IT00DP82-3007 Artificial Intelligence and Machine Learning

peter.hjort@metropolia.fi

Note: please include string IT00DP82 in the subject field of any email correspondence

Search -
Introductory example

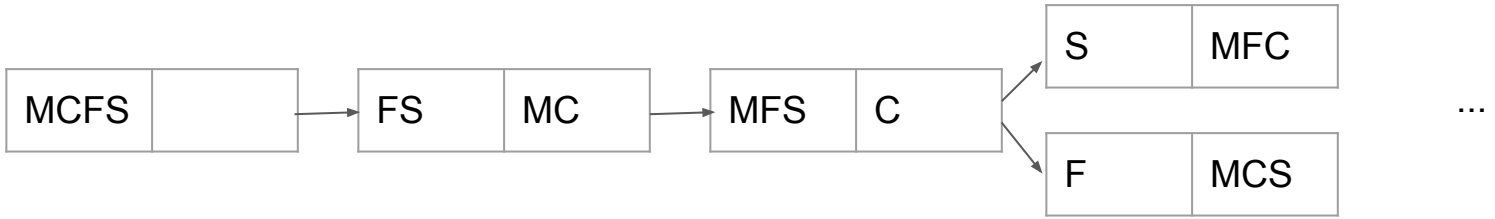
Chicken crossing

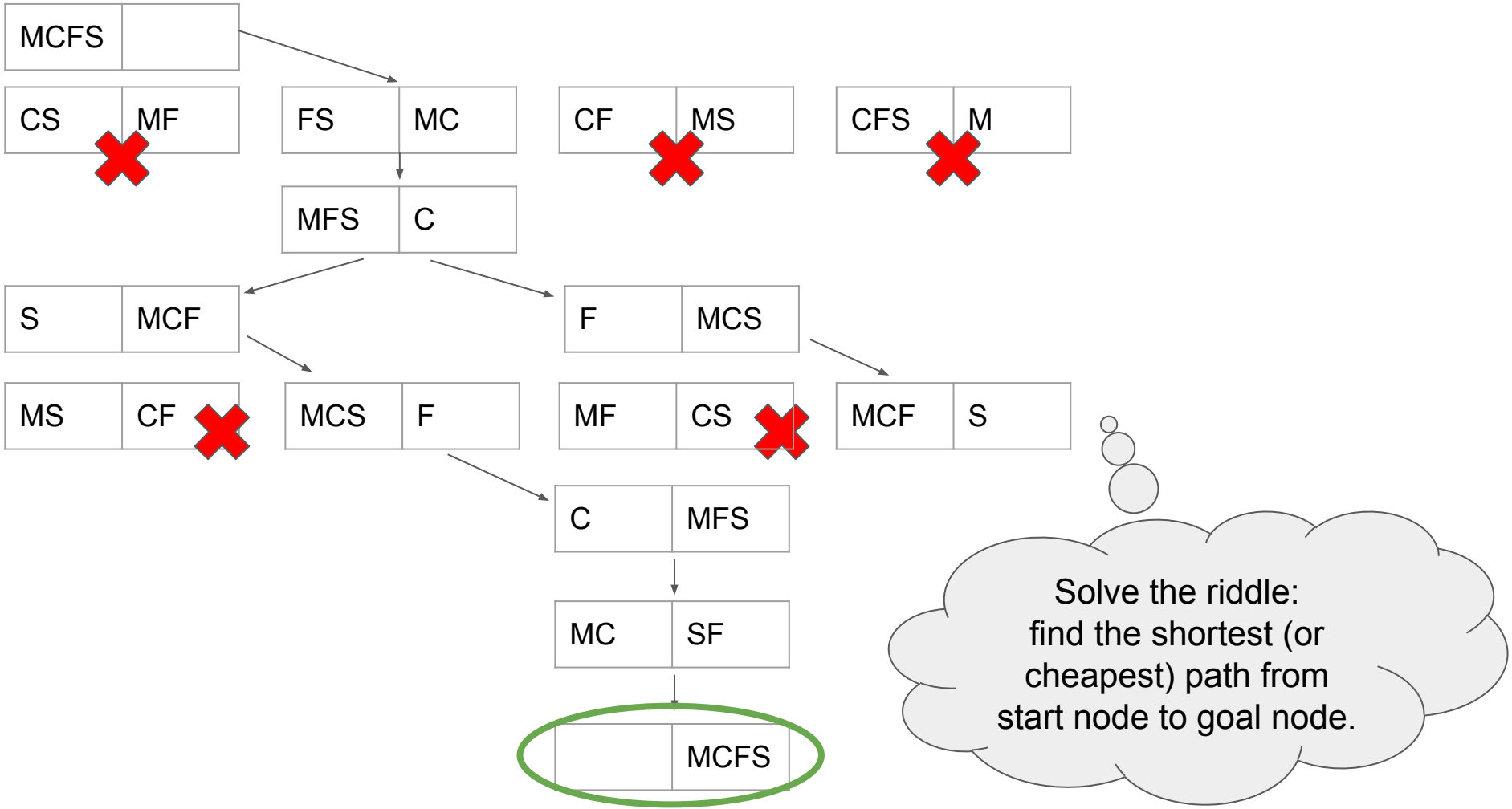
A man (M) has to get a fox (F), a chicken (C), and a sack of corn (S) across a river. He has a rowboat, and it can only carry him and one other thing. If the fox and the chicken are left together, the fox will eat the chicken. If the chicken and the corn are left together, the chicken will eat the corn. How to get across with all objects?

