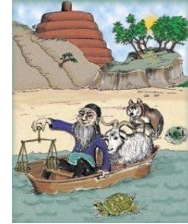


1

Man, Wolf, Goat and Cabbage

- A man owns a wolf, a goat and a cabbage
- He has to cross a river in a boat, but he can only take one item at a time
- If left alone, the wolf will eat the goat or the goat will eat the cabbage
- How can he cross the river without anything being eaten?



2

Searching

- Many problems can be solved by searching for a good solution in a *search space*
- Examples of such problems are
 - 8 puzzle problem
 - Maze
 - TSP



3

3

Representing a Search Space

- A convenient way of representing search spaces is as a *graph*
- A graph consists of nodes and links
 - nodes represent states in a problem solving process
 - links represent transitions or relationships between nodes



4

4

Trees

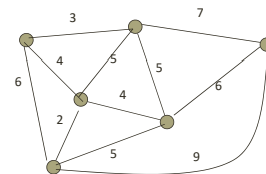
- A special case of a graph is a *rooted tree*
- A rooted tree has:
 - a unique node (the root) from which it is possible to reach all other nodes in the graph
 - at most one link between any two nodes
 - no cycles
- Trees are common in search problems



5

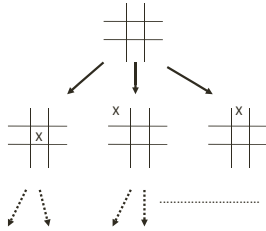
5

Example Graph - Travelling Salesman Problem



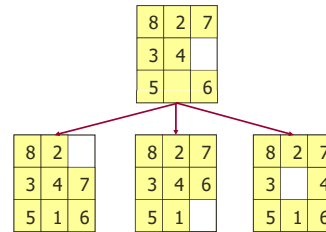
6

Example Tree - TicTacToe



7

8-puzzle



8

8

Representing a problem with a graph

- Typically when representing a problem with a graph, we have:
 - Astart state**(where we are at the start, for example the starting position of a game of chess)
 - Intermediary states** (where we are now, for example the current position of the board of chess)
 - one or more **goal states** where the search for a solution terminates (for example a check-mate position in chess)

9

Forward and Backward Chaining

- Defining the "start state" can be done in two main ways
 - Forward Chaining**
 - start from what you know....
 -work towards a solution
 - Backward Chaining**
 - start from the optimum solution....
 -work out what inputs yield this solution

10

Searching a Graph

- We want *systematic* methods for searching graphs which yield optimum solutions with as little computational effort as possible
- Two basic approaches:
 - depth first search
 - breadth-first search

11

Search Machinery

- A list of *closed* nodes
 - nodes which have been evaluated and their children placed on the open list
- A list of *open* nodes
 - nodes which have not been evaluated yet
 - the first item on this list is evaluated next
- Note that nodes are usually stored as tuples, with their parent node attached. This enables the path to a solution (when we find it) to be reconstructed.
- Before placing a node on the open list, we need to check that it is not already on the closed list

12

Depth First Search

- When we evaluate a state and derive its children, we put these children at the *front* of the open list
 - the list operates as a stack
- The search goes deeper into the space whenever possible and backtracks only at dead ends
- Can be very fast, especially if there are many solutions, but all quite deep in the space
- Uses a small amount of memory
- May fail to terminate even if a solution exists
- Not guaranteed to return the shortest path

13

Implementing Depth First Search

- Implementing depth first search is very easy in any modern programming language:
 - Evaluate the current state using a function Recursively call the
 - evaluation function for each of the children states
 - The recursion mechanism takes care of all the necessary book-keeping and allows the solution path to be generated as the recursion unwinds

14

Breadth First Search

- When we evaluate a state and derive its children, we put these children at the *back* of the open list
 - the list operates as a queue
- The search systematically goes across each level
 - Always finds a solution if one exists
- Always finds the shortest path
- Can be very slow if the branching factor is high
- Can use a huge amount of memory (the open list can get enormous)
- Requires a bit more work to implement

15

Refinements

- There are several possible refinements to straightforward depth-and breadth-first searching:
 - Hybrid breadth/depth using iterative deepening
 - Mixed forward/backward chaining Heuristics
 - Heuristic

16

Iterative Deepening

- A way of carrying out a breadth-first search without excessive memory needs and getting some of the advantages of both methods
- Do a depth first search but limit the depth of the search
- If no solution is found, repeat the depth first search with an increased depth limit

17

Mixed Chaining

- If what we are interested in is finding the optimum path between two (known) nodes, it may make sense to combine forward and backward chaining
 - i.e. expand both simultaneously and try to meet in the middle by matching the open lists
- This is because the number of nodes expands exponentially

18

Heuristics

- AI uses heuristics to address two main difficulties:
 - A problem may be ambiguous, with no exact solution A
 - problem may have an exact solution, but the computational cost of finding it may be too large.
- This second problem is the subject of heuristic search

19

Complexity Considerations

- Since computers are so powerful these days, can't simple breadth and depth first searches solve all our problems?
 - even if not today, maybe tomorrow?
- Unfortunately many problems are of a complexity which we will *never* be able to solve by brute-force methods
 - it is estimated that there are 10^{120} possible board positions in chess (including symmetries).....this is more than the number of molecules in the known universe!

20

Problems Which Grow Exponentially

- Many problems of considerable practical importance are of exponential complexity
 - the travelling salesman problem
 - integer programming optimisation
 - various graph colouring problems (e.g. the 3-colour problem)
 - packing problems (i.e. given some finite storage spaces and some items, what is the best way of packing the items in order to maximise capacity?)

These kinds of problems explode and cannot usually be solved by brute force methods

21

Best First Search

- To tackle these kinds of problems, we need to find some way of "steering" our search towards promising parts of the search space.
- We do this by computing a heuristic estimate of the promise of states put on the open list.
- The list is held in a sorted form, so that the "best" state is expanded first
 - hence the term "best first" search

22

Example of heuristics

- The 8 puzzle is a reduced version of a fun children's puzzle.
- We have a 3 by 3 grid with 8 tiles numbered one to eight, and one space.
- The object of the game is to move the tiles around until you reach a "winning formation", i.e. the numbers in order running clockwise around the puzzle.
- The optimum solution is the one which takes the smallest number of moves.

23

Winning Formation of the 8-Puzzle

1	2	3
8		4
7	6	5

24

A Simple Heuristic for the 8-Puzzle

• A simple heuristic might be as follows:

- $h1(n)$ = number of tiles out of place

25

A Simple Heuristic for the 8-Puzzle

• A better heuristic might be:

- $h2(n)$ = the total city-block distance of all tiles to their correct place

26

How Do We Choose a Heuristic?

- No exact answer can be given
 - heuristics are highly dependent on the problem and require some skill to identify
- A trick is to consider a *relaxed version* of the problem (i.e. with some constraints weakened or removed).

Often an algorithm which solves the relaxed version is a good (and admissible) heuristic for the actual problem

27

Trade-Offs

- Heuristic search involves several trade-offs:
 - space vs time cost of searching vs cost of calculating the heuristic
 - a very complex heuristic might use more computational power than it saves
 - cost of accepting non-optimality vs additional development or running costs to reach optimality

28