Aalto University

Lecture Notes - Week X

Approximations

Fernando Dias, Philine Schiewe and Piyalee Pattanaik

March 11, 2024

CHAPTER 1
Greedy Algorithms

In Mathematics and Computer Science, approximation algorithms are alternative approaches to finding optimal solutions or near-optimal ones with **guaranteed** distance.

It does differ from relaxations (where **some** of the constraints are eliminated/replaced). Two common strategies are **Greedy Algorithms** and **Local Search**.
Starting with greedy methods, a quote from a movie which encapsulates exactly the reason behind those algorithms:

Greed, in the lack of a better word, is good.

Michael Douglas in ”Wall Street” (1987)

In Computer Science and Discrete Mathematics, **greedy** means ”*the best at this moment*”. It looks for a **local optimal** solution at each iteration. Hence, there is **no** clear guarantee of **global** optimality.

However, there are some exceptions.

1.1 GREEDY PROPERTIES

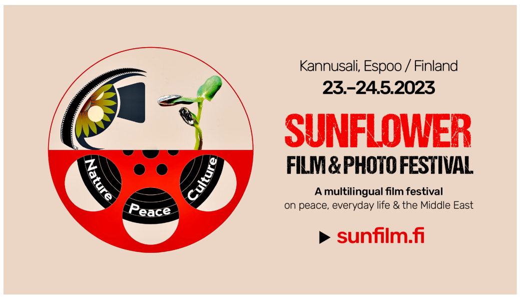
Algorithms such as Dijkstra’s and Prim’s are **optimal** and are **greedy methods**. This leads to the following thought:

Question: *What properties do some problems have that make their greedy approach optimal?* Those are:

Choice Property and **Optimal Substructure**.
As **iterative** problems, at each iteration, greedy methods such as those listed above find the **best** (**local optimal**) solution for that particular version of the problem. They do consider previous solutions, but **do not regard** future choices and remaining solutions from alternative sub-problems. However, **horizon effect** is also a risk.

In addition, making a single decision per iteration is **constantly reducing** the problem. It differs from **dynamic program** because the former exhaustively searches **all** possible solutions of all subproblems and also guarantees an optimal solution.

As present in dynamic programming, greedy algorithms require **sub-optimal structure** to build newer (and improved) solutions.
Consider a brand new problem as an example. Imagine a scenario where you are attending a film festival with all critically acclaimed movies. During your visit, you want to **watch to completion the largest amount of movies as possible**.


Kannusali, Espoo / Finland
23.–24.5.2023
SUNFLOWER
FILM & PHOTO FESTIVAL
A multilingual film festival
on peace, everyday life & the Middle East
sunfilm.fi
Nature
Peace
Culture

You can access the entire list of movies and the **starting** and **finishing** time for each session. The goal is to figure out the **best sequence** of movies to watch at a specific time in order to **maximize** the amount and avoid any **overlap** between two or more distinct movies.

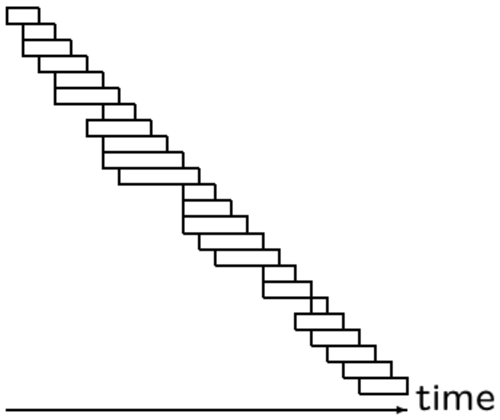


Figure 1.1: Comple list of movies available in the festival

Such a problem is known as **Job Scheduling Problem**.

A **greed approach** to find an optimal solution is as follows:

1. Sort all movies by their finishing time;

2. Book the first film in the list;

3. Go through all the other films in the list (in order) and book all the movies whose start time is at least the same as the previous movie finishing time.

