159.271 Computational Thinking for Problem Solving

5. Recursive Backtracking

5.1 Introduction to Optimisation Problems

Back to top

Very many important and practical computational problems can be expressed as *optimization problems*. Such problems involve finding the best of an exponentially large set of possible solutions. The obvious algorithm, considering each of the solutions in turn, takes too much time because there are too many possible solutions. Some of these problems can be solved in a "reasonable" amount of time (that is, running time a small polynomial function of the size of the input) using techniques such as greedy algorithms or dynamic programming, which we will see later in this course. For some other practical applications, recursive backtracking gives us a way to to find an optimal solution for (most) instances in a "reasonable" amount of time.

Failing this, we can resort to finding an approximately optimal solution, in other words, not quite the best, or to using random algorithms which will give us an optimal solution "most" of the time. These ideas are explored further toward the end of this course. We begin by formally defining what we mean by an optimization problem.

Ingredients: An optimization problem is specified by defining instances, solutions, and costs.

- **`Instances:`** The instances of the problem are the possible inputs for the problem. If we are considering the problem of developing a timetable for the University, an instance will be the set of courses offered by the University, along with the sets of courses that each student requests, the set of time slots available, and the set of rooms available.
- **`Solutions:`** Each instance has an exponentially large set of possible solutions. A solution is *valid* if it meets some set of criteria determined by the instance at hand. In the timetabling example, a solution for an instance will be a schedule that assigns each course a time slot in a particular room. The solution will be valid if no two courses share a room at the same time slot.
- **`Costs:`** Each solution will have an easy-to-compute cost or *measure of success* that is to be minimized or maximized. In the timetabling example, the cost might be the number of students who have clashes for some of their classes, or perhaps the number of (time slot, room) combinations that are not utilized.

Specifications:

- **`Preconditions:`** the input is one instance
- **`Postconditions:`** the output is one of the valid solutions for this instance with optimal measure of success – one of the "best" solutions, note that there may be a number of these, we only need to find one

The brute force algorithm for an optimization problem, computing the cost of each of the exponential number of possible solutions and returning the best, takes exponential time. Another problem is how to write code that enumerates over all possible solutions. Often, the easiest way to do this is to use recursive backtracking.

5.2 Introduction to Recursive Backtracking

Back to top

The basic idea, of course, uses recursion. How to find an optimal solution for one instance using an optimal solution for a smaller instance, or a set of smaller instances? The optimal solutions for the smaller instances are found recursively, until we get to a point where the instances are so small that brute force can be employed. Recursive backtracking in its simplest form effectively enumerates all options, so we haven't won the battle yet. However, it has two important points in favour. Firstly, once you get the hang of recursion, it is the simplest way of writing code to accomplish the task of enumerating over all possible solutions. Secondly, with a little insight, it is often possible to prune off entire branches of the recursion tree, significantly improving the running time. This is the bit that requires creativity.

The way to think about the recursive process is to view the algorithm as a sequence of decisions. Do we include the first object in the solution or not? Do we include the second? At the fork in the road, do we go left or right? At the second fork, which way do we go? As one stack frame in a recursive algorithm, our task is to deal with only the first of these decisions. A recursive friend will deal with the rest.

The basic structure of the strategy is given here.

```
Algorithm(instance I)
        if I is small then
             return(Brute force answer)
        else loop over possibilities k= 1..K:
             temporarily commit to making first decision in kth way
             best[k] = Algorithm(I_k)
        return best of best[k]
```

One way to think about pruning the recursion tree is that it is pretty much exactly like playing 20 Questions. If you are told that the target is an animal that lives in water then you probably don't need to ask how many legs it has, you would be better to ask other questions. In this way you close off a whole set of possible animals to search over. Of course, your assumptions might turn out to be wrong. Once you've found out that it can't swim, you might want to revisit your assumptions and remember that crabs live in water too, as do

water boatmen and plenty of other legged animals! This revising of pruned parts of the space is the backtracking bit. The key, of course, is to ask the right series of questions!

So far, all of the recursive algorithms we have seen have shared one very important property: each time a problem was recursively decomposed into simpler instances of the same problem, only one such decomposition was possible; consequently, the algorithm was guaranteed to produce a solution to the original problem. Now, since we are dealing with a sequence of decisions, we have several possible choices at each step, so several possible decompositions from one instance of the problem to another. That is, each time we make a recursive call, we make a choice as to which decomposition to use. If you look at the pseudocode above, you will see that, in fact, we try out each choice one after another, but we explore as far as we can before trying out the next choice.

