Advanced Algorithms

Dr. Malek Mouhoub

Mathematics and Computer Science Department

University of Lethbridge

Spring 2001

4. Advanced Design and Analysis Techniques

1. Greedy algorithms
2. Divide and conquer
3. Dynamic Programming
4. Randomized Algorithms
5. Backtracking Algorithms

Optimization Problem

- In an optimization problem we are given a set of **constraints** and an **optimization function**.
- Solutions that satisfy the constraints are called **feasible solutions**.
- A feasible solution for which the optimization function has the best possible value is called an **optimal solution**.

Loading Problem

A large ship is to be loaded with cargo. The cargo is containerized, and all containers are the same size. Different containers may have different weights. Let w_i be the weight of the ith container, $1 \leq i \leq n$. The cargo capacity of the ship is c. We wish to load the ship with the maximum number of containers.

- Formulation of the problem :
  - Variables : x_i ($1 \leq i \leq n$) is set to 0 if the container i is not to be loaded and 1 in the other case.
  - Constraints : $\sum_{i=1}^{n} w_i x_i \leq c$.
  - Optimization function : $\sum_{i=1}^{n} x_i$
- Every set of x_is that satisfies the constraints is a feasible solution.
- Every feasible solution that maximizes $\sum_{i=1}^{n} x_i$ is an optimal solution.

4.1 Greedy Algorithms

- Greedy algorithms seek to optimize a function by making choices (greedy criterion) which are the best locally but do not look at the global problem.
- The result is a good solution but not necessarily the best one.
- The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

Machine Scheduling

- We are given n tasks and an infinite supply of machines on which these tasks can be performed.
- Each task has a start time s_i and a finish time f_i, $s_i \leq f_i$, $[s_i, f_i]$ is the processing interval for task i.
- A **feasible** task-to-machine assignment is an assignment in which no machine is assigned two overlapping tasks.
- An **optimal assignment** is a feasible assignment that utilizes the fewest number of machines.

Machine Scheduling

- We have 7 tasks :

task	a	b	c	d	e	f	g
start	0	3	4	9	7	1	6
finish	2	7	7	11	10	5	8

- Feasible assignment that utilizes 7 machines :

$$a \rightarrow M_1, b \rightarrow M_2, \ldots, g \rightarrow M_7.$$

- Is it the optimal assignment ?

Machine Scheduling

- A greedy way to obtain an optimal task assignment is to assign the tasks in stages, one task per stage and in nondecreasing order of task start times.
- A machine is called **old** if at least one task has been assigned to it.
- If a machine is not old, it is **new**.
- For machine selection, we use the following criterion :
  - *If an old machine becomes available by the start time of the task to be assigned, assign the task to this machine; if not, assign it to the new machine.*

Scheduling Problem

- We are given N non preemptive jobs $j_1, \ldots, j_N$, all with known running times $t_1, t_2, \ldots, t_N$, respectively. We have a single processor.
- **Problem**: what is the best way to schedule these jobs in order to minimize the average completion time ?
- Example :

Job	j_1	j_2	j_3	j_4
Time	15	8	3	10

- Solution 1 : j_1, j_2, j_3, j_4. Average completion time : 25.
- Optimal Solution : j_3, j_2, j_4, j_1. Average completion time : 17.75.

Scheduling Problem

- Formulation of the problem :
  - N jobs $j_{i_1}, \ldots, j_{i_N}$ with running times $t_{i_1}, \ldots, t_{i_N}$ respectively.
  - Total cost : $C = \sum_{k=1}^{N}(N-k+1)t_{i_k}$
    $= (N+1)\sum_{k=1}^{N} t_{i_k} - \sum_{k=1}^{N} k t_{i_k}$
- The first sum is independent of the job ordering. Only the second sum affects the total cost.
- Greedy criterion : Arrange jobs by smallest running time first.

Scheduling Problem

Multiprocessor case

- N non preemptive jobs $j_1, \ldots, j_N$, with running times $t_1, t_2, \ldots, t_N$ respectively, and a number P of processors.
- We assume that the jobs are ordered, shortest running time first.
- Example :

Job	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
Time	3	5	6	10	11	14	15	18	20

- Greedy criterion : start jobs in order cycling through processors.
- The solution obtained is optimal however we can have many optimal orderings.

Loading Problem

- Using the greedy algorithm the ship may be loaded in stages; one container per stage.
- At each stage, the greedy criterion used to decide which container to load is the following :
  - *From the remaining containers, select the one with least weight.*
- This order of selection will keep the total weight of the selected containers minimum and hence leave maximum capacity for loading more containers.