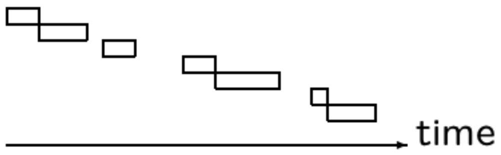


Figure 1.2: Solution for the optimal schedule of your visit to this festival

Question: Now, can we guarantee it is **optimal**?

Considering A as the set of all movie **available**. If you consider B as the subset of movies without **overlapping**, such that $B \subseteq A$.

Let a_x be the movie in A that is **different** than an activity in B. Hence, $A = a_1, a_2, \cdots, a_x, a_{x+1}$ and $B = a_1, a_2, \cdots, b_x, b_{x+1}$.

Since A was chosen by a greedy algorithm, a_x must have a finish time which is **earlier** than the finish time of b_x. Thus, $B' = a_1, a_2, \cdots, a_x, b_{x+1}$ is also a valid schedule, considering that $B' = B - \{b_x\} \cup \{a_x\}$.

The solution proposed in the JSP also reveals the **general structure** for any greedy method:

- Start by **sorting** whichever object that is represented (or partially represented) by your **decision variable**;
  - For instance, in **Dijkstra algorithm**, every "next node" is chosen based on **shortest** path to the current node;
  - In **Kruskal's algorithm**, each edge is selected in **increasing** order of their respective weights;
  - In **Prim's algorithm**, a new node is added to the "cut" based on the **smallest** weight value.
- **Select** one solution for the current problem: all possible values are ignored, a **single** optimal solution is taken
  - In **Dijkstra's**, **Prim's** and other, a **single** solution is taken;
- **Update** the problem;

  - Add a node to the optimal path (as in **Dijkstra's**), an edge (as in **Prim's** and **Kruskal's**) or a movie (as in our previous **JSP** example);

- **Repeat** the process until no more updates to the problem are possible.

CHAPTER 2 Local Search

Different from **greedy algorithm**, local search has a more broad definition and steps. However, its principle is very simple: **look in the neighbourhood**.

Definition 1 *A **neighbourhood** of a solution p is a set of solutions that are in some sense close to p. Normally, it can be easily computed from p or share a significant amount of structure with p.*

The **neighbourhood** generating function may or may not be able to generate the optimal solution. Hence, **optimality** is not guaranteed.

When the neighbourhood function can generate the global optima, starting from any initial feasible point, it is called exact.

In **continuous optimization**, it can be easily derived from **gradient methods**. What about **discrete optimization**?

For each **particular** problem, a **neighbourhood function** is required. This function should be able to map each solution into the feasible region of a problem and allow for **small changes**. Hence, it should have good **tractability**.

The most common type of neighbour function has **operators** (which generate new candidate solutions) using key structures in combinatorics: either **adding** and **removing** elements of solution or **exchanging** two or more elements.

In **combinatorics**, there is no clear way to validate that a candidate solution is globally optimal. Hence, local search provides efficient algorithms for local optimal only.

In addition, the quality of the local optimal is **highly dependent** on the neighbourhood operator.

In conclusion, local search algorithms are very much **tailored** to specific problems, and a general pseudo-code is **not viable**.

Methods relying on local search have a few **limitations**. Mainly:

- Neighbourhood **range**

  - **Only** in the feasible region? **How much** of the feasible region should be considered?

- Efficient neighbourhood **operators**
  - Those are vital to algorithm performance. Which one is the **best**? Is there a ”one size feats all”?
- **Initial** solution
  - How to determine the initial solution? **Randomly**? Via **greedy algorithms**?
- **Strategy**
  - What is the **best** way to **navigate** through the feasible region?
- **Stop** criteria
  - There is **not** a clear way to guarantee that a better solution will not be found. Hence, where and when should the method stop?
- **Performance** guarantee
  - How to **determine** if the effort put into a local search method will be fruitful? **Typically it is**.

Nevertheless, those methods have some remarkable **advantages**:

- **Novelty**
  - Due to highly **specificity** and **tailoring**, most of the local search algorithms are **novel**.
- **Efficient** in performance
  - The effort put into novel work is reflected in their **astonishing performance**, although not yet polynomial
- Research **delights**
  - Many research projects are funded based on local search methods, even being the birthplace for sub-fields such as **genetic algorithms**.