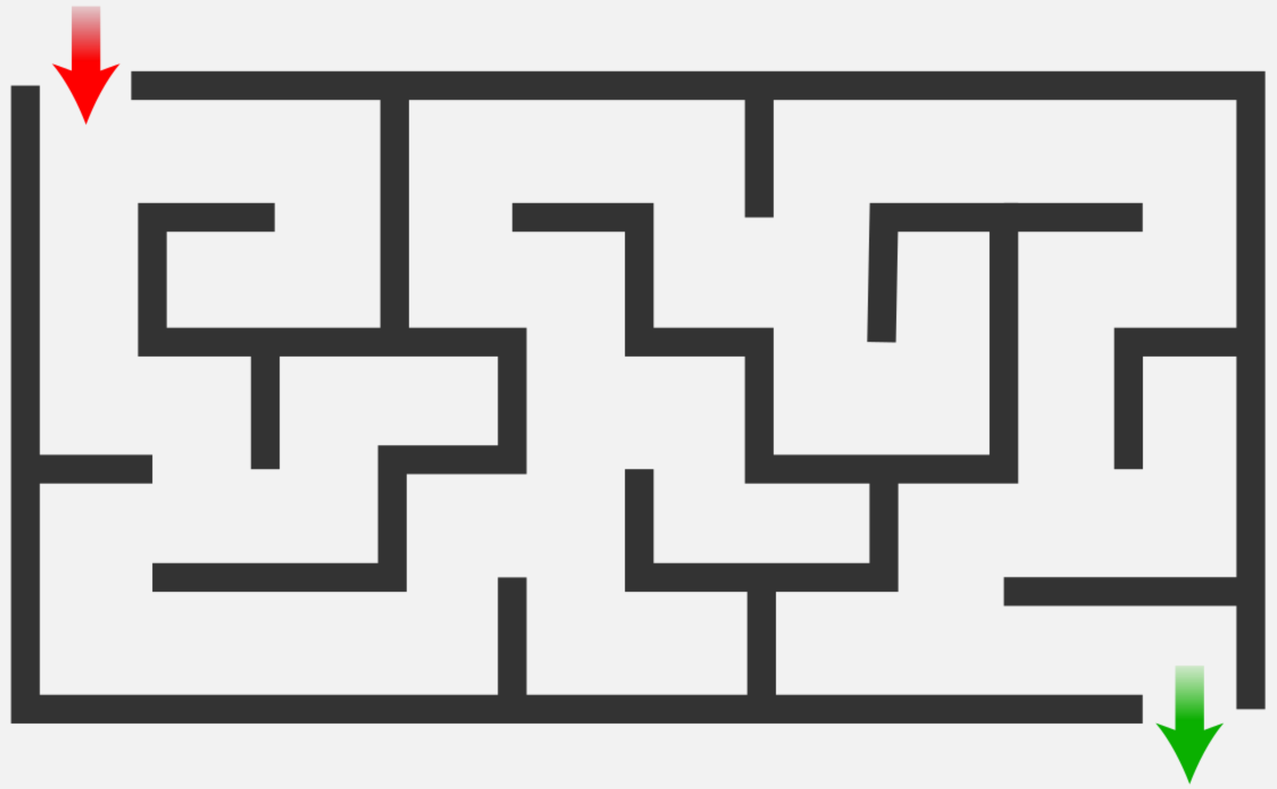
- AI is any program that can perform a task intelligently (typically at a level similar to or greater than that of people)
- We normally think of AI as black box systems that learn from large datasets (like the ones in Hollywood films)
- This is called Machine Learning and is a big subset of AI
- However, a big part of AI is still very much "hard-coded". This refers to algorithms that are cleverly programmed to solve difficult problems well.

OVERVIEW

- Focus on hard coded AI systems today
- Discuss various approaches to navigating a data space
- Look at different algorithms used to optimise for different metrics.

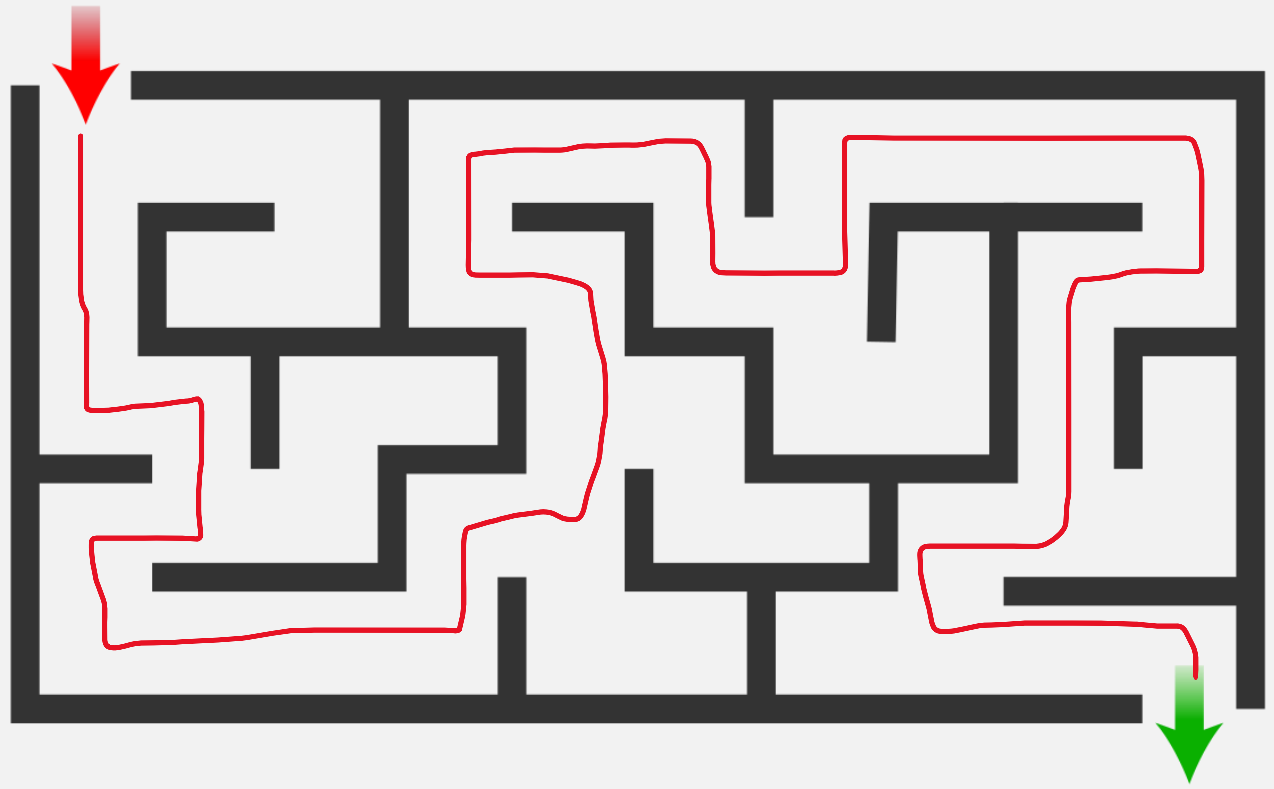
BUT FIRST … MAZES

MAZES!



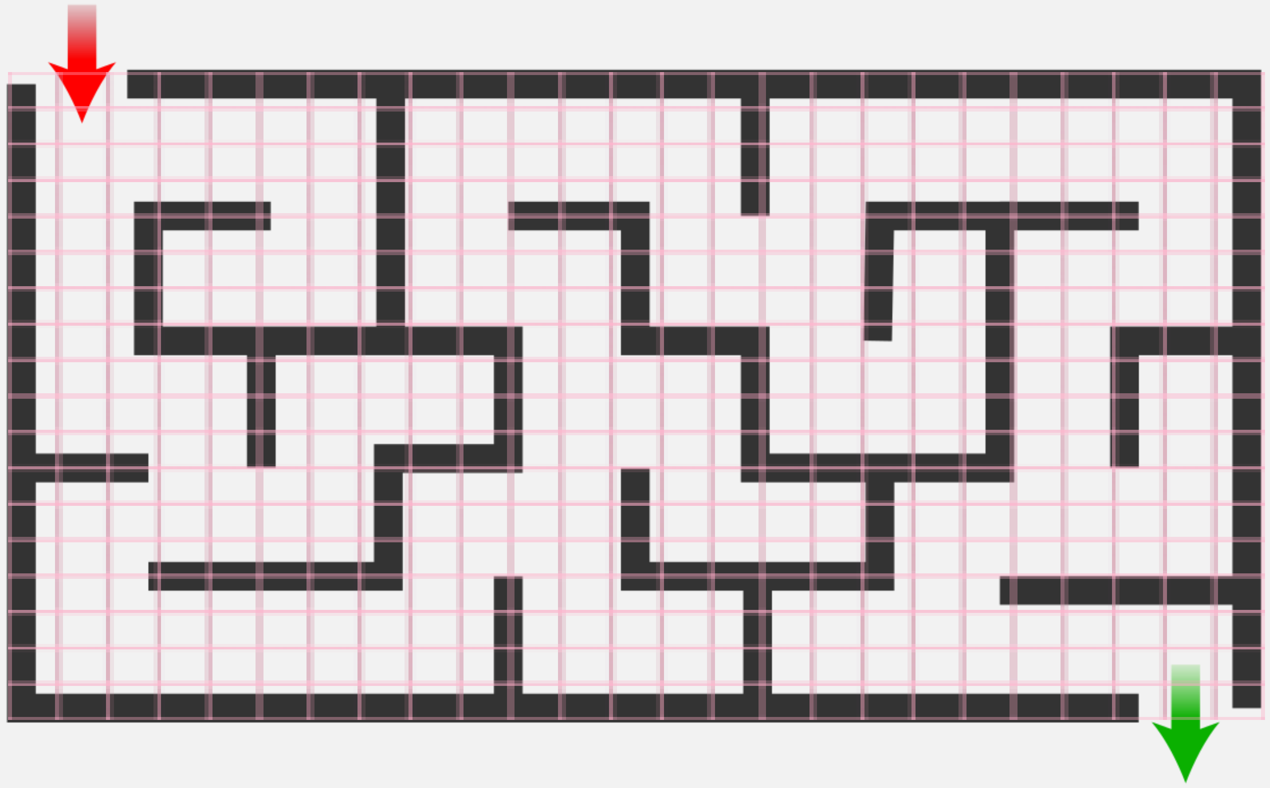
- Get from red arrow to green arrow
- Preferably in shortest possible distance

MAZES!



