M2SI

Master of Information Systems and Intelligent Systems

INSEA

19 NouvembER 2019

RESOURCES ALLOCATION PROBLEM

Supervised by:

Mr BENMANSOUR Rachid

Realised by:

- CHARHBILI Moad
- BOUGADRE Achraf
- NAJAH Ismail
- KABAROUSSE AZZEDDINE

Acknowledgement

Our sincere thanks to Pr.Benmansour, for the support he has given us, throughout the realization of this project, and also for sharing with us his passion for teaching. we greatly appreciated his support, his involvement and his experience throughout the quarter

Contents

List of Figures

The problem:

In this project, we will resolve the problem of resources allocation using the dynamic programming.
We have M units (resources) to share among N activities.
Whenever we assign a resource to an activity i $(1 \leq i \leq N)$ we receive a gain equal to $r(i, j) \geq 0$. Our goal is to maximize the total sum of the gains received from N activities" f=$\sum_{i=1}^{N} r(i, j)$.
Our work is divided into four parts:

- **The first part :** build a dynamic program that allows us to solve this problem: we did that in two different methods:
  1. Recursive method
  2. Iterative method
- **The second part :** implementation of the dynamic program found in the first part in Language C.
- **The third part :** establish another method that will allow us to solve this problem.
- **The fourth part :** application of this algorithm on the example below:
  R = 10, N=4 and the matrix r:

	0	1	2	3	4	5	6	7	8	9	10
1	0	8	29	54	57	61	69	76	76	88	99
2	0	5	41	50	58	62	71	73	76	89	99
3	0	7	11	15	34	37	59	66	78	90	98
4	0	16	25	31	58	59	61	63	77	81	94

- **The fifth part :** as a conclusion, we will compare between the different algorithms we used to solve this problem.

Generalization of the problem

1.1 Definition of the problem

The aim of the resource allocation problem is to allocate a defined number of resources according to different activities and whose performance is measured by a gain, this term is represented by a value that we receive when we allocate a defined amount of resources to a specified activity. We will try to optimize this problem by maximizing the sum of the gains received by the allocation of the resources, that is, we must find a distribution of resources that maximizes our gain.

1.2 Presentation of the problem

We have a M resource to share among N activities, in order to maximize the gain $r(i,j)$ For this, we will try to find a (distribution of M resources) finite set of resources $\{resource_1, ..., resource_i\}$ with $i = 1, ..., N$ and $\sum_{i=1}^{M} resource_i = M$,(the resource i is allocated to the activity i), which allows us to obtain the maximum sum of the possible gains.

- **Objective function :** $f(x) = r(i,x)$
- **Initial condition :** $n = N$
- **Recurrence relationship :** $f_k(j) = max_{\begin{cases} 0 \leq x \leq M \\ 1 \leq k \leq N-1 \end{cases}} (r(k, j-x) + f_{k+1}(x))$
- **Optimal solution:** $f^* = f_1(M)$

First method : resolution of the problem using the dynamic programming

2.1 What is Dynamic programming ?

Dynamic programming is a method that in general solves optimization problems that involve making a sequence of decisions by determining, for each decision, sub-problems that can be solved in like fashion, such that an optimal solution of the original problem can be found from optimal solutions of sub-problems. This method is based on Bellman's Principle of Optimality, which he phrased as follows.

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

More succinctly, this principle asserts that "optimal policies have optimal sub-policies." That the principle is valid follows from the observation that, if a policy has a sub-policy that is not optimal, then replacement of the sub-policy by an optimal sub-policy would improve the original policy.[1]

2.2 Resolution of the problem using a iterative function : forward approach

2.2.1 Our algorithm

We have the function as follows to calculate the optimal solution which is : $f_1(M)$:
$f_k(j) = max_{\begin{cases} 0 \le x \le M \\ 1 \le k \le N-1 \end{cases}} r(k, j-x) + f_{k+1}(x)$ To get calculate $f_n(M)$, we apply this algorithm:
Let $F[a][r]$ that stock our results, the optimal values in every sub-problem. and T the matrix of gains., and $Policy[a][r]$ to stock the policies.

