SOLVING A GENERALIZED RESOURCES ALLOCATION PROBLEM USING DYNAMIC PROGRAMING

IMPLEMENTED BOTH IN *C* AND *Python*

MOHAMED HMINI

Achieved as part of a set of Optimization Techniques Projects

Supervised by : RACHID BENMANSOUR
COMPUTER SCIENCE DEPARTMENT
INSEA
MOROCCO

November 19, 2019

Abstract

SOLVING A GENERALIZED RESOURCES ALLOCATION USING DP

Abstract

in larger companies and organizations resources allocation is as critical as considering the worst to happen if the optimal decision isn't taken in an optimal time, fortunately the field of operations research is focusing on modeling and implementing a number of algorithms which can by far approximate our understanding to the real-world phenomena.

we are used to target the ideal utopia or in our concrete terms the optimum, the problem of optimization occurs everywhere in life from sinks at homes to airplanes, it should be recognized that richard bellman's introduction to dynamic programming technique was a paradigm shit in the way we deal with problems in general.

Keywords Resources Allocation, Operation Research, Constraint Programming

Thanks to our beloved teacher **RACHID BENMANSOUR**

Contents

Listings

List of Figures

List of Tables

Chapter 1

Overview

1.1 Introduction to GRA problems :

In economy resources allocation is the task of assigning resources to a set of tasks which makes GRA problems one of the most known vital subjects of operations research, it's also a very critical and sometimes a time consuming issue, the more constraints are added the harder the problem becomes. (e.g : time, cost-functions ...)

Allocation problems involve the distribution of resources among competing alternatives in order to minimize total costs or maximize total return. Such problems have the following components: a set of resources available in given amounts; a set of jobs to be done, each consuming a specified amount of resources; and a set of costs or returns for each job and resource. The problem is to determine how much of each resource to allocate to each job.

these type of problems can be very difficult to solve indeed, if we consider both the cost and the returned value after one allocation or assignment, assignment is sort of putting one or multiple entities responsible for one or multiple other entities.

1.1.1 Capital Allocation

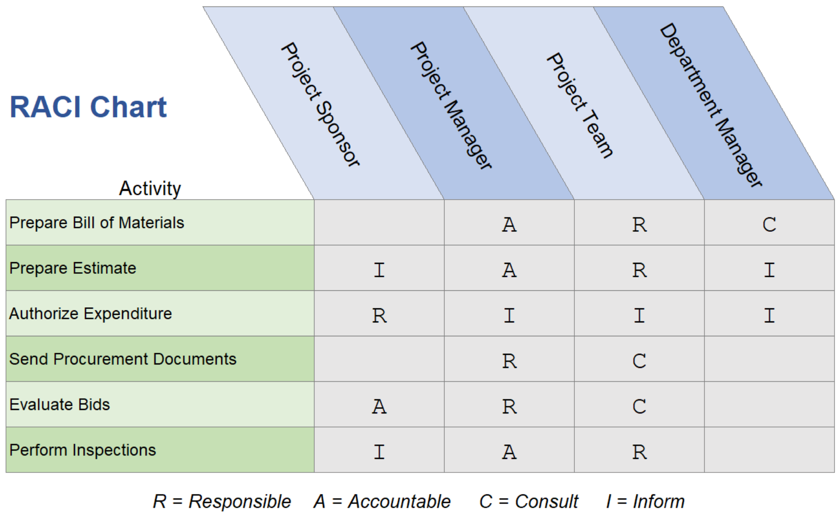
the action of investing can be considered as a resources allocation problem where R is the total allowed investment capital and $A_i(R_j)$ is the return on investment or **ROI** if we invested j resource from R in the i^{th} activity.

	R_j			
A_i	0	50	90	100
	0	10	60	620
	-1	-40	500	1000

Table 1.1: GRA dataset example

1.1.2 Business management (Tasks Assignment)

In the world of management assigning tasks to the appropriate people is a critical problem which can either cause big troubles to the company or generate huge benefits, businessmen usually tend to use a chart known by the name **RACI CHART**, the below's a **figure** 1.1 containing a raci chart example from a typical company.



RACI Chart

Activity	Project Sponsor	Project Manager	Project Team	Department Manager
Prepare Bill of Materials		A	R	C
Prepare Estimate	I	A	R	I
Authorize Expenditure	R	I	I	I
Send Procurement Documents		R	C	
Evaluate Bids	A	R	C	
Perform Inspections	I	A	R	

R = Responsible   A = Accountable   C = Consult   I = Inform

Figure 1.1: RACI CHART EXAMPLE

RACI charts show the project resources assigned to each work package. They illustrate the connections between tasks and project members. RACI charts are an extremely fast and efficient way to communicate the project's roles and responsibility assignments for all of the project members.