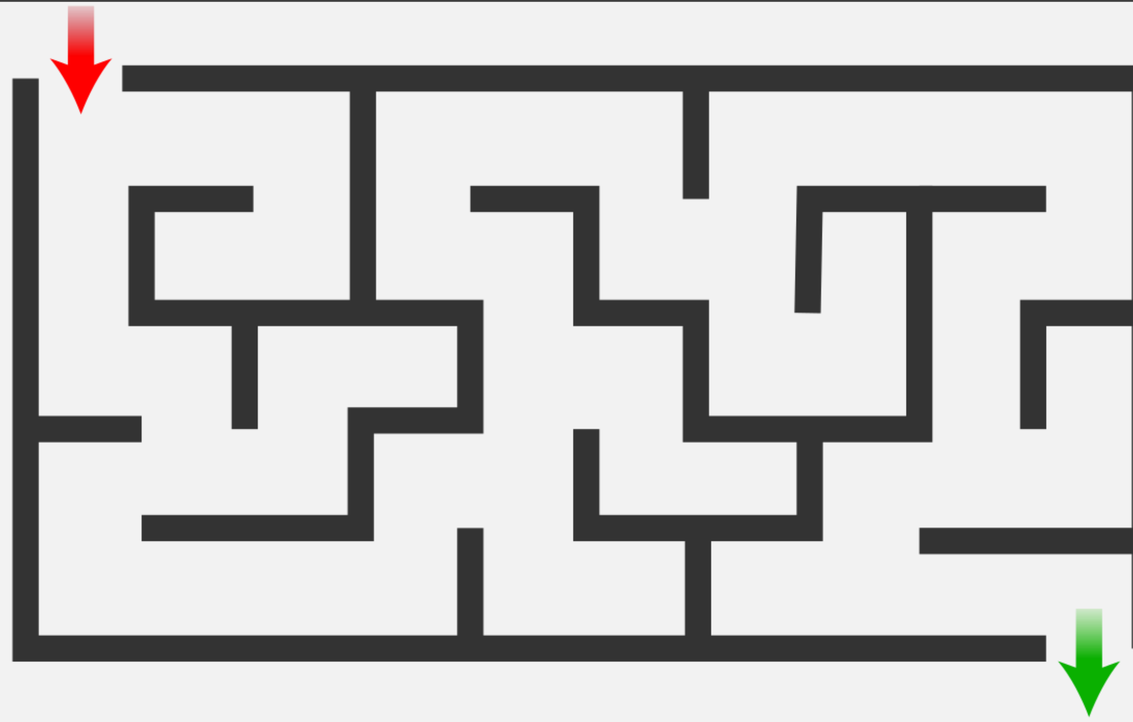
- We intuitively know how to solve a maze.
- When programming AI systems, the challenge often is found in encoding human knowledge to some computer format.

MAZES!



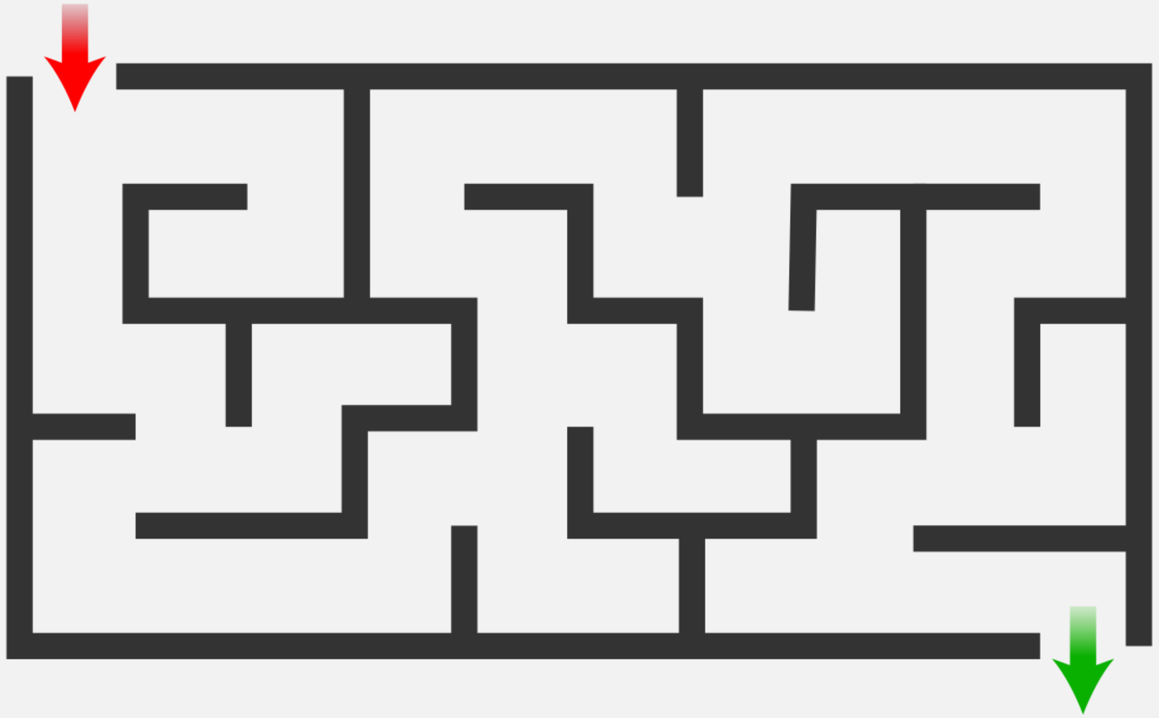
- At any given point, we are able to go left, right, down or up.
- You can't go back on yourself. Why?
- Can break maze down into a series of appropriately sized blocks.

MAZES!



- For most blocks we have no choice of direction.
- However, when we reach certain points. We can choose to move in different directions
- How we handle these decision points determines our solution.

MAZES!



- Let's formalise the way we tend to solve mazes in an algorithmic format.

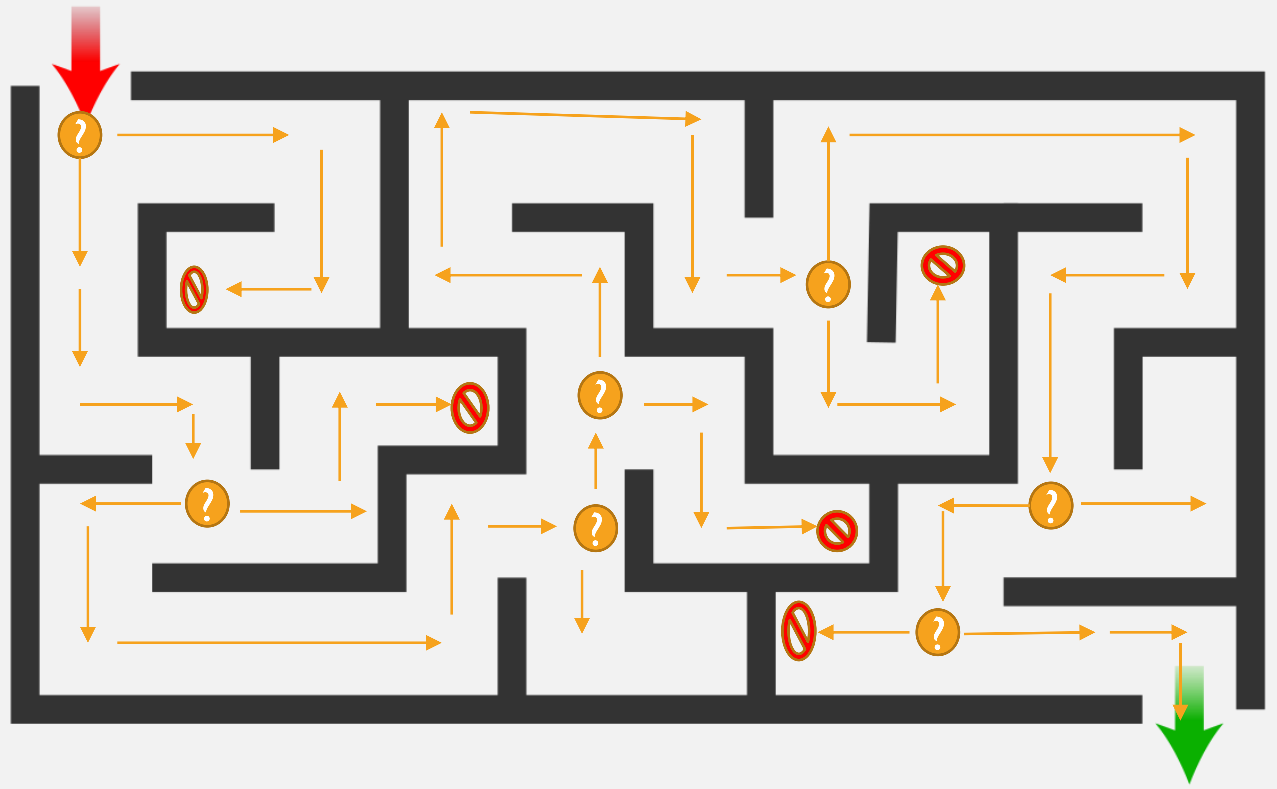
- Every time we reach a branch. Randomly select one of the possible options.
- Keep going along this route. If we hit a dead end, go back up to the most recent decision and choose another branch.
- Repeat until we complete maze

MAZES!

- No choice – keep going
- Decision – pick random path
- Dead end – go back to previous decision
- Maze end reached!

- Every time we reach a branch. Randomly select one of the possible options.
- Keep going along this route. If we hit a dead end, go back up to the most recent decision and choose another branch.
- Repeat until we complete maze

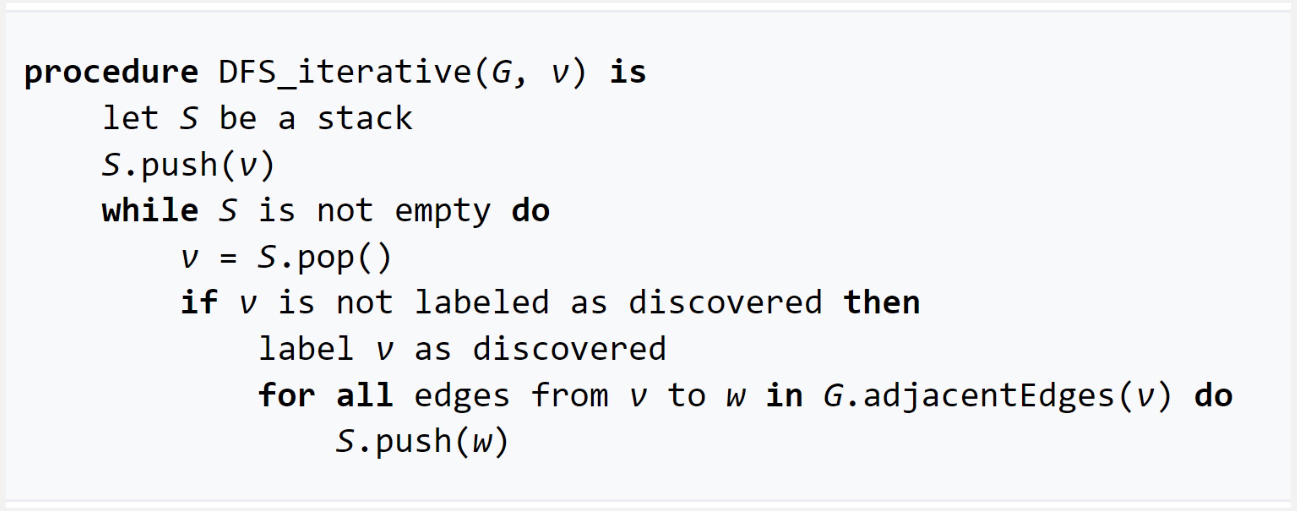
MAZES!



