Greedy Algorithms: Theory

Optimization Problems

Greedy algorithms and dynamic programming algorithms\(^1\) are usually designed for a special class of problems called optimization problems.

**Optimization Problem.** A computational problem in which the object is to find the best of all possible solutions is called an optimization problem.

Typically, an optimization problem comes with an objective function which maps solutions to the problem to a set of values. An optimal solution is a solution that either maximizes or minimizes (depending on the requirements of the problem) the objective function.

**Example.** Making change problem is an optimization problem. Given \(N\), the amount of money to give in change, the problem space consists of a potentially large number of possible solutions: any collection of coins that adds up to \(N\) is a possibility. For example, here are all possible ways in which one can make 11 cents in change:

\(^1\)To be studied next.
The objective function evaluating the quality of a solution \( \text{Sol} = \{\text{coin}_1, \ldots, \text{coin}_s\} \):

\[
\text{val} (\text{Sol}) = |\text{Sol}| = s.
\]

For the Making Change problem, an optimal solution \textit{minimizes} the value of the objective function \text{val}().

Properties of Optimization Problems

Problems solvable using both dynamic programming and greedy algorithms exhibit the \textit{optimal substructure property}.

Additionally, problems solving using \textit{greedy algorithms} exhibit the \textit{greedy choice property}.

Optimal Substructure property.

Subproblems. Given a computational problem \( P \) with input \( I \), a subproblem of \( P_s \) of \( P \) is any computational problem whose input \( I_s \subset I \) and whose desired output should satisfy the same properties as the output of \( P \).

Example. Consider an instance \( P \) of the Making Change problem, asking us for the best way to change 45 cents. A request to make change for any sum that is \textit{smaller} than 45 cents\(^2\), e.g., 36 cents, is a subproblem of \( P \).

Problem decomposition. A problem is called \textit{decomposable} if it can be solved, by solving a number of its subproblems.

Optimal Substructure property. A problem is said to have an \textit{optimal substructure property} iff its optimal solution contains within it optimal solutions to its subproblems.

Theorem 1. Problem Activity Selection has \textit{optimal substructure property}.

Proof. Consider an instance \( P \) of the Activity Selection problem with input \( A[1..M][1..2] \). Let set \( S = \{j_1, \ldots, j_k\} \) \((k < M)\) be an optimal solution to \( P \). (Here, \( j_1, \ldots, j_k \) are indexes of the activities from \( A[][] \) that are being scheduled.)

Consider some activity \( A[j] \) where \( j \in S \). \( A[j] \) splits the list of all activities \( A[] \) into three parts:


2. \( A^{conflict} = \{ A[i] \in \text{[]} | (A[i][1], A[i][2]) \text{ subseteq } (A[j][1], A[j][2]) \neq \emptyset \} \): activities that conflict with \( A[j] \).

\(^2\)Our definition uses \( I_s \subset I \) relation between the inputs of a problem and its subproblem. In case of the Making Change problem, we can view input rendered in a unary alphabet of bits. The set of bits needed to render 45 will be a superset to the set of bits needed to render any smaller number of cents.

Similarly, $A[j]$ partitions the optimal solution $S$ into three parts:

2. $\{ j \}$.

The structure of the problem is shown in Figure 1. The set $S$ of optimally scheduled activities is shown in bold. The partitions of $A$ and $S$ are circled by dotted lines.

We claim that:

- $S^{early}$ is the optimal solution for the Activity Selection problem with input $A^{early}$;
- $S^{late}$ is the optimal solution for the Activity Selection problem with input $A^{late}$;

We prove the first statement. The proof of the second is symmetric.

Suppose $S^{early}$ is NOT an optimal solution for the Activity Selection problem with input $A^{early}$. Consider then a set $S' = \{ f_1, \ldots, f_l \}$, which is an optimal solution for this problem. We know that $\text{val}(S') = |S'| > |S^{early}| = \text{val}(S^{early})$. But then, $S$ cannot be an optimal solution for original problem, because the following set

$$S^* = S' \cup \{ j \} \cup S^{late}$$

is an optimal solution and $\text{val}(S^*) < \text{val}(S)$.

Indeed, $S^*$ is a solution for the Activity Selection problem, because no $A[i]$, such that $i \in S'$ conflicts with $A[j]$ or with any $A[j']$ where $j' \in S^{late}$. And because $|S'| < |S^{early}|$, $|S^*| > |S|$, and therefore $|S|$ cannot be an optimal solution, which contradicts our original assumption.
Greedy Choice Property

Greedy Choice property. An optimization problem has greedy choice property if

- It has optimal substructure property.
- The optimal solution of a problem can be constructed from an optimal solution to a subproblem by extending the solution of the subproblem in a locally optimal way.

Locally optimal way. Consider a problem $P$ with solution $S$. We want to extend the solution of $P$ to a superproblem $P'$. Consider a list of alternatives $A_1, \ldots, A_k$, such that $S \cup A_i$ is a feasible (albeit not necessarily optimal) solution of $P'$. Consider a local quality function $q(A_i)$, which supplies for each possible choice $A_i$ a local goodness estimate.