Algorithm 1: Calculate the optimal solution

```
for i=0 to R do
    F[A-1][i]=T[A-1][i]
end
for a=A-2 to 0 do
    for r=0 to R do
        k ← 0
        max ← -1
        policy ← new array() while $k \leq n$ do
            m=T[a][k]+F[a+1][n-k] if $m > max$ then
                max m
                policy.clear()
                policy.add(k)
            else
                policy.add(k)
            end
            increment k F[a][r]=max Policy[a][r]=policy
        end
    end
end
```

Example: Let M=3 and R=3 and the Matrix of gains is as follows:

	0	1	2	3
1	0	8	29	54
2	0	5	41	50
3	0	7	11	15

Problem resolution is as follows :

$$\textbf{First step: } \begin{cases} f_3(3) = r(3,3) = 15 \\ f_3(2) = r(3,2) = 11 \\ f_3(1) = r(3,1) = 7 \\ f_3(0) = r(3,0) = 0 \end{cases}$$

Second Step:

– $$f_2(3) = max \begin{cases} r(2,3) + f_3(0) = 50* \\ r(2,2) + f_3(1) = 48 \\ r(2,1) + f_3(2) = 16 \\ r(2,0) + f_3(3) = 15 \end{cases}$$

– $$f_2(2) = max \begin{cases} r(2,2) + f_3(0) = 41* \\ r(2,1) + f_3(1) = 12 \\ r(2,0) + f_3(2) = 11 \end{cases}$$

- $f_2(1) = max \begin{cases} r(2,1) + f_3(0) = 5 \\ r(2,0) + f_3(1) = 7* \end{cases}$
- $f_2(0) = 0$

Last step:

- $f_1(3) = max \begin{cases} r(1,3) + f_2(0) = 54* \\ r(1,2) + f_2(1) = 36 \\ r(1,1) + f_2(2) = 49 \\ r(1,0) + f_2(3) = 50 \end{cases}$

The optimal value is : 54

The distribution of resources that lead to this optimal value is:

54 resources allocated to first activity

0 resources allocated to second activity

0 resources allocated to third activity

2.2.2 Implementation in language C

/src/dyn_max_iterative.c

```c
void dyn_max_iterative(Matrix r){
    int A,R;
    A = r->activities;
    R = r->resource+1;

    /*
     *  'F' is a Matrix of size (A, R) used for caching optimal values in a way that
     *  for 0 <= i < activities  and  0 <= j <= resources :
     *  F[i][j] => the optimal possible value for 'j' resources allocated on (i, i+1, .., A-1)
    machines
     */
    Matrix F = createMatrix(A,R);
    // Initializing the last row of F - Last activity/machine - with the objective function s
    values
    for(int i=0;i<R;i++){
       F->values[A-1][i] = r->values[A-1][i];
    }

    /*
     *  'RA' : Resource Affected
     *  is also a Matrix of size (A, R) used for remembering the optimal policies/decisions
    made at a given point
     *  for 0 <= i < activities  and  0 <= j <= resources :
     *  RA[i][j] => a List of  optimal policies/decisions - resources that can be allocated for
     the machine i+1 -
     *  that if made an optimal value of F[i][j] can be reached for 'j' ressources allocated on
     (i, i+1, .., A-1) machines
     */
    List** RA = (List*)malloc(A*sizeof(List));
    for(int i=0;i<A;i++){
        RA[i] = (List*)calloc(R,sizeof(List));
    }

    // Initializing the last row of RA - Last activity/machine's RA - with the Allocated resources (0, 1, ..., resources)
    for(int i=0;i<R;i++){
        RA[A-1][i] = (List)malloc(sizeof(struct List));
        RA[A-1][i]->size = 1;
        RA[A-1][i]->values = (int*)malloc(RA[A-1][i]->size*sizeof(int));
        RA[A-1][i]->values[RA[A-1][i]->size-1]=i;
    }

    /*
     *  Looping through activities From i = (activity-1) into the first one - and for each F and RA s row respectively -
     *  We Fill each of F[i] and RA[i] columns with the optimal value and the appropriate allocated resources
     */
    for(int i=A-2;i>-1;i--){
        for(int j=0;j<R;j++){
            int k=0;
            int optimal=-1;

            List ra = (List)malloc(sizeof(struct List));
            ra->size=0;
            ra->values=NULL;

            while(k<=j){
                int temp = r->values[i][k] + F->values[i+1][j-k];
                if( temp > optimal){
                    optimal=temp;
                    ra->size = 1;
                    ra->values=(int*)realloc(ra->values,sizeof(int));
                    ra->values[0] = k;
                }else if(temp==optimal){
                    ra->size++;
                    ra->values=(int*)realloc(ra->values,ra->size*sizeof(int));
                    ra->values[ra->size-1] = k;
                }
                k++;
            }
            F->values[i][j] = optimal;
            RA[i][j] = ra;
        }
    }

    //  Printing the optimal values stored in the Caching Matrix F[0][resource]
    printf("Optimal value : %d \n",F->values[0][R-1]);


    //  Printing optimal policies by checking the values of the RA Matrix
    int s = (int)calloc(A,sizeof(int));
    print_optimal_paths(RA,A,R−1,1,s);

    //free Memoire allouee par F,RA et s
    freeMatrix(F);
    free3D(RA,R,A);
    free(s);
}
```

