Operational Decision Making Systems Project

10/01/2019

Mohamed	BOUKHEMKHAM	ISCE promo 2020

Table of contents

I. Introduction

The aim of this project is the optimization of the tools position in the loader of a numerical control machine. Indeed, performance of the machine depends of the tools position since the machine spends time to exchange tools. We create a software in C that takes in input a given range and gives in output the best tools order found according the simulated annealing method.

II. Work performed

Write here answers to the questions that have been posed in the guideline. You can divide this section into several subsections if need. Justify your choice.

Range Description

· *Determine input data and decision variables:*

Input data are the attributes of a range: the number of tools (**NB_OUTILS**), the number of places in the store (**NB_PLACES**), the number of range operations (**NB_OPERATIONS**) and the require time to move by one step a tool in the store (**NB_DELTA**). We have also the list of operations describe by the number of the tool and it machine duration. We store the list of tools in the array called **OUTIL** to know the order of tools and the associated machine duration in the array **TPS**.

```
typedef struct config
{
    int NB_OUTILS,NB_PLACES,NB_OPERATIONS,NB_DELTA;
    int *OUTIL, *TPS;
} Config;
```

We decided to put all these variables in a C structure called “config” to simplify the understanding of the code and the programming.

The decision variable will be mostly the **Acceptance probability** variable. This variable will calculate how good is a new solution compared to the old one. We will define it by :

I will explain later how we use it

$$\alpha = e^{\frac{c_{old} - c_{new}}{T}}$$

Modeling part

· *Calculate the total running time of a range for a given range? Since the sum of the machining times is constant, determine the objective function of this problem?*

The total running time is composed of the sum of the machine cost (sum of value in TPS) and the additional cost (when moving time of the tool is more than the machining time of the tool used before). Only the additional cost change according the initial position of the tools. Thus this cost is our objective function

We will first have to build a function that will calculate the cost of any range given and then process an algorithm that generates ranges and use the first function to calculate the cost of these ranges and find the best one.

Annealing programming part

· *From the coding of a solution, propose a method to evaluate the objective function.*

To evaluate the objective function, we have designed a function named **int calcul** that will allow us to calculate the cost to any given range.

In order to do so, we will use the informations from the structure Config that we fetched from the .txt files. The the array **mag** is the array where all the tools and empty places are in a circular way. **mag[0]** index is the position from which we can take and use the tool. So, we are looking for the needed tool from the array OUTIL. We proceed to check right and left for the needed tool and we move the range to the less far side (randomly if they are at the same distance).

Once it is done we permute the tools (or empty spot) calculate how much time we will lose and how is it is hidden. At the end we put back the last tool to an empty spot and we add the lost time together returning the total cost of the range.

· *Propose a neighborhood method that ensures the connectivity of the search space solutions.*

We have tried many ways to determine a neighbor solution of the range, first we tried to randomly take anything in the range and switch it by anything else picked randomly in the array. This method was really bad because we would end up with switching empty places between them.

Then i tried to pick only tools and switch them with everything, switch them with tools, but all these were not ideal.

The final method i decided to use to get a neighbor solution is picking randomly a tool in the range and switching position with the one before him or after him in the array randomly. This allow us to get a good neighbor solution with no useless empty spots and will also allow us to easily get back to the previous neighbor if the move is bad . This is in the function **int neighbor**

· *Propose a method allowing to obtain an initial solution.*

I have made three ways to obtain an initial solution :

The first is to order the tools in a numerical order in the arrays. This function is named **void mag_init,** there is no optimization in the way we order the tools.

For example in the first set of test given the initial solution will be : **1 2 3 4 5 0 0 0 …**

This method allow us to have pretty good results even tho the initial solution is not though about at all.

The second method I have made is to order the range in the order on which the operations will be made (without duplicating tools). This would mean that the initial solution will have less cost and it would maybe be possible to find a cost efficient solution in a shorter time. This is **void mag_init2**

For example in the first set of test given the initial solution will be : **4 5 2 3 1 0 0 0 …**

Although this method seems more optimized the algorithm tends to often go to bad branches of the possibilities and we often find a more optimized solution with the first method

The third method called **void mag_init3** is the same as the previous method except it does duplicate tools until we browsed all operations or if the range has not empty spots (i did not make a checking to see if all the tools are present in the range due to short time, but it would not be complicated and the given tests are not having this issue. I can deliver you a more complete version after)

For example in the first set of test given the initial solution will be : **4 5 2 3 1 4 2 1 0 0 …**

This method got us the best results with very very low costs found very quickly, duplicating tools even in an not so optimized way gives us much more efficiency.