	T1	**T2**	**T3**	**T4**
P1	0.2	0	0	0.8
P2	0.1	0.3	0.3	0.3

Table 1.2: PROBABILITY-BASED GRA

we can use this chart for instance to generate a **task failure probability dataset** as shown in **figure** 1.2 above, and then define a function to minimizing the cost by iteratively minimizing smaller and smaller problems in order to minimize the whole objective function.

1.2 The Essence of Dynamic programming :

In his Article **An Introduction to the Theory of Dynamic Programming**, Mr. **Richard Bellman** Introduces the new notion known by the famous name dynamic programming, he defines it as follows :

> To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time / is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

by which we understand that a certain number of problems can be solved by reducing them into a set of smaller yet larger number of problems, it's a good approach to transform a combinatorial problem into a polynomial one by ignoring the pre-treated sub-problems, there are two approaches to this very same technique :

- **Tabulation** : the typical dynamic programming approach which is based on a bottom-up view of steps.
- **Memoization** : a prog[]rammable recursive version of dynamic programming which creates a tree-like set of operations aka top-down approach.

although that dynamic programming doesn't work its way towards the optimal solution that fast everytimes but it's true that it's considered one of the most prosperous optimization techniques in use right now, one of the vital domains of application could be **Transportation, Scheduling, Resources Allocation ...**, yet one of its cons is its exactness which makes it as bad as other algorithms or techniques to solve optimization problems.

my personal view of this technique is alike what most people think if it, a **careful brute-force algorithm**, however, its nature makes it a bit harsh to be used all the time, cause there isn't a clear general trick for every problem, it's always needed to sit and brainstorm the ideal way of sub-deviding the problem.

Chapter 2

Implementation of a dynamic programming solution to a GRA problem

2.1 Pseudo-code Implementation

2.1.1 Defining the Objective function

let's consider those symbols as the main notations for our problem :

R_j : is the j^{th} resource.

A_i : is the i^{th} activity (also the **step** number).

we'll implement our DP objective function such that the number of activities is the number of the needed steps plus an initial step to allow embarking the process of the successive optimization operations.
then for a step t the optimum is defined as follows :

$$f_t(R_m) = max_{0<i<m}[A_t(R_i) + f_{t+1}(R_j)] \tag{2.1}$$

$$with : R_j = R_m - R_i \tag{2.2}$$

R_m : is the current resource to be assigned.

R_i : is a resource whereby this constraint $0 \leq R_i \leq R_m$ is satisfied.

R_j : is a resource whereby this equation $R_j = R_m - R_i$ is valid from now on, until the end of this section we'll only consider the bottom-up approach therefor the initial value of our recursive function have to be defined as follows :

$$f_{N+1}(x) = 0$$

where N is equal to the total number of steps!

2.1.2 Implementing the respective pseudo-code

we'll consider for the sake of simplicity a pseudo-algorithm for only one optimal policy tracker, so we can move on to the real implementation with multiple optimal-policies solution later when we'll make our way through **C** and **Python**.

P.S : a policy is a path or a certain possible combination of the available elements.

the **Bottom-up Allocator Algorithm** 1 implemented in page 6 is a simplified version of a dynamic programming approach to a GRA problem, the algorithm uses a special type of data-structures called **data-frames** whereby one can store any type of a variable into it as a node making it a multi-level heterogeneous structure, the used function to create a dataframe is called **Dataframe** which accepts the shape of the nodes and the initial value as inputs.

one node is composed with three variants :

- i or the allocated resources during the current activity. (mathematically it represents the **argmax**)
- j or the allocated resources for the next activity.
- v the current stage generated value (current optimum).

```
Algorithm 1: Bottom-up Allocator(R, N, M)
  Input: dataset matrix R,  total # of activities N,  total # of available machines M
  Result: (optimal value V,optimal policy P)

1  $F \leftarrow DecisionMakingMatrixGenerator(R, N, M)$
2  $result \leftarrow getPolicies(F, N, M)$
3  $V \leftarrow result[0]$
4  $P \leftarrow result[1]$

5  Function DecisionMakingMatrixGenerator($R, N, M$):
6      FInitVal = Array(shape = 3, initVal = 0)
         $F \leftarrow Dataframe(shape = (N, M, 3), initVal = FInitVal)$
7      $Step \leftarrow N - 1$

8      while $step > 0$ do
9          for $m \leftarrow 0$ to $M$ do
10             for $i \leftarrow 0$ to $m$ do
11                 $j \leftarrow m - i$ $currentVal \leftarrow R[step - 1][i] + F[N - k - 1][j][2]$ if
                     $currentVal > F[N - k][m][2]$ then
12                     $F[N - k][m] = [i, j, v]$
13                 End if
14             End for
15         End for $step = step - 1$
16     End while
17     return $F$
18 End Function

19 Function getPolicies($F, N, M$):
20     $step = N$
21     $optimalVal \leftarrow F[N - 1][M - 1]$
22     $optimalPolicy = Array(shape = N)$
23     $optimalPolicy.push(F[N - 1][M - 1])$
24     while $step > 0$ do
25         $i \leftarrow optimalPolicy[N - k][1]$ $optimalPolicy.push(F[K - 2][i])$ $step = step - 1$
26     return $[optimalVal, optimalPolicy]$
27 End Function
```