- This algorithm is known as depth first search (DFS).
- It is a deep search as it only looks at one path at a time.
- DFS will always find a solution to the maze.
- This solution is not always optimal. In this case, there is only valid solution, so it does produce an optimal result.

DEPTH FIRST SEARCH

- Search can be applied to many different problems and data structures (not just maze).
- The important thing is how DFS handles searching when it can choose between different nodes.

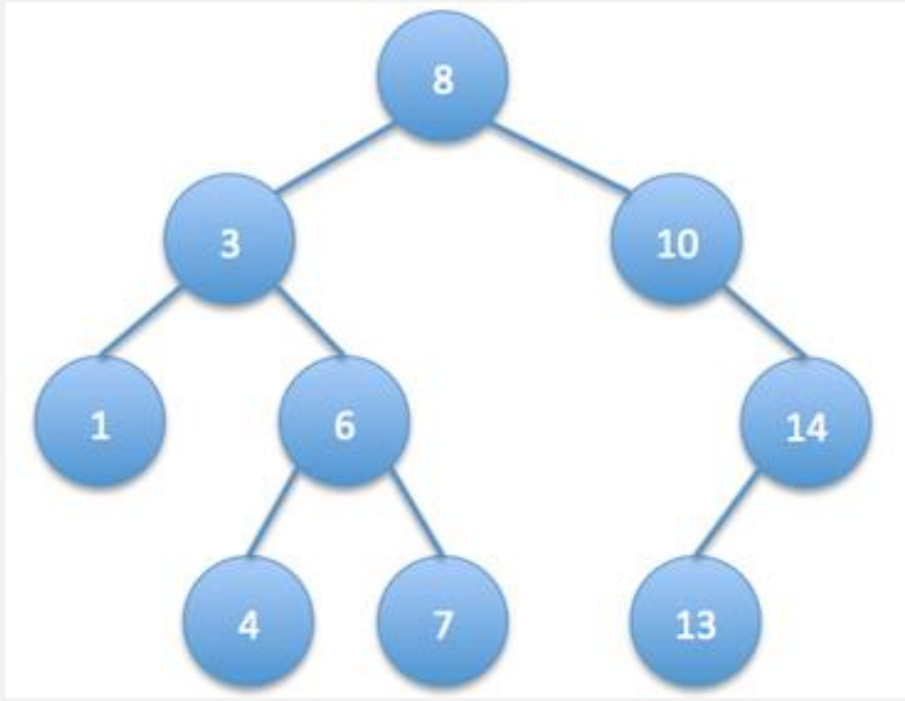


```
procedure DFS_iterative($G$, $v$) is
    let $S$ be a stack
    $S$.push($v$)
    while $S$ is not empty do
        $v$ = $S$.pop()
        if $v$ is not labeled as discovered then
            label $v$ as discovered
            for all edges from $v$ to $w$ in $G$.adjacentEdges($v$) do
                $S$.push($w$)
```

- In our case, how might we change following pseudocode to solve maze?

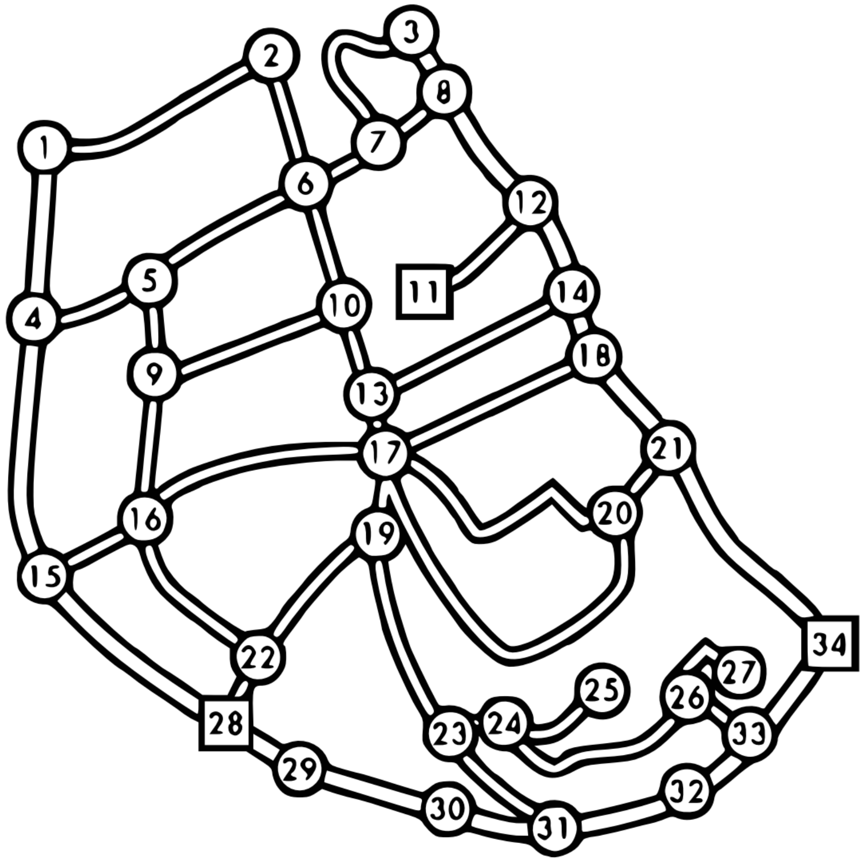
DFS APPLICATIONS

- We can use DFS on a whole range of structures.
- All we need is nodes that are somehow linked together.
- We can generalize this a tree structure:
- In case of maze, each node can be thought of as a point in the maze
- The root node is the start of the maze
- The nodes on the tree could represent anything.



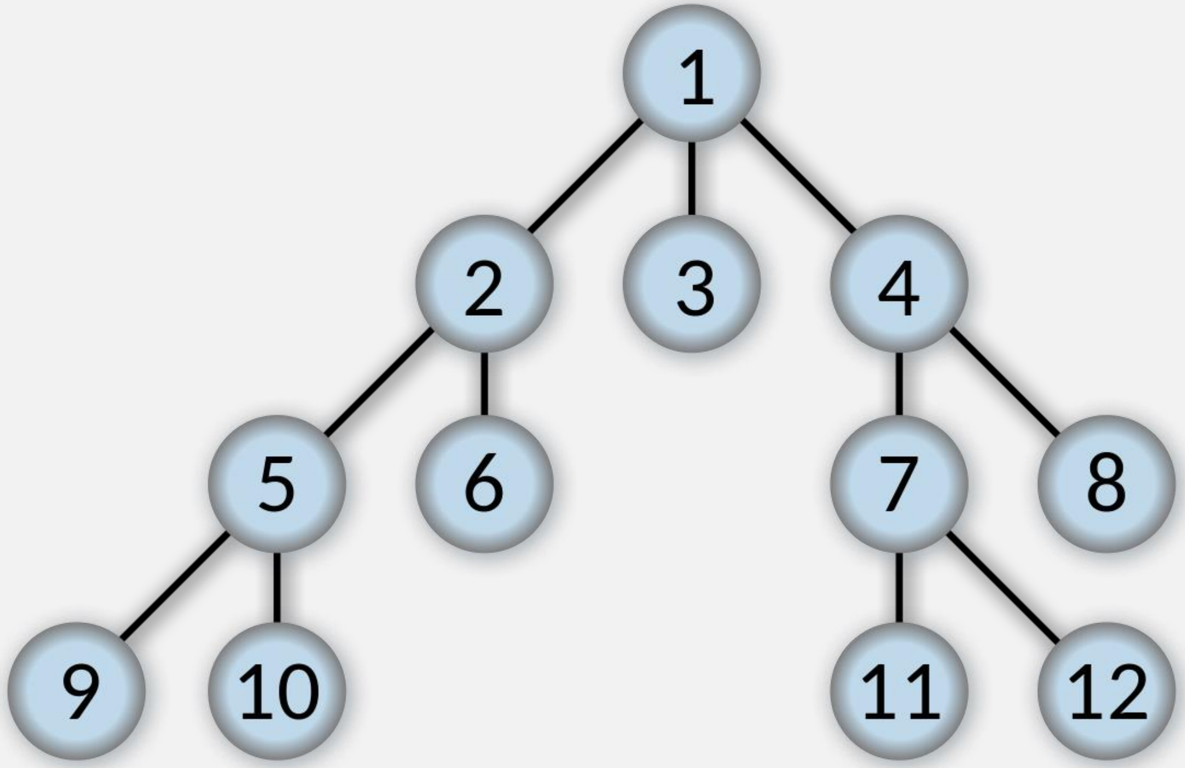
BFS

- DFS could be used as part of Satnav system to find path between two locations.
- DFS always finds a solution, not necessarily the best one – so we might be left with a 48-hour car journey to the shops!
- Often, we want to find the best solution (i.e., shortest possible path).