A locally optimal way of selecting a solution for $P'$ is a solution $S \cup A_i$, such that $q(A_i)$ is optimal$^3$.

Philosophy of Greedy Choice

The key feature of the greedy choice property is the fact that at each step we add into our solution an element with the best local quality estimate. This makes greedy algorithms very efficient, because only one selection needs to be made per algorithm iteration, and that selection is never ”questioned”.

Greedy Choice Property Theorems

Theorem 2. Problem Activity Selection has greedy choice property.

Proof. Consider an instance of the Activity Selection problem with input $A[1..M][1..2]$. Let $A^t$ be a subproblem of this Activity Selection problem containing only activities $A[i]$ that start after time $t$.

Further, Let $A[k] \in A^t$ be the activity with the earliest finishing time, i.e., $A[k][2] = \min_{A[k] \in A^t}(A[j][2])$. We will prove that $A[k]$ is included in some optimal solution for $A^t$.

Let $S^t$ be an optimal solution for the Activity Selection problem for $A^t$. Consider an element $A[i] \in S^t$ with the earliest finish time. We consider two cases:

- **Case 1:** $A[i] = A[k]$, i.e., $A[k] \in S^t$.
- **Case 2:** $i \neq k$. Because $A[k]$ ends earlier than $A[i]$, it follows, that $A[i] \notin S^t$. Consider now a set $S'_k = S^t - \{A[i]\} \cup \{A[k]\}$.

We claim that

(a) $|S'_k| = |S^t|$. This is indeed so, as $S'_k$ is formed from $S^t$ by replacing one activity with another single activity.

---

$^3$Minimal or maximal, depending on the nature of the problem.
Figure 2: Selection of shortest activity is not an optimal greedy strategy for the Activity Selection problem.

(b) $S^k_t$ is conflict-free. Indeed, $S^t - \{A[i]\}$ contains no activities that conflict with $A[i]$ (as $S^t$ is an optimal solution for $A^t$). But $A[k][2] < A[i][2]$ (i.e., $A[k]$ will finish earlier than $A[i]$). Therefore, no activity from $S^t - \{A[i]\}$ can possibly conflict with $A[k]$. But then, $S^k_t$ contains no conflicts.

From these two statements we obtain that $S^k_t$ is an optimal solution for $A^t$.

**Theorem 3.** Problem Fractional Knapsack has greedy choice property.

(Left as exercise)

**Greedy Algorithm Design**

**Greedy algorithms** can be designed for virtually any optimization problem. They can be:

- **optimal** for problems that have optimal substructure property and greedy choice property.
- **approximations** for problems that have only optimal substructure property.

For some problems greedy algorithms give good approximations. For some other problems, greedy algorithms can be really really bad (sometimes, can lead to worst solutions).

**Design.** Given a problem, establish the following:

1. Proper objective function for the problem.
2. Optimal substructure property for the problem.
3. Proper quality function for evaluating solution steps (i.e., your greedy approach).
4. Greedy choice property.

Note, that greedy choice property can only be established after you select your greedy strategy, as computational problems admit optimal greedy algorithms for some greedy approaches, but not for others.
Example. Consider two greedy approaches to the Activity Selection problem.

- Select the activity that finishes the earliest.
  
  - Theorem 2 shows that Activity Selection has greedy choice property w.r.t. this strategy.

- Select the shortest activity.
  
  - Figure 2 shows that Activity Selection with this strategy does not have greedy choice property. There, the optimal solution is the activity set \( \{a_1, a_3\} \), while selecting the shortest activity will lead to the solution \( \{a_2\} \), which is not optimal.

Greedy algorithm outline. Once you either:

- establish that an optimal greedy algorithm exists for a problem, or
- decide that a greedy approximation algorithm is good enough for you

you need to devise the algorithm itself.

Greedy algorithms can have the following components:

- Selection function. On each step, this function, given the currently constructed part of the solution, and current choices for including into the solution, will make the greedy choice.

- Feasibility function. On each step, this function indicates whether the currently constructed part of the solution is feasible, i.e., can lead to a solution.

- Pruning function. On each step, once the greedy choice is made, this function will eliminate all possibilities that conflict with the current portion of the solution.

- Objective function. This function evaluates the current solution or partial solution and returns a numeric value for it.

- Solution function. This function evaluates current partial solution to test if it is, in fact, a complete solution.

The overall organization of a greedy algorithm is:

```cpp
Initialization Step: Set solution \( S \leftarrow \emptyset \);
Solution Step: while \( S \) is feasible and \( S \) is NOT a solution do
    Establish space of choices \( C \);
    Update solution: \( S \leftarrow S \cup \text{Greedy Select}(C) \);
end if
if \( S \) is NOT feasible return FAIL
else return \( S \);
```