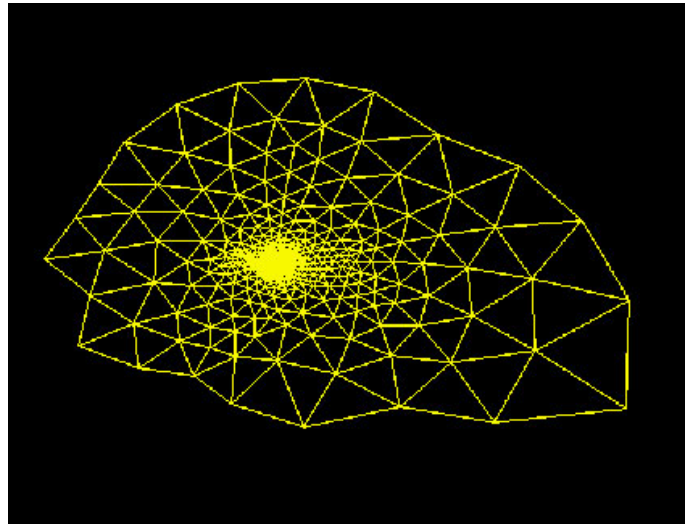
CI583: Data Structures and Operating Systems
Algorithmic strategies to solve NP-hard problems



Greedy algorithms

Greedy algorithms work by looking at a subset of a larger problem and finding the best solution for that subset.

Dijkstra's Shortest Path algorithm is a greedy algorithm developed in the late 50s. At each step it looks at adjacent edges and decides which one to add to the spanning tree. By doing this repeatedly, we end up with a spanning tree that has the minimum overall total, the MST.

Dijkstra's shortest path

The algorithm works by putting all nodes into one of three categories:

1. in the tree: nodes already added to the spanning tree,
2. on the fringe of the tree: those nodes adjacent to the current node, and
3. not yet considered.

Dijkstra's shortest path

The algorithm, informally:

1. Select a starting node.
2. Build the initial fringe from nodes adjacent to the starting node.
3. While there are nodes not yet considered, do
    1. Choose the edge in the fringe with the smallest weight.
    2. Add the associated node to the tree.
    3. Continue with the new selected node and updated fringe.

Backtracking

The class of problems related to finding a path through a maze or avoiding a series of obstacles can be solved by backtracking algorithms. This type of algorithm makes choices at each step and, when it reaches a dead end, retraces its steps to a point at which an alternative choice can be made.

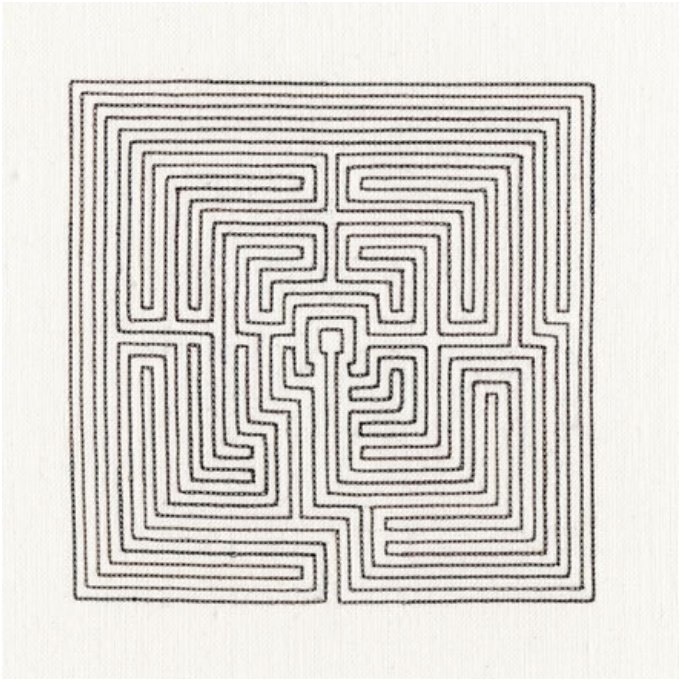
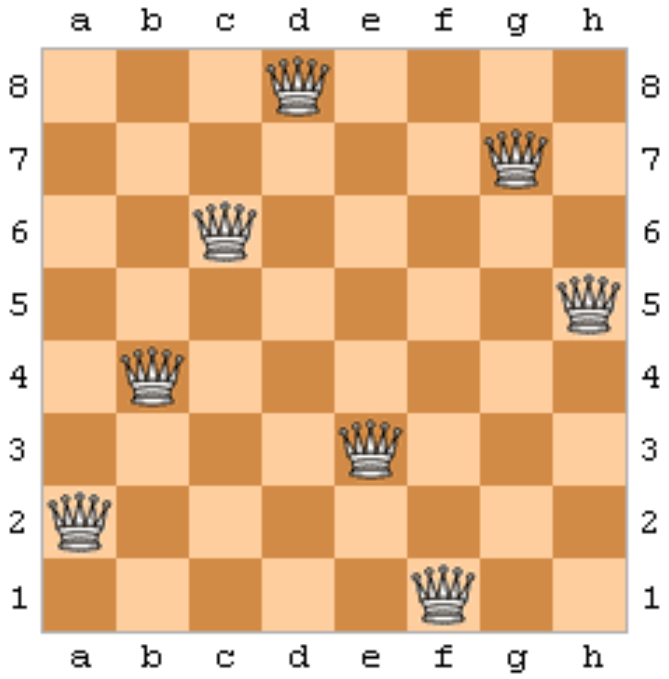