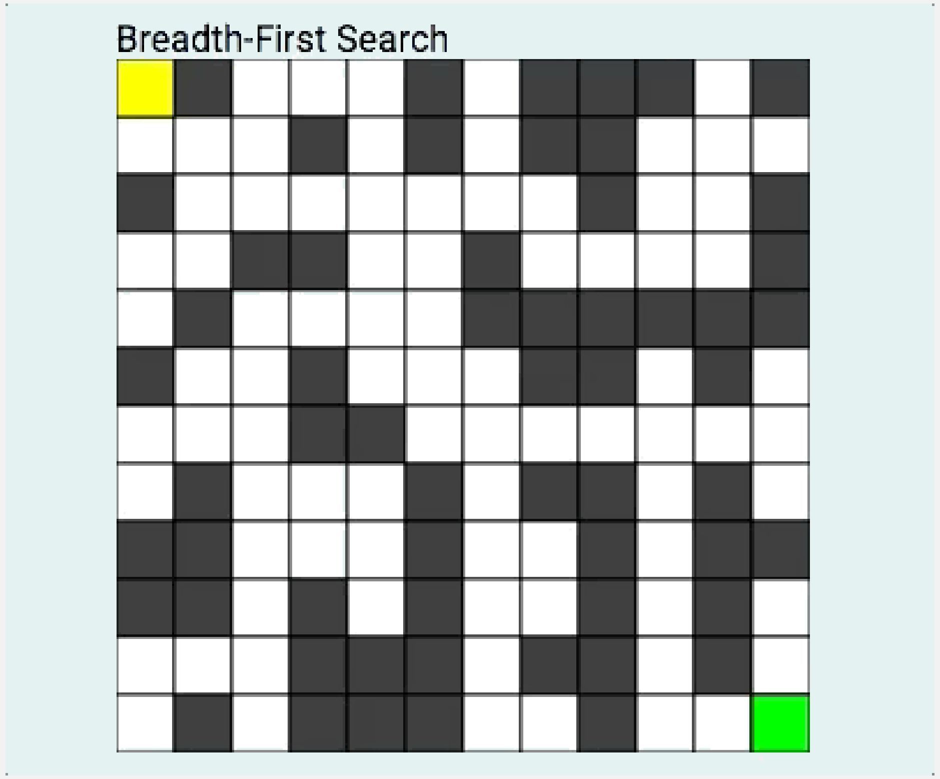


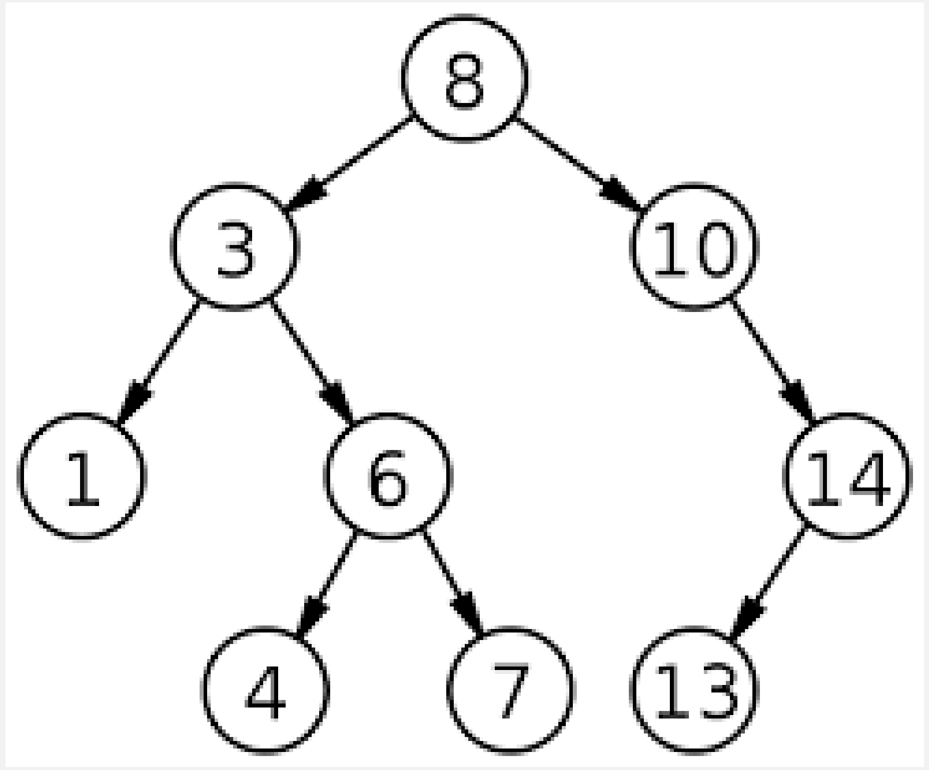
BFS

- Bread first search (BFS) is similar to DFS – it's a way of traversing a tree structure.
- DFS explores one path very deeply.
- BFS explores all paths at the same depth till a solution is found.
- BFS guarantees an optimal solution
- This is useful when multiple solutions may exist.



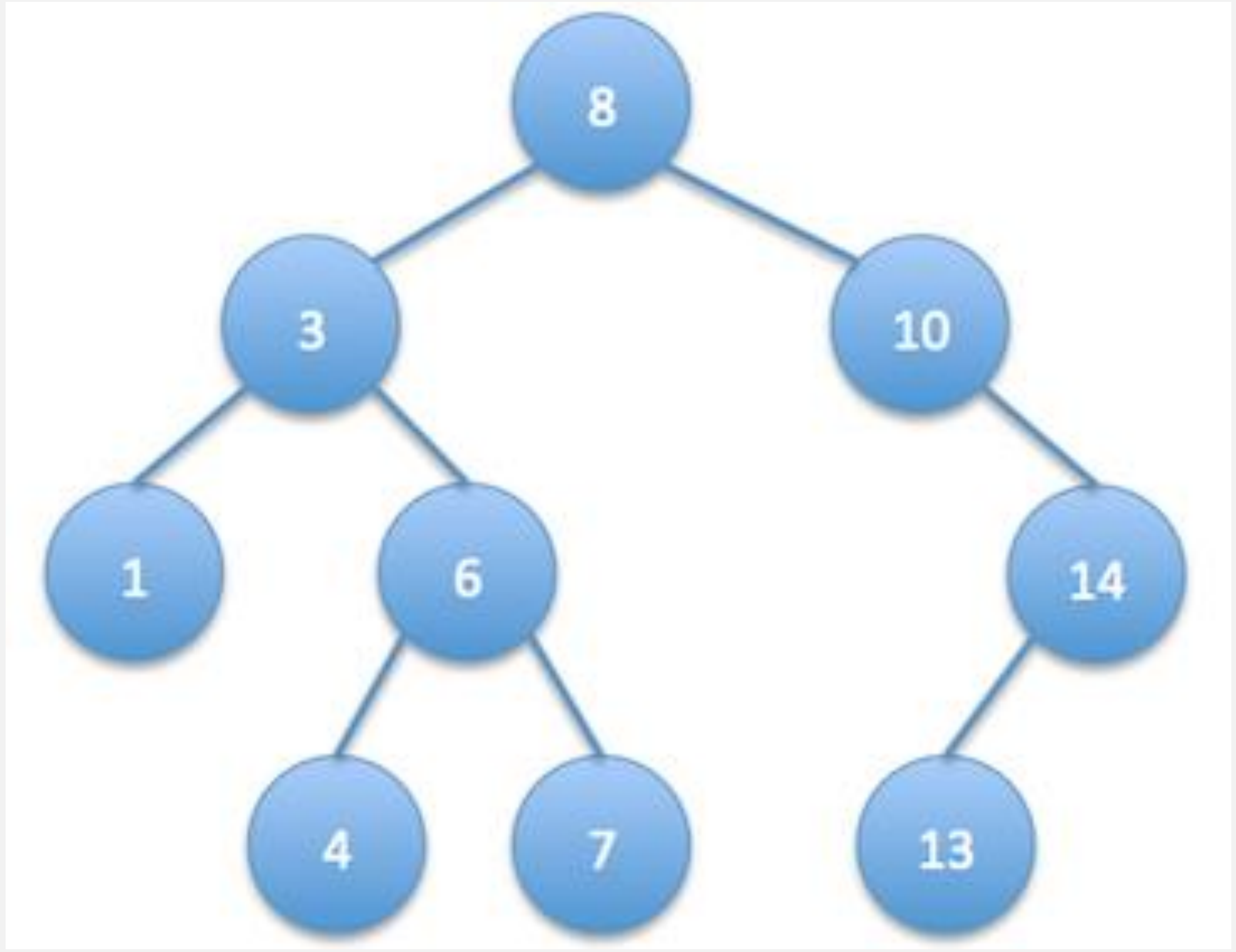
BFS





BFS

- For our maze example, when we have a choice of direction, we will look at all possible directions simultaneously.
- In DFS, we would have only looked at one direction and kept exploring that direction.



BFS

```
procedure BFS_iterative(G, v) is
    let Q be a queue
    Q.push(v)
    while Q is not empty do
        v = Q.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                Q.push(w)
```

BFS VS DFS

```
procedure BFS_iterative(G, v) is
    let Q be a queue
    Q.push(v)
    while Q is not empty do
        v = Q.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                Q.push(w)
```

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Similar pseudocode – difference is how decision branches are handled

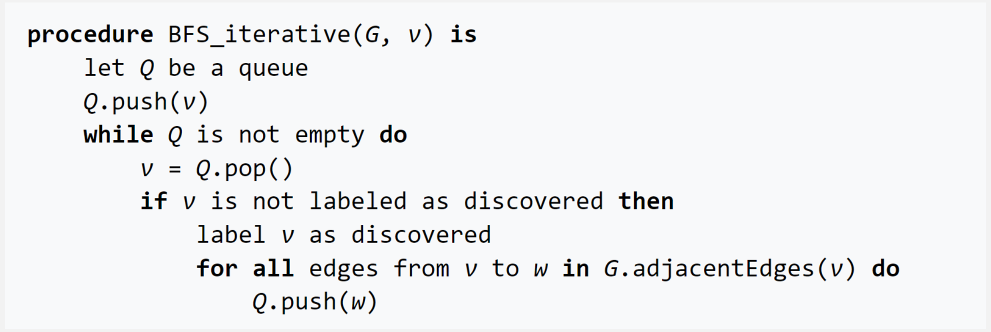
BFS uses a queue (FIFO) structure

DFS uses a stack (LIFO) structure

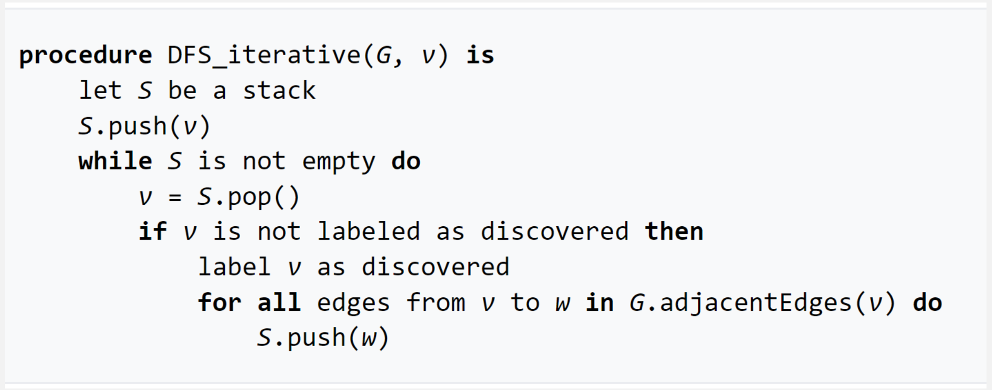
BFS produces an optimal solution by searching whole subspace of distance of shortest path

