CS 341: Algorithms
Module 8: Exhaustive Search

Armin Jamshidpey, Eugene Zima

Based on lecture notes by many previous CS 341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

Backtracking

- Definition from wikipedia:
    - **Backtracking** is a general algorithm for finding all (or some) solutions to decision or optimization problems, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

To illustrate the basic principles of backtracking, we consider the **Knapsack** problem.
Recall that a problem instance consists of a list of profits, $P = [p_1, \cdots, p_n]$; a list of weights $W = [w_1, \cdots, w_n]$; and a capacity, M. These are all positive integers. It is required to find the maximum value of $\sum p_i x_i \leq M$ and $x_i \in \{0, 1\}$ for all i. An n-tuple $[x_1, x_2, \cdots, x_n]$ of 0's and 1's is a feasible solution if $\sum w_i x_i \leq M$.

One naive way to solve this problem is to try all 2^n possible n-tuples of 0's and 1's. We can build up an n-tuple one coordinate at a time by first choosing a value for x_1 , then choosing a value for x_2 , etc. Backtracking provides a simple method for generating all possible n-tuples. After each n-tuple is generated it is checked for feasibility. If it is feasible, then its profit is compared to the current best solution found to that point. The current best solution is updated whenever a better feasible solution is found.
Denote by $X = [x_1, \cdots, x_n]$ the current n-tuple being constructed, and **CurP** will denote its profit. **OptX** will denote the current optimal solution and **OptP** is its profit.

Algorithm 1: Knapsack1(ℓ)

```
global $X, OptX, OptP$
if $\ell = n + 1$ then
    if $\sum_{i=1}^{n} w_i x_i \leq M$ then
        $CurP \leftarrow \sum_{i=1}^{n} p_i x_i$
        if $CurP > OptP$ then
            $OptP \leftarrow CurP$
            $OptX \leftarrow [x_1, \ldots x_n]$
else
    $x_\ell \leftarrow 1$
    Knapsack1($\ell + 1$)
    $x_\ell \leftarrow 0$
    Knapsack1($\ell + 1$)
```

Before invoking Algorithm 1, we would initiakize **OptP** to be 0. Then we call Knapsack1(1) to solve the given problem instance.

The recursive calls to Algorithm 1 implicitly produce a binary tree called the *state space tree* for the given problem instance. When $n = 3$, this tree is given in Figure 1. A backtrack search performs a depth-first traversal of the state space tree.

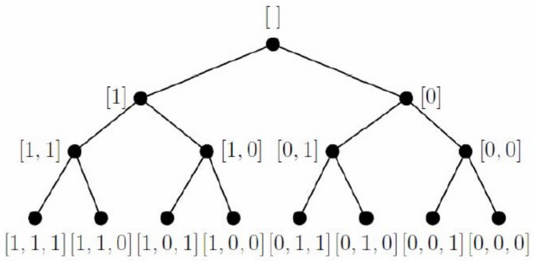


Figure 1: State space tree when $n = 3$.

Algorithm 1 generates the 2^n binary n-tuples in reverse lexicographic order. It takes time $\Theta(n)$ to check each solution, and so the asymptotic running time for this algorithm is $\Theta(n2^n)$. Of course this approach is impractical for $n > 40$, say. Notice that not all n-tuples of 0's and 1's are feasible, and a fairly simple modification to the backtracking algorithm would take this into account. For the **Knapsack** problem, we observe that we must have

$$\sum_{i=1}^{\ell-1} w_i x_i \leq M$$

for any partial solution $[x_1, \cdots, x_{\ell-1}]$.

In other words, we can check partial solutions to see if the feasibility condition is satisfied. If $\ell \leq n$, then denote

$$CurW = \sum_{i=1}^{\ell-1} w_i x_i$$

Our modified algorithm is Algorithm 2. In this algorithm, we check feasibility before making a recursive call. The algorithm is invoked initially with $\ell = 1$ and $CurW = 0$.

Algorithm 2: Knapsack2(ℓ, $CurW$)

```
global $X, OptX, OptP$
if $\ell = n + 1$ then
    if $\sum_{i=1}^{n} p_i x_i > OptP$ then
        $OptP \leftarrow \sum_{i=1}^{n} p_i x_i$
        $OptX \leftarrow [x_1, \ldots, x_n]$
else
    if $CurW + w_\ell \leq M$ then
        $x_\ell \leftarrow 1$
        Knapsack2($\ell + 1, CurW + w_\ell$)
        $x_\ell \leftarrow 0$
        Knapsack2($\ell + 1, CurW$)
    else
        $x_\ell \leftarrow 0$
        Knapsack2($\ell + 1, CurW$)
```

Whenever $CurW + w_\ell > M$, we have a situation known as *pruning*. This means that one of the two recursive calls (the one corresponding to $x_\ell = 1$) is not performed. Because we do not traverse the corresponding subtree of the state space tree, we think of this subtree as having been pruned.

