Aalto University

Lecture Notes - Week VIII

Exact Solutions

Fernando Dias, Philine Schiewe and Piyalee Pattanaik

February 23, 2024

CHAPTER 1
Enumeration Process

In order to find the optimal solution, the most natural process is **list** of all the solutions and **test** each one individually. Each solution can be **easily** tested, but listing is the **challenging** part.
Using an instance of the knapsack problem as an example:

	HP	Hunger Games	LotR	PJ&O	ATTWN	maximal weight
weight	426g	332g	841g	852g	113g	1000g
value	c_1	c_2	c_3	c_4	c_5	-

With 5 elements, all 2^5 combinations have to be tested. For such a small problem, such approach is **acceptable**.
The benefits of such methods relying on the guarantee of an optimal solution; especially if the runtime is not relevant. However, due to scalability, the runtime increases exponentially with the size of the instances.

Instead of enumerating and testing all possibilities, is there a way to **learn** from **small** or previous **solutions**?

Perhaps, an more **structured** and **intelligent** search? **Guided** even?

CHAPTER 2
Dynamic Programming

The first approach that satisfy the conditions listed above is **Dynamic Programming**.

Dynamic programming is a solution method by breaking them into a collection of simpler **sub-problems**, solving them once and storing their solutions. If the sub-problems are **nested recursively** inside a larger problem, dynamic programming is applicable.

A solution method involving dynamic programming requires two main conditions:

- Recursion; solution of larger problems have to be derived from solution small problems;

- Suboptimal Structure; if a small problems has **a guaranteed optimal solution**, any larger problem built upon **will** also have an optimal solution;

Remark: Recursion does not meant "it needs to be recursive".

A classical example of dynamic programming is the infamous **Fibonacci series** $Fib(n)$:

$$Fib(n) = Fib(n-1) + Fib(n-2) \tag{2.1}$$

For each value of n, the value can calculated by solving smaller problems (**sub-problems**) and it is done via recursion (**nested recursion**, in this case).

Shortest Path problem can also be solved via Dynamic Programming. Recalling the definition of such problem:

Problem 1 *In a graph $G = (V, E)$, find the shortest path between node p and q.*

The logic lies on that fact that if R is a node in the minimal path between P and Q, it is implied that the minimal path between P and R is also known. Which is **guaranteed** by the definition of the Dijkstra's algorithm.

In a **general problem**, what are the steps to apply dynamic programming:

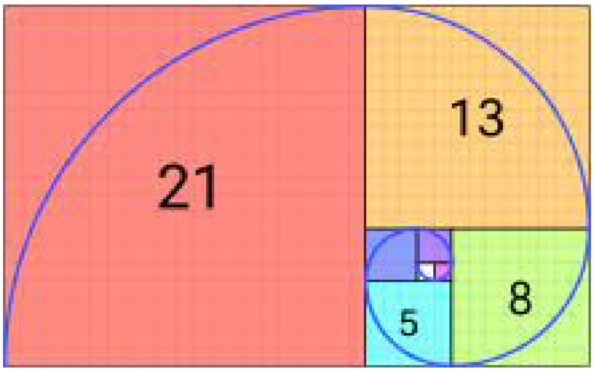

Figure 2.1: Golden spiral and Fibonacci's series

- **Recurrence** relation: **small** problems;
  - **Less** amount of items, i.e. **knapsack problem**;
  - Optimal from A to Z can be calculated by optimal from A to B **then** optimal from B to Z, i.e. **shortest path** or general flow problems;
  - In an iterative process, the solution until this iteration is optimal, and from this point forward is the same problem **but smaller**, i. e. **Fibonacci series**, decomposition problems.

  Recurrent solution $\longrightarrow$ preferably **smaller**;

- Base Cases;
  After smaller problems are found, the smallest possible problem should be trivial to solve;
  - For **Fibonacci series**, for example, the initial values are easily to established;
  - For **shortest path**, the path between the source and the first node in the optimal should be easily found;
  - Same strategies in the knapsack problem.
- **Recursive or Iterative**: depending of the problem, both strategies are valid and provide pros and cons:
  - Recursive: Better memory control; stack overflow; easier reasoning;
  - Iteration: Less memory control; no problems with stack overflow; less intuitive to implementation;

Other choices: top-down (most common) or bottom-up; memoization, etc.

CHAPTER 3

Example I: Knapsack

Back from original example:

	HP	Hunger Games	LotR	PJ&O	ATTWN	maximal weight
weight	4g	3g	8g	8g	1g	10g
value	2	4	2	2	5	-

We can build the following table, where each columns represent a different weighted value (with single integer increment) and each row corresponds to addition of a new item (normally sorted based on value.

0	0	0	0	2	2	2	2	2	2	2
0	0	0	4	4	4	4	6	6	6	6
0	0	0	4	4	4	4	6	6	6	6
0	0	0	4	4	4	4	6	6	6	6
0	5	5	5	9	9	9	9	11	11	11

Each cell is filled such as follows:

$m_{0,weight} \leftarrow 0$
for *each cell* $m_{item,weight}$ **do**
  $m_{item,weight} \leftarrow m_{item-1,weight}$ if $weight_{item} > weight$
  $m_{item,weight} \leftarrow \max(m_{item-1,weight}, m_{item-1,w-weight_{item}} + v_i)$ if $weight_{item} \leq weight$

CHAPTER 4

Example II: Tower of Hanoi

It is a puzzle consisting of **three rods** and a **set of disks with ranging diameters**. It starts with all disk in a single rod, stacked in increase diameter size.

