After our “Haskell in a Nutshell” excursion,

let us move on to

Search in State Spaces

Search in State Spaces

• Many problems in Artificial Intelligence can be mapped onto searches in particular state spaces.
• This concept is especially useful if the system (our “world”) can be defined as having a finite number of states, including an initial state and one or more goal states.
• Optimally, there are a finite number of actions that we can take, and there are well-defined state transitions that only depend on our current state and current action.

Search in State Spaces

• To some extent, it is also possible to account for state changes that the algorithm itself does not initiate.
• For example, a chess playing program can consider its opponent’s future moves.
• However, it is necessary that the set of such actions and their consequences are well-defined.
• While computers are able to play chess at a very high level, it is impossible these days to build a robot that, for instance, is capable of reliably carrying out everyday tasks such as going to a supermarket to buy groceries.

Search in State Spaces

Let us consider an easy task in a very simple world with our robot being the only actor in it:

• The world contains a floor and three toy blocks labeled A, B, and C.
• The robot can move a block (with no other block on top of it) onto the floor or on top of another block.
• These actions are modeled by instances of a schema, move(x, y).
• Instances of the schema are called operators.

Search in State Spaces

• The robot’s task is to stack the toy blocks so that A is on top of B, B is on top of C, and C is on the floor.
• For us it is clear what steps have to be taken to solve the task.
• The robot has to use its world model to find a solution.
• Let us take a look at the effects that the robot’s actions exert on its world.

Effects of moving a block (illustration and list-structure iconic model notation)
Search in State Spaces

- In order to solve the task efficiently, the robot should “look ahead”, that is, simulate possible actions and their outcomes.
- Then, the robot can carry out a sequence of actions that, according to the robot’s prediction, solves the problem.
- A useful structure for such a simulation of alternative sequences of action is a directed graph.
- Such a graph is called a state-space graph.

State-Space Graphs

- To solve a particular problem, the robot has to find a path in the graph from a start node (representing the initial state) to a goal node (representing a goal state).
- The resulting path indicates a sequence of actions that solves the problem.
- The sequence of operators along a path to a goal is called a plan.
- Searching for such a sequence is called planning.
- Predicting a sequence of world states from a sequence of actions is called projecting.

Decision Trees

A decision tree is a special case of a state-space graph.

It is a rooted tree in which each internal node corresponds to a decision, with a subtree at these nodes for each possible outcome of the decision.

Decision trees can be used to model problems in which a series of decisions leads to a solution.

The possible solutions of the problem correspond to the paths from the root to the leaves of the decision tree.

Example: The n-queens problem

How can we place n queens on an n×n chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.

Here, the possible target squares of the queen Q are marked with an x.
Decision Trees

Let us consider the 4-queens problem.

**Question**: How many possible configurations of 4×4 chessboards containing 4 queens are there?

**Answer**: There are 16!(12!4!) = (13×14×15×16)/(2×3×4) = 13×7×5×4 = 1820 possible configurations.

Shall we simply try them out one by one until we encounter a solution?

No, it is generally useful to think about a search problem more carefully and discover **constraints** on the problem’s solutions.

Such constraints can dramatically reduce the size of the relevant state space.

---

**Backtracking in Decision Trees**

• There are problems that require us to perform an exhaustive search of all possible sequences of decisions in order to find the solution.

• We can solve such problems by constructing the complete decision tree and then find a path from its root to a leaf that corresponds to a solution of the problem (breadth-first search often requires the construction of an almost complete decision tree).

• In many cases, the efficiency of this procedure can be dramatically increased by a technique called **backtracking** (depth-first search with “sanity checks”).

---

**Breadth-First vs. Depth-First**

**Uninformed breadth-first search:**

• Requires the construction and storage of almost the complete search tree.
  → Space complexity for search depth n is O(e^n).

• Is guaranteed to find the shortest path to a solution.

**Uninformed depth-first search:**

• Requires the storage of only the current path and the branches from this path that were already visited.
  → Space complexity for search depth n is O(n).

• May search unnecessarily deep for a shallow goal.
Iterative Deepening

Iterative deepening is an interesting combination of breadth-first and depth-first strategies:

- Space complexity for search depth n is O(n).
- Is guaranteed to find the shortest path to a solution without searching unnecessarily deep.

How does it work?

The idea is to successively apply depth-first searches with increasing depth bounds (maximum search depth).

**maximum search depth = 0 (only root is tested)**

**maximum search depth = 1**

**maximum search depth = 2**

**maximum search depth = 3**

**maximum search depth = 4**
Iterative Deepening

But it seems that the time complexity of iterative deepening is much higher than that of breadth-first search!

Well, if we have a branching factor $b$ and the shallowest goal at depth $d$, then the worst-case number of nodes to be expanded by breadth-first is:

$$ N_{bf} = 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} $$

Iterative Deepening

In order to determine the number of nodes expanded by iterative deepening, we have to look at depth-first search.

What is the worst-case number of nodes expanded by depth-first search for a branching factor $b$ and a maximum search level $j$?

$$ N_{df} = 1 + b + b^2 + \ldots + b^j = \frac{b^{j+1} - 1}{b - 1} $$

Iterative Deepening

Therefore, the worst-case number of nodes expanded by iterative deepening from depth 0 to depth $d$ is:

$$ N_{id} = \sum_{j=0}^{d} \frac{b^{j+1} - 1}{b - 1} $$

$$ = \frac{1}{b - 1} \left( b \left( \sum_{j=0}^{d} b^j \right) - \sum_{j=0}^{d} 1 \right) $$

$$ = \frac{1}{b - 1} \left[ \frac{b^{d+1} - 1}{b - 1} - (d + 1) \right] $$

$$ N_{id} = \frac{b^{d+1} - 2b - bd + d + 1}{(b - 1)^2} $$

Iterative Deepening

Let us now compare the numbers for breadth-first search and iterative deepening:

$$ N_{bf} = 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} $$

$$ N_{id} = \frac{b^{d+1} - 2b - bd + d + 1}{(b - 1)^2} $$

For large $d$, you see that $N_{id}/N_{bf}$ approaches $b/(b - 1)$, which in turn approaches 1 for large $b$.

So for big trees (large $b$ and $d$), iterative deepening does not expand many more nodes than does breadth-first search (about 11% for $b = 10$ and large $d$).