CS 341: ALGORITHMS (S18) — LECTURE 17

EXHAUSTIVE SEARCH

ERIC BLAIS

1. Systematic search

Let's say you are given the task to solve a comptuational problem where none of the design techniques we have covered in the class work and—even worse—no-one knows how to design an efficient algorithm for the problem. What do you do then? There are a few options.

- Design a *heuristic* argument that runs quickly but has no provable guarantee of correctness.
- Design an *approximation algorithm* that does not solve the problem optimally but at least gives some guarantees about its solution.
- Design an *exponential-time* exact algorithm for the problem.

Today's lecture aims to explore the last approach: it's never ideal to have an algorithm with time complexity that grows exponentially with the size of the input, but in some cases that's the best option. (If the input data is reasonably bounded, and/or if you have massive amounts of computation available—and if there really aren't any more efficient algorithms out there for the task.) Our goal is to see some techniques that will enable us to design these exponential-time algorithms in a way that guarantees they will be correct, and hopefully keep them as efficient as possible.

2. Backtracking

Consider the following generalization of the 3Sum problem we saw earlier in the course.

Definition 17.1. In the *Subset sum* problem, we are given as input an array of n elements $1, 2, \ldots, n$ with positive weights $w_1, \ldots, w_n$, along with a target weight T. We must determine whether there is a subset $S \subseteq \{1, 2, \ldots, n\}$ of elements with total weight $\sum_{i \in S} w_i = T$ or not.

There is no known polynomial-time algorithm for this problem. In fact, as we will see in the next part of the course, the Subset Sum problem is NP-complete, which means that it is quite possible that no such algorithm exists. So what can we do? We can explore the

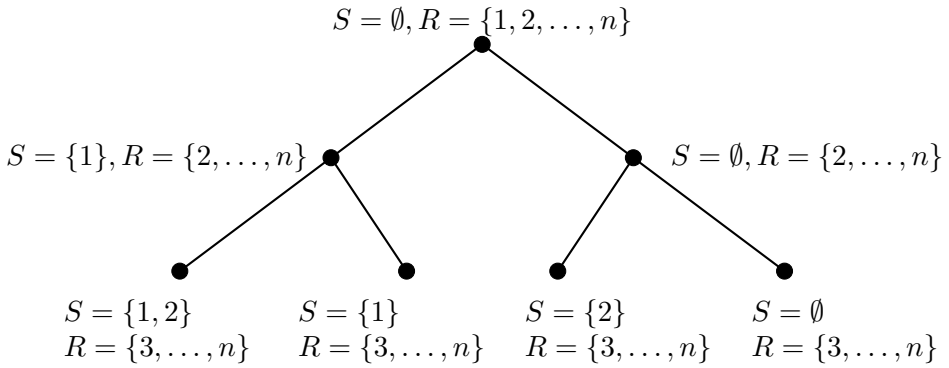
space of all possible sets S, one by one in some fixed order.

$$\begin{aligned} &S = \emptyset \\ &S = \{1\} \\ &S = \{2\} \\ &S = \{1,2\} \\ &S = \{3\} \\ &\quad\vdots \\ &S = \{1,2,\ldots,n\} \end{aligned}$$

Running this algorithm on some examples, however, we might notice some obvious inefficiencies. For example, consider an input where $w_1 > T$. We discover then that with $S = \{1\}$, we already have $\sum_{i \in S} w_i = w_1 > T$, so there's really no point in checking $S = \{1,2\}$, $S = \{1,3\}$, $S = \{1,2,3\}$, etc.

With the *backtracking* technique, we can avoid checking all unnecessary cases while still making sure that we don't miss any potential solutions. The idea is that we can consider the set of all possible *configurations*—corresponding to full or partial solutions—in a tree structure.

For the Subset Sum problem, a configuration corresponds to a set $S \subseteq \{1,2,\ldots,k\}$ of elements that we have decided to add to our solution, and a set $R \subseteq \{k+1,\ldots,n\}$ of elements that we might still add to the set. Then the top of our tree of configurations looks like this:



Every node at level k has two children: one corresponding to the case where we add element $k+1$ to S, and one where we don't. (In both cases, we remove $k+1$ from R.) We never build the tree explicitly, but we design the algorithm to explore this tree of configurations with the following rules at a given node with current sets S and R:

- If $\sum_{i \in S} w_i = T$, SUCCESS! We have solved the problem and can return True;
- If $\sum_{i \in S} w_i > T$, DEAD END. We stop exploring this branch since all nodes underneath the current one will overshoot the target, so we *backtrack* to the parent node directly and continue exploring other paths from there.
- If $\sum_{i \in S \cup R} w_i < T$, DEAD END. We again backtrack since all nodes underneath the current one will undershoot the target.

We can implement this easily in an algorithm for the Subset Sum problem. Or, better yet, we can use the general backtracking algorithm and implement only a few helper functions instead.