DFS produces a solution, not necessarily the best solution

BFS VS DFS



```
procedure BFS_iterative(G, v) is
    let Q be a queue
    Q.push(v)
    while Q is not empty do
        v = Q.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                Q.push(w)
```



```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Both techniques are expensive

DFS is often used for Computer games programming

BFS is often used for navigation systems

Often small tweaks to these general algorithms are used to adapt to specific problems

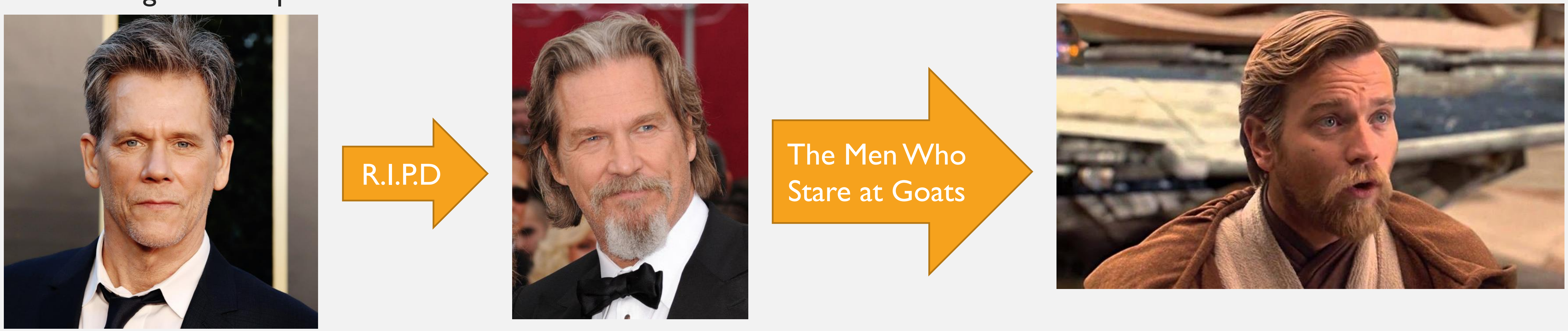
Both have same runtime complexity. Performance largely depends on data structure.

SIX DEGREE OF SEPARATION

= 6

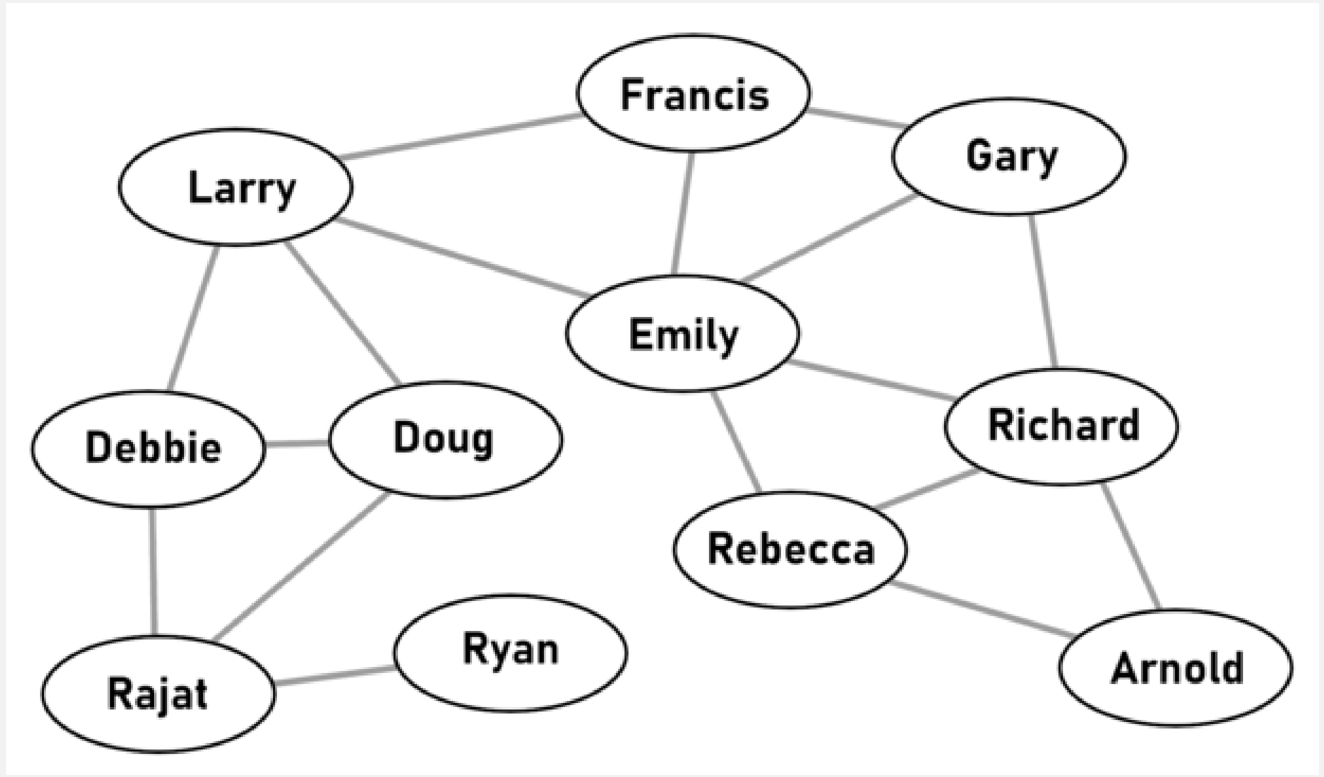
SIX DEGREES OF SEPARATION

- It is theorised that any two people on Earth can be connected together in six steps or less.
- E.g. Kevin Bacon was in the 2013 film “R.I.P.D”. Jeff Bridges also starred in “R.I.P.D”. Jeff Bridges acted in 2009’s “The Men Who Stare at Goats”, which also happens to feature Ewan McGregor. Thus, there are two degrees of separation between McGregor and Bacon.
- There are multiple valid ways to make Kevin Bacon and Ewan McGregor have a degree of two.
- Additionally, there are many ways for Kevin Bacon and Ewan McGregor to have three, four, five (and so on) degrees of separation.



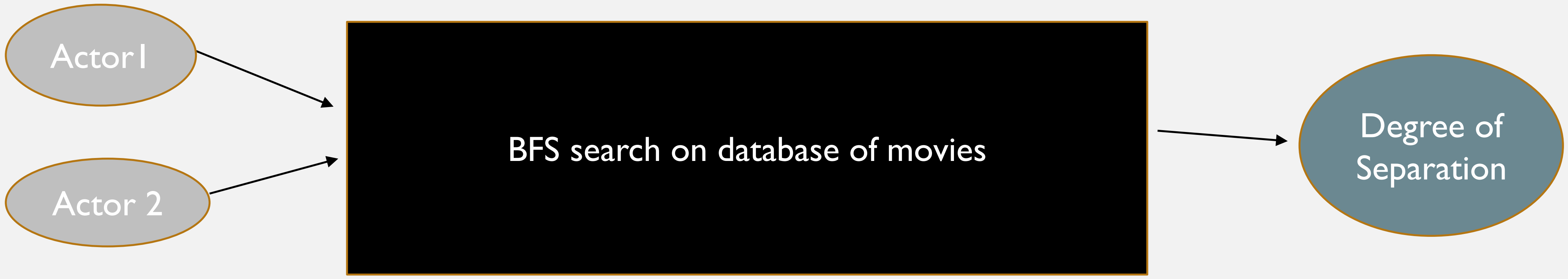
DEGREES OF SEPARATION

- Can think of each actor as a node in a tree. Links between nodes are films shared between actors.
- We know that we can treat a tree in exactly the same way as we treated a maze
- Given a database of films and actors in said films, we can calculate the degree of separation between two actors.



SEPARATION

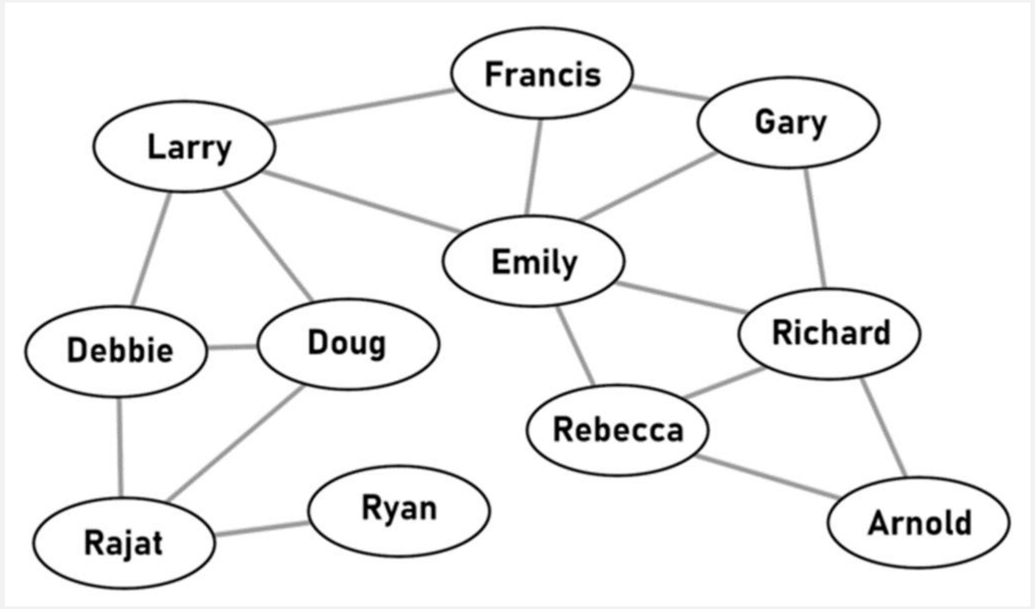
- We want to find lowest number of possible links. Should we apply DFS or BFS?
- Let's design a system that can calculate degree of separation.

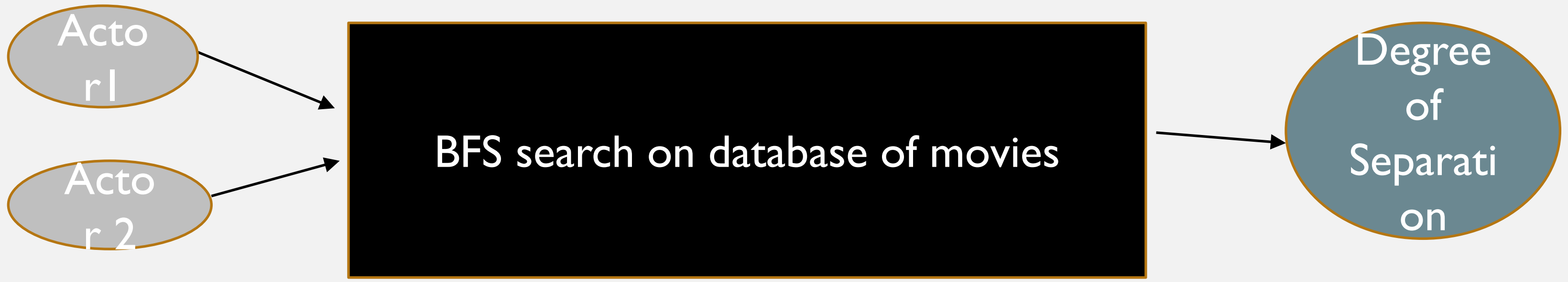


SEPARATION

- Let allFilms(actorName) be the function that gets all films a given actor has been in. This can be though of as getting all tree vertices (lines) leading into a specific node.
- Let allActors(filmName) be the function that gets all actors in a given film. This can be thought of as getting all tree nodes that have a certain vertex (line) leading into them.
- Let f = allFilms(actorName). If we iterate through all films stored in f, we can find all actors that star in the same film.
- getAllActorsLinked(actorName){