Representation

Often a useful first step in solving a problem is to try and find a reasonable representation for it.

For the chicken crossing problem, one possibility to describe a specific situation (or state) is to list the objects on each side of the river. An action - the man moving himself and something else across the river could the be described as a transition between two states, for example (M - man, C - chicken, F - fox, S - sack of corn):




MCFS
CS MF
FS MC
CF MS
CFS M
MFS C
S MCF
F MCS
MS CF
MCS F
MF CS
MCF S
C MFS
MC SF
MCFS
Solve the riddle:
find the shortest (or
cheapest) path from
start node to goal node.

Search problem solving

State space - all the possible states reachable from the initial state.

Transition - moving from a state to another state without visiting other states.

Search through state-space - Find a path (sequence of transitions) from the *start state* to a *final state*. In some problems different transitions might have different **costs** (like roads between crossings might have different lengths).

Note that often the next states are expanded only when a node in the tree is visited - it doesn't make sense to expand just in case. Also, links to nodes that have already been visited are not usually recorded - the structure is maintained as a tree.

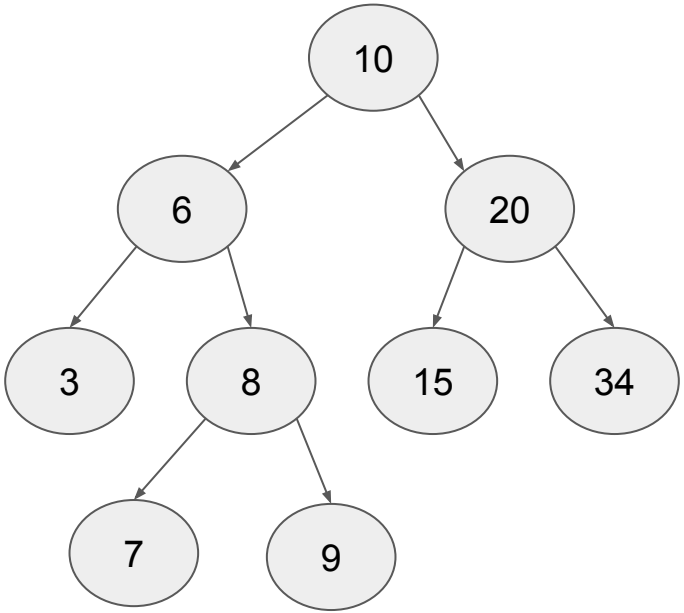
Search methods

Uninformed / informed search

An algorithm for searching through the search tree that has no additional information about the problem is call **uninformed**. There is nothing that directs the search process to the right direction - algorithm is completely generic. Examples of uninformed search (or traversal) methods are **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**.

An algorithm that has information to guide the search is called **informed**. There is some case-specific information that helps the algorithm in finding the solution. The additional information is often given in a form of a **heuristic function**. Example of an algorithm that uses heuristics is **A***.

