# A STAGNATION AWARE COOPERATIVE BREAKOUT LOCAL SEARCH ALGORITHM FOR THE QUADRATIC ASSIGNMENT PROBLEM ON A MULTI-CORE ARCHITECTURE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YAĞMUR AKSAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2016

Approval of the thesis:

**A STAGNATION AWARE COOPERATIVE BREAKOUT LOCAL SEARCH ALGORITHM FOR THE QUADRATIC ASSIGNMENT PROBLEM ON A MULTI-CORE ARCHITECTURE**

submitted by **YAĞMUR AKSAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

# ABSTRACT

A STAGNATION AWARE COOPERATIVE BREAKOUT LOCAL SEARCH ALGORITHM FOR THE QUADRATIC ASSIGNMENT PROBLEM ON A MULTI-CORE ARCHITECTURE

Aksan, Yağmur
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Ahmet Coşar
Co-Supervisor : Asst. Prof. Dr. Tansel Dökeroğlu

June 2016, 83 pages

The quadratic assignment problem (QAP) is one of the most challenging NP-Hard combinatorial optimization problems with its several real life applications. Layout design, scheduling, and assigning gates to planes at an airport are some of the interesting applications of the QAP. In this thesis, we improve the talents of a recent local search heuristic Breakout Local Search Algorithm (BLS) by using adapted Levenshtein Distance metric for similarity checking of the previously explored permutations of the QAP problem instances. In addition to this, the proposed algorithm, BLS-OpenMP-QAP (Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem), makes use of the parallel computation paradigm of the contemporary multi-core architectures using OpenMP programming paradigm. The stagnation-aware search for the (near-)optimal solution of the QAP is executed concurrently on several cores with diversified trajectories while considering the similarity of the already detected local optima. The exploration of the search space is improved by selecting the starting points intelligently and speeding-up the fitness evaluations as many as the number of the processors/threads. The BLS-OpenMP-QAP algorithm is executed on 59 problem instances of the QAP library benchmark and the results shows that it is able to solve 57 of the instances exactly. The overall deviation for the algorithm is obtained as 0.019% on the average; therefore, it can be reported to be among the best algorithms in the literature.

Keywords: Quadratic Assignment Problem, BLS-OpenMP-QAP, BLS, OpenMP, Optimization

# ÖZ

ÇOK ÇEKİRDEKLİ BİR MİMARİ ÜZERİNDE KARESEL ATAMA PROBLEMİ İÇİN İŞ BİRLİĞİ YAPAN DURGUNLUK BİLİNÇLİ YEREL ARAMA KAÇIŞ ALGORİTMASI

Aksan, Yağmur
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ahmet Coşar
Ortak Tez Yöneticisi : Asst. Prof. Dr. Tansel Dökeroğlu

Haziran 2016 , 83 sayfa

Karesel Atama Problemi (KAP) çeşitli gerçek hayat uygulamalarıyla birlikte en zorlu çözümü polinom zamanlı olmayan (NP) kombinatoriyal en iyileme problemlerinden biridir. Yerleşim tasarımı, zaman planlaması ve havaalanındaki uçaklara kapıların atanması KAP'ın ilgi çekici uygulamalarından bazılarıdır. Bu tezde, KAP problem örneklerinin daha önce aranan permütasyonlarının benzerlik kontrolü için uyarlanmış Levenshtein uzaklık ölçüsünü kullanarak yeni bir yerel arama sezgiseli olan yerel arama kaçış (BLS) algoritmasının yeteklerini geliştirdik. Buna ek olarak, önerilen algoritma, BLS-OpenMP-QAP(Karesel Atama Problemi için Açık Çoklu-İşleme ile Yerel Arama Kaçış Algoritması), OpenMP proglama yaklaşımını kullanan güncel çok çekirdekli mimarilerin eş zamanlı hesaplama yaklaşımını kullanır. KAP'ın en iyi çözümü için durgunluk bilinçli arama çeşitli yörüngeleriyle birtakım çekirdekler üzerinde daha önce tespit edilen yerel en iyinin benzerliği göz önüne alınarak eş zamanlı olarak çalıştırılır. Arama alanının keşfi başlangıç noktalarını akıllıca seçerek ve işlemcilerin/iş parçacıklarının sayısı kadar uygunluk değerlendirmelerini hızlandırarak geliştirilir. BLS-OpenMp-QAP algoritması KAP kütüphanesi setindeki 59 problem örneği üzerinde çalıştırıldı ve sonuçlar örneklerden 57 tanesini tam olarak çözebildiğini gösterdi. Algoritma için ortalamada toplam sapma 0.019% olarak elde edildi, böylece literatürdeki en iyi algoritmalar arasında olduğu bildirilebilir.

Anahtar Kelimeler: Karesel Atama Problemi, BLS-OpenMP-QAP, BLS, OpenMP, En iyileme

To my family and people who are reading this page

# ACKNOWLEDGMENTS

Foremost, I would like to express my deepest gratitude to my supervisor Prof. Dr. Ahmet Coşar for his continuous support of my Master study and research, for his patience, motivation, enthusiasm and immense knowledge.

I also would like to express my special thanks to Asst. Prof. Dr. Tansel Dökeroğlu for providing encouragement for this study, his valuable insight, expertise, creative and comprehensive advise during this research. Their guidance helped me in all the time of research and writing of this thesis. The door to their offices was always open whenever I ran into a trouble spot or had a question about my research or writing.

I also would like to express my special thanks to my colleagues at TÜBİTAK BİLGEM YTE for their help, encouragement and support during my graduate study.

I am also grateful to my friends for their patience, continuous encouragement and support throughout my Master study.

Finally, I would like to express my very profound gratitude to all members of my family, especially my brother Çağrı Aksan, for their love, understanding, patience and providing me with unfailing support and continuous encouragement throughout my life and through the process of researching and writing this thesis.

