CSE 202 NOTES FOR OCTOBER 24, 2002

Search and Optimization Problems

Search problems are ubiquitous in computer science. Ideally, you specify a search problem in three pieces:

Instance: Description of input format (eg, a graph $G = (V, E)$)
Solution format: Description of output format (eg, a set $S \subset V$ of nodes)
Constraints: Properties that any solution must meet (eg, $\forall v \in S \exists u \in V$ such that $(u, v) \in E$ and $v \notin S$)

An optimization problem is a search problem with one additional piece: the objective function, which is what you wish to minimize or maximize. In general, these sorts of problems are NP-complete, except under really well-behaved conditions. Notice that sometimes these conditions are very subtle, and it seems like a particular problem must be NP-complete, but when you look very closely at the entire problem, one of the constraints removes it from the set NP. Of course, that doesn't mean that you immediately find the polynomial version of it. We will see later in the course how to cope with NP-completeness.

Backtracking and Search/Optimization Problems

How can we apply our backtracking technique to search problems? Perform the following steps when designing the algorithm:

- Imagine finding a solution as making a series of decisions, each decision with a finite number of options (though this number could be a function of the input size n—it simply has to be finite).
- Do a case-by-case analysis (in the design of the algorithm): how does each option affect the constraints or objective function?
- Prove some sort of "self-similarity" lemma: after we make one decision, the future problem is a smaller instance of the original problem. If that doesn't work out, then provide a slightly more general form of the problem until it has trivial self-similarity, and solve this problem. Specialize this problem to the original problem only at the end of the solution, never while making recursive calls.

As an example of this, we'll consider a new problem: the 3-coloring problem. In the true spirit of self-consistency

Instance: Undirected, simple, connected graph: no self-loops, no parallel edges.
Solution Format: A set S of pairings (v, c) with $v \in V$ and $c \in \{R, G, B\}$. All elements in V are represented in the set S exactly one time.
Constraints: $\forall (u, v) \in E, C(u) \neq C(v)$ (where $C(u)$ is the color associated with u in the solution S).

As an example, consider the graph in figure 1. The decision problem is whether or not there is a legal coloring of G. Before following the advice above, solve the

FIGURE 1. A sample instance of the 3-coloring problem

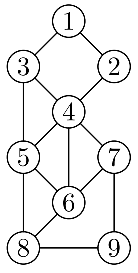
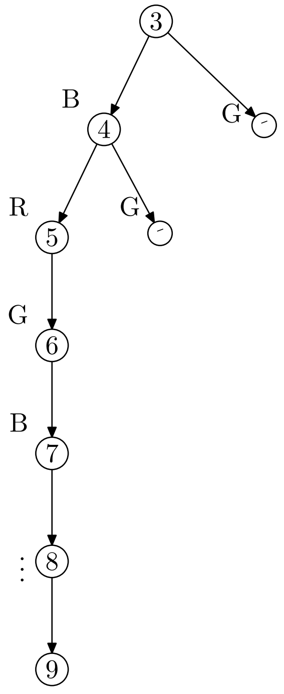


FIGURE 2. A partially complete decision tree for the sample 3-coloring problem



problem in the brute force way: try every assignment of the three colors to each node and see if it's legal. There are three colors, and n nodes. That makes 3^n possible combinations of colorings, which isn't a very good algorithm. Of course, in this algorithm, we're not using the edge information in the graph, which might speed us up. Now let's try to use backtracking to improve on our $O(3^n)$ algorithm by accounting for edge information.

Where do we make decisions?

At the nodes, and the decision that we make is the color associated with the node.

If we color the node, say, red, how does this affect the surrounding nodes?

Adjacent nodes can't be red.

In our example graph above, we can choose 1 and 2 to be R and B by symmetry to start ourselves off. A decision tree looks something like figure 2.

This is clearly recursive—when you make a decision at one node, your graph is a bit smaller and you have a smaller set of colors, but you perform the same decision logic. One of the problems, though, is that the subproblem is not the same as the original problem: you have to remove one of the colors (therefore it's not a 3-color problem any more). Let's apply the generalization process: we'll pass in the set of possible colors for a node into the function along with the graph.

BTG3C($G = (V, E)$: graph, PossColors: array of sets): boolean

```
if $|V| = 0$ then
  return $T$;
end if
if $|V| = 1$ and PossColors[$v_0$] = 1 then
  return $T$
end if
if $\exists v \in V$ such that |PossColors($v$)| = 0 then
  return $F$
end if
if $\exists v \in V$ such that |PossColors($v$)| = 1 then
  Color[$v$] $\leftarrow$ PossColors[$v$]
  $G' \leftarrow (V - v, E)$
  PossColors$'$ $\leftarrow$ PossColors
  $\forall u \in N(v)$, PossColors$'$[$u$] $\leftarrow$ PossColors[$u$] $-$ Color[$v$]
  if BTG3C($G'$, PossColors$'$) then
    return $T$
  else
    return $F$;
  end if
end if
if $\forall v \in V$, |PossColor[$v$]| = 3 then
  pick a $v$, pick a color for $v$, and repeat the above conditional.
end if
Find a $v \in V$ such that PossColors[$v$] = $\{C_1, C_2\}$.
Try the above conditional with color $C_1$, and if it works, return $T$.
Try the above conditional with color $C_2$, and return the outcome.
```

Notice that each time we call `BTG3C` we make (at most) two recursive calls of size $n - 1$, so we have $T(n) = 2T(n-1) + O(n^k)$, and $T(n) = O(2^n)$. Note also that the exhaustive search technique is $O(3^n) \approx 2^{1.6n}$. Again, we can probably find a better bound for this, but this one works. Also notice that there could be a lot of right answers to this problem since colors are essentially symmetric.