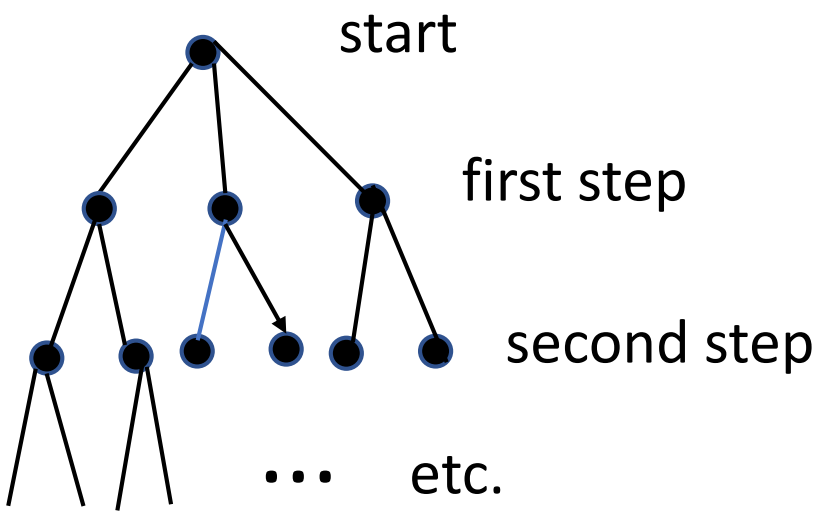
Backtracking I: Concepts

You have seen backtracking before. Among other things it is the basis for the Anagram lab in CSCI 150. We are going to devote a little time to considering backtracking as a programming technique.

Backtracking is a formal way to search all possible elements in the solution space for many problems. It is not efficient. In many situations it produces exponential-time solutions to problems, so it breaks down as the problem size becomes large. Often there are better solutions. However, backtracking does have some advantages. It is fairly general and can be applied in many situations. It is routine; backtracking algorithms all look the same, so once you understand the technique you can apply it quickly and reliably. There are situations where the problem size is known to be small and coding time is more important than runtime. In these situations backtracking can be a good solution.

Backtracking only applies to certain kinds of problems. It requires the problem to have a solution that can be built one step at a time. Such a problem has a solution space that forms a tree:



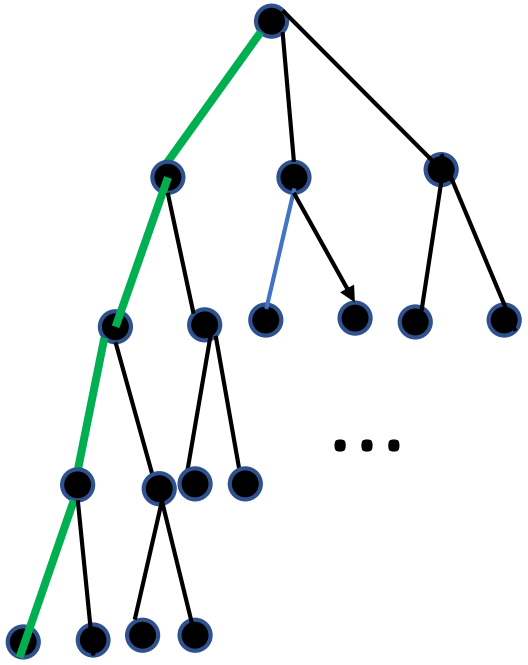
For example, back in that 150 Anagrams lab, each step consisted of trying to make a word out of the remaining letters we had. To find the options for this step we walked through the words of a dictionary, asking if each word could be made out of the remaining letters.

We might use backtracking to find a path through a maze. At each step we look at our current location and we have four options: moving forwards, backwards, left or right.

A famous problem often solved with backtracking is the "n-Queens" problem. Its name comes from a common expression of the problem in terms of chess pieces, but we can state it easily as "Find n squares on an nxn grid so that no two are in the same row, the same column, or the same diagonal." We can think of step 1 as choosing a square for the first column, step 2 as choosing a square for the second column, and so forth. Since each column has n squares, there are n choices for each step.