2.3 Resolution of the problem using a recursive function : backward approach

2.3.1 Our algorithm

```
/* "solution" a type represents a gain received by a specified distribution */
type : solution {
gain : integer
distribution : an array of two dimensions represents a set of distributions
}
```

Function DynamicFunction(resource : integer,activity : integer ,data : 2D array ,r : 2D array)

```
Input: Data : 2D matrix used to save our results
Output: optimal : solution
/* n is the total number of activities                                   */
if $activity = n$ then
    optimal : solution                     // initialize a variable of type solution
    temp.gain← $r[activity][resource]$
    temp.distribution ← [[n]]
    data$[activity][resource]$ ←temp
    return temp
end
optimal : solution
rest = 0
while $rest \leq resource$ do
    if we already calculated the value of data[activity+1][resource] then
        temp ← data[activity+1][resource]
    else
        temp ← DynamicFunction(resource-rest,activity+1,data,r)
    end
    val ← r[activity-1][rest]+temp.gain
    if $optimal.gain < val$ then
        optimal.gain ← val
        optimal.distribution.clear()
        optimal.distribution.add(temp.distribution)
    else if optimal.gain is equal to val then
        optimal.distribution.add(temp.distribution)
    increment rest
end
data[activity][resource]← optimal
return optimal
```

This algorithm will not just find the maximum value of gains but also the distributions of resources whose give us that gain.

2.3.2 Implementation of the algorithm in language c

2.3.2.1 Loading gains' data from a csv file

First thing to do, is loading gains' data to our algorithm, to do that, we defined a new structure to represent a matrix where we will stock the data after loading it from a csv file.
CSV : A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

This is our structure Matrix:

```
struct Matrix{
  int ressource;
  int activities;
  int **values;
};
typedef struct Matrix* Matrix;
```

And to manipulate this Matrix we defined 5 methods:

- $createMatrix$: to create a Matrix by allocating memory for it and returns a the created Matrix.
- $getMatrix$ to load data from a csv file, this method returns a Matrix of the loaded data.
- $getLine$ used by the method $getMatrix$, to get a single line from the csv file.
- $freeMatrix$ used to free a memory allocated by a Matrix.
- $showMatrix$ used to show the matrix in the output.

```
Matrix createMatrix(int,int);
Matrix getMatrix(FILE*);
char *getLine(FILE*);
void freeMatrix(Matrix);
void showMatrix(Matrix);
```

