A General Backtracking Algorithm

Let us say that we can formulate a problem as a sequence of \( n \) successive decisions, in each of which we pick one choice out of a predefined set of options.

For example, in the 4-queens problem we have to make four decisions, each of which consists in placing a queen in one of the four rows.

We could formalize each decision as choosing one element from the set of rows \([1, 2, 3, 4]\).

This set is the same for all four decisions. Therefore, we can describe the overall choices we have for this problem as:

\[
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
\]

(placing the first, second, third, and fourth queen)

Voluntary Homework

1. Write the function \( f\text{Queens} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Bool} \) in Haskell. It is actually easiest if you allow it to check the status of any \( n \)-queens problem instead of only the 4-queens one.

2. Write a function

\[
\text{backtrack} :: (a \rightarrow [a] \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [[a]] \rightarrow [a]
\]

that takes a "sanity" function such as \( f\text{Queens} \), a start plan – usually \([\] \) – and a list of the choices at each level, in our case \([[1,2,3,4], [1,2,3,4], [1,2,3,4], [1,2,3,4]]\) and outputs a solution, i.e., a successful plan (here: a list of four numbers).

I know that this is a tricky task, but it is very useful for not only practicing Haskell but also really understanding backtracking.

If you have extra time and energy, you could also write a function that turns the list output into a console visualization of queens on a chessboard (e.g., an array of "Q"s vs. "."s).

Uninformed Search: Breadth-First

We can build our plan for solving the problem by starting at the root with an empty list \([\] \) and adding (prepending) our decisions as we move down the tree.

For example, if we decide to place the first three queens in rows 3, 1, and 4, our current plan is \([4, 1, 3]\).

Obviously, any solution of the problem is represented by a list of length four.

We also need a "sanity check" function that tells us if a decision, given the current plan, makes sense, i.e., could potentially lead to a solution.

Let us call it \( f\text{Queens} \) for the 4-queens problem:

\[
\begin{align*}
\text{fQueens} 4 \; [4, 1, 3] &= \text{False} \\
\text{fQueens} 2 \; [4, 1] &= \text{True}
\end{align*}
\]
Uninformed Search: Depth-First

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 $j$?

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

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

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

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

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

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

For large $d$, you see that $N_{df}/N_{bf}$ 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 $d$).