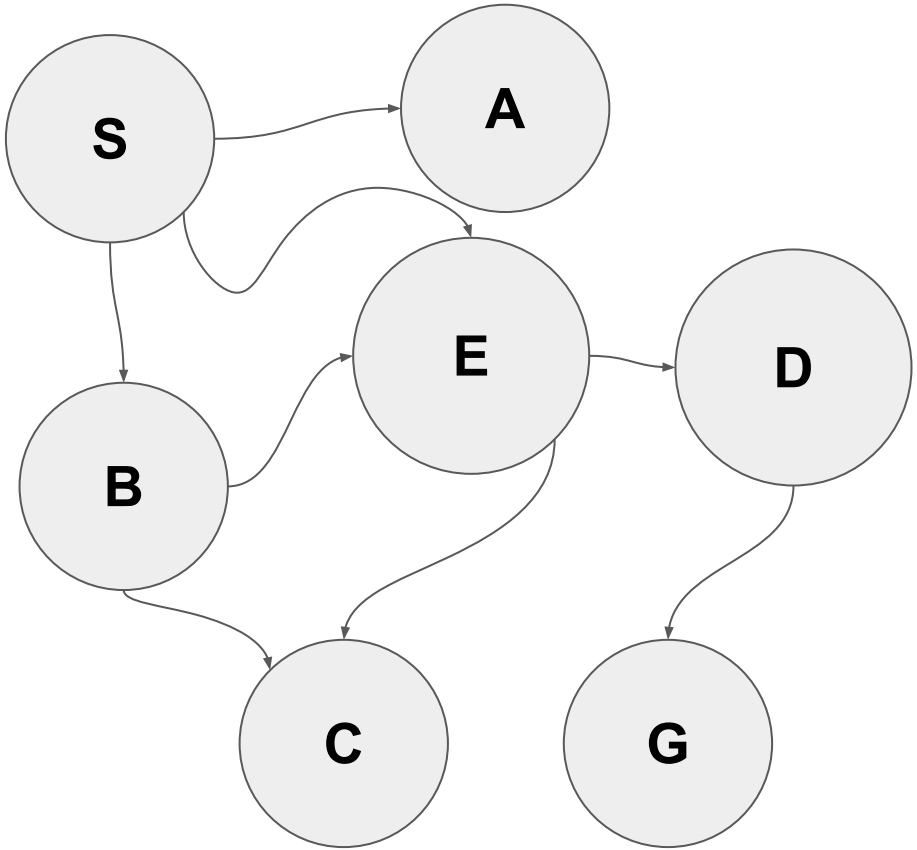
Uninformed search - DFS and BFS



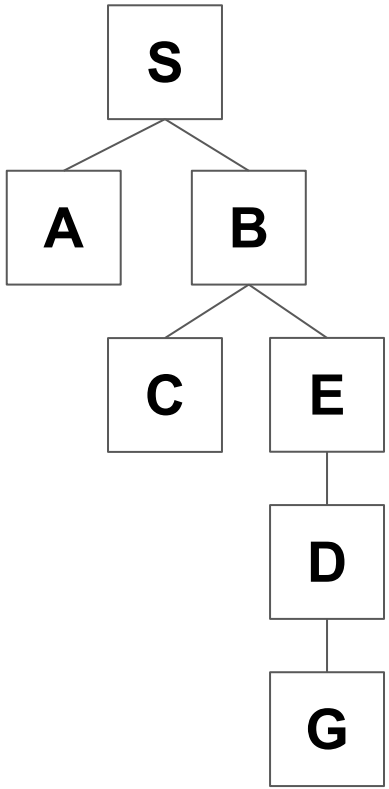
DFS: Starting from root, expand the list of followers of a node, and put the paths into a list (the fringe). Pick next node to be expanded by selecting the longest (deepest) path. Break ties with some criteria (here int ordering).

10 (10-6,10-20) → 6 (10-6-3,10-6-8,10-20) → 3
(10-6-8,10-20) → 8 (10-6-8-7,10-6-8-9,10-20) → 7
(10-6-8-9,10-20) → 9 (10-20) → 20
(10-20-15,10-20-34) → 15 (10-20-34) → 34

