Summary for **Artificial Intelligence Programming** 2021 v1.0

by THOMAS VAN KLINK

Creative Technology – M6: Intelligent Interaction Design

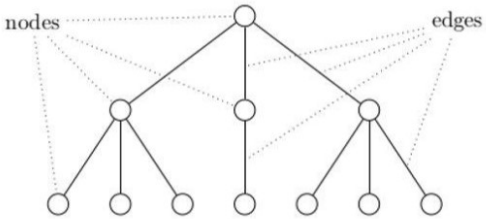
Summary of the content of lecture slides, the course manual and Python book.

Made to fit the sample exam questions.

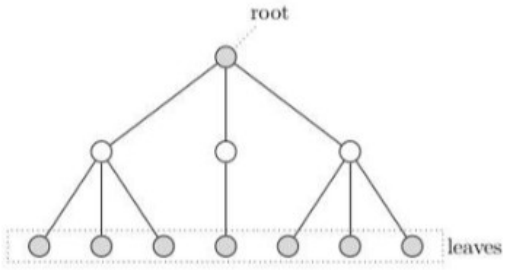
Information taken from the books, manual and lectures.

Search Algorithms

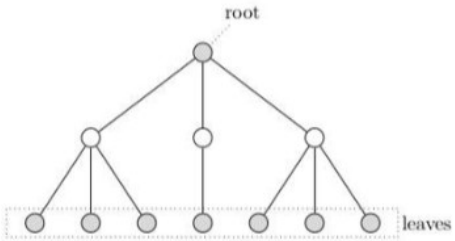
Graphs and Trees



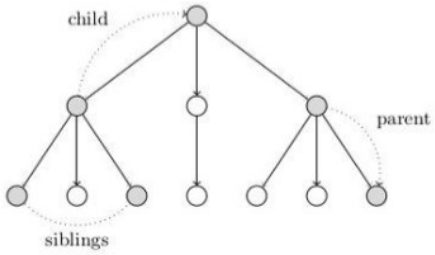
- Graphs have *nodes* and *edges*
- G = (N,E)
- Trees are graphs with certain properties
    - Roots and leaves
    - No cycles 
    - No self-loops 
    - Connectivity 



- Graphs can be directed or undirected

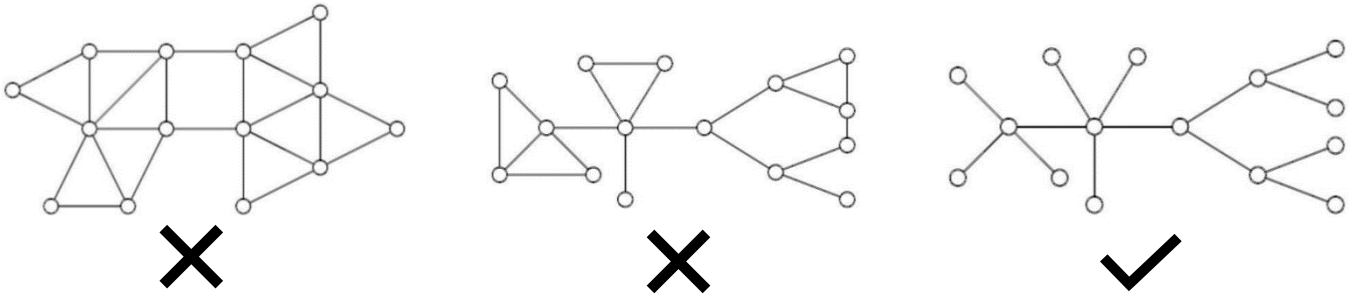


Undirected



Directed

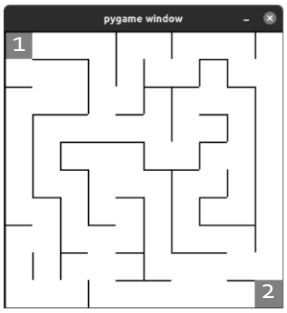
Examples *Tree or not a tree?*

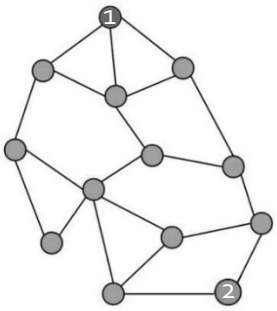


No cycles, no self-loops

In a tree there is a unique path to the root node from every node in the tree.