2.3.2.2 Structure of the problem's solution:

$/src/Headers/dyn\_max\_recursive.h$ $/src/dyn\_max\_recursive.c$ To stock the results of our dynamic function, the value of the maximum gain and the distribution leading to this value, we created a LinkedList as fallows: Policy structure contains 2 variables :

- value : the ressource allocated
- next : pointer points on the next element in the policy

```
struct Policy{
    int value;
    struct Policy *next;
};
typedef struct Policy* Policy;
```

Result structure contains 3 variables :

- optimalValue : the optimal value found in dyn_max
- optimalValue : the optimal value found in dyn_max
- nPolicies : keeps track of number of policies found
- optimalPolicies : list of heads of policies (each policy is a linkedList)

```
struct Result{
    int optimalValue;
    int nPolicies;
    Policy *optimalPolicies;
};
typedef struct Result* Result;
```

```
struct List{
    int size;
    int *values;
};
typedef struct List* List;
```

createCache **:** this function create a matrix of Result structure to store pointers of calculated results.

```

Result **createCache(Matrix r);
```

clearPoliciesTable **:** policies array is a list of heads of linkedlists (each policy is a linkedlist) and this function clears this table of a particulare Result.

```
void clearPoliciesTable(Result r);
```

freeCache **:** it frees the memory allocated by each Result in the Cache

```
void freeCache(Result **cache,int rows,int colums);
```

freeCache **:** Print policies on the output.

```
void showPolicies(Result r,int activities);
```

2.3.2.3 The implementation of our dynamic function using backward approach:

Now, we defining the dynamic function that will calculate the optimal solution:

```c
Result dyn_max(Matrix r,Result **cache,int activity,int resource){
// this part respresents the stop condition of the function, ie if we reach the last activity
    the we affect all the remaining resources
    if(activity == r->activities){
        Result result = (Result)malloc(sizeof(struct Result));
        result->optimalValue = r->values[activity -1][resource];
        result->nPolicies = 1;

        result->optimalPolicies = (Policy*)malloc(sizeof(Policy));
        result->optimalPolicies[0] = (Policy)malloc(sizeof(struct Policy));
        // affectation of the remaining resources
        result->optimalPolicies[0]->value = resource;
        result->optimalPolicies[0]->next = NULL;

        // memorization of the result
        cache[activity -1][ressource] = result;
        return result;
    }
    // if there are still activities
    Result optimal=(Result)malloc(sizeof(struct Result));
    optimal->optimalValue=0;
    optimal->nPolicies = 0;
    optimal->optimalPolicies = NULL;

    Result temp;

    for(int i=0 ; i<=resource ; i++) {
        // if the value is already calculated
        if ( cache[activity][resource-i] != NULL ){
            // the value already calculated
            temp = cache[activity][resource-i];
        }else{
        // calculate cache[activity][resource-i]
            // calculate the value
            temp = dyn_max(r, cache, activity+1, resource-i);
        }

        int gain = r->values[activity -1][i] + temp->optimalValue;

        // if it found a new optimal value we update the Result
        if(optimal->optimalValue <= gain){
            int newSize,start;
            // if it found a new distribution that lead to the optimal value we add it to the
    Result
            if (optimal->optimalValue == gain){
                newSize = optimal->nPolicies + temp->nPolicies;
                start = optimal->nPolicies;
            }else{
            // else we clear the Policies in the Result structure
                if(optimal->nPolicies > 0 )
                    clearPoliciesTable(optimal);
                start = 0;
                newSize = temp->nPolicies;
            }
            // copy the Policies of temp into Result
            optimal->nPolicies = newSize;
            optimal->optimalPolicies = (Policy*)realloc(optimal->optimalPolicies,optimal->
    nPolicies*sizeof(Policy));
            optimal->optimalValue = gain;
            for(int j=start;j<optimal->nPolicies;j++){
                optimal->optimalPolicies[j] = (Policy)malloc(sizeof(struct Policy));
                optimal->optimalPolicies[j]->value = i;
                optimal->optimalPolicies[j]->next = temp->optimalPolicies[j-start];
            }
        }
    }
    //Memorization of the results
    cache[activity -1][resource] = optimal;
    return optimal;
}
```

Second method : resolution of the problem using the brute force

3.1 what is brute force algorithm ?