When we have explored all of the possibilities based on one choice, we will backtrack to a "choice point" and try another alternative. Note that we only back up in the recursion tree as far as we need to go to reach a previously unexplored opportunity. Eventually, more and more of these options will have been explored, and we will backtrack further and further. Once we backtrack to our initial position and find we've nothing else to explore, then we have explored the whole of the recursion tree.

The key to pruning the recursion tree is that, if we have a choice that we can see will lead only to "dead ends", or rather, invalid or sub-optimal solutions, then we never take it.

The two difficult and creative parts of designing a recursive backtracking algorithm are:

1. What question should you ask? Worrying only about the top level of recursion, the designer must formulate one small question about the optimal solution that is being searched for. The question should be such that having a correct answer greatly reduces your search.
2. How should you construct a subinstance? How to express the problem that we want to solve as a smaller instance of the same search problem?

5.3 Developing a Recursive Backtracking Algorithm

Back to top

The n-Queens Problem

Suppose that you want to position n queens on an n x n chessboard so that none of them can capture any of the others (you might want to set n = 8 and try this on a real chessboard - use pawns, but let them move as queens to have enough pieces). We are going to construct a solution to this problem using recursive backtracking. We start by noticing something obvious about the solution: there cannot be more than one queen in any row. This is obvious, but it helps a lot, since it tells us that we can actually solve n subproblems: put a queen in row 1, put a queen in row 2, …, put a queen in row n. Now let's extend that idea to columns. Clearly, we cannot put two queens in the same column, either. This is how we prune: if the solution that we are looking at has two queens in the same column then it is wrong, so we don't need to bother going any further with it. We'll get there soon.

There really isn't too much else to think about. To get started, we will make an algorithm that does the exhaustive search, but in a nice recursive way. Then we will think about the pruning. There is one thing that we will need to include, which is a score for each board, so that we can differentiate between legal and illegal ones. To do this, we will just give a score of 1 to boards that are legal, and 0 to those that are not. **Using those ideas, see if you can now construct the recursive backtracking algorithm for yourself before you read on.**

Specification: We need to be clear about what problem needs to be solved. What is the set of instances? For each instance, what is the set of possible solutions? For each solution, what is its cost?

We start by considering a single input instance for each n, the n x n board. The set of possible solutions is the set of ways of placing n Queens on the board. We don't value one solution over another as long as it is valid. The cost of a solution is 1 if it is a valid solution and 0 if not.

Question: The question should be such that having a correct answer greatly reduces the search. Generally, we ask for the first part of the solution. Where should I place the first Queen?

In order for the final algorithm to be efficient it is important that the number of possible answers is small. Since we must have at least one queen per row, we can place the first queen in the first row. So there are K = n different answers.

Constructing a subinstance: In order to give an instance to a friend I need to tell him where we put the first queen. To develop an effective recursive strategy we need to **provide more information via the input**. An input instance to the problem will specify the location of the queens in the first r rows. So now we have a whole bunch of input instances to consider. A solution is a valid way of putting the queens in the remaining rows. Given such an instance, the question becomes, where does the queen in the next row go?

Costs: In this case, we just need to check if the placement is legal or not.

Base cases: If all n queens have been placed, there is nothing to do.

Here is basic code structure for the problem.

```
Queens(currentBoard,currentRow,n):
        if currentRow == n:
                if currentBoard is legal:
                        return (currentBoard, 1)
                else:
                        return (currentBoard, 0)
        else:
                for k = 1..n:

                        newBoard = currentBoard + (currentRow+1,k)
                        (board[k],success[k]) =
Queens(newBoard,currentRow+1,n)
```

```
        kmax = index of max(success[k])
        return (board[kmax], success[kmax])
```

Convince yourself that this will always find a solution if one exists, and then implement the algorithm. There is obviously still a bit of work to do to ensure that boards are legal, but you should be capable of that. The question is how to change this from exhaustive search, which it currently is, to include pruning. The basic idea is just to stop evaluating any board as soon as a queen is placed on it illegally. So the line **`newBoard = currentBoard + (currentRow+1,k)`** is followed by a check that this does not produce an illegal board, and if it does then the recursive call is not made. This produces a substantial speed-up on the algorithm, as you should be able to see once you implement it.