# TABLE OF CONTENTS

# LIST OF TABLES

TABLES

# LIST OF FIGURES

FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ACO/GA/LS | An Ant Colony Optimization/Genetic Algorithm/Local Search Hybrid |
| APD | Average Percentage Deviation |
| API | Application Program Interface |
| BKS | Best Known Solution |
| BLS | Breakout Local Search Algorithm |
| BLS-OpenMP-QAP | Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem |
| BMA | Population-based Memetic Algorithm |
| CPTS | A Cooperative Parallel Tabu Search Algorithm |
| CPU | Central Processing Unit |
| CUDA | Compute Unified Device Architecture |
| FANT | Fast Ant System |
| GA/C-TS | A Genetic Algorithm Hybrid with Concentric Tabu Search Operator |
| GA/SD | A Genetic Algorithm Hybrid with a Strict Descent Operator |
| GDA | Great Deluge Algorithm |
| ITS | Iterated Tabu Search |
| JRG-DivTS | A Multi-start Tabu Search |
| LD | Levenshtein Distance |
| MA | Memetic Algorithm |
| OpenMP | Open Multi-Processing |
| PHA | A Parallel Hybrid Algorithm |
| PMTS | Parallel Multistart Tabu Search |
| QAP | Quadratic Assignment Problem |
| QAPLIB | Quadratic Assignment Problem Library |
| SC-TABU | Self Controlling Tabu Search |
| TS | Tabu Search |

# CHAPTER 1

# INTRODUCTION

Recently, one of the most popular academic topics all around the world is to find out the most appropriate solution to combinatorial optimization problem that can be formulated as a Quadratic Assignment Problem (QAP). QAP is a well known and well studied challenging NP-Hard combinatorial industrial engineering problem which is used in order to locate $N$ facilities to $M$ different locations under the circumstances of $N \leq M$. Furthermore, facility allocation problem is noteworthy for its capability to formulate great deal of real life problems in various fields. As a result, many heuristic approaches have been widely used to deal with this significant problem near-optimally. Namely, among these several heuristics, local search algorithms, such as Tabu Search [1], Iterated Local Search [2] and Simulated Annealing [3] are the most favourite ones in the literature. In spite of the fact that the problems whose size is less than 30 can be solved with these state-of-the-art algorithms optimally, problems with greater size cannot be solved exactly in acceptable computing time as a consequence of their intractable structure.

In this study, we recommend a BLS-OpenMP-QAP (Breakout Local Search Algorithm with Open Multi-Processing for QAP) algorithm in an attempt to solve the QAP. In BLS-OpenMP-QAP, we principally make use of the effective local search algorithm which is introduced by Benlic et al as Breakout Local Search Algorithm (BLS) [2]. BLS basically uses a steepest descent procedure to find the local optima. According to the number of jump magnitude, perturbation moves, which is determined depending on the present situation of the neighbouring exploration, BLS selects the perturbation method to apply among three separate types including tabu search, recency-based and random perturbation. In addition to the BLS, we introduce a smart beginning mechanism for the generating population section which guarantees to produce diversified candidate solutions for BLS. With the help of the perturbation and the restart mechanism, it consistently escape from one local optimum to another one in the exploration area so that the stagnation situation is prevented. In fact, most of the state-of-the-art algorithms that have been proposed to solve the QAP are executed on a single processor and do not make use of the high performance opportunities that the recent parallel computation technologies provide us. In the light of these opportunities, we make use of the parallel programming with OpenMP (Open Multi-Processing) in an attempt to increase the performance of BLS-OpenMP-QAP algorithm. While applying OpenMP for parallel programming, the most significant thing is to ensure the most efficient usage of threads for implementing the algorithm. That is to say, except for the generating initial solutions part, other processes (steepest descent procedure, determining the jump magnitude and perturbation section) of the BLS-OpenMP-QAP algorithm are executed on threads by organizing a parallel computation using the abstractions of threads in order to take full advantage of parallel programming. In this way, the most time consuming sections of the BLS-OpenMP-QAP algorithm are executed on the parallel threads owing to the OpenMP library. As for the candidate solutions to be generated, they are produced previously on the CPU environment; therefore, each thread has a different candidate solution from the other threads.

As we present in Chapter 4, in an effort to measure the performance of the proposed algorithm, BLS-OpenMP-QAP, and also to evaluate the quality of resulting optimal solutions, BLS-OpenMP-QAP algorithm is evaluated utilising a set of benchmark instances obtained from QAPLIB [4]. As a consequence of these experiments, BLS-OpenMP-QAP algorithm shows a promising speed up.

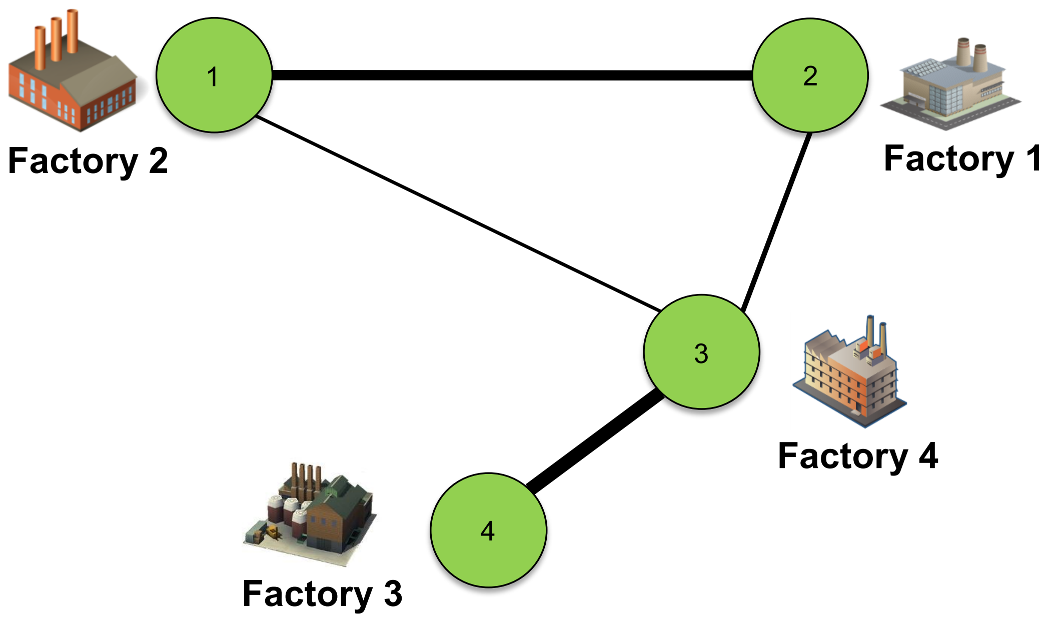


Figure 1.1: Representation of Quadratic Assignment Problem

## 1.1 Quadratic Assignment Problem (QAP)

The quadratic assignment problem (QAP) is a well-known challenging NP-Hard combinatorial optimization problem which was proposed by Koopmans and Beckmann in 1957 [5] on the purpose of modelling the position of indivisible economic activities. In spite of the several academic efforts from its initial formulation to date, it is still one of the hardest combinatorial optimization problems.