Brute Force Algorithms refers to a programming style that does not include any shortcuts to improve performance, but instead relies on sheer computing power to try all possibilities until the solution to a problem is found.

This method consists of two steps:

- Generate all possible cases of resource allocations.
- Test for each distribution generated by the first step if it represents an optimal solution for our problem.

This two steps are not separated we apply its simultaneously, to avoid using two much memory, at each distribution generated we apply the test if it represents an optimal solution for the set of the distribution already tested.

3.2 Generation of all the possible cases of resource distributions

To generate all the possible cases of resource allocation on activities, we generate at first all possible cases of distribution following the constraints below:
Let D be a distribution of the M resources:

- The number of elements of the distribution D is : $1 \leq$ size(D) $\leq$ N, where N is the number of activities we have.
- The sum of all the elements of the distribution R is equal to M, $\sum_{k \epsilon R} k = M$.
- We add zeros to each distribution that has a number of elements strictly smaller than N, such as the number of elements of the distributions is equal to N. which means that we can allocate resources to only a subset of activities.
- Since the gain depends on the activity and the number of resources allocated, the order of the elements of the distribution is important, for that we generate every permutation possible of each distribution of M resource.

Example: We have 5 resources to share among 3 activities:

- The combination of distributions that the sum of its elements is equal to 5 and have a number of elements smaller or equal to 3.
  $C = [[1, 1, 3], [1, 2, 2], [1, 4], [2, 3], [5]]$
- The combination of distributions after adding zeros to the distributions that has a number of elements smaller than 3:
  $C = [[1, 1, 3], [1, 2, 2], [1, 4, 0], [2, 3, 0], [5, 0, 0]]$
- Our set of all possible cases after the permutation of all the distributions:
  P =[[3, 0, 2], [1, 3, 1], [0, 1, 4], [2, 3, 0], [0, 5, 0], [4, 0, 1], [1, 4, 0], [1, 0, 4], [0, 2, 3], [0, 0, 5], [0, 3, 2], [3, 2, 0], [3, 1, 1], [2, 0, 3], [1, 2, 2], [4, 1, 0], [2, 1, 2], [5, 0, 0], [1, 1, 3], [2, 2, 1], [0, 4, 1]]

3.3 The test of all the possible cases of resource distribution

After the generation of each possible distribution of M resources, we apply the test by calculating the sum of gains obtained. The distribution(s) that allows to have the maximum possible gain is(are) the optimal solution(s) of our sub-problem that consists of the distribution already tested.

Example: Applying the tests on the set P obtained by the example in section 3.2, we obtain the optimal solution is : 95 and the distribution that leads to this solution is :(3, 2, 0).
and the Matrix of gains is as follows:

	0	1	2	3	4	5
1	0	8	29	54	57	61
2	0	5	41	50	58	62
3	0	7	11	15	34	37
4	0	16	25	31	58	59

3 resources are given to the activity 1, we gain 54.
2 resources are given to the activity 2, we gain 41.
0 resources are given to the activity 3, we gain nothing.
and the sum of gains is 95.

