Lecture X - Approximations

[1] Department of Mathematics and Systems Analysis,
Systems Analysis Laboratory, Aalto University, Finland

March 11, 2024

Aalto University

Previously on..

PREVIOUSLY ON...

- Enumeration Process;
- Dynamic Programming.

What happens when exact solutions are not viable? Could we settle for **approximations**?

Approximations

In Mathematics and Computer Science, approximations algorithms are alternative approaches to find optimal solution or near-optimal with **guaranteed** distance.

It does differs from relaxations (where **some** of the constraints are eliminated/replaced).

Two common strategies are **Greedy Algorithms** and **Local Search**.

Greedy Algorithms

Greedy

Greed, in the lack of 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, they are some exceptions.

Greedy Properties

Algorithms such as Dijkstra's and Prim's are **optimal** and are **greedy methods**.

Question: *What are the properties that some problem have that make their greedy approach optimal?*

Those are: **Choice Property** and **Optimal Substructure**.

Greedy Properties

As **iterative** problems, at each iteration, it finds the **best** (**local optimal**) solution for that particular version of the problem;

$\longrightarrow$ It does consider previous solution, but **do not regard** future choices and remaining solution from sub-problems.

However, horizon effect is also a risk.

Greedy Properties

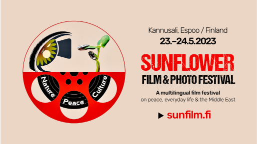
In addition, by making a single decision per iteration, it is **constantly reducing** the problem;

$\longrightarrow$ It differs from **dynamic program,** because the former exhaustively search **all** possible solution of all subproblems and also guarantees an optimal solution.

As present in dynamic programming, greedy algorithms requires **sub-optimal structure** to build newer (and improved) solutions.

Greedy Example - Job Schedulling

Imagine a scenario, where you are attending of a film festival with all critically acclaimed movie. During your visit, you want to **watch to completion the largest amount of movies as possible**.



Greedy Example - Job Schedulling

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

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

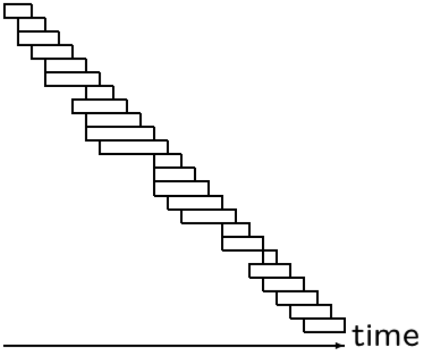
Greedy Example - Greedy Solution

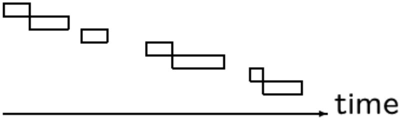
A **greed approach** to find an optimal solution is as follows:

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.

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

Greedy Example - Visuals





Combinatorial Optimization

Previously on..

Greedy Algorithms

Local Search

Greedy Example - Correctness

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 the 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 $B' = B - \{b_x\} \cup \{a_x\}$.

Greedy - General Approach

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**;
  $\longrightarrow$ For instance, in Dijkstra algorithm, every "next node" is chosen based on **shortest** path to the current node;

  $\longrightarrow$ In Kruskal's algorithm, each edge is selected in **increasing** order of their respective weights;

  $\longrightarrow$ In Prim's algorithm, a new node is added to the "cut" based on the **smallest** weight value.

Greedy - General Approach

- **Select** one solution for the current problem: all possible values are ignored, a **single** optimal solution is taken
  $\longrightarrow$ In Dijkstra's, Prim's and other, a single solution is taken;
- **Update** the problem;
  $\longrightarrow$ 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.

Local Search

Definition

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

Definition

A neighborhood 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.

Definition

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

When the neighborhood 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**?

General Structure

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 allows for **small changes**. Hence, it should have good **tractability**.

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

Optimality Conditions

In **combinatorics**, there is no clear way to validate that a candidate solution is global optimal. Hence, local search provide 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 algorithm are very much **tailored** to specific problems and a general pseudo-code is **not viable**.

Limitations

- Neighbourhood range
  $\longrightarrow$ **only** in the feasible region? **How much** of the feasible region should be considered?

- Efficient neighbourhood operators
  $\longrightarrow$ those are vital to algorithm performance. Which one is the **best**? Is there a "one size feats all" ?

- Initial solution
  $\longrightarrow$ how to determine the initial solution? **Randomly**? Via **greedy algorithms**?

Limitations

- Strategy
  $\longrightarrow$ what is the **best** way to **navigate** through the feasible region?

- Stop criteria
  $\longrightarrow$ 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
  $\longrightarrow$ how to **determine** if the effort put into a local search method will be fruitful? **Typically it is**.

Advantages

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


thank you