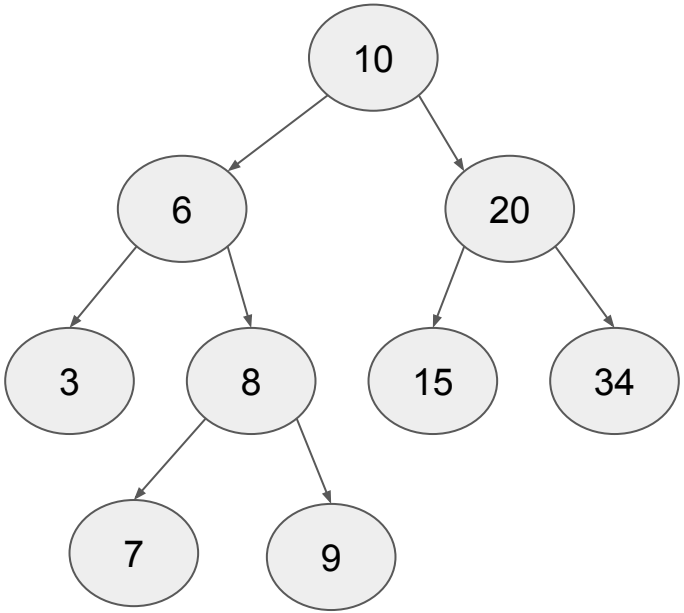
DFS example for a graph



Fringe
S
S-A, S-B, S-E
S-B, S-E
S-E,**S-B-C**,S-B-E
S-E,**S-B-E**
S-E,**S-B-E-D**
S-E,**S-B-E-D-G**



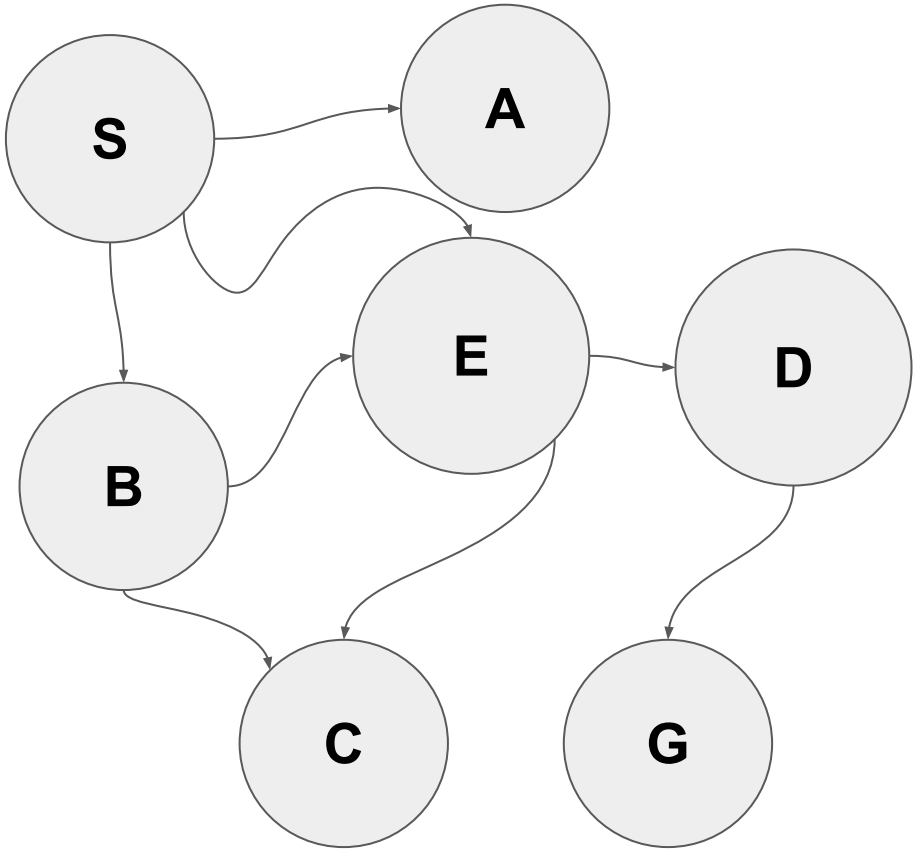
Uninformed search - DFS and BFS



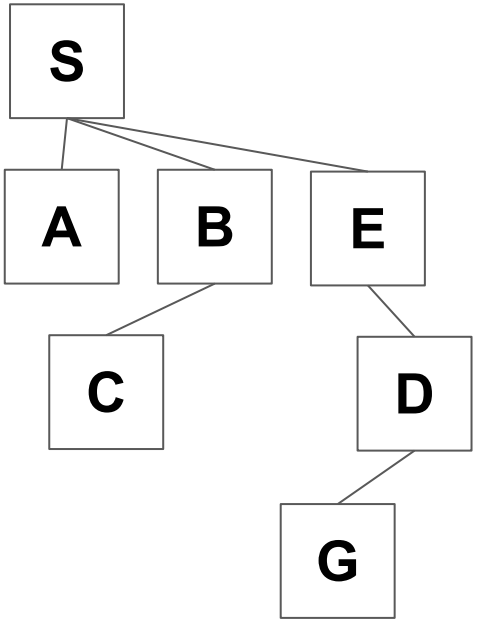
BFS: Starting from root, expand the list of followers of a node, and put the paths into a list (the fringe). Pick next node to be expanded by selecting the shortest (shallowest) path. Break ties with some criteria (here int ordering).

10 (10-6,10-20) → 6 (10-6-3,10-6-8,10-20) → 20 (10-6-3,10-6-8,10-20-15,10-20-34) → 3 (10-6-8,10-20-15,10-20-34) → 8 (10-20-15,10-20-34,10-6-8-7,10-6-8-9) → 15 (10-20-34,10-6-8-7,10-6-8-9) → 34 (10-6-8-7,10-6-8-9) → 7 (10-6-8-9) → 9

BFS example for a graph



Fringe
S
S-A,S-B,S-E
S-B,S-E
S-B-C,S-B-E,**S-E**
S-B-C,S-B-E,S-E-C,S-E-D
S-B-E,S-E-D
S-B-E-D,S-B-E-C,**S-E-D**
S-B-E-D,S-B-E-C,S-E-D-G
S-B-E-D-G,**S-B-E-C**,S-E-D-G
S-B-E-D-G,**S-E-D-G**



