

Topic Notes: Backtracking

No Algorithms course would be complete without a discussion of the *n-Queens Problem*.

The problem is to place n queens on an $n \times n$ chess board such that no queen can attack another in a single move. If you are not familiar with the rules of chess, a queen can move anywhere in her own row (i.e., rank), column (i.e., file), or either diagonal from her current location.

We'll start by considering a fairly small instance of this problem, where $n = 4$, and then the problem on a standard sized chess board where $n = 8$.

We have already considered some techniques that would allow us to come up with a solution here. We could use brute force to do an exhaustive search. The most brute force we could think of is to try all possible combinations of positions where we could place 4 queens among the 16 spaces.

How many such combinations exist? Think back to your discrete math, and we are choosing 4 values from among a set of 16, which is

$$\binom{16}{4} = \frac{16!}{4! \cdot 12!} = 1820$$

For each of these, we would need to do a non-trivial task: check that no two queens are in the same row, column, or diagonal. Of course, the above is worst case when all we want to do is to produce a single example. If we find a solution early in the search, we can report it and stop. To solve the more difficult problem of listing all valid configurations of queens, we would need to consider all of the combinations.

If we move up to the $n = 8$ case, this becomes:

$$\binom{64}{8} = \frac{64!}{8! \cdot 56!} = 4,426,165,368$$

In general, this value grows very quickly.

$$\binom{n^2}{n} = \frac{(n^2)!}{n! \cdot (n^2 - n)!}$$

We can do better.

We can make a significant improvement, even staying with what is still an exhaustive search, if we take into account the fact that each row will need to contain just one queen. For the $n = 4$ case, this means we would have 4 choices for each of the 4 rows, resulting in the much smaller

$$4^4 = 256$$

which is better than 1820.

For $n = 8$, we get

$$8^8 = 16,777,216$$

which is a huge improvement, even though we're still looking at considering n^n possible configurations. That $O(n^n)$ is still an incredibly large growth rate.

But again, we can be a little more careful about using some knowledge of the problem to eliminate parts of the search space. We have been thinking about placing one queen per row, but we can also fairly easily limit to one queen per column. Considering the $n = 4$ case, we can place the queen in the first row in any of the 4 columns. But once we've chosen a column for the first row, we are left with 3 choices for the column for the second row. Once we've chosen that column, there are only two choices for the column for the third row, and that leaves just one choice for the fourth row. This looks like a factorial, and it is. For each of these candidate layouts, we again need to check that the queens cannot attack each other along diagonals.

Normally, we have not considered $O(n!)$ to be a nice growth rate, but at least it's better than $O(n^n)$.

Perhaps a greedy approach would allow us to do even better. In a greedy approach, we would make a locally-optimal selection at each step. Let's say that an optimal selection is simply one that doesn't place the next queen in a location that would be able to attack an existing queen. Further, we will choose the lowest column in each row that results in a valid placement.

This does not lead to us finding a solution for the $n = 4$ case.

We need something that falls between greedy algorithms and a full exhaustive search. A process like a greedy algorithm that lets us undo a bad move.

The kind of algorithms we consider next are called *backtracking* algorithms. With a backtracking algorithm, we try different alternatives at each decision point, in turn, until we find a satisfactory solution.

A fundamental example here is maze running. If a choice to follow one path in the maze fails (leads to a dead end), we go back (backtrack) to the most recent decision point and try a different direction. If all directions from a given point have led to dead ends, we backtrack further to a prior decision point and try a different alternative.

Let's work through this idea with the 4-Queens problem. (In-class demo.)

This can be depicted in a state space search diagram, as seen in Figure 12.2 of Levitin.

The general form of a backtracking algorithm looks like the following, where

- `state` is the current state of the problem (the result of moves made so far)

- `move` is the next move to be attempted to get closer to a solution, which can be null if no move could be determined
- `isSolution(state)` returns whether the given `state` is a complete solution to the problem
- `isLegal(state, move)` returns whether the application of this move to this state would generate a new legal state. Examples of illegal states would be one that requires going through a wall in a maze or placing a queen in a position that would be able to attack queens already placed in the state in the n -Queens problem.
- `nextMove(move)` returns the next move that should be tried after `move` (e.g., try the next direction in a maze, the next possible square to place a queen, etc.)
- `firstMove()` returns the first move that would be attempted in the next step after accepting a candidate move into the state.
- `makeMove(state, move)` creates a new state with the move added to the old state.

```

ALGORITHM backtracker(state, move)
  if isSolution(state)
    // we have a solution!  return it
    return state

  if move = null
    // path failed, return null to indicate this
    return null

  if not isLegal(state, move)
    // cannot apply move to state, so
    // try next move instead
    return backtracker(state, nextMove(move))

  // make move and continue
  result = backtracker (makeMove(state, move), firstMove())
  if result != null
    return result

  // the previous move failed, backtrack to try next
  return backtracker(state, nextMove(move))

```

Specifically for the n -Queens problem, we can use backtracking as follows:

- A `state` is a list of k column positions, specifying where to place queens in rows 1 through k .

- A move is the column position at which to try to place the queen in the next row (row $k + 1$ when state contains k entries).
- `isSolution(state)` checks that the state has n entries.
- `isLegal(state, move)` returns whether adding a queen at column position `move` in the next row in the state would result in a state that contains no pair of queens in the same row, column, or diagonal.
- `nextMove(move)` returns `move+1`, unless `move` is n , in which case it returns null to indicate that the next move would be off the edge of the board.
- `firstMove()` returns 1 to indicate that the first place to try to place a queen in a new row is at column position 1.
- `makeMove(state, move)` returns a new list consisting of the old list with the given move appended, representing the addition of a queen at position `move` in the next row.

So our algorithm becomes:

```

ALGORITHM nqueens(n, positions[1..k], colInNextRow)
  if k = n
    // we have a solution! return it
    return positions

  if colInNextRow = null
    // path failed, return null to indicate this
    return null

  if not isLegal(positions, colInNextRow)
    // cannot apply move to state, so
    // try next move instead
    return nqueens(positions, nextMove(colInNextRow))

  // make move and continue
  result = nqueens(positions+colInNextRow, 1)
  if result != null
    return result

  // the previous move failed, backtrack to try next
  return nqueens(positions, nextMove(colInNextRow))

```

where `isLegal` and `nextMove` are as described above.