Graphs and Mazes





Initial position is the root (1)

Children are available next positions

Target (2) is somewhere in the graph / maze

Note: These graphs are not necessarily a tree.

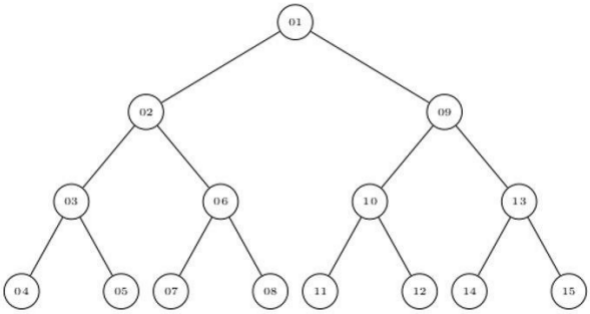
Each position in a maze has max 4 neighbours.

The number of neighbours is further restricted by the walls of the maze.

Search Strategies

There are multiple different strategies to search a graph / tree effectively to find the best route to your target or explore the graph / tree.

Depth-First Search



Explore the graph / maze by going into the "depth first"

Pseudo-code:

```
1. Start at the root

2. As long as the current node
has unvisited children:

- If there is an unvisited
child, move to that child.

- If there is no unvisited
child, go back to your parent
```

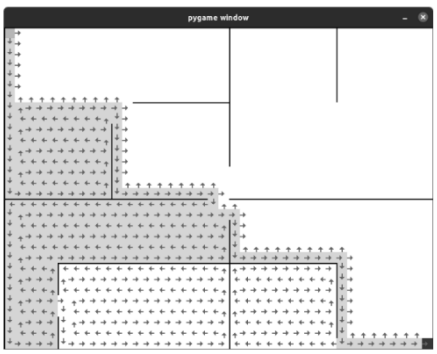
Backtracing (being able to go back in the graph / maze) is made possible by using stacks.

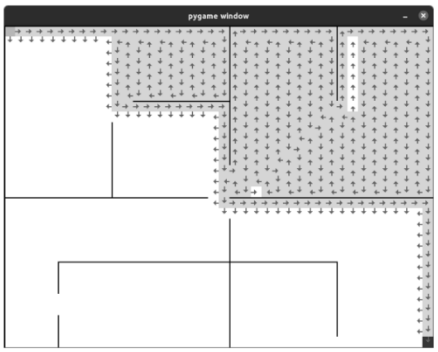
Stacks Data structure that models a *pile* of things

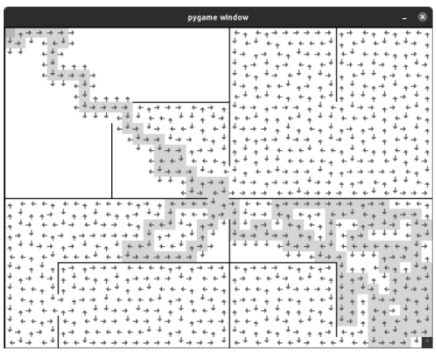
Two important methods:
- Push — Puts an element on top of the stack
- Pop — Gives the element on top of the stack

Uses the principle of LIFO (Last in, First Out)

