How to code it: A simple local search for the Quadratic Assignment Problem

Martin Josef Geiger*

May 15, 2023

Abstract

The article describes a straight-forward implementation of a simple local search for the Quadratic Assignment Problem. It's primary use is educative: The concepts put forward are kept simple on purpose. Note that the entire source code, written in C#, is available, also.

1 The Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) is a well-known optimization problem with numerous applications in manufacturing, planning, and many other areas.

It is best described by means of an application that stems from tactical production planning: The assignment of machines to locations within a production environment. In a more formal way, locations $S_i, i = 1, \ldots, n$ are given, to which machines $M_k, k = 1, \ldots, m$ must be assigned, hence creating an assignment/ solution in the sense of placing the production resources within a plant.

There are two aspects to consider: distances d_{ij} between the locations, and material flows f_{kl} among the machines (transported quantities), and both are typically given (data). As a common side constraint, each location may only accommodate a single machine, and each machine must be assigned to exactly one location. Moreover, it is assumed that $m = n$, therefore, we

*m.j.geiger@hsu-hh.de

consider $M_k, k = 1, \ldots, n$. Note that in practical cases, where $n > m$, we may ensure $m = n$ by introducing additional $n - m$ 'dummy'-machines, and extending the f_{kl}-matrix by adding values of 0 from each machine $M_k, k = 1, \ldots, m$ to the introduced 'dummy' machines.

The overall objective is to minimize the total costs of transportation, expressed as the distance-weighted total transport volume, subject to a feasible assignment. An assignment of a machine M_k to a location S_i is represented by introducing binary decision variables $x_{ik} \in \{0, 1\}$, and x_{ik} assumes 1 if and only if machine M_k is assigned to location S_i and 0 otherwise.

Hence, we obtain the following formal model.

$$\min \sum_i \sum_j \sum_k \sum_l d_{ij} f_{kl} x_{ik} x_{jl} \tag{1}$$

s. t.

$$\sum_i x_{ik} = 1 \quad \forall k \tag{2}$$

$$\sum_k x_{ik} = 1 \quad \forall i \tag{3}$$

$$x_{ik} \in \{0, 1\} \quad \forall i, k \tag{4}$$

The aforementioned mathematical problem is *quadratic* in the sense of the multiplication of the x_{ik}-variables. Note that there is an entire research area devoted towards linearizations of such problems, with considerable progress in past years. Further discussions on this matter are however omitted here.

2 Reflections on the solution representation

While the aforementioned mathematical model of the QAP makes use of binary decision variables, it is easy to see that, in a feasible assignment, most of the values of those variables will be of value 0. More precisely, n variables assume 1, while $n * n - n$ must be 0. It is therefore natural to think about a way of 'only' storing the *actual* assignments, not the 'not'-assignments. This can be achieved by introducing a permutation π: the permutation π, which is of length n, stores, at each position i, the index of the machine assigned to the location S_i of the given position i. As an example, $\pi = (3, 4, 1, 2, 5)$ represent an assignment of machine M_3 to location S_1, machine M_4 for location S_2,

machine M_1 to location S_3, machine M_2 to location 4, and machine M_5 to location S_5, while the corresponding matrix of x_{ik}-variables is as follows:

$$x_{ik} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \Leftrightarrow \quad \pi = (3, 4, 1, 2, 5) \tag{5}$$

On this basis, the objective function can be reformulated as $\min \sum_i \sum_j d_{ij} f_{\pi_i, \pi_j}$, with π_i representing the index of the machine at the *position* i in the permutation π. Note that by representing the assignments in this way, the complexity of the model/ problem is left unchanged, we merely are adding up only the *relevant* material flows of the *actual* machine-location-assignments.

3 Problem instances

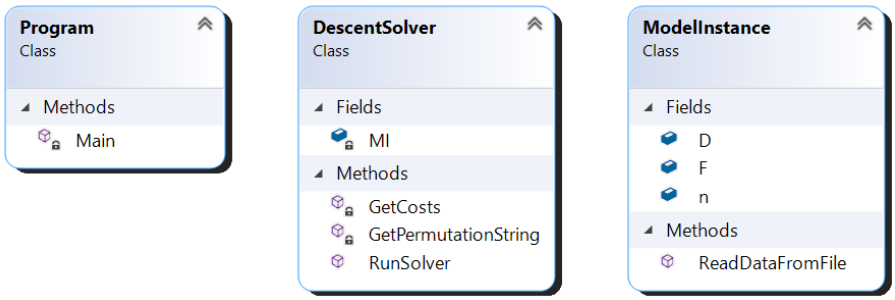
There is a good number of problem instances (data sets) available in the literature. For the purpose of this paper, we refer to the QAPLIB, see `https://www.opt.math.tugraz.at/qaplib/`.

