Recitation Note - CS430 Fall 2014

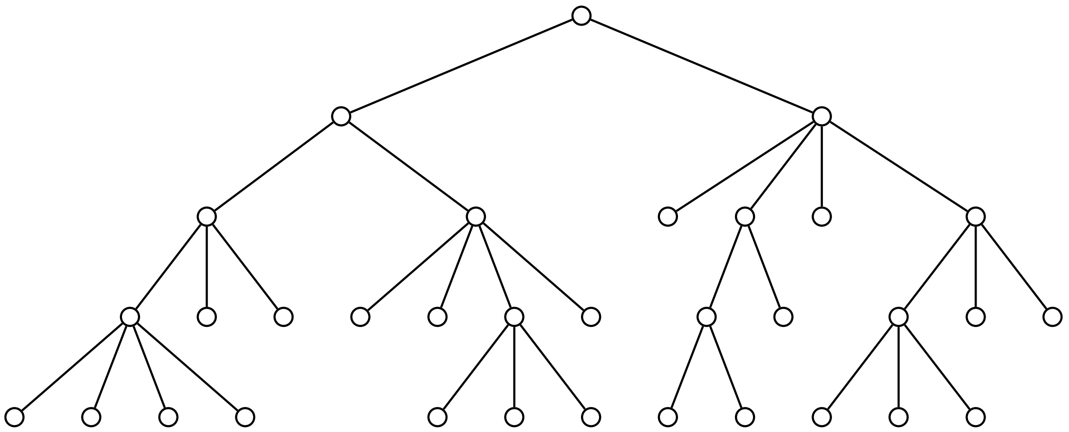
Taeho Jung
Illinois Institute of Technology

Oct 10, 2014

- I do not guarantee I will prepare a note for every recitation.
- Today's recitation is on greedy algorithms.

1 Difference between Dynamic Programming and Greedy Algorithm

Let's suppose given a tree, we need to find a path from root to leaf whose total key values (sum of key values of the nodes in the path) is the maximum.



A stupid algorithm will do a brute-force search. It merely calculates total key value for all paths, and then find the maximum one.

A dynamic programming-like algorithm will do the similar thing. However, at every path, it will keep the records on the sum of keys for the visited part, and it will avoid re-calculating the sum of keys which it has calculated before.

On contrary, a greedy algorithm will try to find the child with largest key at every node, and visit the child with largest key only. It repeats such greedy selection starting from the root until it reaches the leaf node.

As such, dynamic programming visits every case in a brute-force manner, but it avoids unnecessary re-calculation via memoization in the table. On the other hand, greedy algorithm will discard the search space based on the local view at each iteration, and finally find the solution. Because greedy algorithm discards some cases without visiting it, it does not always find the optimum solution.

2 Various Knapsack Problems

There is a set of items $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$, who all have their own sizes s_i and values v_i. We need to select a subset of items to put in a infinitely elastic plastic bag whose capacity is S, but we also want to maximize the total value of the items. We assume the total size of $\mathcal{X}$ is greater than S (otherwise the problem is too easy).

2.1 Fractional Knapsack

In this version, the item can be divided. How do we find such a set?

Algorithm

Calculate $p_i = v_i/s_i$, *i.e.,* value per size. Then, find the item with the largest p_i (say x_{m1}), and put it in the bag. Then, find the next item with the second largest p_i (say x_{m2}), and fill the bag with x_{m2}. So and so forth until the bag is full.

Correctness

Let's first assume the set of items in the bag does not yield the maximum value, and see if there is any contradiction. If there is, it means our assumption is wrong, and the set of items chosen by the algorithm does yield the maximum value.

If the total value is not maximized, it means there is another solution which can further increase the value. Then, at a certain iteration in our algorithm (say x-th iteration of the item choice), we could have chosen another x'_{mx} instead of x_{mx} to fill the plastic bag to further increase the final value. However, we know x_{mx}'s value per size is greater than x'_{mx}'s one (that's why x_{mx} is chosen by the algorithm). That is, x_{mx} is pricier than x'_{mx}. Therefore, if we replaced any portion of x_{mx} with x'_{mx}, the final value would become smaller only, and it is not possible to further increase the total value.

Therefore, it is not possible to increase the total value, and the total value of those selected items is the maximum one.

Time complexity

We first calculate p_i's ($O(n)$), and then sort them to iteratively find the item to fill in ($O(n \lg n + n)$). Therefore, the final complexity of this greedy algorithm is $O(n \lg n)$.

2.2 0-1 Knapsack

In this version, the item cannot be divided, and every item is either put in or not (0 or 1). This is a famous problem which is very hard, and there is no fast algorithm is found yet. It is related to the special class of computation problems (NP-complete and NP-hard) whose polynomial-time solution is not discovered yet.

Instead, we can use the similar greedy heuristics to find a set of items in $O(n \lg n)$ time. The total value of the set may not be maximized in some cases, but it is proven that the total value is at least $v_{max}/2$ times the real optimum solution achieved from the brute-force search (v_{max} is the maximum value among the items). This is called 'approximation' for NP-complete or NP-hard problems, and the approximation ratio of this approximation algorithm is $v_{max}/2$.

3 Minimum Spanning Tree (MST) algorithms

3.1 Prim's Algorithm

Description omitted. Please watch the video or read http://en.wikipedia.org/wiki/Prim's_algorithm

3.2 Kruscal's Algorithm

Description omitted. Please watch the video or read http://en.wikipedia.org/wiki/Kruskal's_algorithm

4 When to consider Dynamic Programming/Greedy Algorithm

A problem has the optimal substructure if part of the optimum solution is the optimal solution to the sub-problem. Then, the optimum solution can be efficiently calculated from the optimum solutions to the sub-problems. In this type of problems, Dynamic Programming or Greedy Algorithm can be used to calculate the solution.

Here are some examples of problems who have the optimal substructure.

1. Finding the shortest path: If $v_1, v_2, \cdots, v_i, \cdots, v_n$ is the shortest path from v_1 to v_n, $v_1, v_2, \cdots, v_i$ is also the shortest path from v_1 to v_i.

2. Knapsack problem: If $x_1, x_2, \cdots, x_i, \cdots, x_n$ is the subset of items out of $\mathcal{X}$ whose total value is maximized, $x_1, x_2, \cdots, x_i$ is also an optimum set of items for the sub-problem with smaller constraints (sum of weights $\leq \sum_{k=1}^{i} w_k$).

3. Activity scheduling problem: If $a_1, a_2, \cdots, a_i, \cdots, a_n$ is the maximum set of the activities in the scheduling problem, $a_1, a_2, \cdots, a_i$ is also the maximum set of the activities in the sub-problem of the scheduling problem (among subset of activities whose deadline is earlier than a_i).

On the other hand, some problems do not have the optimal substructure.

1. Finding the longest path: Even if v_1, v_2, v_3 is the longest path from v_1 to v_3, v_1, v_2 may not be the longest path from v_1 to v_2. An undirected cycle of v_1, v_2, and v_3 is an example.

2. Finding the cheapest airplane ticket: Suppose cheapest airfare from A_1 to A_2 is $A_1 \rightarrow A_3 \rightarrow A_2$. Even so, $A_1 \rightarrow A_3$ is not the cheapest airfare from A_1 to A_3. We can add a transfer to lower the airfare. On the other hand, adding transfer does not always lead to discounts.