There is another speed-up that you can include, which is to notice that the board is initially symmetric, so that you don't have to consider putting the first queen in any except the first n/2 columns, since you could just reflect the board in the vertical axis to put it into those columns. Of course, this isn't going to make much difference if n is fairly large.

Note: If you do 159.272 then you will see a functional programming language, Haskell. This gives a very different way to implement this type of algorithm (and many others) that is very interesting.

5.4 Recursive Backtracking without Costs

Back to top

Recall the basic strategy that we have used so far:

```
Algorithm(instance I)
        if I is small then
             return(Brute force answer)
        else loop over possibilities k= 1..K:
             temporarily commit to making first decision in kth way
             best[k] = Algorithm(I_k)
        return best of best[k]
```

This works for the general class of optimization problems as we have defined it. However, apart from pruning the recursion tree by not expanding "dead ends", you might have noticed that there is another way to reduce the amount of work that we need to do in the case that cost is not an issue. For the n-queens problem, once we have found a valid solution, we know that no other solution can be "better", so we should stop searching. How can we incorporate this idea into our algorithm, while still retaining the recursive nature of our strategy?

The answer is to **return more information about the subinstances**. As well as returning the current board, we will utilize the flag that indicates whether or not the recursive call produces a legal board. If so, we return it immediately to the calling instance, without trying any further recursive calls. Code that incorporates this idea, and the obvious pruning idea, is given below.

```
Queens(currentBoard,currentRow,n):
        if currentRow == n:
```

```
                    if currentBoard is legal:
                            return (currentBoard, 1)
                    else:
                            return (currentBoard, 0)
            else:
                    for k = 1..n:

                            newBoard = currentBoard + (currentRow+1,k)
                            if newBoard is legal:
                               (board[k],success[k]) =
Queens(newBoard,currentRow+1,n)
                               if success[k] == 1 return (board[k],
success[k])
                    # if we get to here, know that no completed legal
board
                    # could be produced from currentBoard
                    return (currentBoard, 0)
```

Here is another problem of this type, where all we really care about is whether we get a valid solution or not, but we have many solutions to search through.

Shrinking A Word (from Julie Zelenski, Stanford)

Consider the simply stated question: Is it possible to take an English word and remove its letters in some order such that every string along the way is also English?

Sometimes it's possible. For example, we can shrink the word "smart" down to the empty string while obeying the restriction that all of the intervening strings are also legitimate words. Check this out:

```
"smart"
"mart"
"art"
"at"
"a"
""
```

We elect to first remove the s, then the m, then the r, then the t, and finally the a. Note that every single string in the above diagram is an English word: That means that, for the purposes of the this problem, it's possible to shrink the word "smart" down to nothingness. Not surprisingly, there are some perfectly good English words that can't be shrunk at all: "zephyr", "lymph", "rope", "father". A reasonable question to ask at this point: Can we programmatically figure out which words can be shrunk and which ones can't be?

In this example, we only commit to some of the recursive calls if a previous one failed to provide an answer. Here is the pseudocode for our **canShrink,** function, which returns true if and only if it's possible to shrink the provided word down to the empty string:

```
def canShrink(aWord, aDictionary):
    if aWord is "":
       return True
```

```
    if !(aWord in aDictionary):
       return False

    for i in range(len(aWord)-1):
       subWord = aWord[:i] + aWord[i+1:]
       if canShrink(subWord, aDictionary):
          return True

    return False
```

The recursive algorithm includes two base bases — that part isn't new. And it is clearly a recursive backtracking algorithm - look at the **for** loop and how it surrounds the recursive call. Each iteration considers the incoming string with some character removed, and then looks to see if that new string is also a word and can be shrunk down to nothingness. Look at this for loop as a series of opportunities to return True. If it just so happens to find some path to the empty string sooner than later, it returns true without making any other **canShrink** calls. Only if every single path turns up nothing do we truly give up and return False.

It's pretty clear why True and False get returned, since the initial call wants a yes or no as to whether or not the provided word can be made to disappear. But what may not be immediately clear is that the recursive calls also rely on those return values to figure out whether or not the path it chose to recursively explore was a good one.

The above version correctly returns True or False, but it doesn't remember what words lead down to the empty string. Can you rewrite the function so that it prints out the sequence of words, if there is one, and "False" otherwise? This is not quite trivial to do.

Another Example - Satisfiability

A famous optimization problem is called *Satisfiability*, or SAT for short. It is one of most important and general problems arising in computer science. We now develop a recursive backtracking algorithm for this problem, the algorithm we develop is often referred to as the *Davis-Putnam* algorithm. It is an important example of an algorithm whose running time is exponential for worst case inputs, yet in many practical situations, can work well.