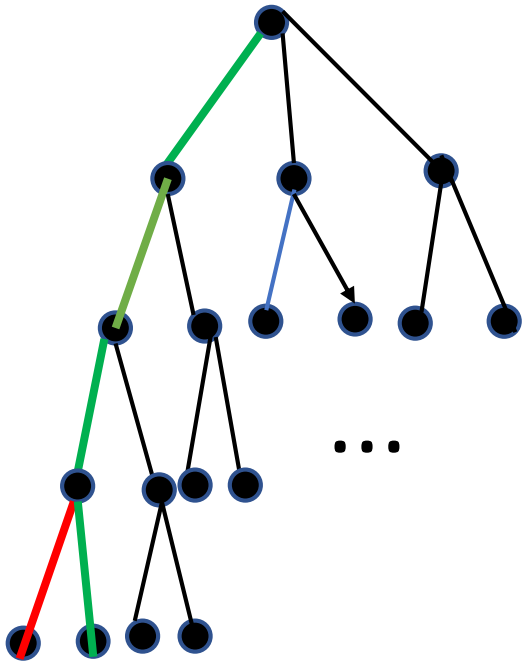
Many problems are not amenable to backtracking. Sorting a list or grouping data into clusters don't lend themselves to this step-by-step solution. But when a problem does fit this model backtracking is at least one way you might set about looking for a solution.

Backtracking peforms a *depth-first search* through the solution space. It tries the first possible value for the first step, then the first possible value for the second, and so forth, walking down the left-most branch of the tree -- that is the path in green:

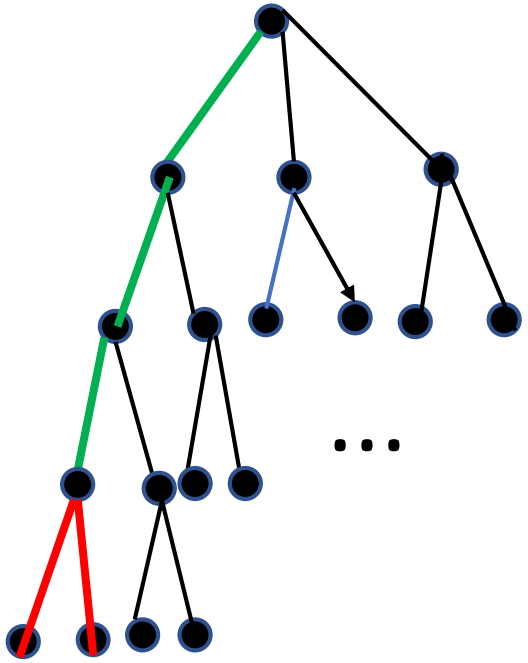


If that gets us to a solution, we are good.

If that does *not* get us to a solution, backtracking undoes the last decision we made. In this case that decision was to use the first possible value in step 3. We move on to the next possible value in step 3:

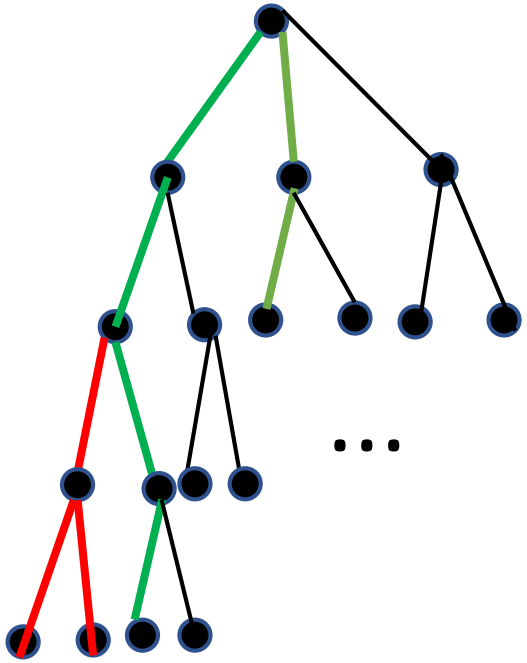


Suppose this still hasn't found a solution and that we are out of options for step 3:

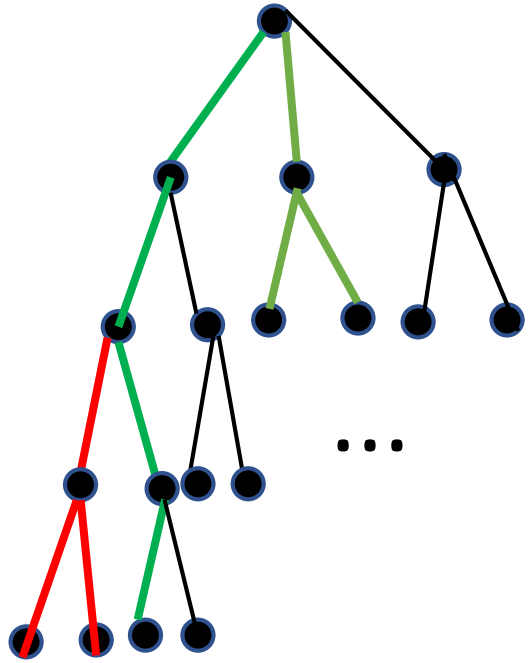


This means our initial choice of the first option for step 2 must have been a mistake. We backtrack to that step and try the next choice:

We repeat the algorithm walking down this new branch of the tree, trying



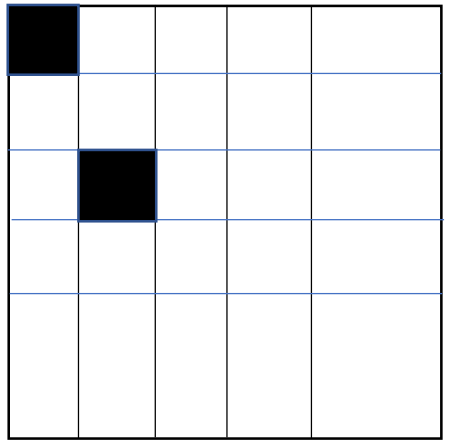
then

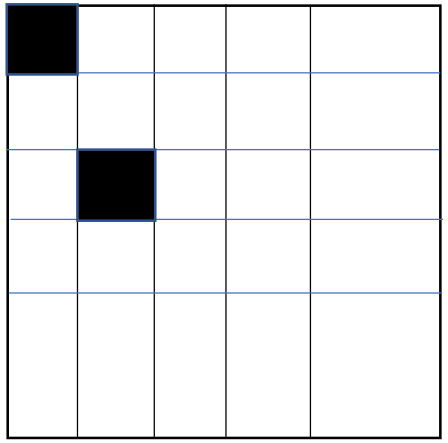


If neither of these work we go all the way back to step 1 and choose the next option. This continues until we have found a solution or explored all options.

Backtracking is never very efficient, but it helps if we can at least determine whether a partial solution is *feasible*, i.e., can potentially be extended to a complete solution. If a partial solution is not feasible there is no point in walking down the tree for it. This lets us prune some useless paths from the tree and saves some time.

For example, remember the n-Queens problem -- finding n squares on an nxn grid with no two on the same row, column or diagonal. We build the solutions by finding a square for each column. Suppose we have already chosen the first square for the first column and the third square for the second column:





For the third column the first square is not feasible since it would be in the same row as the first column; the second square is not feasible since it is on the same diagonal as the choice for the second column, and so forth. We would need to go down to at least square 5 of the third column to get a feasible entry. Our backtracking algorithm checks feasibility and only extends feasible partial solutions.

Here is pseudocode for the basic backtracking algorithm:

```
backtrack(n, sofar)
    ; n, and possibly other parameters, describe the problem
    ; sofar is a list of the steps making up the current partial solution
    ; this returns either a complete solution or null as a failure signal
        if sofar is a complete solution, return sofar
        for each possible value v for the next step:
            if adding v to sofar makes a feasible partial solution
                res = backtrack(n, (cons v sofar))
                if res is not null, return res
                if res is null go to the next value of v
            if adding v to so far is not feasible, go to the next value of v
         If the possible values of v are exhausted,
                return null as a failure signal
```

There are several ways to turn this pseudocode into Scheme code. The only issue is how to represent the for-loop through possible values. Here is a simple solution:

```
(define backtrack (lambda (n curr sofar)
        ; returns the first extension of sofar into a solution with
        ; curr or higher as the value for the current step
        (cond
                [<sofar is a complete solution> sofar]
                [<curr is out of the range of possible values for this step> null]
                [(feasible  curr sofar)
                        (let ([res (backtrack n <first value for next step> (cons curr sofar))])
                                (if (null? res)
                                        (backtrack n  <value after curr> sofar)
                                        res))
                [else (backtrack n <value after curr> sofar)])))
```

One common variant of the backtracking algorithm is to find not one, but *all* solutions to a particular problem. With a procedural language this is a simple change: instead of returning the solution we print it, and backtrack to the next solution. To find all solutions functionally we will build up lists of solutions and have the backtracking function return a list of all solutions it finds.

```
(define allsols (lambda (n)
        (letrec ([backtrack (lambda (curr sofar)
                ; backtrack returns all solutions that extend sofar with value curr or  higher
                   (cond
                        [<sofar is a complete solution> (list sofar) ]
                        [<curr is out of the range of possible values for this step> null]
                        [(feasible  curr sofar)
                            (let ([res (backtrack n <first value for next step> (cons curr sofar))]
                                  [res2 (backtrack n  <value after curr> sofar)])
                                        (append res1 res2)
                        [else (backtrack n <value after curr> sofar)])])
                (backtrack <first value> null)))
```