Watch this…

University of California at Berkeley AI course video (lecturer is Pieter Abbeel):

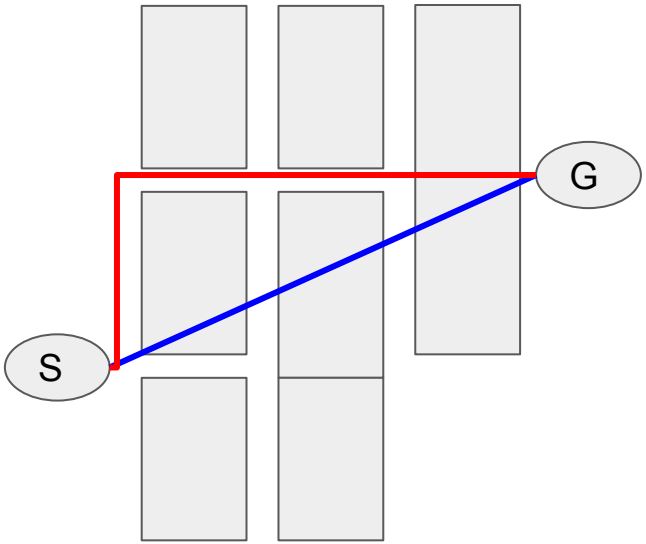
https://www.youtube.com/watch?v=-Xx0QSFYfIQ&list=PL7k0r4t5c108AZRwfW-FhnkZ0sCKBChLH&index=3&t=0s

at 38:05 (search trees), 53:00 (DFS) and 1:00:45 (BFS)

Heuristic function

The purpose of the heuristic function is to include some information about the actual problem in the search algorithm. In search problems heuristic function gives an estimate of the cost from a state to a goal.

There are some mathematical conditions the function must fulfill, most important of which is that is must give an optimistic estimate, ie. not overestimate the remaining cost from a state to goal.



Heuristic function from S to G could be:

- Distance (blue line)
- Manhattan distance (red line)

Both are <= actual distance.

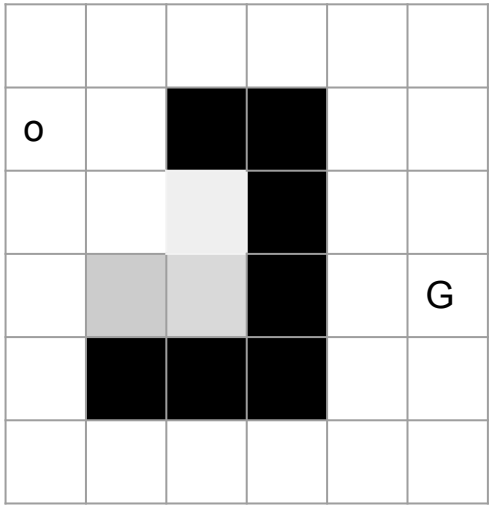
Square of straight line distance, or 2 * Manhattan distance would not make a valid heuristic.