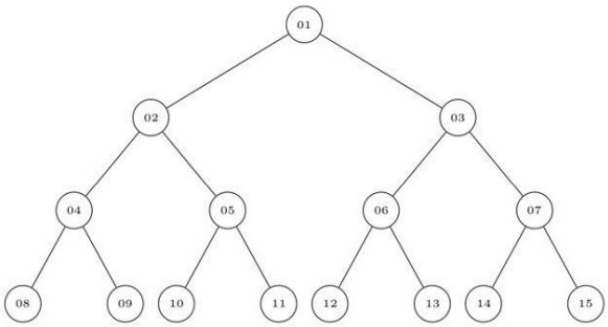
Examples in maze route searches:







Breadth-First Search



Explore the graph / maze by going into the "breadth first"

Pseudo-code:

```
1. Start at layer 0

2. While layer < "height of tree":

- Visit all nodes of level and add children.

- Increase level.
```

Searching per level is organized by using queues.

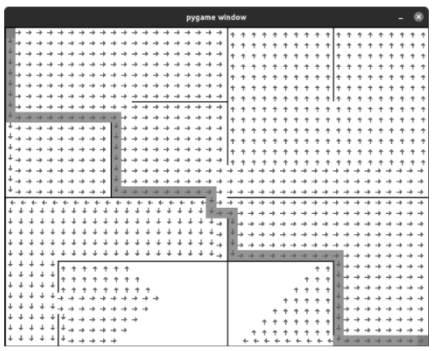
Queues Data structure that model's ordered rows of items.

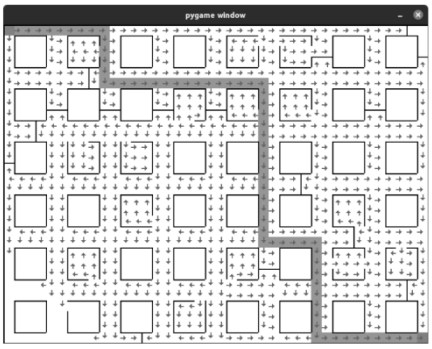
New elements are added to the end, elements are removed at the front.

Two important methods:
- Remove — Removes from the start of the queue
- Add — Adds to the end of the queue

Uses the principle of FIFO (First in, First Out)

Examples in maze route searches:



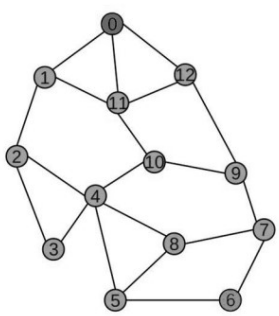


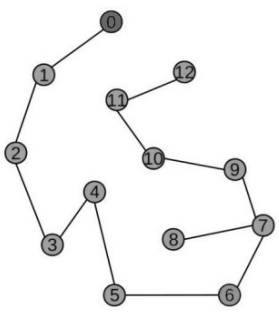
Graphs and Mazes

Each position in a maze has max 4 neighbours.

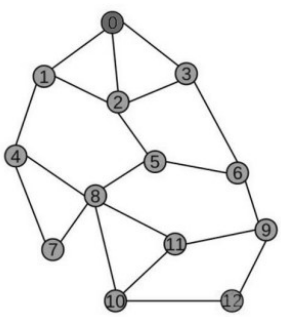
The number of neighbours is further restricted by the walls of the maze.

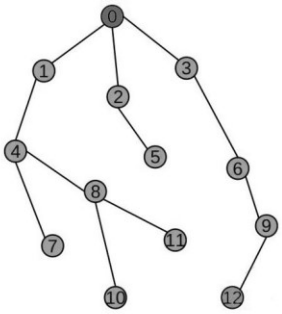
Search starting from the root defines a tree on a graph:





Depth-first search





Breadth-first search

Search Metrics

Different search algorithms construct different trees.