A more sophisticated method of pruning is to use a bounding function. We require a few preliminary definitions, which will apply to any backtracking algorithm for a maximization problem. Let `profit`(X) denote the profit for any feasible solution X. For a partial feasible solution, say $X = [x_1, \cdots, x_{\ell-1}]$, define $P(X)$ to be the maximum profit of any feasible solution which is a descendant of X in the state space tree. In other words, $P(X)$ is the maximum value of `profit`(X') taken over all feasible solutions $X' = [x'_1, \cdots, x'_n]$ such that $x_i = x'_i$ for $1 \leq j \leq \ell - 1$. It follows from the definition that, if $X = []$, then $P(X)$ is the optimal profit of the given problem instance.

In general, $P(X)$ could be computed exactly by traversing the subtree with root node X.
In order to avoid doing this, if possible, we will employ a bounding function. A *bounding function* is a real-valued function, denoted B, defined on the set of nodes in the state space tree and satisfying the following condition:

$$\text{For any feasible partial solution } X,\ B(x) \geq P(X)\,.$$

This property says that $B(X)$ is an upper bound on the profit of any feasible solution that is a descendant of X in the state space tree. (For a minimization problem, the definition is the same, except that the inequality is reversed.)

Once a bounding function $B(X)$ has been specified, it can be used to prune the state tree, as follows. Suppose that at some stage of the backtracking algorithm, we have a current partial solution $X = [x_1, \cdots, x_{\ell-1}]$, and $OptP$ is the current optimal profit. Whenever $B(X) \leq OptP$, then we have that

$$P(X) \leq B(X) \leq OptP$$

This implies that no descendants of X in the state space tree can improve the current optimal profit. Hence we can prune this entire subtree, i.e., we perform no recursive calls from this partial solution X.

We now show how to define a useful bounding function for the **Knapsack** problem using the **Rational Knapsack** problem. Recall that the **Rational Knapsack** problem is exactly the same as the **Knapsack** problem we have been considering, except that the condition $x_i \in \{0, 1\}$ is replaced by the weaker condition $0 \leq x_i \leq 1$ (i.e., the x_i's are allowed to be arbitrary rational numbers in the interval $[0, 1]$). Recall also that the **Rational Knapsack** problem can be efficiently solved by a greedy algorithm, in which we consider the objects in non-decreasing order of their profit-to-weight ratios.

Assume we have a greedy algorithm **GreedyRKnap**$(k; p_1, \cdots, p_k, w_1, \cdots, w_k, M)$ to find the optimal solution to the **Rational Knapsack** problem for an arbitrary problem instance $p_1, \cdots, p_k, w_0, w_1, \cdots, w_k, M$ which consists of k objects. We use the algorithm **GreedyRKnap** to define a bounding function for the **Knapsack** problem as follows. Given a (feasible) partial solution $X = [x_1, \cdots, x_{\ell-1}]$ for **Knapsack**, define

$$\begin{aligned} B(X) &= \sum_{i=1}^{\ell-1} p_i x_i + \textbf{GreedyRKnap}(n-\ell+1; p_\ell, \cdots, p_n, w_\ell, \cdots, w_n \\ &\quad , M - \sum_{i=1}^{\ell-1} w_i x_i) \\ &= \sum_{i=1}^{\ell-1} p_i x_i + \textbf{GreedyRKnap}(n-\ell+1; p_\ell, \cdots, p_n, w_\ell, \cdots, w_n \\ &\quad , M - CurW) \end{aligned}$$

Thus $B(X)$ is equal to the sum of:

1. the profit already obtained from objects $1, \cdots, \ell - 1$, plus
2. the profit from the remaining $n - \ell + 1$ objects, using the remaining capacity
$$M - CurW,$$
but allowing rational x_i's.

If we restricted each x_i to be 0 or 1 in part 2 above, then we would obtain $P(X)$. Allowing the x_i's with $\ell \leq i \leq n$ to be rational may yield a higher profit, so $B(X) \geq P(X)$ and B is indeed a bounding function. It is also easy to compute, and thus this may be useful for pruning.

Suppose we want to solve an instance of the **Knapsack** problem. It will be useful to sort the objects in decreasing order of profit/weight ahead of time, before we begin the backtracking algorithm. Then when we wish to evaluate our bounding function for a subset of objects $\ell, \cdots, n$, they are already given the appropriate order. Thus we will assume that

$$\frac{p_1}{w_1} \geq \cdots \geq \frac{p_n}{w_n}\ .$$

The improved algorithm is given as Algorithm 3.