· Program in language C the Simulated Annealing method that corresponds to this problem and adjust the parameters from sets of test

Apply this method to other sets of test and check the setting of the parameters.

We have used a fairly basic Simulated Annealing algorithm for this problem but is is very efficient and optimized.

The main function is called **int *anneal** and it is structured as a normal simulated annealing algorithm. We used obviously a temperature variable.

Now for the functioning of the annealing: We are first calculating the initial solution and keeping it as the base of our evaluations. Then we generate a neighbor solution and we check its cost. Now comes the use of the **Acceptance probability** we fixed. Acceptance probability is defining if we are going to accept a solution or not.

$$\alpha = e^{\frac{c_{old} - c_{new}}{T}}$$

I have modified the common AP formula to this one because I made tests and came to the conclusion that this formula is well adapted to our program.

This variable will give a number that will tell us if the new neighbor solution is better than the old one we kept. If the solution is the same, α will be 1, if it is better α will be greater than 1 .

But it the solution is worse than the previous accepted one, α will start to quickly decrease. Even if those solutions are not better, some of them can lead us to much better solution than the previous ones. Thus, we decided after many tests to accept solution in which α is greater than 0.1

When we accept a solution, we will continue to search for better ones and if we do not accept it, we will carry on with the last accepted one.

We have now to take count for the temperature decreasing throughout the work of the loop, this causes the acceptance rate to be lower and accept less not so worse new solutions. In order to still accept more solution and explore more branches, i have set a loop using an incremented variable and number of maximum loop that we can modify based on how complicated is our problem.

This variable allows us to test more solutions before decreasing the temperature.

To illustrate, for the first test given, we can set the loop of the increment variable to 5 and find the optimal solution quickly. This is because there are not many possibilities in the problem, and it is fairly simple.

For the second test, which is far more complicated, doing 5 loops only will lower the temperature too quickly and we will get a bad solution with a high cost. However, if we set this loop to execute 50 times, the algorithm has much more time to find a good solution and we usually end up with a very good solution. However, this increases the execution time.

Then we keep going through the algorithm keeping trace of the best solution and displaying it with the configuration.

Result part with comments

After finishing everything we proceeded to test in order to find the best configuration possible for each problem. These tests are pretty much explained in the previous part but i will show a bit more here.

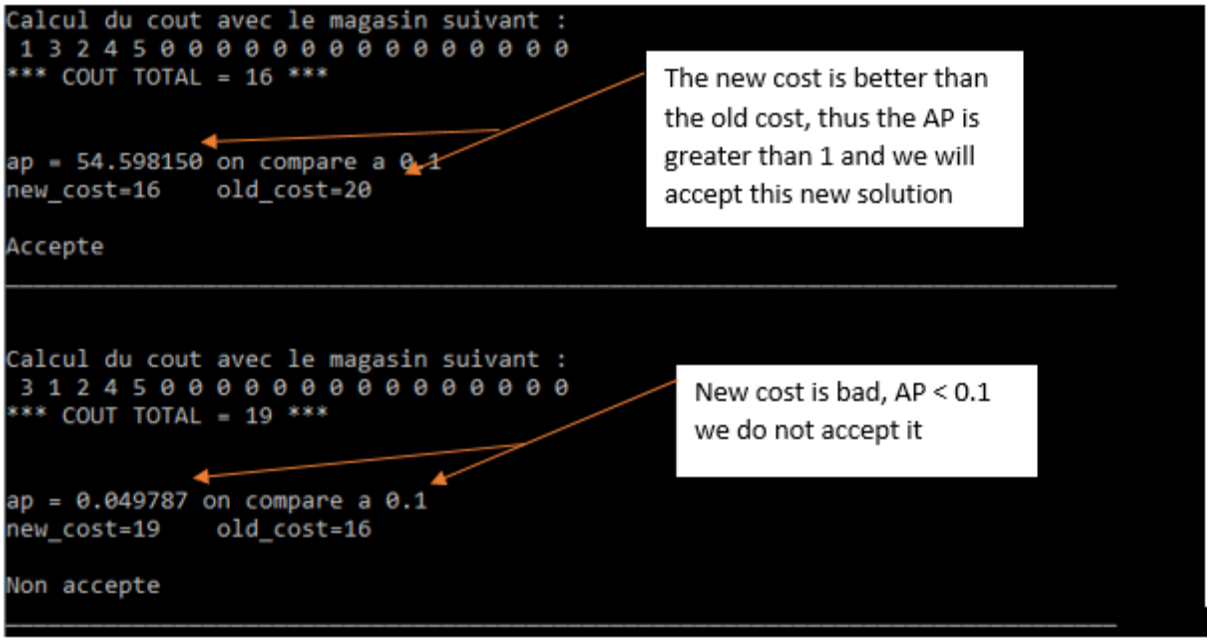
Here is a bit of the function **int calcul** it continues until the end of the wanted work.

```
Calcul du cout avec le magasin suivant :
 1 2 3 4 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 OUTIL 4
nous allons decaler de 3

nous utilisons actuellement l'outil 4
0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3
Le temps de mouvement est de 12 -> cout 12



OUTIL 5
nous allons decaler de 1

nous utilisons actuellement l'outil 5
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 0
Le temps de mouvement est de 4 et le temps d'usinage 10 -> temps masque
```