- Nodes/mazes are accessed in different orders
- Efficiency of algorithms varies

Metrics on this are used inside algorithms often to reach your goal the fastest

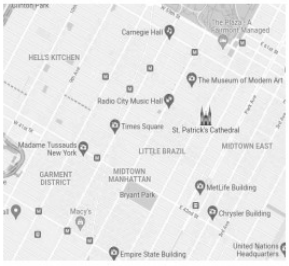
(which is not always the shortest or most efficient)

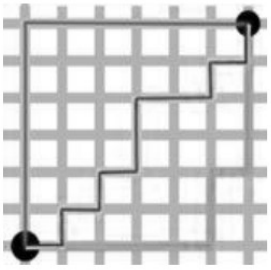
Greedy Search

This method defines a measure to determine the distance to the goal.

For maze solving, which is in a grid, it uses:

Manhattan Distance Measures distance in blocks rather then Euclidian (straight line)





The fastest path to the goals is the one with the shortest Manhattan distance.

This requires finding nodes in a queue that have the smallest distance.

Prioritiy Queue In a sorted list it only requires *one* step to find the next smallest distance.

2	7	11	14	18	23	27	31	33

In an unsorted list, it can take up to the whole length of the list to find it.

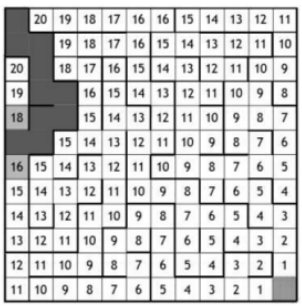
7	23	33	18	31	14	11	2	27

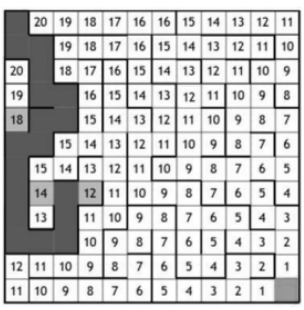
In a priority queue element are taken from the front of a queue just as a regular one.

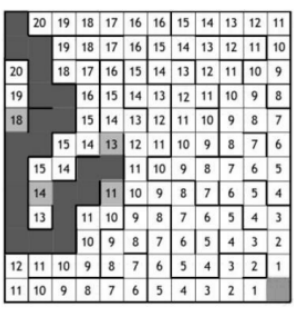
But they are *added accordingly to priority*.

Greedy search uses a priority queue with the states of the nodes to pick the node with the lowest distance to the current one and then continue to repeat this until it reached it's target/goal.

It finds the fastest, but not necessarily the shortest path.

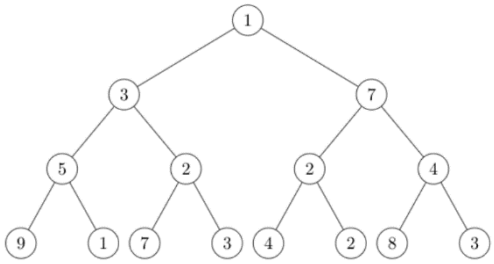




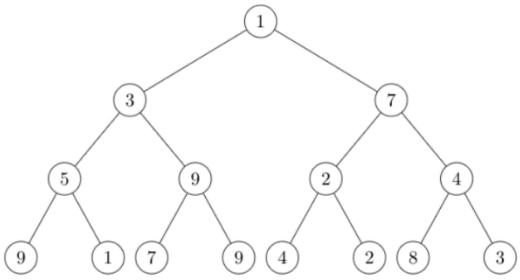


Examples of a greedy search:

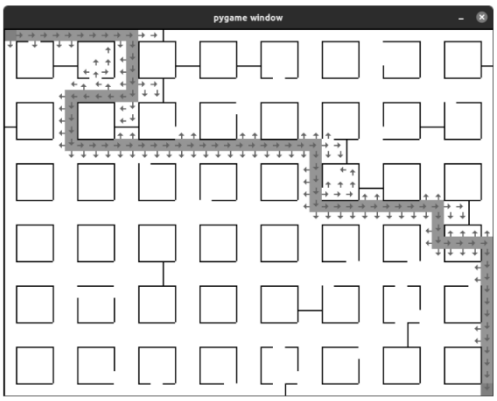
Finding a path with lowest score to a leaf node:



Finding a path with highest score to a leaf node:



In a maze:



A* (A-star) Search

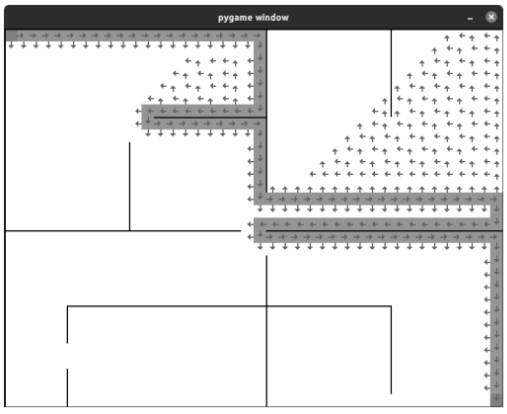
Greedy does not find the shortest path.

A* considers the path that has been travelled so far in addition to the Manhattan distance.

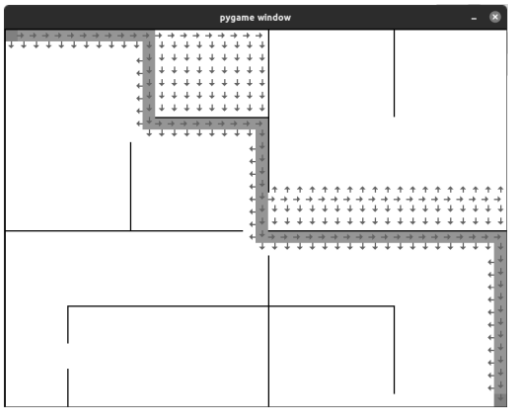
In considering this the scores of nodes might change and are then reinserted.

This algorithm is more provides shorter routes and explores less of the possible routes then Greedy.

Example in a maze:



Greedy



A*

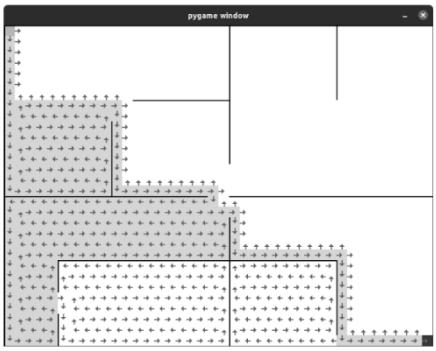
Searching in Grids

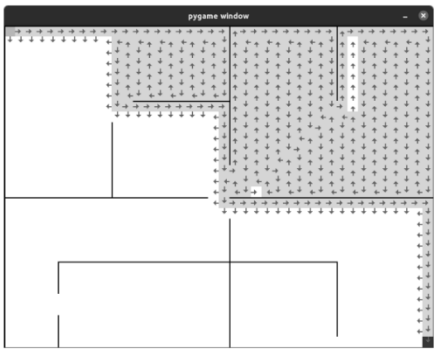
You need to be able to explain why the different algorithms generate certain routes in tree different settings of a rooms like mazes, obstacles and rooms.

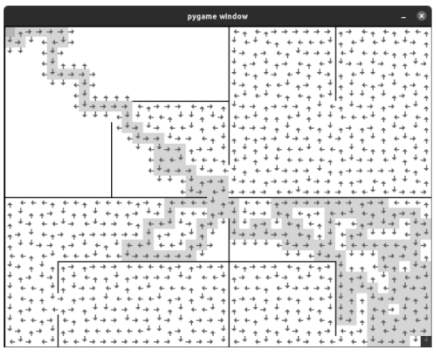
Depth-first search

Not very efficient, explores a big part of the room, tends to go up and down before moving to the side.

This is because it traces a route all the way to the last child before returning.



