Combinatorial Optimization : Basics

Houcine Senoussi

March 19, 2020

What is it about ?

Optimization :

An optimization problem can be defined as follows :

- find x^* to :
  - minimise $f(x)$
- where f is a real function of $x \in S$ for some set S.

Categories :

Optimization problems can be divided into two categories :

1. problems with **continuous** variables.
2. problems with **discrete** variables, i.e.variables belonging to a finite (e.g. the set V of a graph's vertices) or countably infinite set (typically the set N of natural numbers).

What is it about ?

- When the set S is the set of all the subsets 2^E of some finite set E, we have a **combinatorial optimization** problem.
- Examples :
  1. Traveling Salesman Problem (TSP).
  2. Minimum Spanning Tree (MST).
  3. Knapsack Problem.

Knapsack

Definition :

- Given a set O of p items, $O = \{o_1, ..., o_p\}$,
- each item o_i has a value v_i and a weight w_i.
- Given the total capacity W
- Find $(n_1, ..., n_p) \in \{0, 1\}^p$ to :
  - maximize the total value $V = \sum_{i=1}^{p} n_i * v_i$,
  - subject to constraint : $\sum_{i=1}^{p} n_i * w_i \leq W$.

Knapsack-2

- First method : brute force (try all 2^p possibilities).
- A better method : Dynamic programming.

Divide & Conquer

1. To solve a problem P :
   1. Decompose it into smaller problems $P_1,...P_k$.
   2. Use optimal solutions of smaller problems to discover those of larger ones.
2. But .... what if (larger) subproblems share (smaller) subproblems ?
   - We will repeatedly solve the common subproblems !
   - Example : Computing Fibonacci series.

Fibonacci Series

1. $F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$
2. From this definition we deduce an obvious recursive algorithm (the naive algorithm).
3. Example : to compute F_6
   - Compute F_5, F_4.
   - Compute F_4, F_3, F_3, F_2.
   - Compute F_3, F_2, F_2, F_1, F_2, F_1, F_1, F_0.
   - Compute ....

Fibonacci Series-A better algorithm

1. $tab[0] = 1$
2. $tab[1] = 1$
3. For $i = 2, ..., n$
   - $tab[n] = tab[n-1] + tab[n-2]$

- Every value is calculated only once but we need more space (linear space complexity).

Dynamic Programming-Principle

1. To solve a problem P :
   1. Decompose it into smaller problems $P_1,...P_k$.
   2. Use optimal solutions of smaller problems to discover those of larger ones.
   3. Compute subproblems' solutions only **once** and **store** them in a table so that they can be **reused**.

Dynamic Programming-Principle

1. Bellman's Optimality principle "From any point on an optimal trajectory, the remaining trajectory is optimal for the corresponding problem initiated at that point".

Dynamic Programming-Knapsack problem

- We define a matrix $V[0..p, 0..W]$ as follows :
  - For $k = 0, ..., p$, $w = 0, ..., W$, $V[k, w]$ stores the maximal value if we consider the k first objects and w as a capacity. In other words it represents the solution of the subproblem corresponding to the k first objects and the capacity w.
  - it follows that $V(p, W)$ will contain the "solution" of the whole problem.

Dynamic Programming-Knapsack problem

- The solution can be defined recursively as follows:
  1. For $k = 1, ..., p,\ w = 0, ..., W,$
     $V(k, w) = max(V(k-1, w), v_k + V(k-1, w-w_k))$
     1. O_k belongs to the solution of the problem $(k, w) \rightarrow V(k, w) = v_k + V(k-1, w-w_k)$
     2. O_k doen't belong to this solution of the problem $() \rightarrow V(k, w) = V(k-1, w)$
  2. For $w = 0, ..., W,\ V(0, w) = 0.$
  3. For $k = 1, ..., p,\ w < 0,\ V(k, w) = -\infty.$

Dynamic Programming-Knapsack problem

- Input : p, W, $w[1..p]$, $v[1..p]$

1. For $w = 0, ..., W$, $V[0, w] = 0$.
2. For $k = 1, ..., p$
    1. For $w = 0, ..., W$
        1. If (w[k] <= w)
           $V[k, w] = max(V[k-1, w], v[k] + V[k-1, w - w[k]])$
        2. Else
           $V[k, w] = V[k-1, w]$

- Output : $V[p, W]$

Dynamic Programming-Knapsack problem

- In this version of the algorithm an important part of the solution is missing :
  - Which set of items gives the optimal solution ?
- We add a boolean table $Belongs[1..p, 1..W]$ defined by :
  - if O_k the solution of the problem (k, w) $Belongs[k, w]$=1
  - else $Belongs[k, w]$=0.
- We use this table as follows :
  1. If $Belongs[n, W]$=1 add O_n to the solution and continue with $Belongs[n-1, W-w[n]]$
  2. If $Belongs[n, W]$=0 continue with $Belongs[n-1, W]$

Dynamic Programming-Knapsack problem

- Input : p, W, $w[1..p]$, $v[1..p]$
- Part I

1. For $w = 0, ..., W$, $V[0, w] = 0$.
2. For $k = 1, ..., p$
    1. For $w = 0, ..., W$
        1. If ((w[k] <= w) And (V[k-1, w]<v[k]+ V[k-1, w-w[k]]))
           $V[k, w] = v[k] + V[k-1, w-w[k]]$
           $Belongs[k, w]$=1
        2. Else
           $V[k, w] = V[k-1, w]$
           $Belongs[k, w]$=0

Dynamic Programming-Knapsack problem

- Part *II*

1. $T = \emptyset$
2. $w = W$
3. For $k = p, ..., 1$
    - If $(Belongs(k, w) == 1)$
      $T = T \bigcup \{k\}$
      $w = w - w[k]$

- Output : $V[p, W], \ T$

Dynamic Programming-Knapsack problem

- An example :
  - W=10
  - $w = [5, 4, 6, 3]$
  - $v = [10, 40, 30, 50]$

Conclusion

We introduced Combinatorial optimization through three well known problems. In the next chapter we will study the third problem (Knapsack problem) and its dynamic programming resolution.

References

- Ralph Otten, "Combinatorial algorithms", `http://www.es.ele.tue.nl/education/5MC10/`.