Loading Problem

- Suppose that :
  - $n = 8$,
  - $[w_1, \ldots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$,
  - and $c = 400$.
- When the greedy algorithm is used, the containers are considered for loading in the order 7,3,6,8,4,1,5,2.
- Containers 7,3,6,8,4 and 1 together weight 390 units and are loaded.
- The available capacity is now 10 units, which is inadequate for any of the remaining containers.
- In the greedy solution we have :

  $[x_1, \ldots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1]$ and $\sum x_i = 6$.

0/1 Knapsack Problem

- In the 0/1 knapsack problem, we wish to pack a knapsack (bag or sack) with a capacity of c.
- From a list of n items, we must select the items that are to be packed into the knapsack.
- Each object i has a weight w_i and a profit p_i.
- In a **feasible** knapsack packing, the sum of the weights of the packed objects does not exceed the knapsack capacity :

$$\sum_{i=1}^{n} w_i x_i \leq c \text{ and } x_i \in [0, 1], 1 \leq i \leq n$$

- An **optimal** packing is a feasible one with maximum profit :

$$\text{maximize } \sum_{i=1}^{n} p_i x_i$$

- The 0/1 knapsack problem is a generalization of the loading problem to the case where the profit earned from each container is different.

0/1 Knapsack Problem

- Several greedy strategies for the 0/1 knapsack problem are possible.
- In each of these strategies, the knapsack is packed in several stages. In each stage one object is selected for inclusion into the knapsack using a greedy criterion.
- First possible criterion : from the remaining objects, select the object with the maximum profit that fits into the knapsack.
- This strategy does not guarantee an optimal solution.
- Example : $n = 3, w = [100, 10, 10], p = [20, 15, 15], c = 105$.
- Solution using the above criterion : $x = [0, 1, 1]$, profit = 20.
- Optimal solution : $x = [0, 1, 1]$, profit = 30.

0/1 Knapsack Problem

- Second criterion : greedy on weight
  - *From the remaining objects, select the one that has minimum weight and also fits into the knapsack.*
- This criterion does not yield in general to an optimal solution.
- Example : $n = 2, w = [10, 20], p = [5, 100], c = 25$.
- Solution : $x = [1, 0]$ inferior to the solution : $x = [0, 1]$.

0/1 Knapsack Problem

- Third criterion : greedy on the profit density p_i/w_i. *From the remaining objects, select the one with maximum p_i/w_i that fits into the knapsack.*
- This strategy does not guarantee optimal solutions either.
- Example : $n = 3, w = [20, 15, 15], p = [40, 25, 25]$ and $c = 30$.

Note : The 0/1 knapsack problem is an NP-hard problem. This is the reason why we cannot find a polynomial-time algorithm to solve it.

4.2 Divide and Conquer

To solve a large instance :

1. Divide it into two or more smaller instances.
2. Solve each of these smaller problems, and
3. combine the solutions of these smaller problems to obtain the solution to the original instance.

The smaller instances are often instances of the original problem and may be solved using the divide-and-conquer strategy recursively.

Closest-Points Problem

- Find the closest pair of N points in the plane.
- A brute force algorithm requires $O(N^2)$ in the worst case since there are at most $N(N-1)/2$ pairs.
- A divide and conquer algorithm for this problem works as follows :
  - Sort the points by x coordinates and draw a vertical line that partitions the point set into two halves, P_L and P_R.
  - Either the closest points are both in P_L, or they are both in P_R or one is in P_L and the other is in P_R (this is similar to solving the maximum subsequence sum using a divide and conquer method).
  - Our goal is to have a complexity better than $O(N^2)$.
  - According to the master theorem, computing the 3rd case should be performed in $O(N)$ in order to have $O(N \log N)$ in total.

Closest-Points Problem

- The brute force algorithm to perform the 3rd case requires $O(N^2)$. Why ?
- To do better, one way consists in the following :
  1. Compute $\delta = min(d_L, d_R)$.
  2. We need then to compute d_C only if d_c improves on δ. The two points that define d_C must be within δ of the dividing line. This area is general refereed as a **strip**.
  3. The above observation limits in general the number of points that need to be considered. For large points sets that are uniformly distributed, the number of points that are expected to be in the strip is of order $O(\sqrt{(N)})$ in average. The brute force calculation on these points can be performed in $O(N)$.
  4. However, in the worst case all the points could be in the strip which requires $O(N^2)$ to expect all the points.
  5. Can we do better ?