Instances: An instance consists of a set of n Boolean variables $x_1, x_2, \ldots, x_n$, and a set of m constraints on these variables, e.g. $(x_1 \text{ OR } x_4 \text{ OR } \overline{x_5})$.

Solutions: An assignment of Boolean values **TRUE** and **FALSE** to the variables. A solution is valid if it satisfies all constraints, that is, all constraints evaluate to **TRUE** given the particular assignment of values to the variables.

Costs: 1 for satisfying assignments, 0 for non-satisfying assignments. An alternative version of the problem considers the weight of the assignment, that is, the number of variables set to **TRUE**, and tries to minimise this.

Question: Should x_1 be assigned **TRUE** or **FALSE**?

Constructing a subinstance: An input instance to the problem will specify a partial assignment, to variables $x_1, x_2, \ldots, x_r$. A solution is an assignment for the remaining variables that results in a valid solution satisfying all constraints. Given such an instance, the question becomes, should x_{r+1} be assigned **TRUE** or **FALSE**?

Base case: Once all variables have been assigned a Boolean value, there is nothing more to do.

Pruning: The complete recursion tree will contain 2^n nodes, since we branch two ways for each decision made. However, there are several ways in which we can shrink the tree.

Consider the constraint $(x_3 \text{ OR } x_4)$. If x_3 has already been assigned **FALSE** then x_4 is forced to be assigned **TRUE**. So we commit to this assignment immediately and don't consider the branch where x_4 is assigned **FALSE**.

Consider the constraints $(x_3 \text{ OR } x_4)$, $(x_3 \text{ OR } \overline{x_4})$. If x_3 is assigned **FALSE** then this set of constraints is unsatisfiable. So we don't consider the branch where x_3 is assigned **FALSE**.

We make progress by shrinking the set of constraints. Consider constraints $(x_3 \text{ OR } x_4)$, $(x_2 \text{ OR } x_3)$. If x_3 is assigned **TRUE** then both of these constraints can be removed from the set of constraints we need to consider. Once the set of constraints is empty, we are done. So we look for variables that occur in a large number of constraints, either positively or negatively, and make assignments for these as early as possible.

Note that we can stop as soon as we encounter a valid solution. Even if all assignments to the variables haven't yet been made, if all constraints are satisfied, we can assign arbitrary values to the remaining variables.

Here is the pseudocode, we use 1 and 0 in place or **TRUE** and **FALSE**.

```
Algorithm DavisPutnam(c)
        # Precondition: c is a set of constraints on the assignment
to x_1, x_2, ... , x_n
        # Postcondition: if there is one, Sol is a satisfying
assignment and Cost is 1,
        # otherwise Cost is 0

        begin:
          if c has no variables or no constraints:
          # c is trivially satisfiable
            return (empty set, 1)

          elif c has a constraint forcing x_i to 0 and another forcing
x_i to 1:
          # c is trivially not satisfiable
            return (empty set, 0)

          else:
            for any variable forced by a constraint to some value:
```

```
      substitute this value into c

    let x_i be the variable that appears most often in c

    # loop over possible assignments to x_i
    for k = 0 to 1:
      let c' be constraints c with k substituted for x_i
      (subSol, subCost) = DavisPutnam(c')
      Sol[k] = (forced values, x_i = k, subSol)
      Cost[k] = (subCost)

    kmax = index of max(Cost[k])
    return (Sol[kmax], Cost[kmax])
  end
```

5.5 Practice Questions

Back to top

1. What is the running time for the n Queens algorithm when there is no pruning?
2. Consider the following Scrabble problem. An instance consists of a set of letters and a dictionary. A solution consists of a permutation of a subset of the given letters. A solution is valid if it is in the dictionary. The value of a solution depends on its placement on the board. The goal is to find a highest-value word that is in the dictionary. Develop a brute-force recursive backtracking algorithm for this problem. How could you prune the recursion tree?
3. Graph 3-coloring: Given a graph, determine whether its nodes can be colored with three colors so that two nodes don't have the same color if they have an edge between them. Let's redefine the problem so that the input consists of a graph and a partial coloring of the nodes. The new goal is to determine whether there is a coloring of the graph consistent with the coloring given. What question should you ask about the input? How would you construct a subinstance for a friend?
4. Implement the Davis-Putnam algorithm described above.