```
getAllActorsLinked(actorName){
        allActors is list
        f = allFilms(actorName)
        for (FilmName fName : f){
                allActors.add(allActors(fName))
        }
        allActors.removeDuplicates()
        allActors.remove(actorName)
        return allActors
}
```





SEPARATION

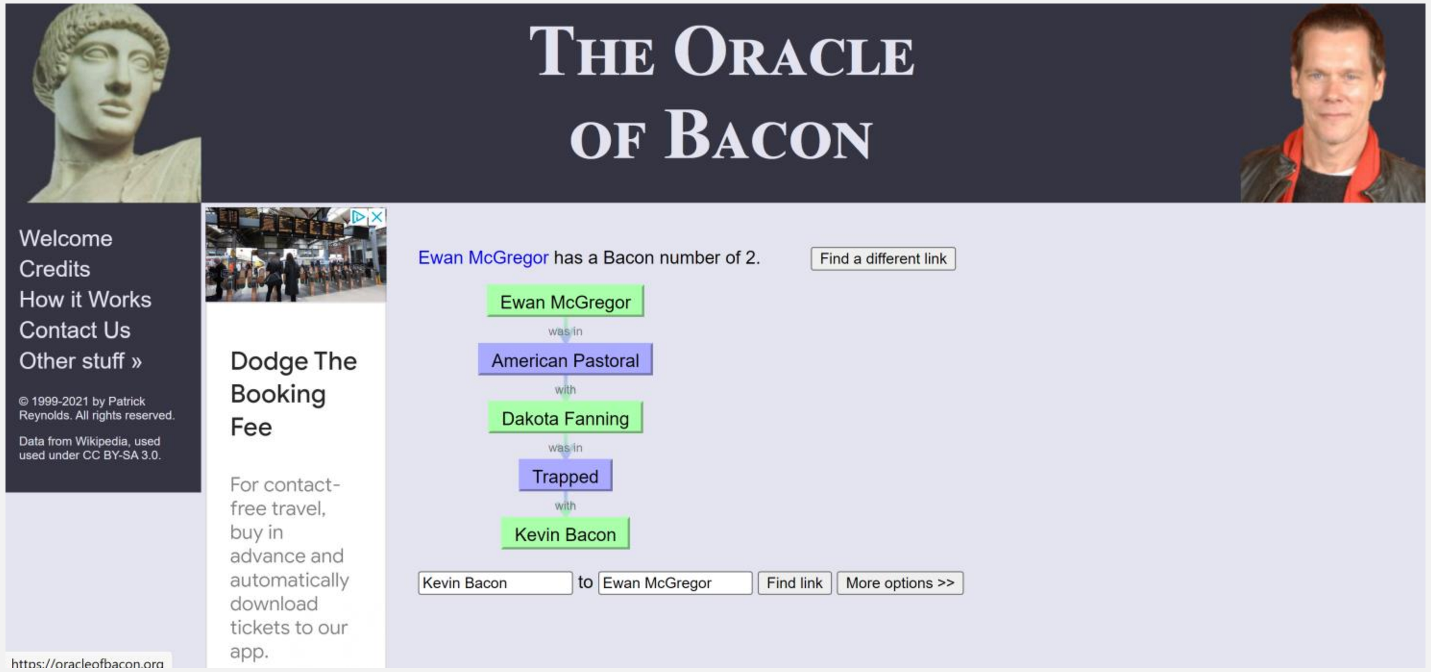
- Using pseudocode for BFS, we can now formulate algorithm.

```
BFS(actor1, actor2)
        degree = 0
        let Q be a queue
        Q.push(actor1)
        while Q is not empty do
                degree = degree + 1
                v =  Q.pop()
                if actor v has not been discovered yet then
                        label v as discovered
                        actorsLinked = getAllActorsLinked(v)
                        for all actors, a, in actorsLinked
                                if a == actor2 then
                                        return degree
                                else
                                        Q.push(a)
        return "No Link found"
```

```
procedure BFS_iterative(G, v) is
    let Q be a queue
    Q.push(v)
    while Q is not empty do
        v = Q.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                Q.push(w)
```

SEPARATION

- BFS will always give us the lowest possible degrees of separation.
- In fact, there is a website that calculates degree of separation using BFS: https://oracleofbacon.org/


THE ORACLE OF BACON

Welcome
Credits
How it Works
Contact Us
Other stuff »

© 1999-2021 by Patrick Reynolds. All rights reserved.

Data from Wikipedia, used used under CC BY-SA 3.0.

Dodge The Booking Fee

For contact-free travel, buy in advance and automatically download tickets to our app.

Ewan McGregor has a Bacon number of 2. Find a different link

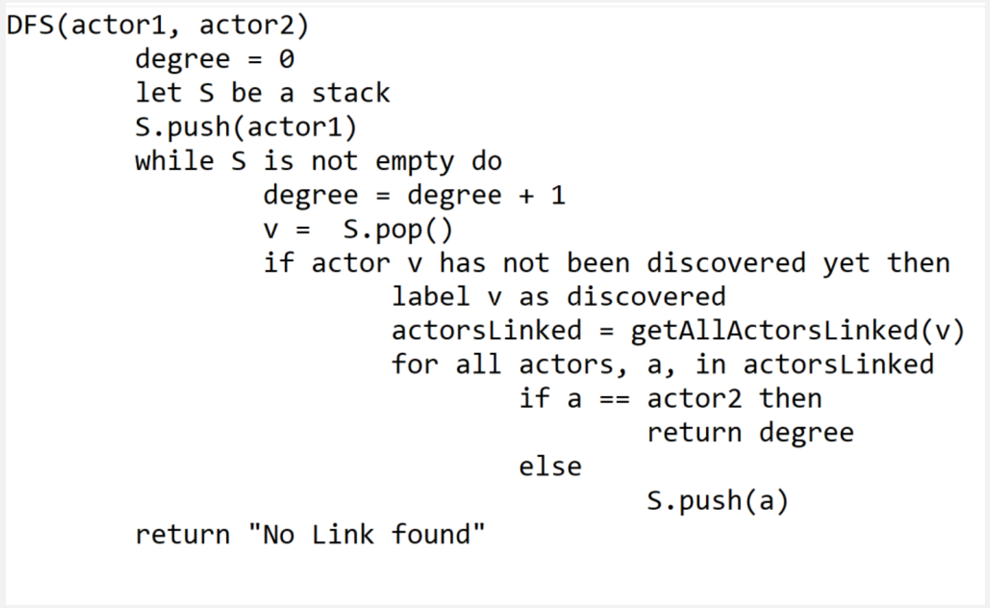
Ewan McGregor
was in
American Pastoral
with
Dakota Fanning
was in
Trapped
with
Kevin Bacon

Kevin Bacon to Ewan McGregor Find link More options >>

https://oracleofbacon.org

SEPARATION

- What would we expect to happen if we use DFS instead of BFS?
- How would we change algorithm to support DFS?
- Simply changing the queue data structure to a stack allows us to perform DFS.