The purpose of the QAP is to locate a number of different facilities to the different locations in such a way which minimizes the total cost of the placement as possible. That is to say, it seeks to locate $N$ facilities to $N$ fixed locations in the most cost-effective way.

To shed light on the QAP, the factory assignment problem along with four factories and four locations is considered as an illustration and one possible assignment among the several permutations of the problem size 4 is represented in Figure 1.1 [6]. In this representation factory 2 is placed to location 1, factory 1 is placed to location 2, factory 4 is placed to location 3, and finally factory 3 is placed to location 4. In other words, this allocation of the factories can be presented as the permutation $p = \{2, 1, 4, 3\}$. Furthermore, in the figure, the connection link between a pair of factories indicates that there is a compulsory flow between these two factories, and the bolder the link is, the more flows there are between the factories.

Table1.1: Flows between factories for QAP example

| factory $i$ | factory $j$ | flow $(i, j)$ |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 4 | 2 |
| 2 | 4 | 1 |
| 3 | 4 | 4 |

Table1.2: Distances between locations for QAP example

| location $i$ | location $j$ | distance $(i, j)$ |
|---|---|---|
| 1 | 3 | 53 |
| 2 | 1 | 22 |
| 2 | 3 | 40 |
| 3 | 4 | 55 |

In order to compute the allocation cost of the permutation, the information about the flows between factories and the distances between locations are required. In Table 1.1 and 1.2, flows and distances matrices of the facility location problem are presented. As a matter of fact, the cost of the placement for a pair of factories is a product of the flow between the factories and the distance between the locations of the factories. In other words, it is the sum of the flow between a pair of factories multiplied by the distance between their assigned locations. Therefore, the placement cost of the permutation in the example can be calculated as :

$$\begin{gathered} f(1,2)d(2,1) + f(1,4)d(2,3) + f(2,4)d(1,3) + f(3,4)d(3,4) \\ = 3 * 22 + 2 * 40 + 1 * 53 + 4 * 55 = 419 \end{gathered}$$

For the QAP, the most fundamental thing is to locate facilities to locations in the most economical way. However, the permutation illustrated above is not the optimal solution.

As for the computational complexity of the QAP, it can be formulated by using three $nxn$ matrices including A, B, and C [7].

$$\mathrm{A} = (a_{ik})$$

where $a_{ik}$ stands for the flow value from facility $i$ to facility $k$.

$$\mathrm{B} = (b_{jl})$$

where $b_{jl}$ stands for the distance value from location $j$ to location $l$.

$$\mathrm{C} = (c_{ij})$$

where $c_{ij}$ stands for the cost value of assigning facility $i$ to the location $j$.

The Koopmans-Beckmann formulation of the QAP is presented as :

$$min_{\phi \in S_n}(\sum_{i=1}^{n}\sum_{k=1}^{n} a_{ik}b_{\phi(i)\phi(k)} + \sum_{i=1}^{n} c_{i\phi(i)})$$

where $S_n$ stands for the permutation of integers $1, 2, ..., n$ chosen from the set of all possible placements for $n$. The range of the indexes $i, j, k, l$ is $1, ..., n$ and $n$ is the problem size. Each individual product $a_{ik}b_{\phi(i)\phi(k)}$ indicates the cost of the transportation from facility $i$ at location $\phi(i)$ to facility $k$ at location $\phi(k)$. Moreover, $c_{i\phi(i)}$ represents the total cost for locating facility $i$ at location $i$ and the sum of the transportation costs to all other facilities $k$ assigned at locations $\phi(1), \phi(2), ..., \phi(n)$. As a consequence, the main purpose is to seek the most optimal solution $S_*$ that minimises the cost function.

From a theoretical point of view, the QAP belongs to the NP-Hard class of problems which means there is no accurate algorithm that is able to solve the problem in polynomial time. That is to say, the problems whose size are greater than 30 cannot be solved in rational time since it is practical to compute the accurate permutation only for comparatively small instances whose size is up to 30. However, existing approaches for seeking the optimal solution are costly and can be difficult for even problem sets with the size of $n = 30$. In addition, this hardness encourages the heuristics for the development of new near-optimal solution techniques for QAP. Indeed, scientists pursue a research on solving this problem and propose several heuristic algorithms — Simulated Annealing [3][8][9], Tabu Search [1][10][11][12][13][14][15], Ant Colony Optimisation [16][17][18][19], Genetic Algorithms [20] and Memetic Algorithms [21][22][23], Scatter Search [24], Iterated Local Search [25], to name a few.

The main reason why QAP is such an outstanding problem in scientific world and so it attracts a great deal of attention of scientists is that it can formulate a number of practical real life problems in various areas such as Steinberg wiring problem [26], dartboard design [27], keyboard layout [28], hospital layout [29], planning university campus [30], data visualisation [31][21], DNA micro array layout [32] and so on.

## 1.2 Switching to the Parallel Computing

Traditionally, applications use Central Processing Unit (CPU) for primary calculations over the course of many years. Nevertheless, with the increased necessity in improving the performance of computing and the intensive demand in solving a problem more quickly, a number of software developers, scientists and researchers conduct a survey on finding alternative ways to accelerate the speed of calculations. Thereby, the parallel processing techniques are taken into more consideration by them in various areas.

At the very beginning of an innovation of the parallel programming, it was seemed as an exotic pursuit and commonly categorised as a speciality within the area of computer science. Afterwards, this perception has altered thoroughly and it is understood that the parallel programming has an increasingly important role in the computer science. That is to say, from being limited to a particular people, the computing world has turned into the area where almost entire programmers are eager to be trained and practise in parallel programming so as to become outstandingly powerful in computer science.

As a matter of fact, much has been made for switching to the simultaneous computing in the computer industry widespread. Therefore, more and more developers need to deal with a great diversity of parallel computing platforms and technologies in order to provide sophisticated users with novel experiences. Nowadays, instead of command prompts, multi threaded graphical interfaces are in demand. Moreover, cellular phones that are used only for making calls are unfavourable and instead people prefer versatile phones like simultaneously browsing the Web, playing music, and providing Global Positioning System (GPS) services.