algorithm 1 consist of two stages, the first is to make the so called **decision making matrix** which i called **F** and a stage whereby the optimal policies are extracted (**getPolicies** function), the idea is as simple as traversing the F matrix from left to right and vice versa at the same time in an opposite way (i **and** j) for every coordinate ($step$ **and** m).

2.1.3 practical example of a GRA problem

	M			
N	0	5	30	30
	0	4	20	40
	0	2	10	80

Table 2.1: RANDOMLY GENERATED GRA DATASET PROBLEM (usecase)

instead of trying to explain the plain code of this algorithm let's try to use it with a small data-set and see how it works much more clearly, the used dataset is filled with randomly generated values optained in the above's **table 2.1**.

	0	**1**	**2**	**3**
4	0	0	0	0
3	0	2	10	80
2	0	4	20	80
1	*	*	*	80*

Table 2.2: GRA-EXAMPLE-DECISION-MAKING-MATRIX

the solution to that problem is given starting from this page 8, as clear as it is we can iteravely go from the initial step and solve all of the possible sub-problems and then store them in some sort of a matrix to access them later on, such matrix is given in **figure 2.2** represented above.

the solution of our respective example given above is underneath this paragraph :

STEP t = 4

$$f_4(x) = 0$$

STEP t = 3

$$f_3(0) = max \left\{ A_3(0) + f_4(0) = 0^* \right.$$

$$f_3(1) = max \begin{cases} A_3(0) + f_4(1) = 0 \\ A_3(1) + f_4(0) = 2^* \end{cases}$$

$$f_3(2) = max \begin{cases} A_3(0) + f_4(2) = 0 \\ A_3(1) + f_4(1) = 2 \\ A_3(2) + f_4(0) = 10^* \end{cases}$$

$$f_3(3) = max \begin{cases} A_3(0) + f_4(3) = 0 \\ A_3(1) + f_4(2) = 2 \\ A_3(2) + f_4(1) = 10 \\ A_3(3) + f_4(0) = 80^* \end{cases}$$

STEP t = 2

$$f_2(0) = max \left\{ A_2(0) + f_3(0) = 0^* \right.$$

$$f_2(1) = max \begin{cases} A_2(0) + f_3(1) = 2 \\ A_2(1) + f_3(0) = 4^* \end{cases}$$

$$f_2(2) = max \begin{cases} A_2(0) + f_3(2) = 10 \\ A_2(1) + f_3(1) = 6 \\ A_2(2) + f_3(0) = 20^* \end{cases}$$

$$f_2(3) = max \begin{cases} A_2(0) + f_3(3) = 80^* \\ A_2(1) + f_3(2) = 14 \\ A_2(2) + f_3(1) = 22 \\ A_2(3) + f_3(0) = 40 \end{cases}$$

STEP t = 3

$$f_1(3) = max \begin{cases} A_1(0) + f_2(3) = 80^* \\ A_1(1) + f_2(2) = 25 \\ A_1(2) + f_2(1) = 34 \\ A_1(3) + f_2(0) = 30 \end{cases}$$

2.2 C Implementation

2.2.1 overview of HLCio

the functioning of our **resoruces allocator** algorithm depends on another project i created named **HLCio** which can be found in the following link https://github.com/MohamedHmini/HLCio, the importance of this library consists of providing a handy way of dealing with I/O and data-structure in c, it's a high-level c library.

the **HLCio** library contains two main headers :

- **advio.h** : provides functions to read CSV files.
- **dataframe.h** : provides a different representation of high dimensionality data such as nested matrices or arrays called a dataframe.