Image ©http://www.liyenchong.com

Backtracking

Backtracking techniques save the state of the problem each time a choice is made. These techniques are not only useful for path-finding.

Consider the N-queens problem. This problem asks how to position N queens on an $N \times N$ chess board in such a way that they don't threaten each other.

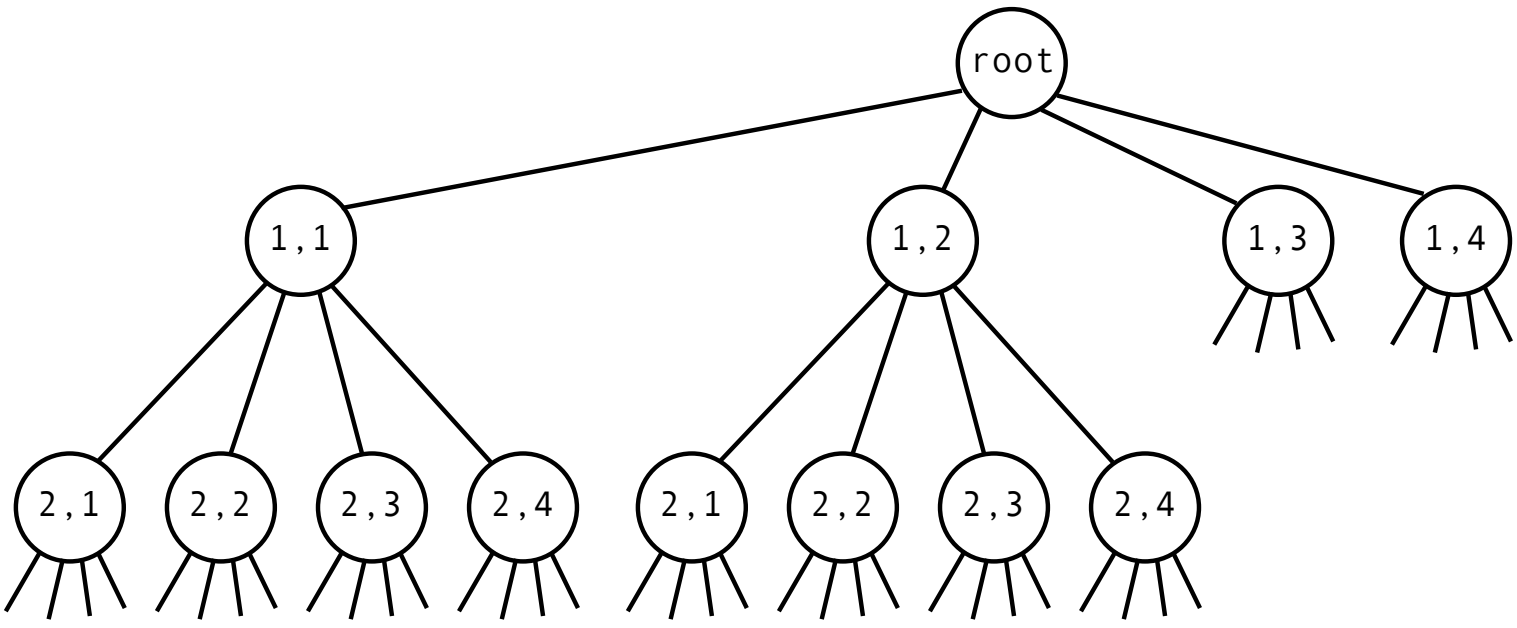


4-queens

We can develop a state space tree to describe the 4-queens problem. Note that no row, column or diagonal can contain more than one queen. Each level of the state space tree represents the possible places where each queen can be placed for one of the rows of the board.