In recent years, because of the fact that using multiple processors in parallel means solving problems more rapidly than with a single processor, parallel programming is preferred in several fields. For this purpose, some parallel machines are come out. To give an example, a Cluster Computer which contains multiple personal computers combined together with a high speed network; a Shared Memory Multiprocessor (SMP) by connecting multiple processors to a single memory system; a Chip Multi-Processor (CMP) containing multiple processors on a single chip.

For a long time, CPU manufacturers make an effort for increasing the clock speed of the processors as one of the significant methods applied to in an effort to enhance the performance of consumer computing devices. In the early 1980s, CPUs have clock speeds around 1 MHz whereas current desktop processors ran with internal clocks operating between 1GHz and 4 GHz, approximately 1000 times quicker than the first personal computer.

Recently, however, the attempt of improving the clock speeds of the processors by taking additional power from existing architectures is not applicable any more on the occasion of several fundamental limitations in the production of integrated circuits. As a consequence, scientists and manufacturers have begun to look for alternatives for the improvement of the performance of consumer computing devices.

In the exploration for alternative methods to obtain massive performance gains for CPUs, the idea of placing more than one processing core in a personal computer rather than concentrating solely on enhancing the performance of a single processor is brought about. In 2005, CPU manufacturers introduce the processors with dual-core instead of one. In the later years, they keep developing multi-core processors like 3-, 4-, 6-, 8- and 16-core central processor unit. In the year 2010, almost all consumer computers are shifted to the architecture with multi-core central processors. This trend is referred to as multi-core revolution in the computing world. With multi-core revolution, due to its efficiency and better quality with higher computing power, the parallel computation gain notable popularity. For instance, electronic devices such as mobile phones and portable music players also proceed to make use of simultaneously computing capabilities so as to offer preferably better functionality.

## 1.3 Open Multi-Processing (OpenMP)

Apart from the evolution of central processors in clock speeds, increase of core number also experienced the huge revolution. In the necessity of more and more computational power, researchers conduct a research on taking advantages of the multi-processors. Nowadays, all new computers are parallel multi-core computers and we use multi-core phones, laptops and desktops to meet our personal needs. More and more scientist use multi-node clusters and supercomputers for their researches. As a consequence of multi-core revolution in the computing world, the new increasing interest in parallel programming among threads by using CPU cores is brought about. Therefore, a new industry standard API (Application Program Interface) of C/C++ and Fortran, OpenMP, is developed with the aim to serve as a worthy interface for the development of parallel programs on shared memory. The OpenMP specification is introduced by the OpenMP Architecture Review Board, which is a join of the companies actively participate in the development of the standard shared-memory programming interface. The consequence of a large agreement between hardware vendors and compiler developers, OpenMP is brought about and considered to be an industry standard. That is to say, it defines a group of compiler instructions, library routines, and environment variables.

OpenMP is used in order to realise parallel programming for multi-core machines. To be more precisely, OpenMP is API for shared-memory parallel programming and works on one multi-core computer. It principally provides a portable and scalable model for developers of shared memory parallel applications. In shared memory model, the globally shared memory can be accessed by all threads. As a matter of fact, we can define the variables to be shared or private. If we state the data as shared, it can be accessed by all the threads. Hence, threads read from and write on shared variables. There is no need for explicit communications via messages. On the other hand, private variables can be reached only by the threads in which it is defined; therefore, threads use their own private variables to do work that does not need to be globally visible outside the parallel region. This model is represented in Figure 1.2.

Parallel computing is about data processing. In practice, memory models determine how we write parallel programs. Data corruption is possible when multiple threads attempt to update the same memory location. The most significant thing to pay high attention is that synchronisation must be used in order to protect data against race conditions.

As for the execution model of OpenMP, it follows the Fork and Join Model. That is to say, OpenMP programs are initialised with a single thread, the master thread (Thread 0). At the beginning, master thread creates a number of parallel worker threads. This process is named as FORK. Instructions in parallel blocks are executed in parallel by every thread. At the end, all threads synchronize and join master thread and this process is named as JOIN. This execution model is represented in Figure 1.3.

OpenMP provides outstanding acceleration in computing performance by benefiting from the power of the computer's cores. What is more, it is well-known for executing parallel and compute intensive tasks.