2.2.2 what is a dataframe?

due to the unfortunate low-level nature of the C programming language i was sort of forced to create some code to at least approximate the high-level language when dealing with data, a data-frame is composed mainly by three elements the dimensions number of rows and columns as well as a data-container or a pointer of rows as represented in the **figure 2.1** below.

```c
typedef struct DATAFRAME{

  int len_rows;
  int len_cols;
  DF_ELEMENT_TYPE type;
  DF_DATA_CONTAINER data;

}DATAFRAME;
```

Listing 2.1: C DATAFRAME STRUCTURE

the **DF DATA CONTAINER** is just another pointer to the rows of dataframe given as a set of **DF ELEMENT CONTAINER** elements, its description is given in the **figure 2.2** below.

```c
// plays the role of the element's value holder :
typedef DF_ELEMENT* DF_ELEMENT_CONTAINER;

// plays the role of an array :
typedef DF_ELEMENT_CONTAINER* DF_DATA_CONTAINER;
```

Listing 2.2: CONTAINERS

the **DF ELEMENT CONTAINER** is an **DF ELEMENT** holder as represented in the **figure 2.3**.

```c
typedef struct DF_ELEMENT{
  DF_ELEMENT_TYPE type;
  DF_NODE node;
}DF_ELEMENT;
```

Listing 2.3: DF ELEMENT

the last stage is the **DF NODE** or the very last element container in the theoretical representation of a data-frame, it can be understood from the **figure** 2.4.

```
  // the actual element :
  typedef union DF_NODE{
    char* Str;
    int Int;
    double Double;
    struct Array *Arr;
  } DF_NODE;
```

Listing 2.4: DF NODE

when the **DF NODE** is more complex than when it holds a simple primitive type such as Str, Int or a Double i use a new nested element called **Array** represented in **figure** 2.5.

```
  typedef struct Array{
    int size;
    DF_ELEMENT_CONTAINER data;
  }Array;
```

Listing 2.5: ARRAY

then to keep track of nodes types i use an enumeration called **DF ELEMENT TYPE** represented in **figure** 2.6.

```
  typedef enum DF_ELEMENT_TYPE {
    DF_ELEMENT_TStr,
    DF_ELEMENT_TInt,
    DF_ELEMENT_TDouble,
    DF_ELEMENT_TNaN,
    DF_ELEMENT_TArray
  } DF_ELEMENT_TYPE;
```

Listing 2.6: DF ELEMENT TYPE

the same header contains a number of functions which enables the exploitation of these data-structures, such functions can provide array operations such as **PUSH, POP** or **create new data-structures**, the user of this library isn't in need to allocate memory himself thus no need to deal with memory leak in a personal way, **figure** 2.7 shows the declaration of these functions.

```
 DATAFRAME *Dataframe(int ,int , DF_DATA_CONTAINER );
  void df_free(DATAFRAME *df);
 DF_ELEMENT arrcreate(int);
  void arrpush(DF_ELEMENT*, DF_ELEMENT);
  void arrfree(DF_ELEMENT*);
  void arrpop(DF_ELEMENT*);
 DF_ELEMENT df_element_copy(DF_ELEMENT);
 DATAFRAME *df_full(int, int ,DF_ELEMENT_TYPE ,DF_ELEMENT);
  void DF_STR_TO_INT(DF_ELEMENT*);
  void DF_INT_TO_STR(DF_ELEMENT*);
  void df_map(DATAFRAME *,void (*fun)(DF_ELEMENT* df_element), ...);
  void df_retype(DATAFRAME *, DF_ELEMENT_TYPE );
  void display_df(DATAFRAME *);
  void show(DATAFRAME *);
```

Listing 2.7: DATAFRAME HEADER FUNCTIONS

the other header with the name **advio.h** enables the conversion of **CSV** files to **DATAFRAMES**, the below's **figure** 2.8 represent the declaration of function used in advio.c.

```c
FILE *open_file(char*, char*);
bool close_file(FILE *);
char* get_line(FILE *);
char** read_lines(FILE *);
DF_ELEMENT_CONTAINER tokenize_line(DATAFRAME *, char* , char* );
void tokenize(DATAFRAME *, char** , char* );
DATAFRAME *csv_to_df(FILE *);
```

Listing 2.8: ADVIO HEADER FUNCTIONS

please refer to the Github Repo given above for extra details!