Informed search - Greedy Best-First Search

GBFS overall operation is like DFS or BFS - new nodes to consider are inserted into a fringe, and nodes are selected for expansion from the fringe.

In GBFS heuristic function value *h(s)* is used as priority for a new node *s* in the fringe.

Items in the fringe are kept in priority order (from smallest h(s) value to largest). This means that each time a new node is selected for consideration, it will be the one with lowest value. This behaviour leads to GBFS stubbornly explore the direction with low heuristic value. Better overall behaviour is reached with more balanced approach.



Informed search - A*

In A* search the basic principle is again the same as in the algorithms discussed earlier - in case of A* the value with which nodes are arranged in the fringe is

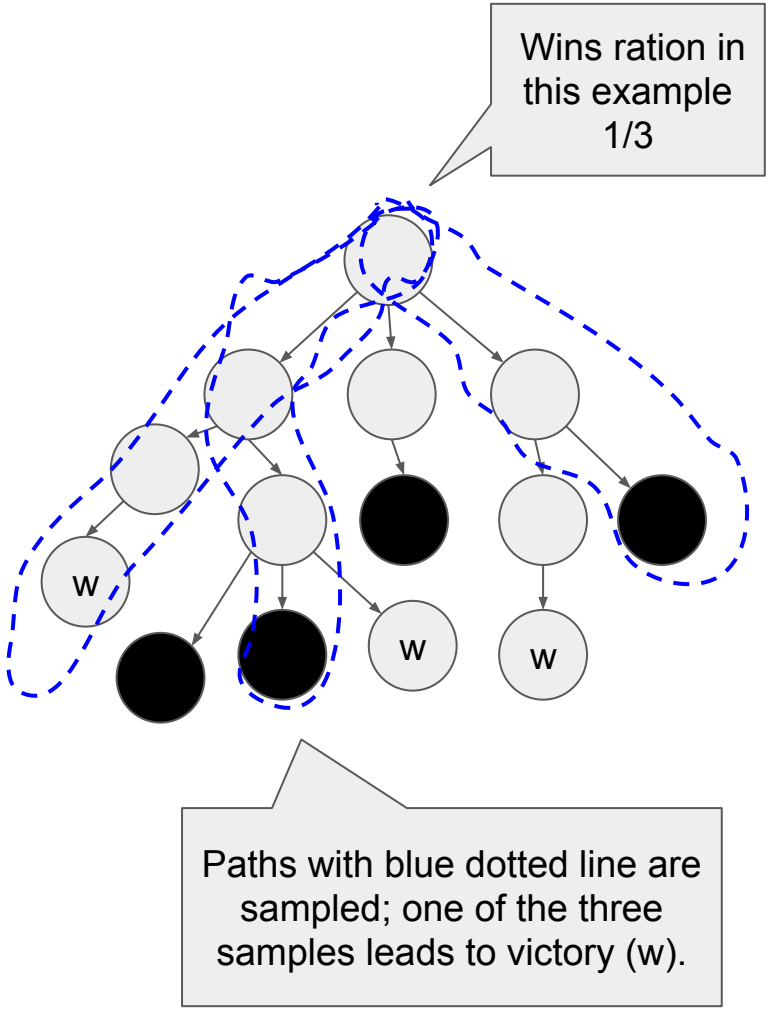
f(s) = *g(s)* + *h(s)*, where *g(s)* is the accumulated cost to *s* (that is known for sure), and *h(s)* heuristic function.

Value *f(s)* balances between the cost accumulated so far with the estimated remaining cost. In practise it means compared to GBFS that following the direction towards lowest *h(s)* will stop unless real progress is made (because *g(s)* grows). In that case A* backtracks to a state visited earlier, and continues search there.

Monte Carlo Search Trees (MCTS)

Exhaustive search is, like discussed, not an option for realistic systems. For some problems, like the game of Go, an effective heuristic function is hard, or impossible, to design.