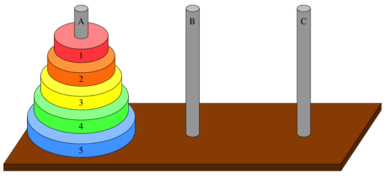

Figure 4.1: Tower of Hanoi puzzle

The goal is to **move all disk to another rod** using the following rules:

- Only one disk can be moved at the time;
- A disk has to be moved to the top of another stack or an empty rod;
- In a rod, the diameter should increase from top to bottom.

The following algorithm solves larger instance of this puzzle:

Algorithm: HANOI(BFS)

Input: Disk, Source, Destination, Auxiliary Rod

if *Disk == 1* **then**
  move Disk from Source to Destination;
else
  Hanoi(Disk-1,Source,Auxiliary Rod, Destination)
  move Disk from Source to Destination;
  Hanoi(Disk-1, Auxiliary Rod, Destination, Source)

Example:

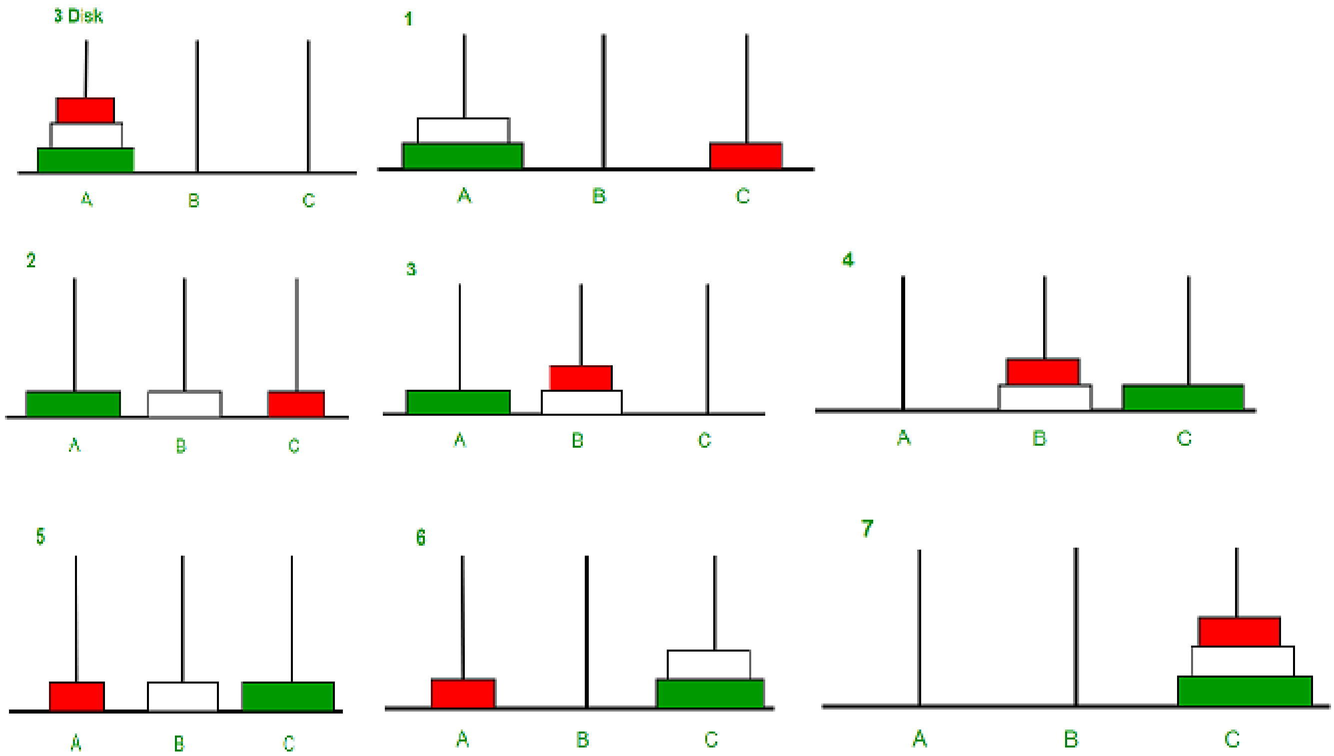

Figure 4.2: Tower of Hanoi with three disks

CHAPTER 5

Example III: Bellman-Ford Algorithm

An **alternative version** of the Dijkstra's algorithm can be derived directly from dynamic programming principles. It is slower, but allows **negative weights**.

Algorithm: BELLMAN-FORD'S ALGORITHM

Input: undirected, connected graph G, weights $c\colon E(G) \to \mathbb{R}$, nodes V, source s

d_v distance to reach node v
p_v node predecessor to node v
$Q \leftarrow \emptyset$ set of "unknown distance" nodes.
for *each node v in V* **do**
    $d_v \leftarrow \infty$
    $p_v \leftarrow FALSE$
$d_s \leftarrow 0$ **for** *each node v in V* **do**
    for *each edge* $(u, v) \in E$ **do**
        temp-dist $\leftarrow d_u + c_{uv}$
        if *temp-dist* $< d_v$ **then**
            $d_v \leftarrow$ temp-dist
            $p_v \leftarrow u$
    for *each edge* $(u, v) \in G$ **do**
        if $d_u + c_{uv} < d_v$ **then**
            return *Error: Negative Cycle Exist*
return d_v,p_v