2.2.3 Bottom-up Dynamic programming implementation

the actual project which contains the respective code for our DP solution can be found in the following **Github Repo** https://github.com/MohamedHmini/GRA, i've implemented a bottom up solution for this problem due to its easiness and beauty.

the code we'll see uses a new type of data-structure to store the results of the given dataset, the result is composed of two elements, **optimal value** which has been found at the last stage of our algorithm and **the policies** which define the set of different optimal paths or combinations to generate the same optimum, the code of the result data structure is given in **figure** 2.9.

```c
typedef struct GRA_RESULTS{
  int optimal_value;
  DF_ELEMENT policies;
}GRA_RESULTS;
```

Listing 2.9: GRA RESULT DATA STRUCTURE

we'll use the functions defined in the **GRA.h** header which can be seen in **figure** 2.10.

```c
GRA_RESULTS *gracreate();
DATAFRAME *fit_to_find_one_optimal_path(DATAFRAME*);
DATAFRAME *fit_to_find_multiple_optimal_paths(DATAFRAME*);
void load_policies(DATAFRAME*, int , int , DF_ELEMENT*, DF_ELEMENT*);
void display_results(GRA_RESULTS);
```

Listing 2.10: GRA HEADER FUNCTIONS

let's discuss the algorithm itself and then see a use case of its usage, first we create an array of length 3 contains only zeros and then we'll initialize our decision making matrix **F** nodes with that array's value, we use an array to store the current allocation, the next allocation and the local optimum generated value, the variable **K** in the code represents the **current step**, the initialization part of the algorithm is given in **figure** 2.11 .

```c
DATAFRAME *fit_to_find_multiple_optimal_paths(DATAFRAME *r){

    DF_ELEMENT e = arrcreate(1);
    e.node.Arr->data[0] = arrcreate(3);

    for(int i = 0;i < 3; i++){
        e.node.Arr->data[0].node.Arr->data[i].type = DF_ELEMENT_TInt;
        e.node.Arr->data[0].node.Arr->data[i].node.Int = 0;
    }


  DATAFRAME *f = df_full(r->len_rows + 1, r->len_cols , DF_ELEMENT_TArray, e
    );

    int N = f->len_rows;
    int k = N - 1;
    int M = r->len_cols;
    .
    .
    .
}
```

Listing 2.11: ALLOCATOR ALGO INITIALIZATION

the second thing to consider is the skeleton of our algorithm, we need three loops to make sure we are considering every node in our **F** data-frame, the first loop we'll iterate over steps starting from step number **N - 1** then the second loop will iterate over all of the **M** resources starting from 0, the last but not least loop will move accordingly and iterate over resources which are less or equal to the second's loop output, the sekeleton is given in **figure** 2.12.

```c
DATAFRAME *fit_to_find_multiple_optimal_paths(DATAFRAME *r){
    . . .

    while(k > 0){

        int m ;
        for(m = 0; m < M; m++){
            int i;
            for(i = 0; i< m + 1; i++){
                int j = m - i;
                . . .

            }
        }
        k--;
    }
}
```

Listing 2.12: ALLOCATOR ALGO SKELETON

the last step takes palce underneath the third loop shown above, we add the max value found from the given formula 2.2 to the previously stored max values if we met the same optimum otherwise we clear the previously stored arrays in that node and initiate it with the current max value, the main content of this algorithm is represented in **figure 2.13**.

```c
DATAFRAME *fit_to_find_multiple_optimal_paths(DATAFRAME *r){
    .
    .
    .
    int v = r->data[k - 1][i].node.Int + f->data[N - k - 1][j].node.Arr->
   data[0].node.Arr->data[2].node.Int;

    if(v >= f->data[N - k][m].node.Arr->data[0].node.Arr->data[2].node.Int)
   {
        if(v > f->data[N - k][m].node.Arr->data[0].node.Arr->data[2].node.
   Int){
            arrfree(&f->data[N - k][m]);
            f->data[N - k][m] = arrcreate(0);
        }

        DF_ELEMENT e = arrcreate(3);
        int coords[3] = {i,j,v};

        for(int c = 0;c < 3; c++){
            e.node.Arr->data[c].type = DF_ELEMENT_TInt;
            e.node.Arr->data[c].node.Int = coords[c];
        }
        arrpush(&f->data[N - k][m], e);
    }
    .
    .
    .
```

Listing 2.13: ALLOCATOR ALGO MAIN CONTENT

the second function iterate recursively over all the optimum nodes (argmax) and create a matrix of optimal policies which have been found during the execution of our algorithm, the code is given in **figure 2.14**.

```c
void load_policies(DATAFRAME *f, int k, int i, DF_ELEMENT *policy,
   DF_ELEMENT *policies) {
    if(k == 0){
        DF_ELEMENT copy = df_element_copy(*policy);
        arrpush(policies, copy);
        return;
    }

    for(int j = 0; j < f->data[k - 1][i].node.Arr->size; j++){
        arrpush(policy, f->data[k - 1][i].node.Arr->data[j].node.Arr->data
   [0]);
        load_policies(f, k - 1, f->data[k - 1][i].node.Arr->data[j].node.
   Arr->data[1].node.Int, policy, policies);
        arrpop(policy);
    }
}
```

