

CSE 101 Class Notes

Map Coloring, Greedy Algorithms

May 12, 2004

Schedule

Upcoming Topics

- May 11–18: Backtracking, Greedy algorithms
- May 20–27: Dynamic programming
- June 1–3: NP Completeness

Upcoming due dates

- May 25: Hw. 3
- June 3: Hw. 4
- June 9: Final

Backtracking: Map coloring

Backtracking is a search and optimization procedure where at each step we consider a number of possible choices, and for each one *recursively* solve a smaller version of the original problem. If the subproblems are *not* similar to the original, we may still be able to restate the problem to give it this similarity, usually by generalizing our original problem statement. Map coloring, our final example of a backtracking algorithm, is one such problem.

Given a map, we want to find a coloring using 3 colors such that no two adjacent countries share a color. That is, given a graph $G = (V, E)$ and a set of labels $\{R, G, B\}$, can we guarantee that $(v, w) \in E \implies \text{color}(v) \neq \text{color}(w)$?

This can be formulated as a backtracking problem where we choose a color for some node, then look for a solution for the rest of the graph. However, this subgraph will contain some already-colored nodes. Therefore, we generalize our original problem statement to include a partial color assignment as well as a graph: given a graph $G = (V, E)$, a set of labels $\{R, G, B\}$, and a partial labeling $C : V \rightarrow \{R, G, B, \text{nil}\}$, can we guarantee that $(v, w) \in E \implies \text{color}(v) \neq \text{color}(w)$ and either $C(v) = \text{color}(v)$ or $C(v) = \text{nil}$?

This leads to the following backtracking algorithm:

```
1   C <- { array of nil for each node of G }
2   BT3Color(G, C)
3     if C has no nil's
4       return true
5     if (there is some x with 3 neighbors (a,b,c)
6         and C[a] = R, C[b] = G, C[c] = B)
7       return false
8     if (there is some uncolored x with 2 colored neighbors (a,b)
9         and C[a] != C[b])
10      C[x] <- { R,G,B } - { C[a], C[b] }
11      return BT3Color(G, C)
12    if (there is some uncolored x with 1 colored neighbor a)
13      for each color k such that k != C[a]
14        C' <- C
15        C'[x] <- k
16        if BT3Color(G, C')
17          return true
18      return false
19    x <- any node in G
20    C[x] <- R
21    return BT3Color(G, C)
```

Running time

Exhaustive search would try every color for every node, for a cost of $O(3^n)$. In our backtracking solution, the case at lines 12-18 contains the most recursive calls. Each of these calls has size $n - 1$, so the running time is

$$T(n) = 2T(n - 1) = O(2^n)$$

an improvement comparable to those we saw last time (e.g. 5x faster for size 4, 656x faster for size 16, 431,000x for size 32).

Greedy Algorithms

At each decision, a backtracking algorithm tries each feasible choice and returns the best solution. A greedy algorithm, on the other hand, tries only the (locally) best option, ignoring locally less-appealing choices. The first hard part of coming up with a greedy algorithm is finding the right definition of “locally best” – often a naive approach will fail where a more sophisticated version of local optimality will work.

To show that a greedy algorithm works, we must show that choosing greedily will do no worse than searching exhaustively. This is the second hard part of coming up with a greedy algorithm – many problems have greedy “solutions” that are as appealing and natural as they are wrong.

Example: Job scheduling

Given a list of jobs with start- and finish-times, maximize the number of non-overlapping jobs performed. We can imagine several definitions of local optimality, choosing at each step the job with:

1. shortest duration
2. fewest conflicts
3. earliest start
4. earliest finish

None of definitions 1-3 works; to see this, we can come up with counterexamples for each. Definition 4 succeeds in these cases, but next time we'll actually prove that it works for all cases.