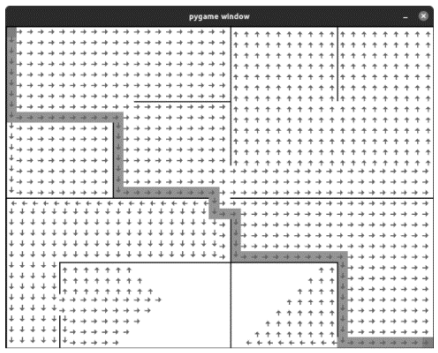


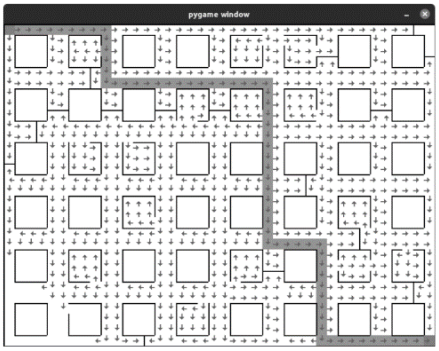


Breadth-first search

Not very efficient, explores a big part of the room, tends to go left and right from previous node direction before moving up or down.

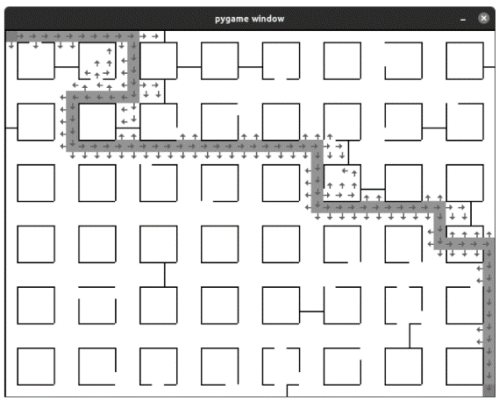
This is because it visits all nodes of a *level* in a tree before increasing the level and moving further.





Greedy Search

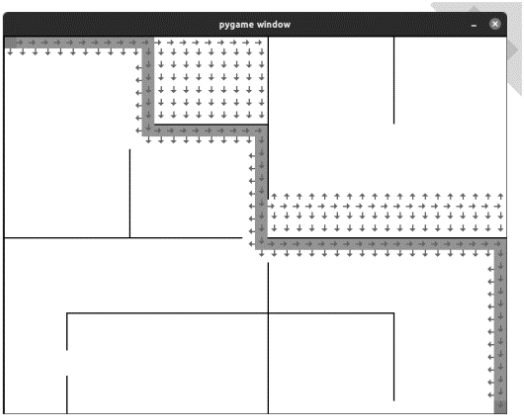
A more efficient algorithm by using the (Manhattan) distance to the target in the calculation.



Explores a much smaller area when comparing nodes, calculates the fastest not the shortest route.

A* (A-star) Search

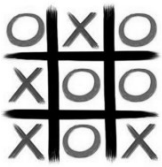
In addition to using the Manhattan distance this algorithm considers the path it already has taken.



Explores more strategically and efficient, bridges gaps by exploring in straight lines left/right or up/down.

Minimax Algorithm

Minimax in a game



Tic-tac-toe (American English), **noughts and crosses** (Commonwealth English), or **Xs and Os** (Irish English) is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with *X* or *O*. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players.

And still complex enough to make our brain melt when implemting it in code lmao

For a game like this you can build a tree with different game states represented in nodes.

Algorithms can search such a tree for a game like Tic-tac-toe (melk-kaas-boter) very well as it has perfect information (no random input/results) and is turn based.

The algorithm

A game of Tic-tac-toe can be represented in a tree / graphs with different game boards.

The nodes in this tree are to be a X or an O, representing the two players of the game.

One player is the max-player, it tries to use the minimax strategy to win the game.

The other player is it's opponent, the min-player.

The algorithm takes three arguments: the current node (X/O), the current player and the max-player.