Algorithm 1: BACKTRACKING()

```
Initialize the set A of active configurations;
while A ≠ ∅ do
    C ← next configuration of A;
    if C is a solution return True;
    if C is not a dead end then
        EXPAND(C) and add the resulting configurations to A;
return False;
```

Note that there are two ways that we can implement the BACKTRACKING algorithm: by keeping the set of active configurations in $\mathcal{A}$ in a queue or in a stack. The two options correspond to designing an algorithm that explores the configuration tree using BFS or DFS—which is best depends on the width and the height of the configuration tree. For the Subset sum problem, the best option if we want to optimize the set complexity of the algorithm is DFS, as it will require keeping only $O(n)$ active configurations in memory at any time, versus up to $\Omega(2^n)$ active configurations if we use BFS instead.

2.1. **Time complexity analysis.** What is the time complexity of this algorithm? It is still $\Theta(2^n)$! This is a good exercise to try: can you come up with concrete examples where the backtracking algorithm explores $\Omega(2^n)$ nodes of the configuration tree before returning? In the *worst-case* time complexity setting, these hard examples show that we haven't gained anything over the simple exhaustive search, but in practice there are many situations where the gains obtained by the backtracking algorithm over naïve search are quite significant.

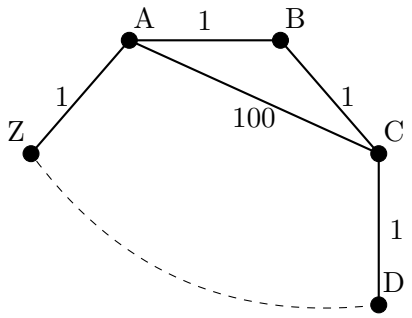
3. Branch-and-bound

There is another similar algorithm technique when we consider optimization problems instead of decision problems. Consider for example the famous Travelling Salesman problem.

Definition 17.2. In the *Travelling Salesman problem (TSP)*, we are given a weighted undirected graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^{\geq 0}$. Our goal is to find a cycle C that goes through each vertex in V exactly once (called a TSP tour) with minimum weight $\sum_{e \in C} w(e)$.

The Travelling Salesman Problem is also NP-complete. We don't have any polynomial-time algorithm for the problem—and in fact the optimal tour that visits all $< 2M$ cities on Earth is still not known (see `http://www.math.uwaterloo.ca/tsp/world/index.html` for all the details and instructions on how to submit your improved tour when you beat the world record!).

We can again solve this problem by computing the lengths of all $|V|!$ possible tours of the graph, but again this approach feels wasteful. Consider the following graph:



If we consider the tour $A \to B \to \cdots \to Z$ first, we obtain a tour with total weight 26. This means that *any* tour which uses the edge $A \to C$ will not be optimal, since it will have weight strictly greater than 26. To avoid multiple unnecessary checks, we can use the *branch and bound* technique. At each step, we

- Consider a set A of possible configurations;
- *Branch* by splitting A into subsets $A_1, \ldots, A_t$ of possible configurations; and
- *Bound* the minimum value of any solution in configuration A_i; we discard A_i if this value is greater than the optimal solution discovered so far.

The general branch-and-bound algorithm looks very similar to the BACKTRACKING algorithm.

Algorithm 2: BRANCHANDBOUND()

Initialize the set $\mathcal{A}$ of active configurations;
Initialize `best` $\leftarrow \infty$;
while $\mathcal{A} \neq \emptyset$ **do**
  $C \leftarrow$ next configuration of $\mathcal{A}$;
  if *C is a solution and* VALUE(C) $<$ `best` **then**
    `best` $\leftarrow$ VALUE(C);
  else if *C is not a dead end and* MINVAL(C) $<$ `best` **then**
    EXPAND(C) and add the resulting configurations to $\mathcal{A}$;
return `best`;

For the travelling salesman problem, a configuration C consists of

- a set $I \subseteq E$ of edges that are *included* in the tour, and
- a set $X \subseteq E$ of edges that are *excluded* in the tour.

(We maintain $I \cap X = \emptyset$ throughout.)

Initially, $I = X = \emptyset$. We *branch* by considering any edge $e \in E \setminus (I \cup X)$ and constructing the two configurations $C_1 = (I \cup \{e\}, X)$ and $C_2 = (I, X \cup \{e\})$.

If I forms a tour and has smaller weight than the best tour found so far, we update the minimum weight and the best tour found so far to I.

Otherwise, we *bound* the solution by checking whether:

- The total weight $\sum_{e \in I} w(e)$ is not larger than the weight of the best tour found so far;
- Every vertex has degree at most 2 in I;
- There is no cycle in I;

- The graph $G' = (V, E \setminus X)$ is 2-connected.

If any of the condition is not satisfied, then the current configuration cannot lead to an optimal tour so we discard it.

There are a number of optimizations to this algorithm that we can implement: we can get a stronger lower bound on the cost of any tour obtained from some configuration, we can use clever branching strategies to explore more promising configurations first, etc. None of those enhancements lead to a polynomial-time algorithm, but they do yield algorithms that perform reasonably well on instances of TSP encountered in practice.