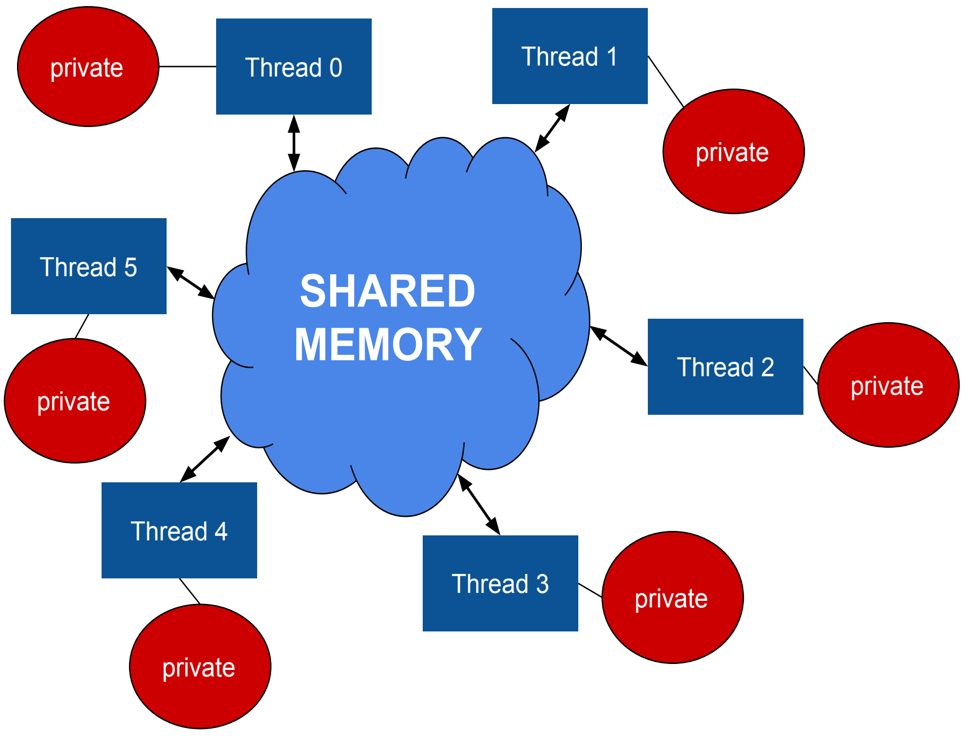


Figure 1.2: Shared Memory Model

As a matter of fact, OpenMP must be well studied in order to program threads for executing parallel computing tasks. By learning OpenMP, high performance applications can be written. Especially for the large application, most of the code can be executed on parallel regions by threads. By the time extremely computationally intense is occurred, the program can simply call the OpenMP directive. Namely, the main idea of OpenMP is that it should be used for the most computationally intense portions of the program in order to take full advantage of parallel programming. OpenMP architecture provides executing the same instructions in multiple threads simultaneously. While CPU is designed for very complicated control logic which executes the sequential programs in optimized way, OpenMP has a simpler control logic that is optimized for the execution of parallel tasks using the abstractions of threads. Consequently, we make use of the heterogeneous computing techniques in our algorithm so as to make use of OpenMP in the most effective ways.

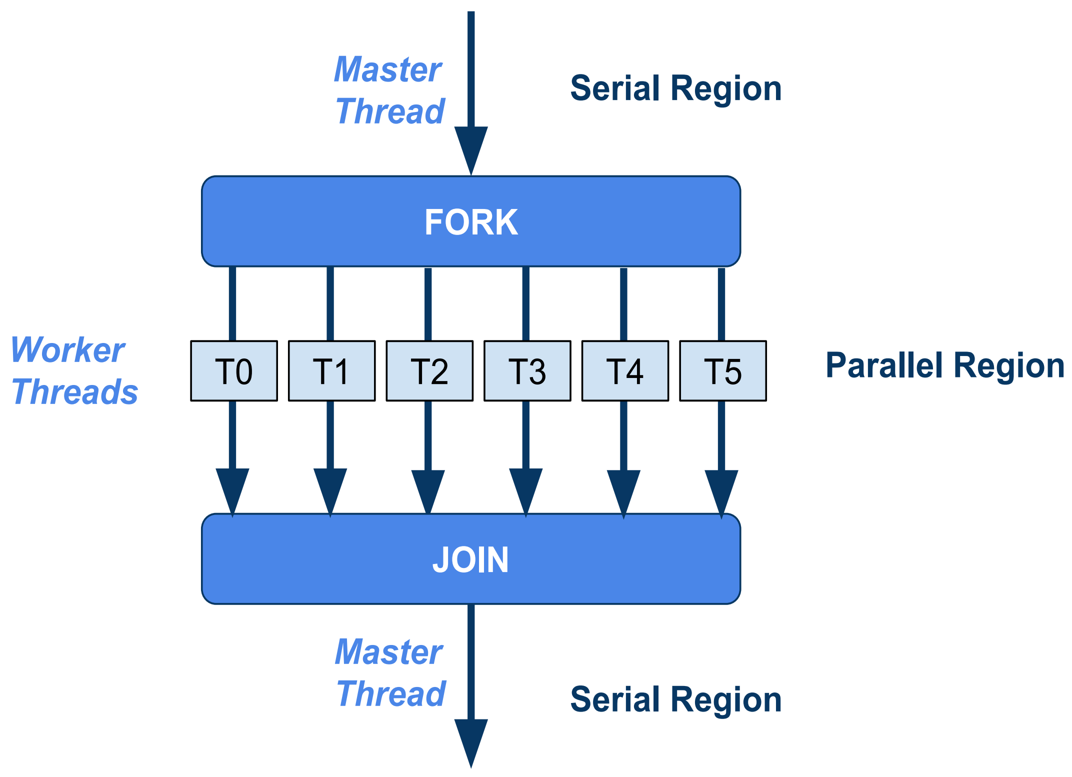


Figure 1.3: The OpenMP Execution Model

As for the heterogeneous computing architecture, the workload is distributed properly to the serial region or parallel region according to the intensity of the computation in this design pattern. Initially, program executes on serial region and then intense procedures are performed on parallel region by threads. Finally, the solutions are presented in serial region. The most significant thing we do is that we run multiple threads simultaneously; therefore we get excessive speed-up.

As for the threads, they are determined while launching the parallel processing since the dynamic change in the number of threads is allowed just before a parallel region is entered. In order to initiate threads, execution parameters for the size of threads must be given inside the parenthesis in the directive clause as the following :

$$\#pragma\ \ omp\ \ parallel\ \ num\_threads(<< numberOfThreads >>);$$

By the configuration of the execution parameters such as the number of threads to be initialised and the data access type to be shared or private, how the OpenMP is going to launch the threads is determined. As for the performance of the algorithm and protecting data against the race conditions, these parameters are very significant. As a consequence, we adjust them in our algorithm so as to extract the massive performance gains.

## 1.4 A Simple OpenMP Example

In this section, we would like to give a simple OpenMP example to present the massive computational power of the multi-core architectures. This example is named as inner product or dot product. It makes a computation for a simple dot product across two arrays, $A$ and $B$. That is to say, it multiplies the corresponding elements of these two vectors and finally calculates the sum of the products. The following equation shows the mathematical formulation of the simple dot product :

$$(x_1, x_2, x_3, x_4) \text{ x } (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

There are several useful OpenMP directives which we need to ensure to be declared before the compilation of the code and they are located in *omp.h*. Therefore, we include it in the header in order to benefit from the functions of OpenMP library. The next step of the dot product is to create some OpenMP threads and allocate values to the arrays $A$ and $B$. Afterwards, we separate the computation of dot product into two for loops. The first one deals with computing the array product $A[i] * B[i]$ for all i values and the second for loop calculates the sum of all products. The code of the example is presented below with the demonstration of the OpenMP parallel for and OpenMP reduction compiler directives.