Closest-Points Problem

- Yes, using the following observation :
  - The y coordinates of the two points that define d_C can differ by at most δ. Otherwise, $d_C > \delta$.
  - If p_i and p_j's coordinates differ by more than δ, then we can proceed to p_{i+1}.
  - In the worst case, for any points p_i, at most 7 points p_j are considered. The time to compute d_C that is better than δ is $O(N)$.
  - It appears that we have now an $O(N \log N)$ solution to the closest-points problem. However it is not the case :
    * An extra work is needed to sort, for each recursive call, the list of points by y coordinates which gives an $O(N \log^2 N)$ algorithm. Well, it is better than $O(N^2)$.
    * Can we reduce the work in each recursive call to $O(N)$, thus ensuring an $O(N \log N)$ algorithm ?

Closest-Points Problem

- Yes. We can maintain two lists. One is the point list sorted by x coordinate, and the other is the point list sorted by y coordinate. Let's call these lists P and Q, respectively.
- First we split P in the middle. Once the dividing line is known, we step through Q sequentially, placing each element in Q_L or Q_R as appropriate.
- When the recursive calls return, we scan through the Q list and discard all the points whose x coordinates are not within the strip. The remaining points are guaranteed to be sorted by their y coordinates.
- This strategy ensures that the entire algorithm is $O(N \log N)$, because only $O(N)$ extra work is performed.

4.3 Dynamic Programming

- Using a table instead of a recursion.
- Remember the problem given to Mr Dupont :

$$\begin{aligned} F(1) &= 3 \\ F(2) &= 10 \\ F(n) &= 2F(n-1) - F(n-2) \end{aligned}$$

What is $F(100)$?

1st Method

Translate the mathematical formula to a recursive algorithm.

if (n==1) $\{$ return 3; $\}$
else
  if (n==2) $\{$return 10;$\}$
  else $\{$ return 2 $*$ f(n-1) - f(n-2); $\}$

Analysis of the 1st method

- $T(N) = T(N-1) + T(N-2)$
- This is a Fibonacci function : $T(N) \approx 1.62^N$
- Complexity of the algorithm : $O(1.62^N)$ (exponential).

2nd Method

- Use an array to store the intermediate results.

```
int f[n+1];
f[1]=3;
f[2]=10;
for(i=3;i¡=n;i++)
   f[i] = 2 ∗f[i-1] - f[i-2];
return t[n];
```

- What is the complexity in this case ?
- What is the disadvantage of this method ?

3rd Method

- There is no reason to keep all the values, only the last 2.

```
if (n==1) return 3;
last = 3;
current = 10;
for (i=3;i<=n;i++) {
    temp = current;
    current = 2*current - last;
    last = temp; }
return current;
```

- What is the advantage of this method comparing to the 2nd one ?

Fibonacci Numbers

- We can do use the same method to compute Fibonacci numbers.
- To compute F_N, all that is needed is F_{N-1} and F_{N-2}, we only need to record the two most recently computed Fibonacci numbers. This requires an algorithm in $O(N)$.

Dynamic Programming for Optimization Problems

- In dynamic programming, as in greedy method, we view the solution to a problem as the result of a sequence of decisions.
- In the greedy method, we make irrevocable decisions one at a time using a greedy criterion. However in dynamic programming, we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequences.

0/1 Knapsack Problem

- In the case of the 0/1 Knapsack problem, we need to make decisions on the values of $x_1, \ldots, x_n$.
- When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called **dynamic-programming recurrence equations** that enable us to solve the problem in an efficient way.
- Let $f(i, y)$ denote the value of an optimal solution to the knapsack instance with remaining capacity y and remaining objects $i, i+1, \ldots, n$:

$$f(n, y) = \begin{cases} p_n & \text{if } y \geq w_n \\ 0 & 0 \leq y < w_n \end{cases}$$

$$f(i, y) = \begin{cases} max\{f(i+1, y), f(i+1, y - w_i) + p_i\} & \text{if } y \geq w_i \\ f(i+1, y) & 0 \leq y < w_i \end{cases}$$

Ordering Matrix Multiplications

Goal : *Determine the best order in which to carry out matrix multiplications*

- To multiply a $p \times q$ matrix and a $q \times r$ matrix we do pqr element-wise multiplications.
- Matrix multiplication is associative : $A(BC) = (AB)C$ however the order to perform the intermediate multiplications can make a big difference in the amount of work done.

Ordering Matrix Multiplications

Example

We want to multiply the following arrays :

$$A_1\,(30\times1) \quad \times \quad A_2(1\times40) \quad \times \quad A_3(40\times10) \quad \times \quad A_4(10\times25)$$

How many multiplications are done for each ordering ?