To make your change of winning as high as possible the algorithm is to decide a route of moves which is the least favourable for your opponent (*maximimum over their minima)*.

One of the branches of the tree provides this route of moves and we need to find that one. But we only know this when we have calculated the lower parts of the tree.

By assuming the opponent uses the same algorithm as us we can determine outcomes of future moves of the opponent without it playing them. This is done by recursing through the tree. (It tries moves).

These moves then receive a score (-1,1,0) based on how favourable they are.

Based on these scores it finds the best route and plays the according move. In the next turns the whole process repeats until the game is won.

In Tic-tac-toe if both players play perfectly a tie is possible but in practise this algorithm is very hard to beat as a human player.

Recursion

As stated, the minimax algorithm relies heavily on recursion.

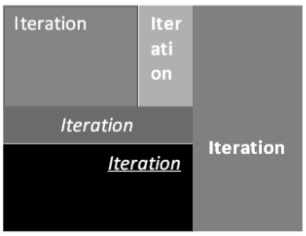
It is a solution to solving high complexity problems by:

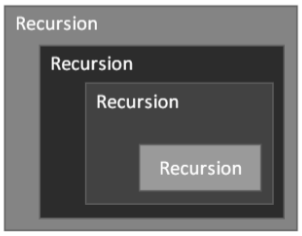
- Iteration Dividing problems, solve them individually, then merge them.
- Recursion Reduce problems to a smaller instance of itself.

The calculation you do occurs repeatedly to simplify the task it needs to do.

It starts from a base case, which then further refines/simplifies itself.

Useful to progress through a graph while (re)calculating nodes in a non-linear fashion (forward/backward)





Memoisation

This technique can be used to increase the performance / effectiveness of an algorithm.

Some algorithms including minimax calculate scores for nodes each time they iterate over a node.

In many routes they then pass over notes that already have been calculated before.

With memoisation you save the scores of new nodes you pass over into a dictionary or other storage type in the programming language and if you the happen to pass it again in future iteration of the algorithm you use the score that was previously calculated and not calculated the same node and score again.

This saves unnecessary computing and with that valuable time.

Heuristics

By definition:

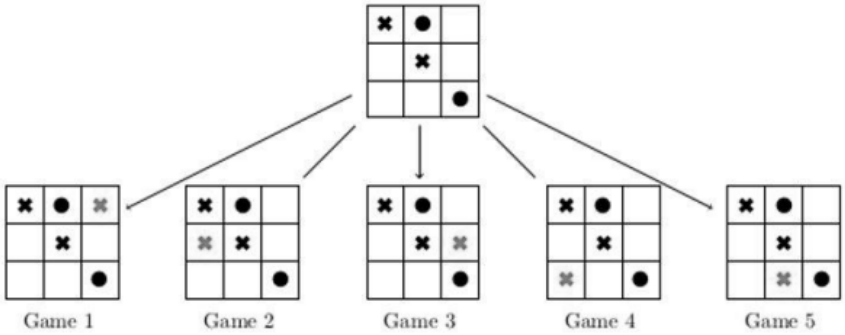
"Is an approach to problem solving that employs a practical method that is not guaranteed to be optimal, perfect or rational, but is nevertheless sufficient to reach an immediate or short-term goal or approximation."

By using heuristics, you can save certain game situations that would benefit the goal of your search (which in case of a game is of course winning).

They might not always be completely accurate, but they suffice and are valuable.

From certain game parameters you define states that are beneficial for the goal and describe them in a formula. The score generated by the formula can then be used per node to be used in the Minimax algorithm to determine the best move to make.

This technique cannot be combined with memoisation as you recursively need the new calculated heuristics scores which are not stored in the dictionary.



$$Eval(s) = \underbrace{(3 \cdot X_2(s) + X_1(s))}_{\text{good for us}} - \underbrace{(3 \cdot O_2(s) + O_1(s))}_{\text{good for opponent}}$$