The arrays $A$, $B$, $C$ and the variable *sum* are defined as *shared* in the directive. That is to say, it is simultaneously accessible by all threads. On the other hand, $i$ and $tId$ are specified as *private*. As a matter of fact, by default, all variables in the parallel area are defined as shared except the loop iteration counter or thread number.

```c
#include <stdio.h>
#include <omp.h>
#define N 16

int main() {
  // Declare the variables
  long A[N], B[N], C[N];
  long sum = 0;
  long i, tId;
  // Initialize the arrays A and B
  for (i = 0; i < N; i++) {
    A[i] = i;
    B[i] = i;
  }

  // Parallel Computation on OpenMP
  #pragma omp parallel num_threads(N) shared(A, B, C, sum) private(
      i, tId)
  {
    tId = omp_get_thread_num();
    printf("Hello from thread %d\n", tId);

    // Parallel vector multiplication
    #pragma omp for
    for (i = 0; i < N; i++) {
      C[i] = A[i] * B[i];
    }
    // Parallel reduction for the sum
    #pragma omp for reduction (+:sum)
    for (i = 0; i < N; i++) {
      sum = sum + C[i];
    }
  }
  printf("Sum = %d\n", sum);
  return 0;
}
```

When we run this code, we observe that we have created 16 threads, from thread 0 to thread 15, and these are running simultaneously in no apparent order. In order to compute $C[i]$ for all $N$ elements, we make use of the compiler directive, *#pragma omp for*. This clause is used to distribute loop iterations among the threads.

Apart from the computation of the products, the sum of them are calculated. In order to do that, we use a compiler directive, *for reduction*, in our OpenMP implementation of dot product example. This directive is very useful if a particular operation on a data is executed iteratively. That is to say, its value at a particular iteration is computed according to the previously calculated one. As for our example, the variable of *sum* has a local copy for each thread and the values of the local ones are reduced into the global shared variable, *sum*.

Output of this example is presented below.

```
Hello from thread 0
Hello from thread 7
Hello from thread 12
Hello from thread 10
Hello from thread 5
Hello from thread 8
Hello from thread 13
Hello from thread 11
Hello from thread 14
Hello from thread 9
Hello from thread 2
Hello from thread 3
Hello from thread 6
Hello from thread 4
Hello from thread 1
Hello from thread 15
Sum = 1240
```

Above all, the graphical representation of the example is presented in Figure 1.4 in order to show the execution of threads on OpenMP.

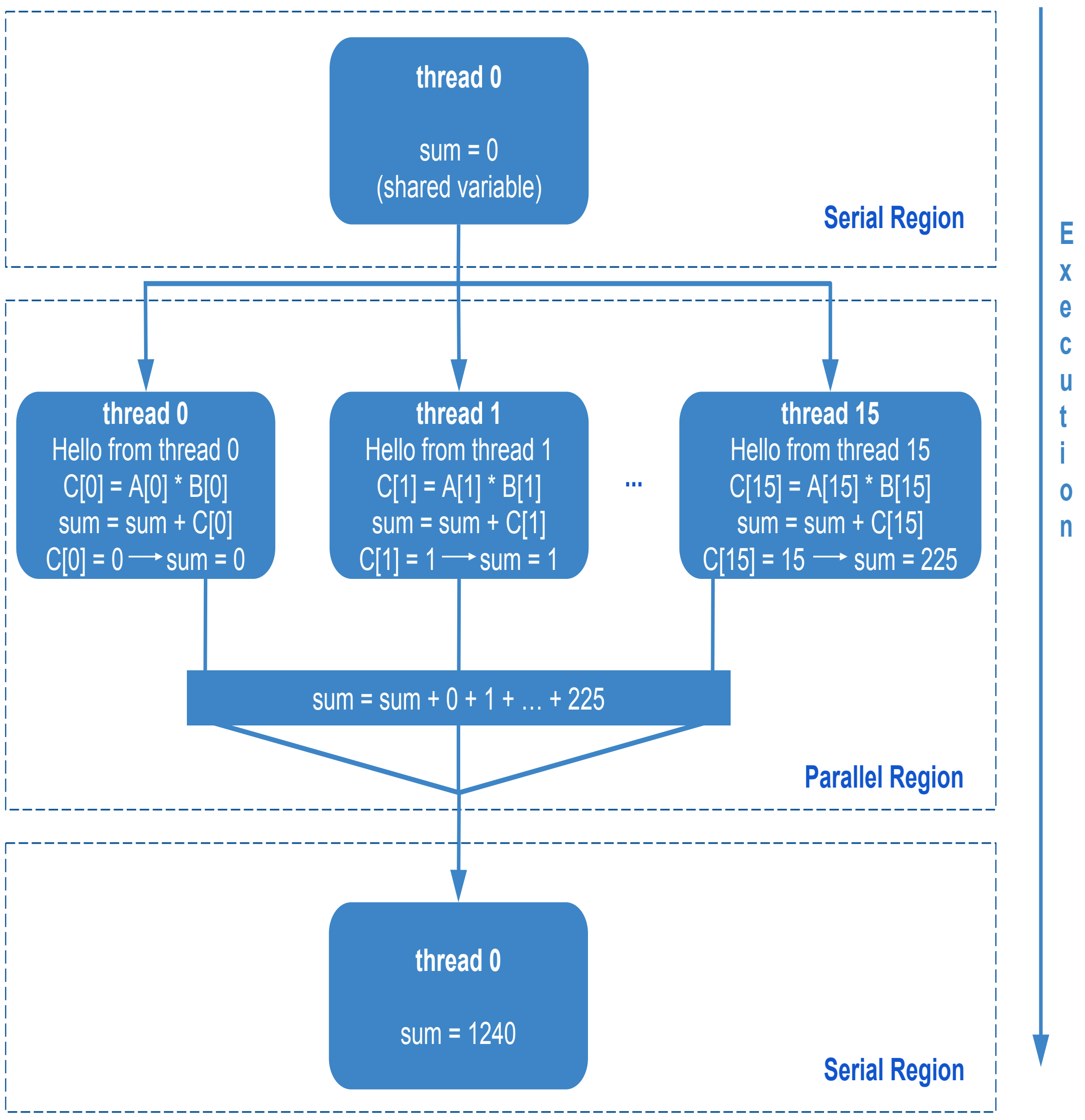


Figure 1.4: The Graphical Represantation of the Example : Dot Product

The *pragma omp parallel* instructs the program should start a number of threads equal to what is passed in via the *num_threads* directive. Then, each thread executes the function of printing "*Hello*" message and calculating the values of array $C$. By the help of the *pragma omp for reduction* directive, all products are accumulated to *sum* variable. The threads then rejoin the main thread at which point they are terminated. Finally, the main thread is itself terminated. Thanks to the parallel programming, programs can execute faster than their sequential counterparts by sharing work among the threads. In fact, the more the number of threads is, the higher parallel efficiency we can obtain.

