# 7- Exploration Problems

Bruno MARTIN,
University of Nice - Sophia Antipolis
`mailto:Bruno.Martin@unice.fr`
`http://deptinfo.unice.fr/~bmartin/mathmods.html`

---

## Methods for solving exploration problems

- **Greedy Algorithm:** a *heuristic strategy*: try to make, at each step, the optimal choice compatible with the previous and hope that this sequence of choices leads to the optimal solution. *Generally linear algorithms*
- **Dynamic Programming:** divide the “problem entries” into as many subsets as needed. The problem is solved on every subset using the previous solutions to compute the result of the current subset. Finding a way to split the set is not always possible. *Typically polynomial algorithms*
- **Brute-Force Search:** When the previous methods don’t work. Consider every subset of elements and find the optimal one. *These algorithms are clearly exponential*

---

## Exploration Problems

We search algorithms for solving the following class of problems:

- We are given a set $E$ with $N$ elements
- Each element $e$ has a cost: $value(e)$
- We have a predicate $C$ on the subsets of $E$
- The problem is to find a set $F \subseteq E$ for which:
  - the predicate $C(F)$ evaluates to *true*
  - $\sum_{e \in F} value(e)$ is maximal (or minimal in some cases)

---

## Greedy algorithm pattern

- Find a clever way to order the elements ⇒ ordered set
- start from the empty set ($F = \varnothing$) and iterate on the ordered set
- add the elements one by one, adding the current element if it is compatible with the previous ones
- at the end of the iteration, you might have an optimal solution

Finding the best ordering is not always possible :
Greedy algorithms don’t always lead to an optimal solution

## Problems tractable by greedy algorithms

Only some problems are known to be solvable by greedy algorithms:

- the Huffman's codes for data encoding (data compression)
- the (dummy) unique resource allocation
- the Dantzig's algorithm for the graph shortest path problem
- the Kruskal's algorithm for graph's minimum spanning tree
- the (dummy) task-scheduling problem

---

## Kruskal's method for minimum spanning tree

$G = (V, E)$ is a weighted graph of order $n$; find a spanning tree with the minimum weight

- consider each vertex of $V$ as a tree
- sort the edges $v_i \to v_j$ by increasing weight
- For each edge $e = v \to u$
  - if $v$ and $u$ belong to the same tree, you can't add the edge without creating a cycle and the edge is discarded
  - else you add the edge and do the union of the two trees
  - In fact, at a step, you add the shortest edge $e = v_i \to v_j$ that doesn't add a cycle
- stop when no more edge can be added

The running time of kruskal's algorithm is $O(\sharp E \log \sharp E)$

---

## Unique resource allocation : the (dummy) car renter

A renter wishes to rent **1** car to the **max. number** of customers. There's a set of **rental requests**, the **constraint** is that the **chosen requests don't overlap**

- the requests are sorted in increasing order of ending dates
- iterate on the requests in order. If the current request does not overlap the last chosen request, select it

This algorithm gives an **optimal solution**
**Notice** : if you want to **maximize** the **renting duration** this algorithm **won't give the optimal solution** even if you sort the requests by increasing duration : a 3-day renting is incompatible with two 2-day rentings)

---

## Dynamic Programming

There's only a polynomial number of subproblems and thus some subproblems might have to be solved many times (remember Fibonacci)

The solution is to store intermediate solutions in a table

- divide the set into as many subsets as required
- solve the problem on every subset using the previous solutions to compute the result of the current subset

*These are polynomial-time algorithms*

# Dynamic Programming

Finding the way to split the set is not always possible.
There is often no way to divide a problem into a small number of subproblems whose solution can be combined to solve the original.
In such a case you may divide the problem (and the subproblems) into as many subproblems as necessary.
The latter clearly has an exponential time complexity.

---

# Example of Dynamic Programming

DP Examples:

- Floyd's algorithm for solving the all-pairs shortest paths problem in a graph
- Warshall's algorithm for transitive closure
- matrix chain product
- ...

---

# DP is not a divide-and-conquer strategy

In a divide-and-conquer problem:

- you solve a large problem by spliting it into **independent** smaller subproblems
- Solving them **independently** solves the global problem
- Example: quicksort algorithm

---

# The Floyd's Example of Dynamic Programming

The most famous example of dynamic programming is **Floyd's algorithm** which finds all the shortest paths in a valuated graph.