Listing 2.14: POLICIES MATRIX GENERATOR

Chapter 3

Testing the Allocator algorithm

3.1 testing on a small data-set with one optimal policy

	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

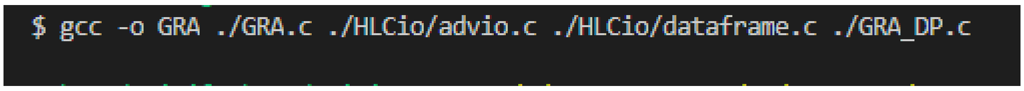
Table 3.1: DATASET 1

we'll consider first a small data-set which is given in **table** 3.1, let's look at the main code now which uses the previously viewed part of our project, the file we are interested in is **GRA_DP.c** represented in **figure** 3.1, .

```
void main(int argc, char *args[]){
    char *filename = args[1];
    FILE *file = open_file(filename, "r");
  DATAFRAME *r = csv_to_df(file);
    // display_df(r);
  df_retype(r, DF_ELEMENT_TInt);

    DATAFRAME *f = fit_to_find_multiple_optimal_paths(r);
    // display_df(f);

    GRA_RESULTS *results = gracreate();

    results->optimal_value = f->data[f->len_rows - 1][f->len_cols - 1].node
   .Arr->data[0].node.Arr->data[2].node.Int;
    results->policies = arrcreate(0);

    DF_ELEMENT policy = arrcreate(0);
    load_policies(f, f->len_rows, f->len_cols - 1, &policy, &results->
   policies);

    display_results(*results);

    arrfree(&results->policies);
    arrfree(&policy);
    df_free(f);
    df_free(r);
  fclose(file);
}
```

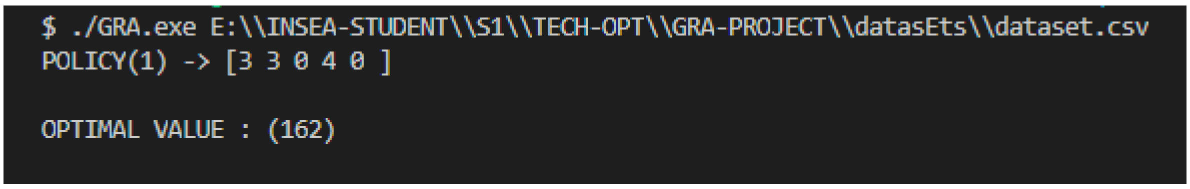
Listing 3.1: GRA_DP.c FILE (MAIN)

from the command line we compile our program by running the following command :



```
$ gcc -o GRA ./GRA.c ./HLCio/advio.c ./HLCio/dataframe.c ./GRA_DP.c
```

which uses gcc compiler to create object files and then the GRA executable file.
the last step is to execute the GRA.exe file which is then this way :



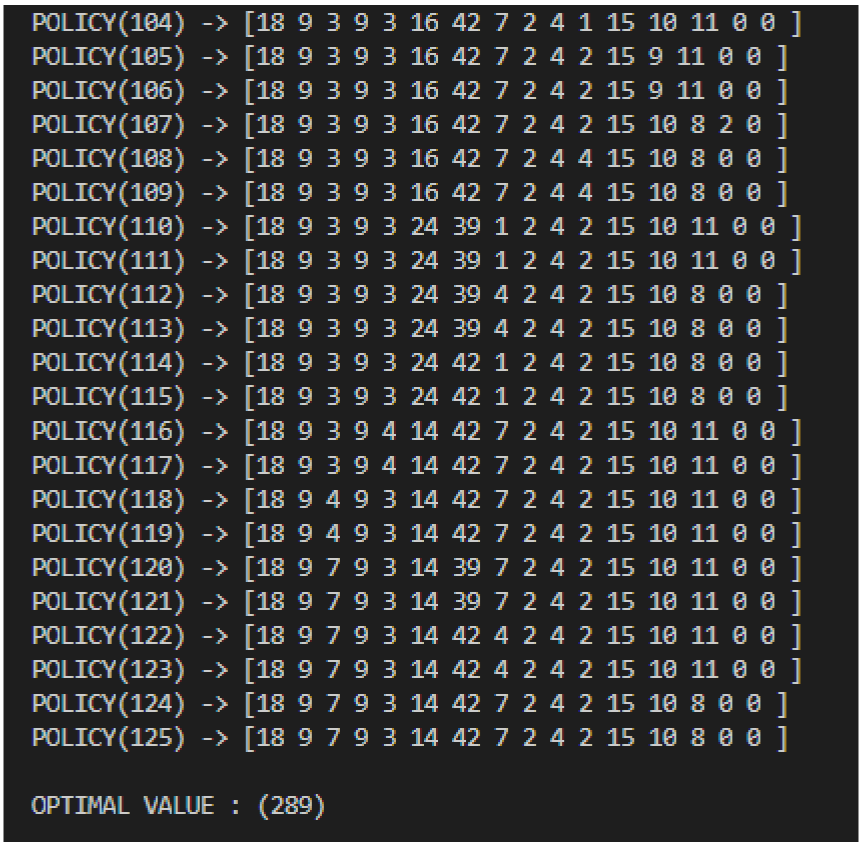
```
$ ./GRA.exe E:\\INSEA-STUDENT\\S1\\TECH-OPT\\GRA-PROJECT\\datasEts\\dataset.csv
POLICY(1) -> [3 3 0 4 0 ]

OPTIMAL VALUE : (162)
```