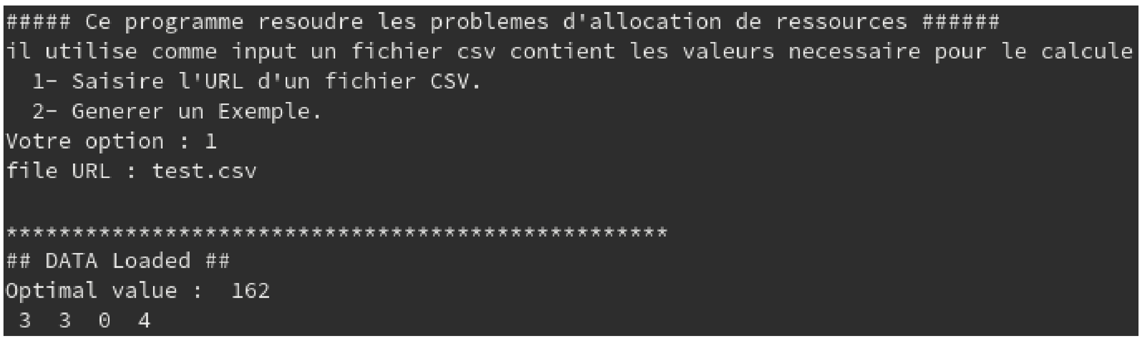
The implementation of this algorithm is in :
$/src/bruteforce.c$ as shown in part 6

Application of the first method on an example of resources allocation problem

In this example,we have 10 resources, and 4 activities and the matrix r representing the gains is as follow:

	0	1	2	3	4	5	6	7	8	9	10
1	0	8	29	54	57	61	69	76	76	88	99
2	0	5	41	50	58	62	71	73	76	89	99
3	0	7	11	15	34	37	59	66	78	90	98
4	0	16	25	31	58	59	61	63	77	81	94

Figure 4.1: Application of the first method on an example of resources allocation problem



```
##### Ce programme resoudre les problemes d'allocation de ressources ######
il utilise comme input un fichier csv contient les valeurs necessaire pour le calcule
  1- Saisire l'URL d'un fichier CSV.
  2- Generer un Exemple.
Votre option : 1
file URL : test.csv

**************************************************
## DATA Loaded ##
Optimal value :  162
 3  3  0  4
```

Conclusion

5.1 Execution time

5.1.1 Brute force algorithm vs Dynamic programming : forward approach

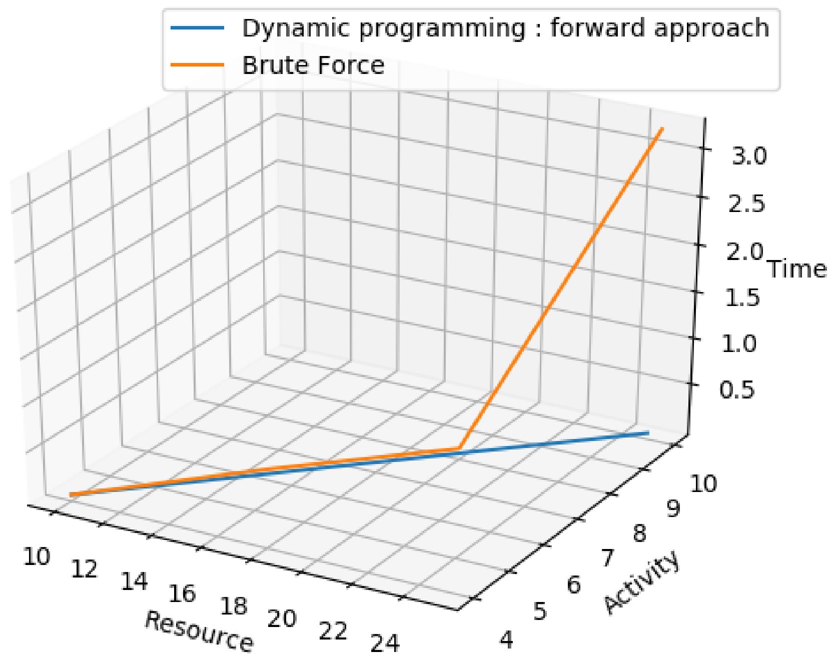


Figure 5.1: Brute Force vs Dynamic programming : forward approach

As the graph shows, the execution time function for the Brute Force algorithm increases exponentially as the number of resources and the activities increase, while the execution time of the forward approach is increases slowly, it still can solve the problem even if the number of resources and activities is too big.