Idea: how about trying to create a less perfect view of the game state-space by using randomness. Generate (sample) random (moves picked by random in basic case) gameplays, and use wins ratio as a way to decide in which parts of the tree to pay more attention. (Real MCTS implementations are more elaborate, see https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/).



Watch this…

University of California at Berkeley AI course video (lecturer is Pieter Abbeel):

https://www.youtube.com/watch?v=Mlwrx7hbKPs&list=PL7k0r4t5c108AZRwfW-FhnkZ0sCKBChLH&index=4&t=0s

at 19:00 - 42:00 (A*) and

1:02:50 - 1:06:40 (comparisons and demos of search algorithms)

Games and search

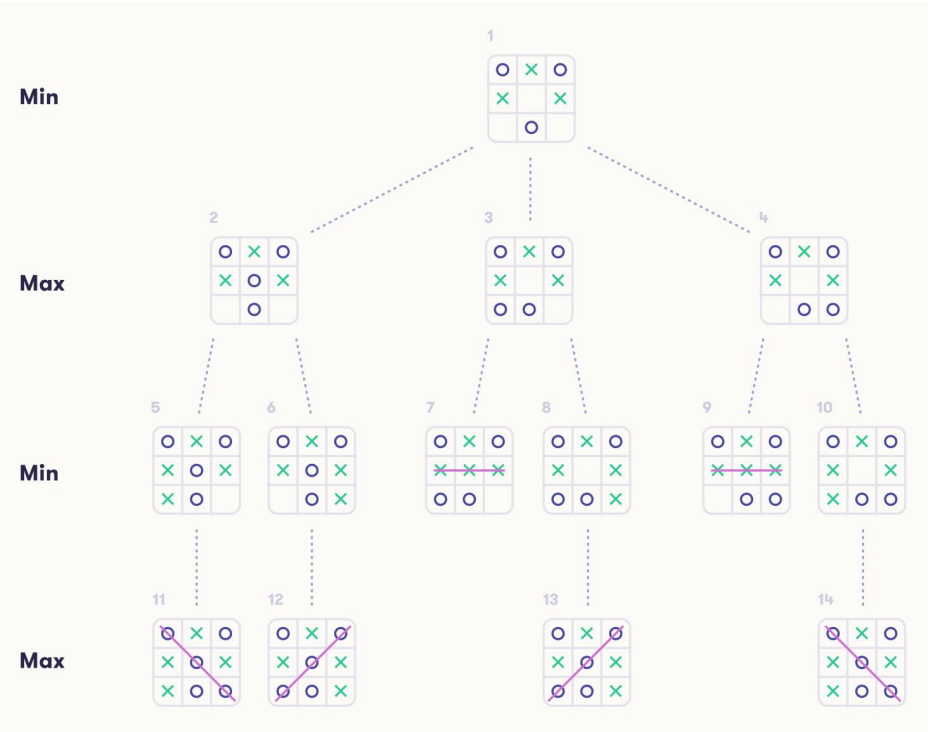
Games and searching

Games are a traditional area where AI based solutions have been studied. Some reasons for this are:

- Games are an example of small worlds; even quite complicated games can be modeled
- Playing games, such as chess, well has been considered a sign of intelligence

Games can often be modeled in the same way as our chicken crossing problem. The state of a board game could be represented as the board situation - the transitions depend on which player has the turn.