```
DFS(actor1, actor2)
        degree = 0
        let S be a stack
        S.push(actor1)
        while S is not empty do
                degree = degree + 1
                v =  S.pop()
                if actor v has not been discovered yet then
                        label v as discovered
                        actorsLinked = getAllActorsLinked(v)
                        for all actors, a, in actorsLinked
                                if a == actor2 then
                                        return degree
                                else
                                        S.push(a)
        return "No Link found"
```

SEPARATION

- I made program that calculates degree of separation using both BFS and DFS. Made quickly so does have bugs.
- https://github.com/philipmortimer/AI-Course/tree/main/Programs/Part%201/Degree-of-Seperation-Calculator
- To run, download and extract. Run ".jar" file, ensuring that Java is installed. ".java" files reveal source code.
- Let's try some names!

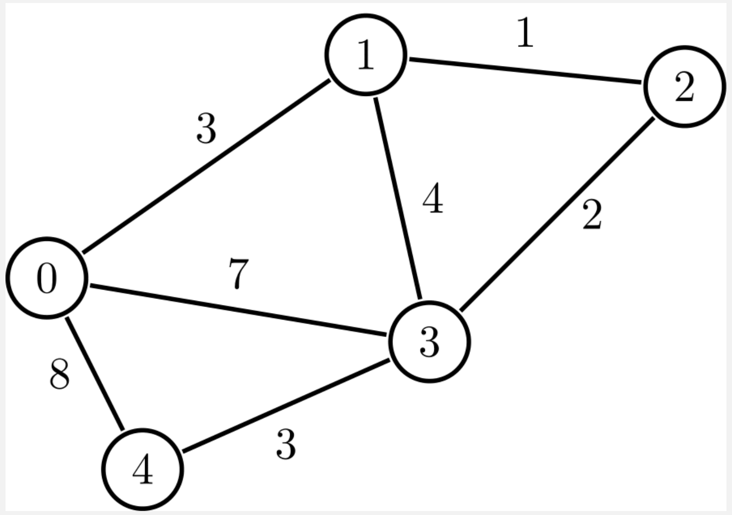
Actor 1	Actor 2
Tom Hanks	Mark Hamill
Zendaya	Scarlett Johansson
Shane Carruth	Kevin Bacon

SEPARATION

Actor 1	Actor 2	BFS		DFS	
		Degree	Time(s)	Degree	Time(s)
Tom Hanks	Mark Hamill	2	1.4	1440	13.4
Zendaya	Scarlett Johansson	2	0.1	370	1.1
Shane Carruth	Kevin Bacon	3	0.6	667	3.1
Kevin Bacon	Shane Carruth	N/A	N/A	N/A	N/A

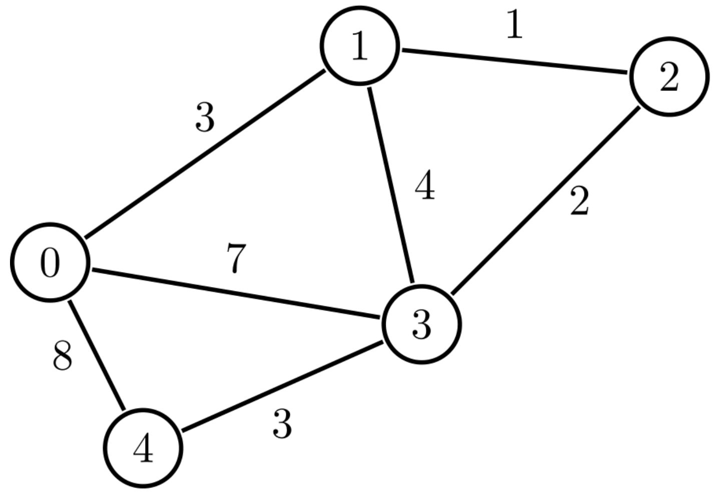
SATNAV SYSTEMS

- Many systems use concepts of DFS and BFS
- GPS systems use many ideas from DFS and BFS to produce navigation route
- BFS alone is too slow when calculating long distance journeys
- We often need algorithms that produce an answer that is good enough
- Satnav is just like a maze or a collection of nodes. Each node can be thought to represent some location
- Up till now, each line on tree has the same cost. This won't always be true – some nodes will be more expensive to reach than others.

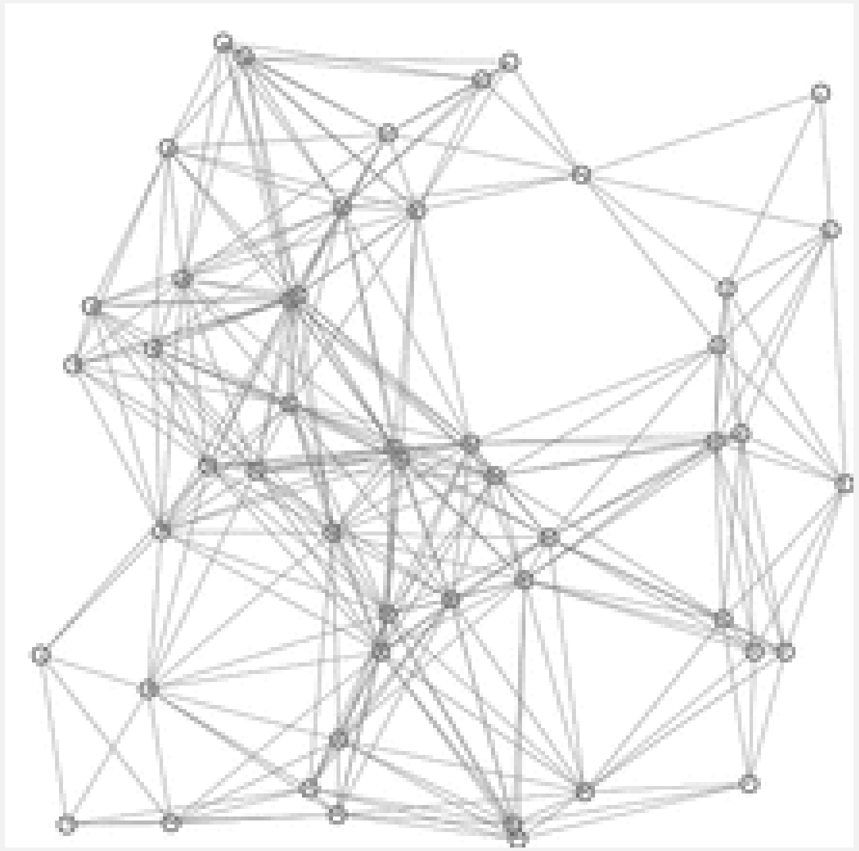


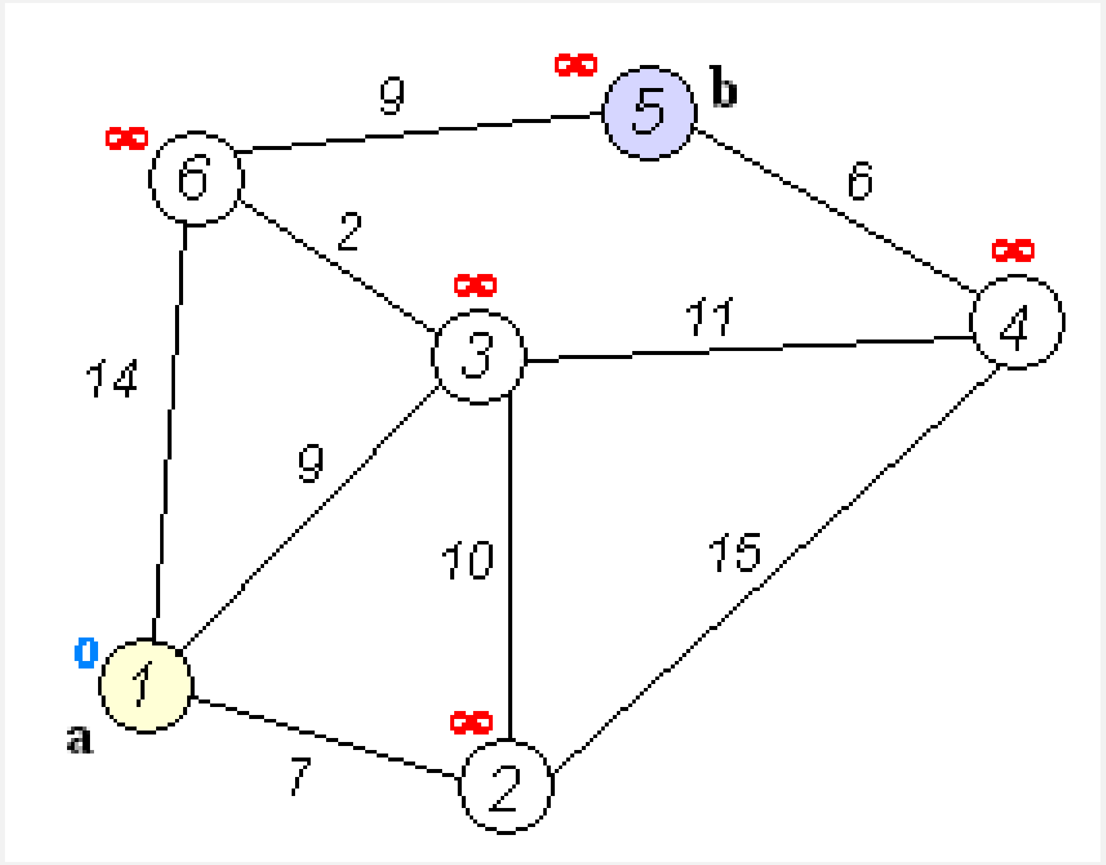
DIJKSTRA'S ALGORITHM

- Always finds optimal solution (analogous to BFS)
- We want to find distance between two nodes. Say node 0 and node 2.
- Initially we label the distance from node 0 to all nodes as infinity.
- We look at neighbours of node 0 and update this distance estimate.
- We then say node 0 has been visited.
- We repeat this process using the node currently perceived to be closest to node 0.
- This repeats until node 2 has been visited.

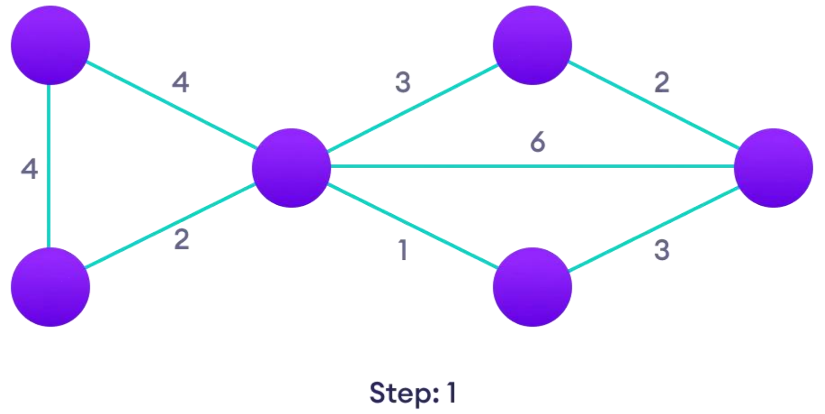


DIJKSTRA'S ALGORITHM



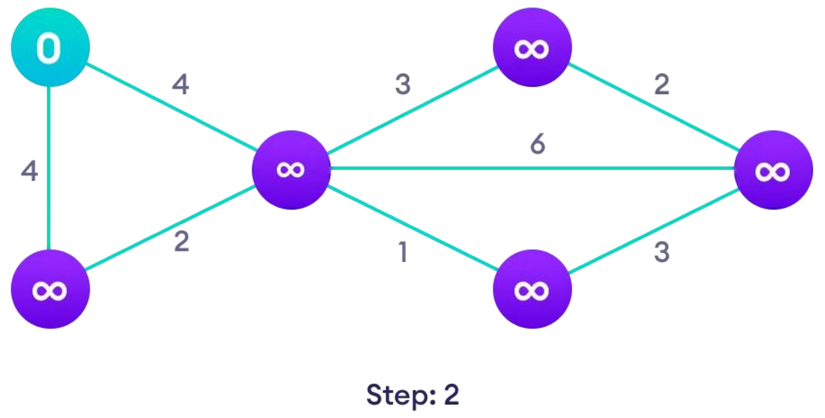


DIJKSTRA'S ALGORITHM



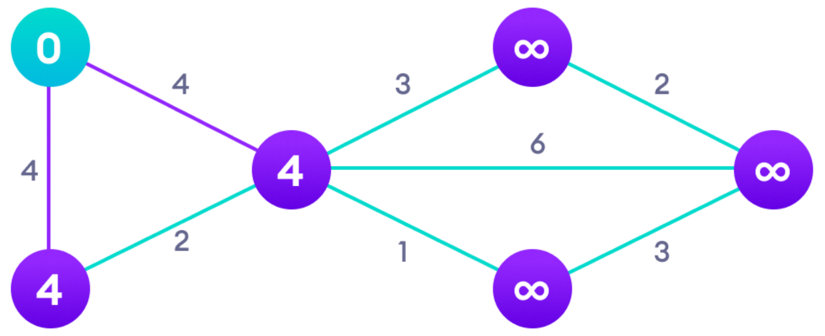
Start with a weighted graph

DIJKSTRA'S ALGORITHM



Choose source node and assign infinity to all other nodes.

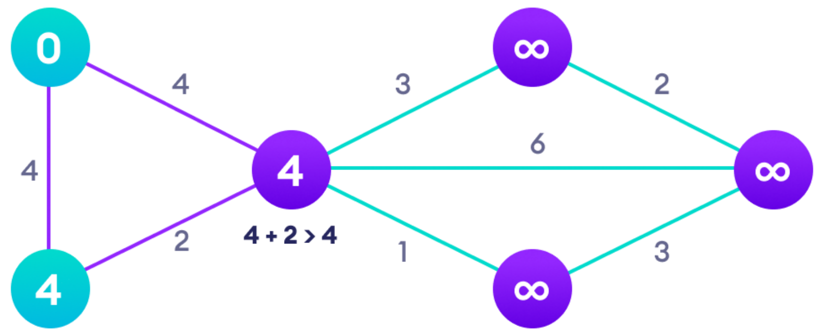
DIJKSTRA'S ALGORITHM



Step: 3

Update distance values for all neighbours of chosen node

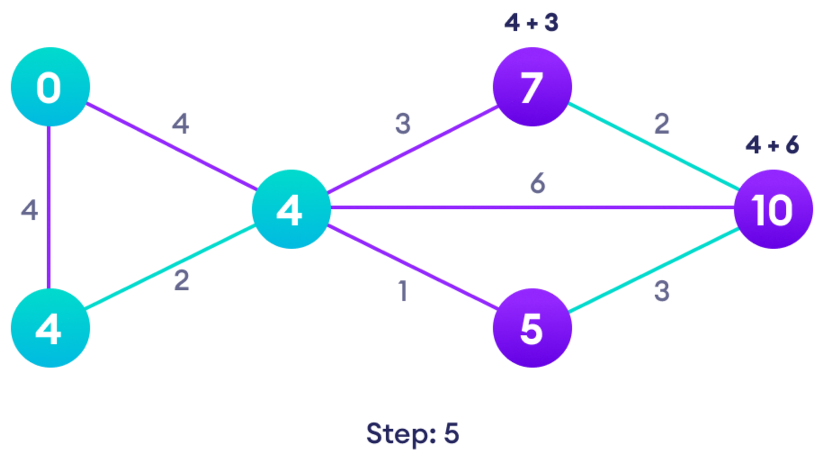
DIJKSTRA'S ALGORITHM



Step: 4

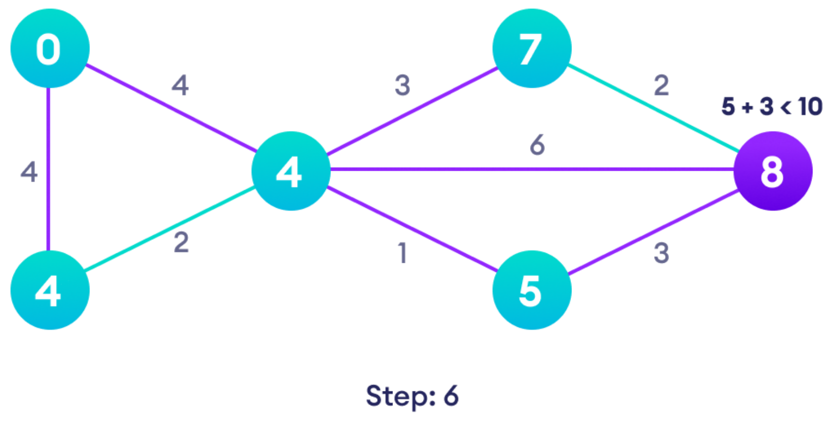
If the path length is more than the current length, don't update the distance value

DIJKSTRA'S ALGORITHM



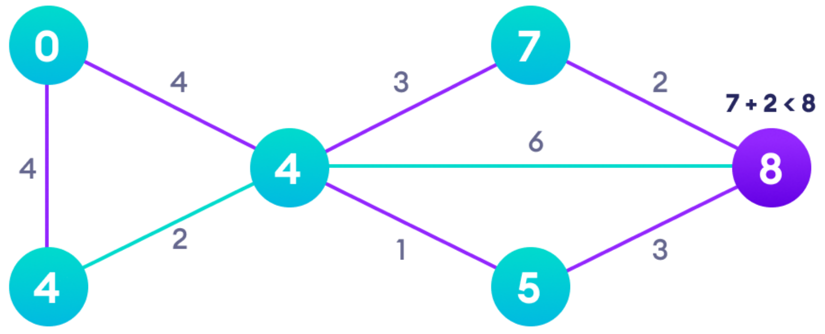
No need to update distance for all visited nodes

DIJKSTRA'S ALGORITHM



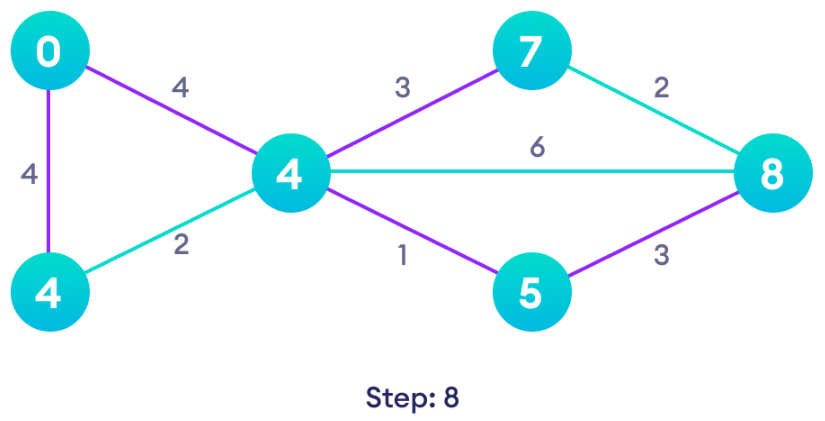
After each iteration, our current node is the unvisited one with the lowest path length.

DIJKSTRA'S ALGORITHM



Step: 7

DIJKSTRA'S ALGORITHM



Repeat until all vertices have been visited (or until destination node has been visited)

DIJKSTRA'S

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