Using forward approach (Dynamic programming) we were able to solve problem of 10000 resources among 100 activities in just 17.62551 seconds.

On the other hand, the brute force algorithm take a long time when the resources exceed 30, and the activities exceed 12, moreover it couldn't solve big problems.

5.1.2 Dynamic programming : forward approach vs backward approach

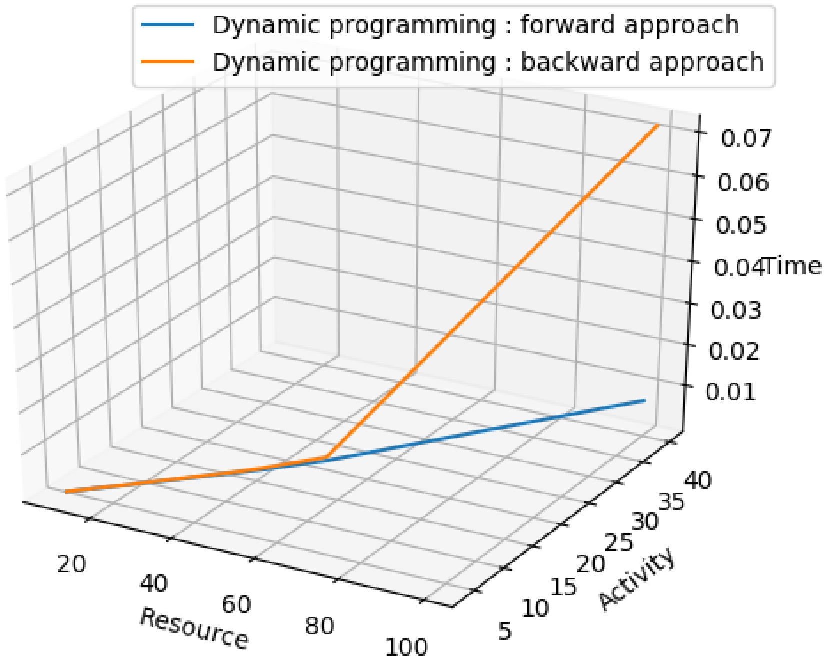


Figure 5.2: Dynamic programming : forward approach vs backward approach

When the resources M is $M \leq 50$ and the activities A is $A \leq 20$, the execution time of the two approaches is almost equal, if it is not the case, the execution time of the backward approach increases to be 10 time bigger than the execution time of the forward approach.

