CSE 202 NOTES FOR OCTOBER 22, 2002

Backtracking and Maximum Independent Set

We have been trying to cover general techniques for algorithm design, and so far we've examined divide and conquer. This technique works really well if you can divide problem into multiple subproblems that are all some multiplicative factor smaller than the original—if you consider the recurrence relation where you simply subtract off an integer at each step, you'll see that the problem quickly becomes exponential. Nonetheless, even this latter type of recursion—often called *backtracking*—is occasionally useful, at least in practice.

A vague taxonomy of algorithm design techniques is shown in figure 1.

Consider a new problem: calculate the maximum independent set of a graph G. A MIS of a graph $G = (V, E)$ is a subset of V such that $\forall e = \{u, v\} \in E$, either $u \in S$ or $v \in S$. The problem is to find the largest independent subset in G: maximize $|S|$. In English, the idea is to find the largest subset such that at most one endpoint of an edge is present, in other words that no two vertices in the new set are neighbors.

For example, consider the graph in figure 2. An independent set would be $\{A, C\}$, but not $\{A, B\}$. More formally,

Input: Graph $G = (V, E)$
Solution Space: $S \subset V$
Constraint: $\forall e \in E, (u \in S \implies v \notin S) \vee (v \in S \implies u \notin S)$.

In this case, exhaustive search generates the power set of V, where every combination of v in the set and v out of the set for every v is represented. There

Figure 1. A rough taxonomy of algorithm design techniques that we'll learn about in this class

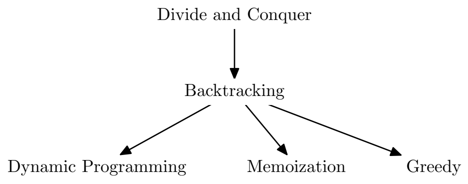
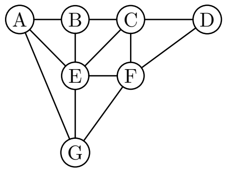


Figure 2. An undirected graph. $\{A, C\}$ form an independent set, while $\{A, B\}$ do not.



are 2^n possible sets, and we have to measure the size of each, so the exhaustive search technique takes $O(2^n)$ time. We can improve on this a little bit by deciding to imagine the solution space as a set of binary numbers, and then ordering the binary numbers by the number of 1's in their representation and then starting from $11111\cdots111$ and looking until $0000\cdots000$, stopping at the first independent set. It's still slow as hell, and $O(2^n)$.

Backtracking applies a different approach: pick a decision point and perform a case analysis for each option for constructing other decision points. Then solve each subproblem recursively and take the best solution (the optimization method — a "search" problem would simply terminate).

In general you need to have self-similarity in backtracking—by doing your case analysis you should be solving recursively the same problem (otherwise, how is it recursive?). The book makes a big deal about this, but Russell basically implies that you can generalize a problem until it is necessarily self-similar. There's a bit of an art to that, from what I've seen, but it appears to hold.

The Maximum-independent set algorithm is shown in algorithm 1.

```
MIS($G = (V, E)$: a graph): largest set of independent vertices
1: **if** $|V| = 0$ **then**
2:     return .
3: **end if**
4: **if** $|V| = 1$ **then**
5:     return $V$.
6: **end if**
7: pick $u \in V$.
8: $G_{\text{out}} \leftarrow G - \{u\}$ {remove $u$ from $V$ and $E$}
9: $G_{\text{in}} \leftarrow G - \{u\} - N(u)$ {$N(u)$ are the neighbors of $u$}
10: $S_{\text{out}} \leftarrow \text{MIS}(G_{\text{out}})$
11: $S_{\text{in}} \leftarrow \text{MIS}(G_{\text{in}}) \cup \{u\}$
12: return maxsize($S_{\text{out}}, S_{\text{in}}$)
    {return $S_{\text{in}}$ if there's a tie — there's a reason for this.}
```

Obviously, first check to see if you're in a base case. If so, you terminate nicely. If you're not, you choose a vertex in the vertex set, and then posit that the vertex is in the MIS. If it is, then you can remove the vertex from consideration, along with all of its neighbors, and then recursively call the same MIS routine on the smaller problem. At this point, you might have returned a non-empty MIS, but you don't know if it's as large as possible: you assumed that the vertex was in the set, so now assume that the vertex is not in the set—remove it from the graph and recurse. The idea here is that you're decreasing the size of the search space by the degree of u each time you visit a point u. At least some of the time you're getting a reasonable savings, but what about the worst case? What is the worst case? We have $T(n) \leq T(n-1) + T(n-1-d_u) + \Theta(n^k)$, where the k doesn't particularly matter (in other words, this recurrence is very bottom heavy for this exponential algorithm, so we're more concerned with the number of leaves, rather than the polynomial work done). We'll ignore the $\Theta(n^k)$ term and just deal with $T(n) \leq T(n-1) + T(n-1-d_u)$. If $d_u = 0$ then $T(n) \leq 2T(n-1)$, which implies that $T(n) \in O(2^n)$. That's no good. We can shortcut this case by noticing that if

$d_u = 0$ that we can just leave the node u in S, so we only recurse on S_{out} if $d_u \neq 0$. Otherwise, just recurse S_{in}.

Now what's the worst case? Try $d_u = 1$, then $T(n) \leq T(n-1) + T(n-2)$, and we get a Fibonnaci sequence, $T(n) \in O(\left(\frac{1+\sqrt{5}}{2}\right)^n) \approx O(2^{.7n})$. This is still exponential, but it's much better than 2^n (well, maybe not *much* better, but it's still better). Maybe we can use a linear sequence of nodes as the next worst case? Or a binary tree. Then we can put new decisions into the algorithm and find new worst cases and so on. The point is that we generally end up with more efficient exponential algorithms, not poly-time algorithms.

Backtracking is usually a very general solution technique—you can exploit almost any problem-specific characteristic to improve the efficiency, and the algorithm frequently does well in practice; worst-case inputs are often unlikely in the real world. One caution about backtracking algorithms, though, is that a rough upper bound is generally very easy to see, but narrowing in on the upper bound gets very difficult very quickly.