the argument passed is required to be the path to the csv file containing the required data-set R.

as we can see the optimal value is definitely **162** due to the exactness of our algorithm, and the respective policy (read from left to right starting by the first activity) which contains the allocated resources in each step.

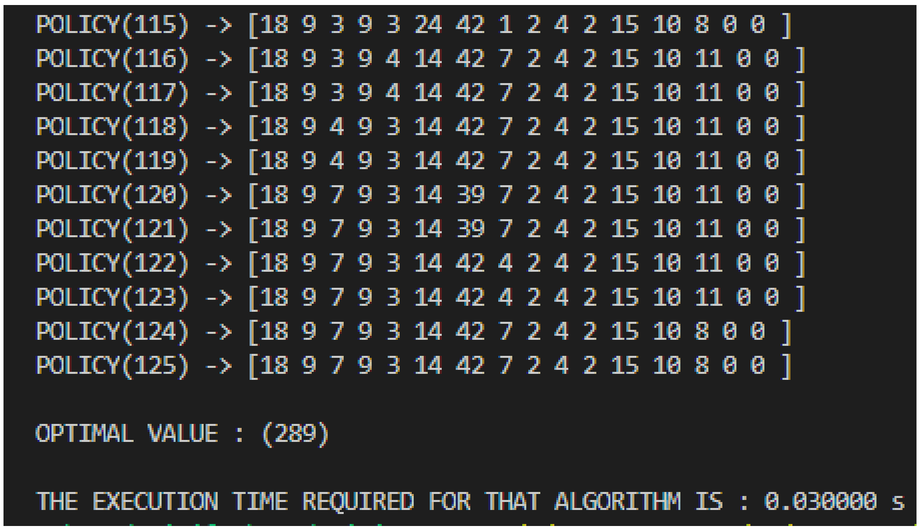
3.2 testing on a larger data-set with multiple optimal policies

unfortunately i can't show the dataset due to its size but it's enought to know its dimensions at least which are **16 rows and 101 columns**, after executing the same algorithm we get **289 as an optimal value** and **125 number of optimal policies**, **figure** 3.1 shows a sample of the found solutions.



```
POLICY(104) -> [18 9 3 9 3 16 42 7 2 4 1 15 10 11 0 0 ]
POLICY(105) -> [18 9 3 9 3 16 42 7 2 4 2 15 9 11 0 0 ]
POLICY(106) -> [18 9 3 9 3 16 42 7 2 4 2 15 9 11 0 0 ]
POLICY(107) -> [18 9 3 9 3 16 42 7 2 4 2 15 10 8 2 0 ]
POLICY(108) -> [18 9 3 9 3 16 42 7 2 4 4 15 10 8 0 0 ]
POLICY(109) -> [18 9 3 9 3 16 42 7 2 4 4 15 10 8 0 0 ]
POLICY(110) -> [18 9 3 9 3 24 39 1 2 4 2 15 10 11 0 0 ]
POLICY(111) -> [18 9 3 9 3 24 39 1 2 4 2 15 10 11 0 0 ]
POLICY(112) -> [18 9 3 9 3 24 39 4 2 4 2 15 10 8 0 0 ]
POLICY(113) -> [18 9 3 9 3 24 39 4 2 4 2 15 10 8 0 0 ]
POLICY(114) -> [18 9 3 9 3 24 42 1 2 4 2 15 10 8 0 0 ]
POLICY(115) -> [18 9 3 9 3 24 42 1 2 4 2 15 10 8 0 0 ]
POLICY(116) -> [18 9 3 9 4 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(117) -> [18 9 3 9 4 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(118) -> [18 9 4 9 3 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(119) -> [18 9 4 9 3 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(120) -> [18 9 7 9 3 14 39 7 2 4 2 15 10 11 0 0 ]
POLICY(121) -> [18 9 7 9 3 14 39 7 2 4 2 15 10 11 0 0 ]
POLICY(122) -> [18 9 7 9 3 14 42 4 2 4 2 15 10 11 0 0 ]
POLICY(123) -> [18 9 7 9 3 14 42 4 2 4 2 15 10 11 0 0 ]
POLICY(124) -> [18 9 7 9 3 14 42 7 2 4 2 15 10 8 0 0 ]
POLICY(125) -> [18 9 7 9 3 14 42 7 2 4 2 15 10 8 0 0 ]

OPTIMAL VALUE : (289)
```