It stores the shortest path between each pair of vertices in a matrix
It works by considering all vertices one by one. For each vertex $k$, it considers every pair of vertices $i \to j$. When there exists a shortest path from $i$ to $j$ going through $k$ it stores the new cost in the $Matrix[i, j]$
When we end the three loops $O(n)$, the Matrix contains every shortest path
Already seen in a previous lecture

# The Matrix Chain Product

You multiply these three matrices $A[4,3] \times B[3,5] \times C[5,1]$:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \end{pmatrix} \times \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \\ c_{41} \\ c_{51} \end{pmatrix}$$

There are two possible parenthesizations
$(A[4,3] \times B[3,5]) \times C[5,1]$ and $A[4,3] \times (B[3,5] \times C[5,1])$
The numbers of scalar multiplications are :
$4 \times 3 \times 5 + 4 \times 5 \times 1 = 80$ for the first parenthesization and
$4 \times 3 \times 1 + 3 \times 5 \times 1 = 27$ for the second
**Problem**: When multiplying $A_1 A_2 \ldots A_n$, find the parenthesization that minimizes the total number of scalar multiplications required

---

# The DP Solution to the Matrix Chain Product

Notice that to compute $M[i,j]$ we need to have previously computed $M[i,k]$ and $M[k+1,j]$
We take a matrix $M_{[n \times n]}$ to store the intermediate computations
We record the number of multiplications needed to compute $A_1$ by $A_2$, $A_2$ by $A_3$, ... in $M[1,2], M[2,3], \ldots$
To find the best way to compute $A_1 A_2 A_3$:

- For $(A_1 A_2) A_3$ the result is $M[1,2] + r_0 r_2 r_3$
- For the other $A_1 (A_2 A_3)$ the result is $r_0 r_1 r_3 + M[2,3]$
- We compare them and store the smallest in $M[1,3]$

---

# The Matrix Chain Product

Let $M[i,j]$ be the minimum number of scalar multiplications required to compute $A_i A_{i+1} \ldots A_j$
When $i = j$ the cost is clearly 0
When $i < j$, the optimal parenthesization splits the product in $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ *for* $i \leq k < j$ ($A_i$ is of size $[r_{i-1} \times r_i]$)

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ min(M[i,k] + M[k+1,j] + r_{i-1} r_k r_j) & \text{if } i \leq k < j \end{cases}$$

The recursive algorithm of the above recurrence is exponential

---

```
m[2, 5] = min{
       m[2, 3] + m[4, 5] + r1 r3 r5
       m[2, 4] + m[5]    + r1 r4 r5
       m[2]    + m[3, 5] + r1 r2 r5
              }
```

## The DP Solution to the Matrix Chain Product

We continue for all triples, then for successive group of four, ..., by continuing that way we obtain at the end the best way to multiply the matrices
The time-complexity for computing the optimal parenthesization of the matrices is in $O(n^3)$
The space-complexity is in $O(n^2)$ (the auxiliary matrix $M_{[n,n]}$)

---

## Exhaustive Search

When greedy algorithms and dynamic programming don't work we need an exhaustive search of the subsets of $E$
The algorithm becomes exponential
You must try to cut the choices during the computation
Examples of exponential algorithms are :

- The Queens problem
- The Travelling Salesman problem
- ...

---

## Problems with Dynamic Programming

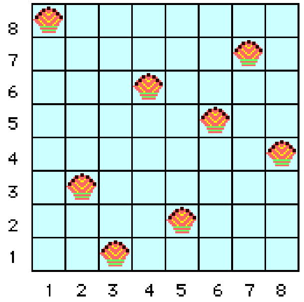
- It may be impossible to combine the solutions of smaller subproblems to form the solution of the larger one.
- The number of small subproblems to solve may be unacceptably large
- None has precisely characterized which problems can be solved with dynamic programming
- There are many hard problems for which dynamic programming does not seem to be applicable (TSP)
- There are many "easy problems" for which it is less efficient than a standard algorithm (for example the ones for which the greedy algorithm applies)

---

## The 8 queens problem

Place 8 queens on a chess board so that none of them catches any other one.
Exponential algorithm explores the $\binom{64}{8}$ solutions.
Some choices can be "cut" by testing the conflicts every time you try to place a queen. Eg, when adding new queen, you can test for each previously placed queen whether there's a conflict:

- on the line, complexity is $8^8$
- on the line and the column, complexity is 8!
- on the line, the column and on the 2 diagonals



# The Travelling Salesman Problem

Most famous problem of exhaustive search: *Given n cities, find the shortest route connecting them all with no city visited twice.*

Arises naturally in a number of important applications and has been extensively studied. Still unthinkable to solve huge instances. Difficult problem because it seems there's no way to avoid checking the length of a very large number of possible routes. [`http://www.tsp.gatech.edu`]

A TSP on 33 cities for a contest in 1962.