Algorithm 3: Knapsack3(ℓ, $CurW$)

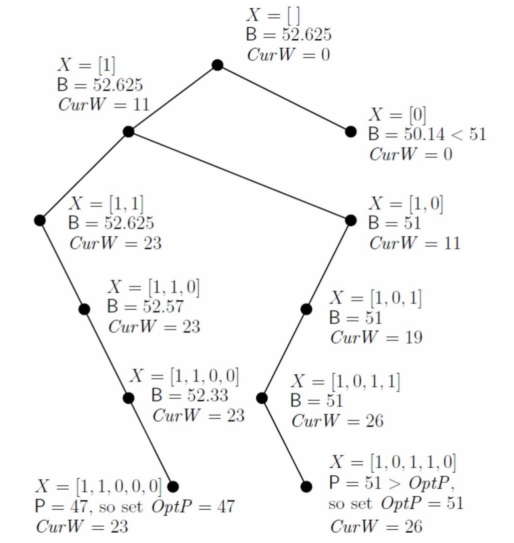
```
1  external GreedyRKnap()
2  global $X$, $OptX$, $OptP$
3  if $\ell = n + 1$ then
4      if $\sum_{i=1}^{n} p_i x_i > OptP$ then
5          $OptP \leftarrow \sum_{i=1}^{n} p_i x_i$
6          $OptX \leftarrow [x_1, \ldots, x_n]$
7  else
8      $B \leftarrow \sum_{i=1}^{\ell-1} p_i x_i + \mathrm{GreedyRKnap}(n - \ell + 1;$
           $p_\ell, \ldots, p_n, w_\ell, \ldots, w_n, M - CurW)$
9      if $B \leq OptP$ then return
10     if $CurW + w_\ell \leq M$ then
11         $x_\ell \leftarrow 1$
12         Knapsack3($\ell + 1$, $CurW + w_\ell$)
13     if $B \leq OptP$ then return
14     $x_\ell \leftarrow 0$
15     Knapsack3($\ell + 1$, $CurW$)
```

It is very important to note that we check to see if the pruning condition, $B \leq OptP$, is true before every recursive call made during the algorithm. This is because the value of $OptP$ can increase as the algorithm progresses, and so we check to see if we can prune every time we are preparing to choose a new value for x_ℓ.

Suppose we have five objects, having weights $11, 12, 8, 7, 9$; profits $23, 24, 15, 13$ and 16 (respectively); and capacity $M = 26$. Note that the objects are already arranged in decreasing order of profit/weight. We draw in Figure 2 the space tree traversed in the course of the backtrack algorithm **Knapsack3**. At each node, we record the current values of $X, B(X)$ and $CurW$.
Observe that pruning using the bounding function happens three times in the example:

1. When $X = [1, 0]$ and $[1, 0, 1]$, we have $OptP = 51$ after the first recursive call. Since $B(X) = 51$, the second recursive call is not performed. Therefore, the subtree with root nodes $[1, 0, 0]$ and $[1, 0, 1, 0]$ are pruned.
2. When $X = [0]$, we have $OptP = 51$ and $B(X) = 50.14 < 51$, so no recursive calls are performed. The subtree with root node $[0]$ is pruned.

Also, pruning due to the capacity being exceeded occurs at nodes $[1, 1], [1, 1, 0], [1, 1, 0, 0]$ and $[1, 0, 1, 1]$. At each of these nodes, the first recursive call, with $x_\ell = 1$, is not performed.



$X = [\,]$
$\mathsf{B} = 52.625$
$CurW = 0$
$X = [1]$
$\mathsf{B} = 52.625$
$CurW = 11$
$X = [0]$
$\mathsf{B} = 50.14 < 51$
$CurW = 0$
$X = [1, 1]$
$\mathsf{B} = 52.625$
$CurW = 23$
$X = [1, 0]$
$\mathsf{B} = 51$
$CurW = 11$
$X = [1, 1, 0]$
$\mathsf{B} = 52.57$
$CurW = 23$
$X = [1, 0, 1]$
$\mathsf{B} = 51$
$CurW = 19$
$X = [1, 1, 0, 0]$
$\mathsf{B} = 52.33$
$CurW = 23$
$X = [1, 0, 1, 1]$
$\mathsf{B} = 51$
$CurW = 26$
$X = [1, 0, 1, 1, 0]$
$\mathsf{P} = 51 > OptP$,
so set $OptP = 51$
$CurW = 26$
$X = [1, 1, 0, 0, 0]$
$\mathsf{P} = 47$, so set $OptP = 47$
$CurW = 23$