Figure 3.1: SAMPLE OF THE SECOND DATASET RESULT



```
POLICY(115) -> [18 9 3 9 3 24 42 1 2 4 2 15 10 8 0 0 ]
POLICY(116) -> [18 9 3 9 4 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(117) -> [18 9 3 9 4 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(118) -> [18 9 4 9 3 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(119) -> [18 9 4 9 3 14 42 7 2 4 2 15 10 11 0 0 ]
POLICY(120) -> [18 9 7 9 3 14 39 7 2 4 2 15 10 11 0 0 ]
POLICY(121) -> [18 9 7 9 3 14 39 7 2 4 2 15 10 11 0 0 ]
POLICY(122) -> [18 9 7 9 3 14 42 4 2 4 2 15 10 11 0 0 ]
POLICY(123) -> [18 9 7 9 3 14 42 4 2 4 2 15 10 11 0 0 ]
POLICY(124) -> [18 9 7 9 3 14 42 7 2 4 2 15 10 8 0 0 ]
POLICY(125) -> [18 9 7 9 3 14 42 7 2 4 2 15 10 8 0 0 ]

OPTIMAL VALUE : (289)

THE EXECUTION TIME REQUIRED FOR THAT ALGORITHM IS : 0.030000 s
```

Figure 3.2: SAMPLE OF THE SECOND DATASET RESULT (WITH EXECUTION TIME)

the used data-set in here could be found in the **Github repo** given above with the following path **GRA/datasets/g_test.csv**, now let's check the execution time of that algorithm, executing the same script with some modifications to consider the timeline trajectory as well, you'll find the results in **figure 3.2**.

3.3 why not to use a brute-force method?

we have discussed a **dynamic programming approach** to solve our resources allocation problem but it was also possible for us to create a theoretical tree which-by we can consider every single possible combination, this type of techniques are called brute-force methods, a sample implementation of such algorithm is given in python in **figure 3.2**, we can't use such algorithms cause they are time consuming, inefficient and factorial time complexity kind **O(n!)**.

```python
def getVal(r, arr):
    s = 0
    for i,v in enumerate(arr):
        s += r[i][v]
    return s
    pass

mx = 0

def bruteforce(arr,N,M):
    global mx

    if sum(arr) > M:
        return

    v = getVal(r, arr)

    if v > mx:
        mx = v

    for i in range(N):
        arr_copy = arr.copy()
        arr_copy[i] += 1
        bruteforce(arr_copy, N, M)
        pass

    pass
```

Listing 3.2: GRA BRUTE FORCE ALGORITHM

due to the nature of our burteforce solution which is to consider every signle combination we are recursively iterating through the two given dimensions N and M, the exit condition is obviously to check if the allocated resources are larger than the total number of resources available to us or **M** then we just check if the current instance is the new max thus storing it is a must.

this program we'll consider all of the combinations and not only the where the sum is equal to **M**, we can easily change the exit condition but let's put it this way for the sake of generality, there is no need to calculate the execution time of this algorithm cause it's a common sense that it will take a life-time to process one tiny dataset let alone a sort of large one used in the previous section.

Conclusion

the comparison between dynamic programming and brute-forcing is far away from rationality, it's clear cut that the use of such techniques does make a difference even in small data-sets such as the ones treated in the previous chapter.

however, dynamic programming isn't the ideal technique for every optimization problem, the use of it consist of the way we view a solvable problem by which we consider smaller problems and then iteratively solve them where by each smaller problem is contributing to the resolution of bigger problems, the result of this debate is nothing but a reality, real-life problems are far away from getting captured by such algos this is way we use heuristic functions for most of the time to approximate our local solution to the global one by the use of a semi-random functions.

Bibliography

[1] Richard Bellman *RAND CORPORATION.* [*An Introduction to the Theory of Dynamic Programming*] USA, 1953.