# CHAPTER 2

# RELATED WORK

Recently, one of the most important topics around the academic world is to seek optimal solution to combinatorial optimization problem which can be defined as a quadratic assignment problem (QAP). Therefore, there are a number of exact and approximate heuristic approaches in literature as many researchers worldwide conduct a research on this problem.

In this chapter, we examine the contributions to the progress of exact and heuristic solution methods for QAP brought about by the study of various approaches. As a consequence, we provide a literature review of state-of-art heuristic adaptations, some of the most popular recent studies for combinatorial optimization problems and proposed algorithms that solve the QAP by parallel programming techniques.

## 2.1 State-of-art Heuristic Adaptations

In the literature, QAP is specified as the problem of finding a minimum cost of assignment facilities to locations, calculating costs as the total of whole distance-flow products [33]. This problem firstly arisen by Koopmans and Beckmann in 1957 as a mathematical model associated with economic activities [5]. In this research, there is a discussion about problems in the assignment of independent resources interpreted as plants. Afterwards, it has been used in various practical applications to solve real-life problems in different fields.

QAP is one of the most troubling combinatorial optimization problems. That is to say, problem whose size is greater than 30 cannot be worked out in rational time. On the occasion of its practical and theoretical significance and its complexity, QAP has been an outstanding research area for the researchers all over the world since its first formulation [33]. In 1976, Sahni and Gonzales declared that QAP is an NP-hard combinatorial optimization problem [34]. Hence, there is no doubt, the algorithm solving the problem in a polynomial time does not exist. In fact, there is a huge number of NP-hard combinatorial optimization problems that can be modeled as the QAP. To name a few, travelling salesman, scheduling, transportation systems, typewriter keyboard design, the graph partitioning problem, backboard wiring, signal processing, bin-packing, maximum clique, statistical data analysis, data allocation, layout design, minimum-inbreeding seed orchard layout and so on [7][35][4][26][36][37][38].

In 1961, for the backboard wiring problem, Steinberg focused on QAP in order to decrease the connections between components [26]. In 1972 and 1980, Heffley used it to resolve economic problems [39][40]. In 1976, Geoffrion and Graves applied QAP for scheduling problems [41]. In 1976, Pollatsche applied it to the layout planning problem in archaeology which attract building planners and operation researchers' great attention [42]. In 1987, Hubert focused on statistical analysis [43]. In 1994, Forsberg et al. applied it in the field of chemistry [44]. In 2000, Brusco and Stahl focused on numerical analysis [45].

As a matter of fact, numerous applications to the QAP is for the facilities layout problem. In 1972, Dickey and Hopkins used it to allocation of constructions in a University campus [30]. In 1977, Elshafei applied it for designing hospital [29]. In 1993, Bos focused on forest parks [46]. In 2002, Benjaafar declared a formulation of the facility layout problem to reduce work-in-progress [47]. In 2003, Rabak and Sichman [48], in 2005, Miranda et al. [49] and in 2007 Duman and İlhan [50] applied it to the allocation of electronic components on a printed circuit board or on a microchip. In 2005, Ben-David and Malah studied the index assignment problem on which the effect of channel errors on the coding system performance depends [51].

As for the proved optimal solutions that declared as the local optima, we can exemplify like : Bur26 in 2004 and Tai25a in 2003 by Hahn; Ste36a in 2001 by Brixius and Anstreicher; Bur26a in 2001 and Kra30a by Hahn; Kra30b, Kra32 and Tho30 in 2000 and Nug30 in 2000 by Anstreicher, Brixius, Goux and Linderoth; Ste36b, and Ste36c in 1999 by Nystrom. By means of enhanced tabu search algorithm, Misevicius improved the best known local optima for Tai50a, Tai80a and Tai100a in 2003 [33]. In the light of these proved solutions, in 2003, Anstreicher published the article about latest improvements in QAP solutions and proposed the heuristics [52]. Moreover, in order to test efficiency of solutions, the new instances are appeared by Burkard et al. in 1991 [4] and in 1997 [53], Li and Pardalos in 1992 [54] and QAPLIB in 2004 [55]. Consequently, new problem sets which are specified as hard for meta-heuristics are declared by Palubeckis in 1999 [56] and 2000 [57], Drezner et al. in 2005 [58] and Stützle and Fernandes in 2004 [59].

## 2.2 Recent Studies to Solve the QAP

In the literature, there are several heuristics which make use of the tabu search algorithm. Drezner proposed an algorithm to improve the concentric tabu search for the QAP in 2005 [60]. In 2012, Iterated Tabu Search (ITS) algorithm is proposed by Misevicius in order to practically solve the medium- and large-scale QAP [61]. ITS is a combination of intensification mechanism to seek for good solutions in the neighbourhood and diversification mechanism in an attempt to run away from local optima and continue on the different searching area. Moreover, it raises a novel formula for quick computation of the objective function so that the results can be obtained more quickly. The consequences of the tests for ITS algorithm show promising efficiency particularly for the random QAP instances. Fescioglu-Unver and Kokar proposed a self controlling TS algorithm for the QAP in 2011 [62]. They come up with novel mechanisms for the TS algorithm. These mechanisms aim to adjust the algorithm parameters for intensification and diversification. Thanks to the self-controlling mechanisms of the algorithm, well accomplishments on different QAP instances can be obtained. Acan and Unveren proposed an algorithm called GDA, a combination of great deluge and tabu search algorithms, for QAP in 2015 [63]. GDA benefits from the accumulated experience in memory. When better solution obtained in a specific iteration, GDA apply to the level-based acceptance criterion and update the first stage of external memory. Furthermore, second stage is updated after the first stage is updated for a determined number of times. The elements in second stage are controlled to be dissimilar according to the similarity degree. Therefore, it stores promising elements from different regions of exploration area. As a result of the experiments, it is apparent that GDA can be an alternative way to solve the QAP since it shows promising results.

In 2006, ANGEL combining the ant colony optimization (ACO), the genetic algorithm (GA) and a local search method (LS), is proposed as a hybrid metaheuristic to solve QAP by Tseng and Liang [64]. It consists of ACO and GA phases. ANGEL uses the local search method along with the constructing initial population mechanism. As a result of the experiments, it is shown that ANGEL is successful for obtaining the optimal solution with a high rate.

Breakout Local Search Algorithm (BLS), proposed by Benlic et al. [2] executes local search algorithm that finds local optima with best improvement move method and escape from this local optima to another local optimum with its adaptive perturbation mechanism that makes use of tabu search, recency-based and random perturbation methods.