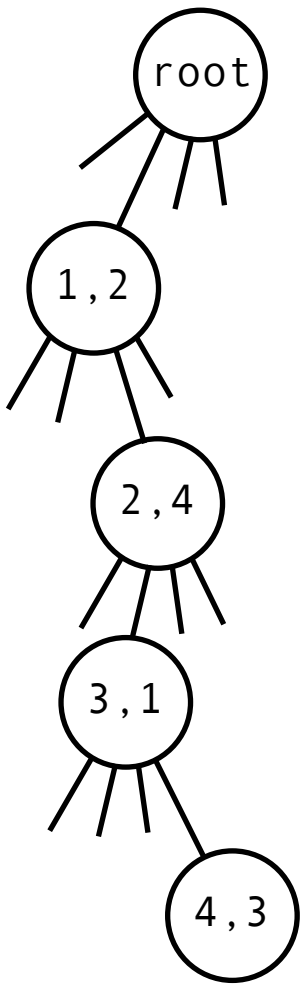
The state space tree will have 256 leaves, with each path from the root to a leaf representing a possible solution. Most of these are not solutions of course.

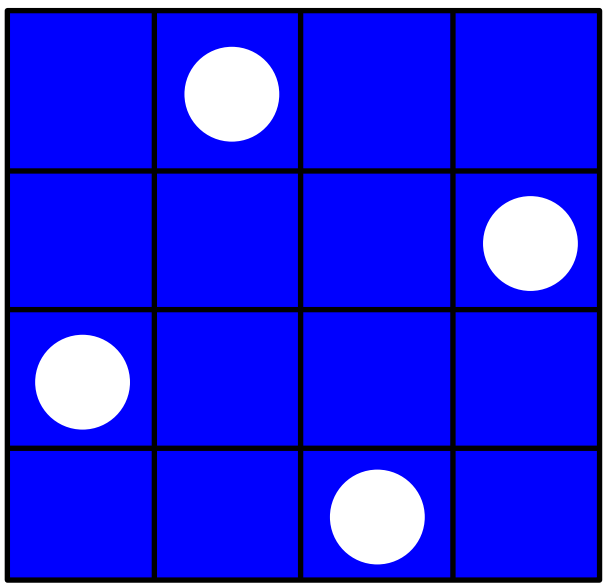
4-queens



4-queens

Given the state space tree, we can carry out a depth-first traversal to place 4 queens on the board, then check whether they can attach each other:





4-queens

What is wrong with this approach?

4-queens

What is wrong with this approach?

Using this approach, much more work is done than necessary. As soon as we place a queen so that it threatens another, a path that passes through that node will not lead to a solution and we call that a nonpromising node. So there is no need to populate or search subtrees with a nonpromising root.

A backtracking recursive solution to n-queens would stop recursing whenever it reaches a nonpromising node. Note that, in this case, the algorithm does not literally need to retrace its steps.

Backtracking

The general strategy:

```
procedure SearchSpace(i)
    if there is a solution then
        output the solution
    else
        for every possible next step do
            if the step is promising then
                SearchSpace(i+1)
            end if
        end for
    end if
end procedure
```

Dynamic programming

Dynamic programming techniques include algorithms in which the most efficient solutions depend on choices that might change with time.

The key feature to dynamic programming is memoisation – this is the technique of storing the result of expensive computations in order to reuse them.

Dynamic programming

This Java method calculates the *n*th element in the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8 ...):

```
1  int fibonacci(int n) {
2    if (n<2) return n;
3    return fibonacci(n-1)+fibonacci(n-2);
4
5  }
```

Dynamic programming

This is very inefficient. To calculate `fibonacci(10)` we calculate `fibonacci(9)` and `fibonacci(8)`. To calculate `fibonacci(9)` we calculate `fibonacci(8)` (again) and `fibonacci(7)`, and so on.

A solution that uses memoisation:

```
1  int fibonacci(int n) {
2    if (n<2) return n;
3    int[] data = new int[n];
4    data[0] = 1;
5    data[1] = 1;
6    for(int i=2;i<n;i++)
7      data[i] = data[i-1]+data[i-2];
8    return data[n-1];
9
10 }
```

An even more efficient solution would store just the last two values, as these are all we will need.

Next time

We have described the set of problems for which there is (currently!) no polynomial time solution, looked at some examples, and at some algorithmic techniques that can be used to tackle these problems.

In a later lecture we will look at other important algorithmic techniques: ones which make use of probability and chance, and ones which are suited for paralellisation.