Technical remark: In the QAPLIB, the distance-matrix is referred to as A, and the flow-matrix as B.

4 A simple descent algorithm

4.1 Classes

The program is kept deliberately simple. Starting from the default-function `Main`, a 'solver'-object, derived from the class `DescentSolver` is created, and the function `RunSolver` is executed. This function encapsulates the problem-solving logic, i. e., the actual descent algorithm.



The data on the other hand is kept in a separate object, see the class `ModelInstance`.

4.2 Reading the data

Reading the data from a file comes as a first natural step. Unfortunately, in case of the QAP and the data obtained from the QAPLIB, the formatting of some instances is somewhat all over the place. Line breaks are not consistent, there is a variable amount of blanks separating the numbers, and therefore, reading the data for the distance- and the flow-matrix is a bit of a challenge. Fortunately, this can be resolved as follows.

- First, we read the entire set of numbers into a long list.
- Second, we check if the data read from the file is reasonably consistent, i. e., if we can assume that we have read the file correctly.
- Finally, we assign the values to the matrices.

The following source code implements this concept.

```
1
2   class ModelInstance
3   {
4       public int n;
5       public int[,] D, F;
6
7       public void ReadDataFromFile(string file)
8       {
9           List<string> AllStrings = new List<string>();
10
11          System.IO.StreamReader sr = new
                System.IO.StreamReader(file);
```

```
12          while (!sr.EndOfStream)
13          {
14              string line = sr.ReadLine();
15              string[] splitrow = line.Split(new char[] { ' '
                    }, StringSplitOptions.RemoveEmptyEntries);
16              for (int i = 0; i < splitrow.Length; i++)
17              {
18                  AllStrings.Add(splitrow[i]);
19              }
20          }
21          sr.Dispose();
22
23          n = Convert.ToInt32(AllStrings.First());
24
25          if (AllStrings.Count != (1 + 2 * n * n))
26          {
27              Console.WriteLine("Error reading the file");
28              Environment.Exit(0);
29          }
30
31          D = new int[n, n];
32          F = new int[n, n];
33          int row = 0;
34          int column = 0;
35          for (int pos = 1; pos < (1 + 2 * n * n); pos++)
36          {
37              if (row < n)
38              {
39                  D[row, column] =
                        Convert.ToInt32(AllStrings[pos]);
40              }
41              else
42              {
43                  F[row - n, column] =
                        Convert.ToInt32(AllStrings[pos]);
44              }
45              column++;
46              if (column == n) { row++; column = 0; }
47          }
48      }
49  }
```

Objects derived from the class `ModelInstance` have a single function only: `ReadDataFromFile` for reading the data. The filename must be passed as a parameter to this function. Once the function is run, the actual values

of the instance are kept in the variable `n` and in the matrices `D` and `F`, both of which are of size $n \times n$. Contrary to the notation given in the QAPLIB, we use `D` for the distance-matrix, and `F` for the flow-matrix.

A `StreamReader`-object is employed to read the file. We read the file line by line, and separate the acquired strings in line 15 of the source code. Note that by using the split-parameter `StringSplitOptions.RemoveEmtyEntries`, unnecessary blanks are automatically removed. Therefore, we only read the actual data, and keep each value in a growing list of string variables, denoted as `AllStrings`.

Lines 25–29 perform a quick check: Either the correct number of values has been read (for an instance of size n, we must have $1 + 2 * n * n$ values), or not. In the latter case, the program terminates.

The subsequent lines update the `D`- and the `F`-matrices. This is done by two auxiliary variables, one the for column and one for the row in which the current number should be placed. As the distance-matrix precedes the flow-matrix in the files from the QAPLIB, the values are first allocated to `D`, and then, following with `row = n`, to `F`. See the condition in line 37 on this.

4.3 The descent algorithm

In a nutshell, the descent algorithm presented here can be described as follows.

Algorithm 1 Simple Descent

1: generate first assignment
2: **while** improvement possible **do**
3: swap the assignment of two machines if this improves the objective function value
4: **end while**

Starting with a first alternative, we try improving this current alternative by swapping (exchanging) the positions of exactly two machines. This procedure is run until no further swaps are possible, i. e., until no improvements are found on the basis of this move. In more detail, we describe the approach in the following Algorithm 2.

