Lecture VIII - Exact Solutions

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

February 23, 2024

Aalto University

Previously on..

- NP-Completeness;
- Polynomial transformation;
- Common problems "transforms".

PREVIOUSLY ON...

Now, when you have a problem that **is NP-Complete**, what are the **alternatives** besides ILPs?

Enumeration Process

Enumeration

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.

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**.

Challenges

- Guaranteed optimal solution;

  if runtime is not relevant, an optimal solution will be found;

- Scalability.

  the runtime increases exponentially with the size of the instances;

Alternatives

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?

Dynamic Programming

Definition

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.

Requirements

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".

Simple Example

Fibonacci series $Fib(n)$:

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

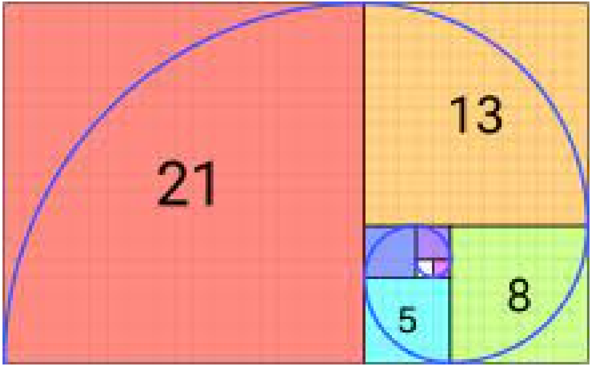

Figure: Golden spiral and Fibonacci's series

Dijkstra's Attempt

Shortest Path problem can also be solved via Dynamic Programming:

Problem

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

$\longrightarrow$ 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.

This is **guaranteed** by the definition of the Dijkstra's algorithm.

Application

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

- 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;

Applications

- 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.

Applications

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.

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	-

Example I: Knapsack

Vertical: items;
Horizontal: weights;

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 $weight_{item} \leq weight$

Example I: Knapsack

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

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.

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.

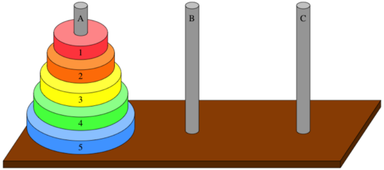

Figure: Tower of Hanoi puzzle

Example II: Tower of Hanoi

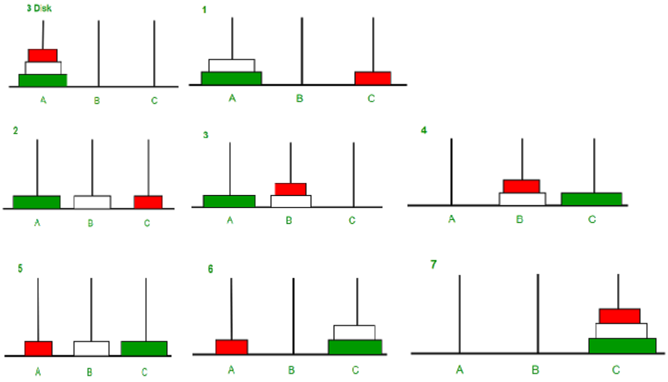

Figure: Tower of Hanoi with three disks

Example II: Tower of Hanoi

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 III: Bellman-Ford Algorithm

An alternative version of the Dijkstra's algorithm.
It is slower, but allows negative weights.

Algorithm: Bellman-Ford's Algorithm - Preparation

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 "unkown distance" nodes.
for *each node v in V* **do**
    $d_v \leftarrow \infty$
    $p_v \leftarrow FALSE$
$d_s \leftarrow 0$

Example III: Bellman-Ford Algorithm

Aalto University

Algorithm: BELLMAN-FORD'S ALGORITHM - Calculation

Output: d_v,p_v

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


thank you