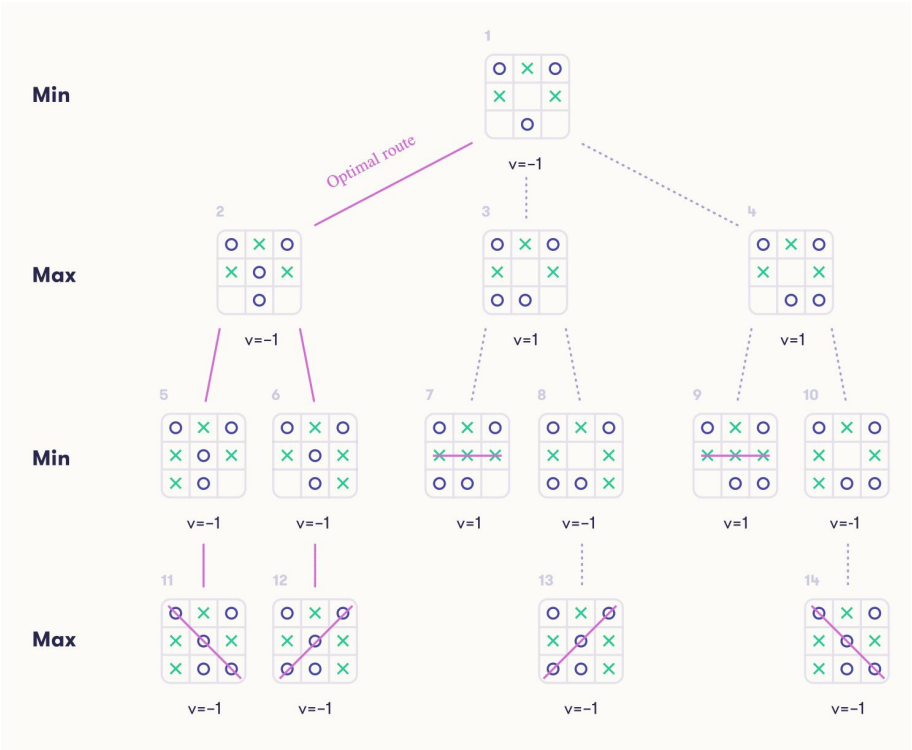
Game transition diagram (tree)



Players in the tic-tac-toe example are named Min and Max (for reasons apparent on next slide). Min plays with circles and Max with crosses. The tree branches represent the results of choices each player has made.

(See https://course.elementsofai.com/2/3)

Minmax algorithm for tic-tac-toe



If the whole decision tree for the game can be generated, then the winning strategy can be found out with minmax algorithm.

Start from the leaves (down), and mark the ones where max (x) wins with +1, min (o) wins with -1.

Depending on turn (indicated on left), mark either max or min of the children node values to a node.

State-space size: need for heuristics

The size of state space for tic-tac-toe is certainly manageable - no tricks or advanced techniques are necessarily needed to cope with that.

For chess, the state space is about 10^120, for Go it is 10^174 (approximate numbers, there are different ways to represent the state space). Representing that space in computer memory is impossible.

Instead of trying out all possibilities, heuristic guidance must be employed. Reinforcement learning, Monte Carlo search (randomised search) and deep learning techniques are employed in state-of-the-art systems for playing chess and go.

Logic and satisfiability

Logic in search problem solving

The algorithms used for searching through the states of the problem all need to be adjusted to the search problem in hand - data structures for representing states, etc. must be designed. Also, the algorithms themselves should be fine-tuned for best performance. Could this be done in a more generic manner?

In logic programming the idea is to represent the problem using the language of logic (propositions, their truth values, combining propositions with connectives such as AND, OR, NOT etc to express conditions), and then trying to find a truth value setting for propositions that satisfies the given conditions.

Some connectives

NOT: NOT(A) is true if and only if A is false

AND: A AND B is true if both A is true and B is true, false otherwise

OR: A OR B is true is either A is true, B is true, or they both are true, false otherwise

→ **(implication)**: A → B truth value is given by NOT(A) OR B

A	B	and	or	→
false	false	false	false	true
true	false	false	true	false
false	true	false	true	true
true	true	true	true	true

Example

Let’s consider a student and describe the state of student with three propositions:

A: “Student works hard.”

B: “Student is lucky.”

C: “Student gets a good grade.”

Additionally:

A AND C and (NOT(A) AND B) → C. A student is happy if they get a good grade from an exam not by being lucky: NOT(B) AND C. In what case(s) is student happy? Let’s construct a truth table:

A	B	C	A AND C	(NOT(A) AND B) →C	NOT(B) AND C
false	false	false	true	true	false
false	false	true	false	true	true
false	true	false	false	false	false
false	true	true	false	true	false
true	false	false	false	true	false
true	false	true	true	true	true
true	true	false	false	true	false
true	true	true	true	true	false

Satisfiability checking

The assignment of truth values to propositions A, B and C in the preceding example is called a *valuation*, and finding valuation(s) that satisfy conditions is called the *satisfiability problem*. Satisfiability problem (SAT) is well-studied and known to be NP-hard, however, there are techniques that have allowed solving SAT in cases of systems with 100.000 or more variables and millions of clauses (conditions).

In real-life cases, the system description is written in a specification language that allows for compact description in more complicated situations. The description can then be translated into (propositional) logic description, and an assignment of truth values that satisfies the given conditions is searched for.