5.2 Memory usage

5.2.1 Brute force algorithm vs Dynamic programming : forward approach

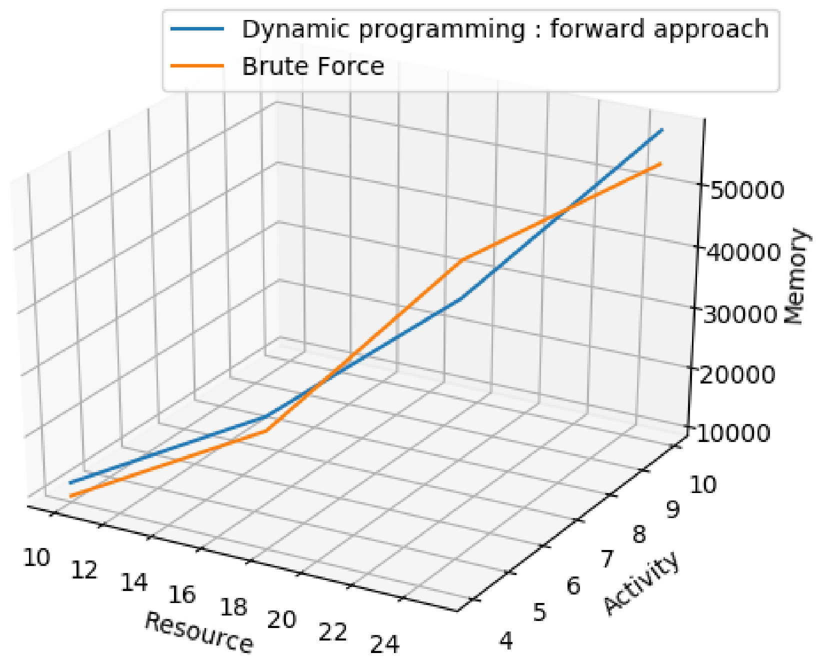


Figure 5.3: Brute Force vs Dynamic programming : forward approach (Memory in bytes)

When we compare the memory usage between the brute force algorithm and the forward approach (Dynamic programming) the memory usage is almost equal for both methods, the difference might be obvious when the number of the resources and the activities.

5.2.2 Dynamic programming : forward approach vs backward approach

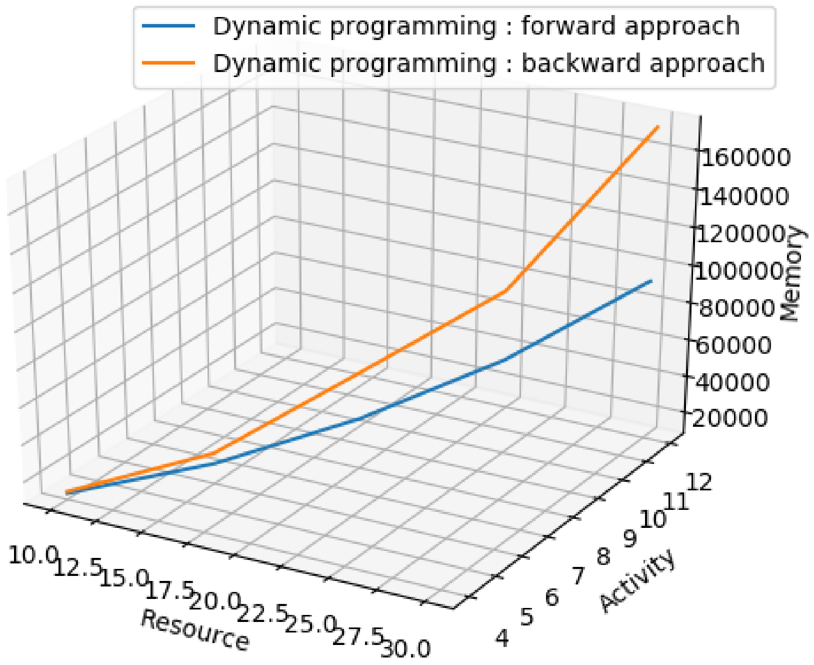


Figure 5.4: Brute Force vs Dynamic programming : forward approach (Memory in bytes)

As it shown in the graph, the backward approach uses more memory than the forward approach since we used too much memory saving the distribution of the resources that might be the optimal solution.

5.3 Which is the best approach to solve the resources allocation problem?

Using the dynamic programming we were able to solve complex the resources allocation problems in a short time while the brute force couldn't since it tests all the possible cases that the problem could encounter, while the dynamic programming divide the problem to multiple sub-problems which make it too much easier to solve in a reasonable time.

And by using the forward approach we were able to solve problem in a short time and using less memory comparing it with the backward approach.

Project Structure:

Github :

$https://github.com/ismailnajah/Ressource\_Allocation\_Problem\_DP$

$/src/dyn\_max\_iterative.c$: dynamic programming : forward approach

$/src/dyn\_max\_recursive.c$: dynamic programming : backward approach $/src/bruteforce.c$: Brute force algorithm

$/src/CSV\_Parser.c$: all function to manipulate the Matrix structure and to read the csv files.

$/src/main.c$: the main program $/python_version$: the python version of the algorithms

Bibliography

[1] Art Lew and Holger Mauch. *Dynamic programming: A computational tool*. Vol. 38. Springer, 2006.