---

# Three Variants of the Knapsack Problem

**Greedy algorithm:** When items are fractionnable (gold's powder, flour, ...). You can then compute and sort by decreasing order all *value*/*weight*. You fill your knapsack by taking the largest quantity of the greatest value per kilo, then the second more expensive, ...
**Dynamic-Programming:** When the items are not fractionnable and when the capacity is an integer: you consider an array $C[n]$ where $C[i]$ stores the highest value that can be achieved with a knapsack of capacity $i$. You combine the already computed values to achieved the next one
**Exhaustive-search:** When the items are not fractionnable and when the capacity is a real number, no efficient algorithm is known. You need to explore all the possibilities. When inserting a new item, you can just cut the choices by verifying that you do not exceed the capacity of the knapsack

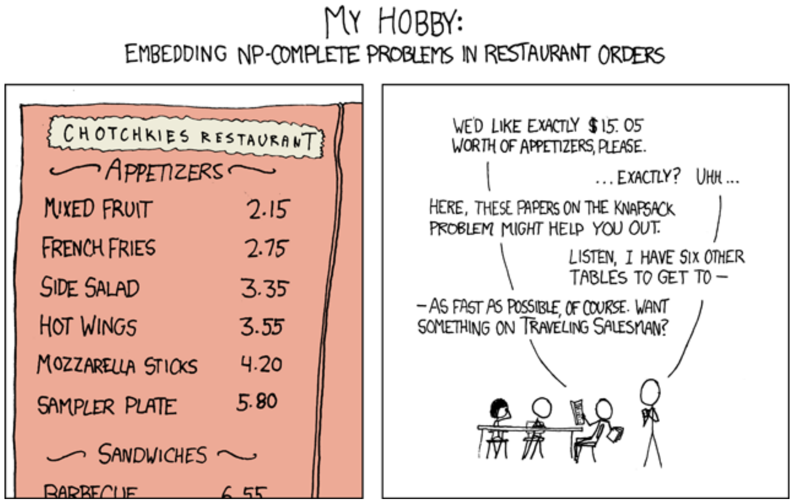
---

# The Knapsack Problem

A robber is in a room filled with $N$ types of items of varying weight and value. He has a knapsack of capacity $M$ to carry the goods.
**The knapsack Problem** : Find the combination of items that the robber should choose for his knapsack in order to maximize the total value of all the items he takes.
Depending on the kind of items and the capacity's value $M$, this problem can be solved with the three types of algorithms previously introduced

---

# A particular knapsack problem



# The dynamic-programming of the Knapsack Problem

A robber is in a room filled with $n$ items of varying integer weights and values over positive reals. He has a knapsack of capacity $W$ (a weight limit) to carry the goods

Find a subset $S$ st. the constraint $\sum_{k\in S} w_k \leq W$ is observed in order to maximize the total value $max = \sum_{k\in S} v_k$. We embedd the problem in an $n \times W$ array of problems and solve those problems successively.
For $0 \leq i \leq n$ and $0 \leq j \leq W$, $m[i,j]$ denotes the max value of the knapsack problem restricted to $S \subseteq \{1,\ldots,i\}$ under weight limit $j$.
The heart of the solution is the recurrence

$$m[i,j] = \max\{m[i-1,j], v_i + m[i-1, j-w_i]\}$$

if in the optimal solution $i \notin S$ then $m[i,j] = m[i-1,j]$; otherwise we gain value $v_i$ and have to maximize from the remaining objects under the remaining weight limit $j - w_i$ (assuming $j \geq w_i$ ). The optimum will be the greatest of these two values.

---

# The Subset-Sum Problem

In a set of integer-valued items, you search a subset where the sum of the item's values is a given integer $b$.
You have a finite set $A$ of items with integer values, and an integer $b$, is there a subset $A'$ such that $\Sigma_{a\in A'} value(a) = b$

---

# The DP algorithm for the knapsack problem

We have $m[0,k] = m[k,0], \forall k \geq 0$

```
for i in 0..n
  m[i,0]=0
end
for j in 1..W
  m[0,j]=0
end
for i in 1..n
  for j in 1..W # expresses the value of the next m[i,j]
    if j<w[i]
       m[i,j]=m[i-1,j] # item i cannot be selected
    else m[i,j]=max { m[i-1,j], v[i]+m[i-1,j-w[i]] }
    end
   end
end
```

---

# The Partition Problem

You partition a set of integer-weighted items into two subsets of equal weight
You have a finite set $A$ of items with integer values, is there a subset $A'$ such that $\Sigma_{a\in A'} value(a) = \Sigma_{a\in A\setminus A'} value(a)$

# Polynomial Problem versus "Exponential" Problem

When $N$ becomes very big :

- Polynomial Problems remain tractable
- While Non deterministic Polytime Problems become practically intractable

The difference between those two classes of problem have been formalized and is the object of the study of the NP-completeness