1. $((A_1A_2)A_3)A_4 \quad 30.1.40+30.40.10+30.10.25 = 20,700$
2. $A_1(A_2(A_3A_4)) \quad 40.10.25 + 1.40.25 + 30.1.25 = 11,750$
3. $(A_1A_2)(A_3A_4) \quad 30.1.40+40.10.25+30.40.25 = 41,200$
4. $A_1((A_2A_3)A_4) \quad 1.40.10 + 1.10.25 + 30.1.25 = 1,400$

Ordering Matrix Multiplications

General Problem :

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ where the dimensions of A_i are $(d_{i-1} \times d_i)$ (for $1 \leq i \leq n$). What is the optimal way (ordering) to compute :

$$A_1(d_0 \times d_1) \times A_2(d_1 \times d_2) \times \cdots \times A_n(d_{n-1} \times d_n)$$

Ordering Matrix Multiplications

- The number of possible orderings is :

$$T(N) = \sum_{i=1}^{N-1} T(i)T(N-i)$$

  - Suppose that the last multiplication is :
    $(A_1 A_2 \ldots A_i)(A_{i+1} A_{i+2} \ldots A_N)$
    * there are $T(i)$ ways to compute $(A_1 A_2 \ldots A_i)$
    * and $T(N-i)$ ways to compute $(A_{i+1} A_{i+2} \ldots A_N)$.

- The solution of this recurrence is called *Catalan numbers* and it **grows exponentially**.

Ordering Matrix Multiplications

- $m_{Left,Right}$ is the number of multiplications required to multiply $A_{Left}A_{Left+1}\cdots A_{Right}$
- $(A_{Left}\cdots A_i)(A_{i+1}\cdots A_{Right})$ is the last multiplication.
- The number of multiplications required is :

$$m_{Left,i} + m_{i+1,Right} + c_{Left-1}c_i c_{Right}$$

- The number of multiplications in an optimal ordering is :

$M_{Left,Right} =$

$$min_{Left \leq i < Right}\{M_{Left,i} + M_{i+1,Right} + c_{Left-1}c_i c_{Right}\}$$

4.4 Randomized Algorithms

- A randomized algorithm is an algorithm where a random number is used to make a decision at least once during the execution of the algorithm.
- The running time of the algorithm depends not only on the particular input, but also on the random numbers that occur.

Primality Testing

- **Problem :** Test whether or not a large number is prime ?
- The obvious method requires an algorithm of complexity $O(\sqrt{N})$. However nobody knows how to test whether a d-digit number N is prime in time polynomial in d.
- Fermat's Lesser Theorem :

  If P is prime, and $0 < A < P$, then $A^{P-1} \equiv 1(\text{mod } P)$

Primality Testing

- Using Fermat's Lesser Theorem, we can write an algorithm for primality testing as follows :
  1. Pick $1 < A < N - 1$ at random.
  2. If $A^{N-1} \equiv 1 (\text{mod } P)$ declare that N is "probably" prime, Otherwise declare that N is definitely not prime.
- Example : if $N = 341$, and $A = 3$, we find that $3^{340} \equiv 56 (\text{mod } 341)$.
- If the algorithm happens to choose $A = 3$, it will get the correct answer for $N = 341$.
- There are some numbers that fool even this algorithm for most choices of A.
- One such set of numbers is known as the *Carmichael numbers*. They are not prime but satisfy $A^{N-1} \equiv 1 (\text{mod } N)$ for all $0 < A < N$ that are relatively prime to N. the smallest such number is 561.

Primality Testing

- We need an additional test to improve the chances of not making an error.
- **Theorem :** If P is prime and $0 < X < P$, the only solutions to $X^2 \equiv 1(\text{mod } P)$ are $X = 1,\ P - 1$.
- Therefore, if at any point in the computation of $A^{N-1}(\text{mod } N)$ we discover violation of this theorem, we can conclude that N is definitely not prime.

4.5 Backtracking

- Backtracking is a systematic way to search for the solution to a problem.
- This method attempts to extend a partial solution toward a complete solution.
- In backtracking, we begin by defining a **solution space** or **search space** for the problem.
- For the case of the 0/1 knapsack problem with n objects, a reasonable choice for the solution space is the set of 2^n 0/1 vectors of size n. This set represents all possible ways to assign the values 0 and 1 to x.
- The next step is to organize the solution space so that it can be reached easily. The typical organization is either a graph or a tree.
- Once we have defined an organization for the solution space, this space is searched in a depth-first manner beginning at a start node.
- The start node is both a **live** node and the **E-node** (expansion node).
- From this E-node, we try to move to a new node.
- If such a move is possible, the new node becomes a live node and also becomes the new E-node. The old E-node remains a live node.
- If we cannot move to a new node, the current E-node dies and we move back to the most recently seen live node that remains. This live node becomes the new E-node.
- The search terminates when we have found the answer or when we run out of live nodes to back up to.