```
S ← empty sequence
u ← target
if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
    while u is defined:                       // Construct the shortest path with a stack S
        insert u at the beginning of S        // Push the vertex onto the stack
        u ← prev[u]                           // Traverse from target to source
```

A* ALGORITHM

- Almost exactly the same as Dijkstra’s algorithm
- In Dijkstra's algorithm, we choose next node based on the cost to get there from the current node
- In A* search, we introduce some heuristic function h, which estimates how far we are from the current goal
- Let g denote the cost of going from the current node to the next node.
- Dijkstra simply selects the neighbour that minimises g.
- A* selects the neighbour that minimises g + h.

A* ALGORITHM

- A* can be thought of as a smart algorithm, as it cleverly chooses which nodes are most likely to actually lead to the goal.
- A* will produce optimal result if heuristic is consistent and admissible
- In practice, Heuristics won't be perfect, but often good enough.
- For maze finding, a good heuristic would be the Manhattan distance of the next node from the end point of the maze. This would result in nodes that take us closer to the goal being expanded first.
- This traditionally offers massive performance boosts and makes it a perfect choice for Satnav systems.

A*

```
function A*(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        heuristic[v] ← heuristic(v)
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min (dist[u] + heuristic[u])

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

```
S ← empty sequence
u ← target
if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
    while u is defined:                       // Construct the shortest path with a stack S
        insert u at the beginning of S        // Push the vertex onto the stack
        u ← prev[u]                           // Traverse from target to source
```

MAZE

- We finally have a range of algorithms which can effectively solve a maze.
- DFS, BFS, Dijkstra's and A* all offer different trade-offs depending on type of maze and solution constraints.
- If you want any path, DFS is often suitable. Why?
- If you want optimal path, BFS, Dijkstra's and A* are all viable. Why?
- If you are using really big mazes, which searches may be most appropriate?

SUMMARY

- AI is hard to understand – concepts are difficult and may take time and reinforcement to understand.
- Depth first search – human like search
- DFS – goes as deep as possible down one path at a time. No guarantee that solution is optimal. Often good when only one solution exists.
- Bread first search – exhaustive search.
- BFS – go down all possible paths one step at a time simultaneously. This guarantees an optimal solution. Can be very memory and processor intensive.
- Both DFS and BFS are key to many AI systems.
- Whole range of problems can be represented as maze / tree like structure

SUMMARY

- Dijkstra’s algorithm searches through all nodes of a weighted graph / tree and finds the shortest path between two nodes.
- Dijkstra’s algorithm calculates the minimum distance of all nodes from a given source node. Nodes with smallest estimated distance are expanded first in a fashion similar to BFS. An optimal solution is always found.
- A* algorithm is very similar to Dijkstra’s, finding a short path between two chosen nodes. Some heuristic function is used to prioritise expanding nodes that are more likely to be closer to destination node.
- An heuristic like Manhattan distance if often used. If a heuristic meets certain criteria, a solution will be optimal. Typically, solutions are not optimal, but are close to optimal.
- A* search is much faster (often 60 times faster) than Dijkstra’s and thus is perfect choice for real world problems like navigation.

QUESTIONS

FURTHER READING

- DFS - https://en.wikipedia.org/wiki/Depth-first_search (contrary to popular belief, Wikipedia is often an excellent resource!) https://www.programiz.com/dsa/graph-dfs
- BFS - https://www.programiz.com/dsa/graph-bfs
- Dijkstra's - https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

  https://brilliant.org/wiki/dijkstras-short-path-finder/
- A* - https://brilliant.org/wiki/a-star-search/

  https://www.youtube.com/watch?v=-L-WgKMFuhE
- General Searching - https://youtu.be/qzhEB8FxxRs