The source code presented below in the class `DescentSolver` implements this idea in a rather straight-forward way. There are, however, some differences to the pseudo-code. While above (and also in the mathematical

Algorithm 2 Simple Descent – more detailed

```
1: generate first assignment:
2: for i = 1 to n do
3:     assign machine M_i to location S_i
4: end for
5: declare boolean (flag) variable ImproventFound
6: repeat
7:     ImprovementFound ← false
8:     for i = 1 to n − 1 do
9:         for j = i + 1 to n do
10:            if swapping the machine at S_i and S_j improves the solution
   then
11:                make the swap
12:                ImprovementFound ← true
13:            end if
14:        end for
15:    end for
16: until ImprovementFound = false
```

model), we denote the indices of the locations and machines from 1 to n, the program allocates arrays with an index starting at 0. See line 59: We declare an array `CurrentAlternative` of n integer variables and subsequently assign the indices of the machines, starting with machine 0 at position 0 (the first) to machine $n-1$ at position $n-1$ (the last).

```
50  class DescentSolver
51  {
52      private ModelInstance MI;
53
54      public void RunSolver(string file)
55      {
56          MI = new ModelInstance();
57          MI.ReadDataFromFile(file);
58
59          int[] CurrentAlternative = new int[MI.n];
60          for (int i = 0; i < MI.n; i++)
61          {
62              CurrentAlternative[i] = i;
63          }
64          int CurrentCosts = GetCosts(CurrentAlternative);
```

```
65          Console.WriteLine(CurrentCosts.ToString());
66
67          bool improvement;
68          do
69          {
70              improvement = false;
71              for (int pos1 = 0; pos1 < MI.n - 1; pos1++)
72              {
73                  for (int pos2 = pos1 + 1; pos2 < MI.n; pos2++)
74                  {
75                      int machine_at_pos1 =
                            CurrentAlternative[pos1];
76                      int machine_at_pos2 =
                            CurrentAlternative[pos2];
77                  CurrentAlternative[pos1] = machine_at_pos2;
78                  CurrentAlternative[pos2] = machine_at_pos1;
79                  int NewCosts =
                        GetCosts(CurrentAlternative);
80                  if (NewCosts < CurrentCosts)
81                  {
82                      improvement = true;
83                      CurrentCosts = NewCosts;
84                      Console.WriteLine(CurrentCosts.ToString());
85                  }
86                  else
87                  {
88                      CurrentAlternative[pos1] =
                            machine_at_pos1;
89                      CurrentAlternative[pos2] =
                            machine_at_pos2;
90                  }
91              }
92          }
93          }
94          while (improvement);
95          Console.WriteLine(GetPermutationString(CurrentAlternative));
96      }
97
98
99      private int GetCosts(int[] permut)
100     {
101         int c = 0;
102         for (int i = 0; i < permut.Length; i++)
103         {
104             for (int j = 0; j < permut.Length; j++)
```

```
105             {
106                 int dst = MI.D[i, j];
107                 int machine_at_i = permut[i];
108                 int machine_at_j = permut[j];
109                 int trnsprtvol = MI.F[machine_at_i,
                        machine_at_j];
110                 c += dst * trnsprtvol;
111             }
112         }
113         return c;
114     }
115
116
117     private string GetPermutationString(int[] permut)
118     {
119         string ps = "(";
120         for (int pos = 0; pos < permut.Length - 1; pos++)
121         {
122             ps += (permut[pos] + 1).ToString() + ",";
123         }
124         ps += (permut.Last() + 1).ToString() + ")";
125         return ps;
126     }
127 }
```

Evaluating the current permutation is done by means of function `GetCosts`, see the code from line 99. This function evaluates the permutation passed to it (see the parameter `int[] permut`) from scratch. It is first called after creating the initial assignment of machines to locations, and then whenever two machines swap their spots. The return values are kept in variables such as `CurrentCosts` and `NewCosts`, respectively.

The functionality provided by the function `GetPermutationString`, see the source code from line 117, is rather obvious. It performs a string concatenation and returns the assignment in a more readable form, i. e., a permutation of the machine indices (by incrementing each internal number by +1).

4.4 Reflections on the algorithm and the implementation

The implementation above comes with some design choices that we should reflect upon.

- The creation of the initial alternative is somewhat simplistic. Clearly, this procedure could be improved, e. g. by sorting the machines and the locations. Besides, a randomization of this procedure could be considered as otherwise, the algorithm always starts with the same assignment.

- While the move – an exchange of the assignment of two machines – is standard, it's evaluation is less so. In the current implementation, we make the move (see lines 75–76) before evaluating it. Ion case the modification gets reject, it is undone, see lines 88–89. Contrary to that, it is more common to first evaluate a move and then, in case of acceptance, modify the current alternative. However, the current implementation is only able to *fully* evaluate an alternative, and in order to keep this (simple approach), the implementation is as such.

- Evaluating alternatives only from scratch is costly: $\mathcal{O}(n^2)$. While we are aware of faster implementations, those come with more complex implementations – which we here avoid on purpose.