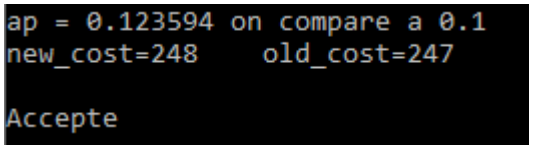
We end up with the total cost:

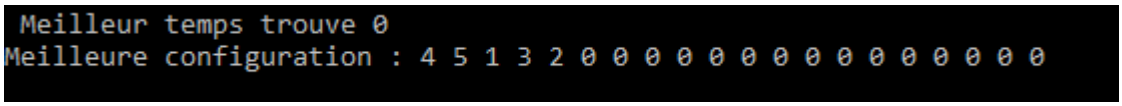
```
Nous allons nous positionner a un emplacement videnous allons decaler de 1

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 5 4 0 2
Le temps de mouvement est de 4 et le temps d'usinage 4 -> temps masque

*** COUT TOTAL = 20 ***
```

Now we will only show the total cost and apply the annealing



New cost is not better, but we accept it.





For the first test, we easily find a solution with 0 cost. We did set the loop before changing temperature to 5.

Now the second and more complicated test. Using 5 for incrementation we find 145 for best cost which is not really good.

Using 50, the program execution is longer but we find a cost of 15 which is way better.

```
******** FIN DE LA RECHERCHE ********


 Meilleur temps trouve 15
Meilleure configuration : 11 0 5 12 3 7 9 17 0 8 2 6 14 18 15 13 20 4 1 16 19 0 0 0 0 10 0 0 0 0
```

We have set the AP to 0.1 because it gives us the best results overall.

Using the **void mag_init** i talked about earlier initializing the initial range in numerical order, we get very good results overall.

Using the **void mag_init2** which sets the initial range in the order the tools will be used, we do not get optimal results as the annealing does not always find a good branch to go.

Using the **void mag_init3** which duplicate elements in the array we find extremely good results for example :

```
******** FIN DE LA RECHERCHE ********


 Meilleur temps trouve 0
Meilleure configuration : 11 5 6 14 13 6 17 12 11 10 4 1 14 19 16 20 2 8 15 8 9 3 18 9 5 12 7 7 15 13
```

We have found a optimal solution for the second test by duplicating elements.

To conclude we have noticed that we do not find always a good solution even using the best configuration because when the annealing is stuck in a certain solution it is too hard to come back to exploitable solution. But this is not a problem because executing the program from the beginning is a way to start over and get to another better solution. The program is not meant to be used only once, we can use it many times with many configurations to find the result we want.

III. Conclusion

The program is therefore perfectly functional, but the solution found will depend on certain parameters predefined by the user. Depending on the acceptance rate and the number of iterations allowed, the final cost found may be different. It is a matter of finding the right one in the middle between the cost that the program will be able to give and the calculation time that this one can take before finding a final solution.

The different methods for finding a neighbor and initializing the initial range are also important and must be considered. Such as duplication.

Finally, this project allowed us to approach a more adapted methodology facing system optimization issues. Despite difficulties encountered in developing the algorithm to find the best initial layout tools, we managed to answer the problem of the subject.

We were proud to produce an algorithm that can resolve complex problem. Indeed, it is interesting to think and produce ways to solve a problem in the best way possible. Moreover, it was interesting to try many ways to optimize the code in order to make it find a better solution at a faster speed for any given problem.

It was interesting to know the stake of the problem and understand it.

Also simulated annealing is a really good algorithm to understand how we can browse a great number of solutions within a reasonable time and allow us to comprehend the ways of making smart algorithms.

This project was rewarding and also pleasing in its way.