BLS consists of two main points including local search procedure and diversification mechanism that dynamically determines perturbation moves to run away from one local optimum to another. In LS procedure, it searches for the best move using the steepest descent procedure. By the time it reaches to local optima it runs the second part of algorithm which is the perturbation phase. According to the present situation of exploration, the magnitude of jump is determined. Perturbation mechanism plays a significant role in varying the search areas. With the benefit of diversifying jump magnitude, it improves the quality of investigation. These two main phases turn in rotation until it hits the best moving cost or it extends the time limit set to 2 hours for instances whose size is equal and less than 100, and to 10 hours for the two instances which are biggest in the library(tai150b and tho150) [2].

BLS has been even applied to various complex combinatorial problems that are maximum clique (both weighted and unweighted cases) [65], maximum cut [66], and minimum sum coloring, to name a few [2]. Pretty well success is achieved, as a result of these experiments made on the well known problem instances of the QAPLIB benchmark. What is more, it shows quite well performance on this problem set by reaching the current best known solutions in acceptable computation time, except for 2 cases.

In 2015, Benlic and Hao proposed a population-based memetic algorithm (BMA) which incorporates with BLS, a powerful local optimization algorithm [67]. In more detail, BMA consists of a uniform crossover, a fitness-based pool updating methodology and an adjustable mutation strategy. As for the experiments, they carry out the research on the 135 well-known problem sets from the QAPLIB. As a result of the ability to attain current best known solutions by 133 of the whole selected benchmark instances, BMA outperforms favourably when compare to the present most remarkable QAP algorithms.

## 2.3 Optimization with Parallel Programming for Solving the QAP

Recently, there is a continuous attention in parallel programming taking advantage of graphics processing unit (GPU) which is developed by NVIDIA Corporation. A GPU has thousands of cores whereas a central processing unit (CPU) has a few cores designed for sequential serial working. By the help of having an almost entirely parallel architecture, GPUs handle multiple tasks simultaneously in efficient way. That is to say, it is capable of billions of calculations per second. Therefore, more and more people worldwide use GPU along with a CPU in order to speed up computationally-intensive tasks in several scientific, analytic, engineering, and enterprise applications.

Since it offers speed-up opportunity which outperforms current multi-core processors, Tsutsui and Fujimoto applied GPU with compute unified device architecture (CUDA) for solving the QAP in 2011 [68]. By combining tabu search algorithm, they propose ant colony optimisation for the QAP. In this research, Tabu search moves create two groups according to their computing cost. For the computation of these groups' moving cost, threads of CUDA are applied. In order to minimize disabling time, Move-Cost Adjusted Thread Assignment (MATA) is used. Moreover, Cunning Ant System is used for the ACO algorithm. The result of this study is that GPU computation with MATA presents outperforming acceleration when compared with the speed of computation in CPU.

In 2015, Dokeroglu and Cosar proposed a novel Parallel Multi-start Hyper-heuristic algorithm (MSH-QAP) for the solution of the QAP [7]. With the help of hyper-heuristics, appropriate heuristic for the given instance is dynamically determined along with the parameters of the heuristic. In this research, MSH-QAP benefits from the state-of-the-art (meta)-heuristics which have been declared as the best performing algorithms solving the QAP with large instances. These algorithms consist of Simulated Annealing (SA), Robust Tabu Search (RTS), and Ant Colony Optimisation-based Fast Ant System (FANT). For the experiments, they use 134 problem instances in QAPLIB which are classified into four types according to the categorisation by Stützle [25]. Besides, they run MSH-QAP on 64 processors with various multistarts to acquire high quality solutions. As a result, 119 of the problem sets are solved successfully in terms of reaching best known results whereas remaining 15 problems end up with %0.07 average percentage deviation (APD) from the best known solutions (BKS) in literature.

Harris et al., in 2015, proposed a memetic algorithm (MA) which makes use of ternary tree structure for its population and TS algorithm which runs simultaneously for its local search mechanism [69]. In fact, MA is an improvement of the algorithm mentioned in the study of Meneses in 2011 [23]. As a result of the experiments, using instances taken from QAPLIB [4], this algorithm shows exactly great performance in the sense of not only less time but also more qualified solution.

In 2012, Czapinski proposed Parallel Multistart Tabu Search (PMTS) algorithm which consists of remarkable and outstandingly rapid heuristic for QAP. Furthermore, it is implemented on a significantly powerful GPU hardware intended for high performance computing with CUDA platform [70]. Therefore, PMTS is shown to perform competitively with a single-core or a parallel CPU implementation on a high-end six-core CPU. This noteworthy results from experiments are owing to the neighbourhood evaluation in parallel architecture, effective memory design, sensible access patterns and almost entirely GPU application of Tabu Search.

In 2015, a parallel hybrid algorithm (PHA) with three phases is proposed by Tosun [71]. PHA initially benefits from a genetic algorithm to get a high quality initial seed on which a diversification mechanism will be run. Finally, this modified solution is used for a robust tabu search in order to find a near-optimal result. Moreover, PHA makes use of parallel computing; therefore it obtains considerable speed-up. As a result of the experiments, it is obviously seen that in the sense of solution quality and execution time.

To the best of our knowledge, there is no heuristic approach that takes the advantage of the threads and similarity checking of the previously explored areas by using OpenMp to solve the QAP. The heuristic search algorithm which uses a log-based similarity checking approach for the previously searched permutations of the QAP problem instances has not brought about yet. Recent approaches also try to get rid of the stagnation and do this by randomly generating new starting points on a single core. Therefore, we propose Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem algorithm (BLS-OpenMP-QAP) that is explained in detail in the following Chapters. BLS-OpenMP-QAP algorithm executes on different candidate solutions instead of only one permutation thanks to the similarity check mechanism of the proposed algorithm.

# CHAPTER 3

# PROPOSED ALGORITHM

This chapter is dedicated to our proposed algorithm named BLS-OpenMP-QAP (Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem). In this research, we aimed to make use of parallel programming with OpenMP on the purpose of attaining remarkable solutions on the well-known problem instances from the QAPLIB benchmark.

With the help of parallel programming with OpenMP, our recommended algorithm shows a promising speed-up when compared to its CPU implementations counterpart. As we will describe in the following Chapter 4, BLS-OpenMP-QAP can obtain outstandingly competitive results on a wide range of well-known problem sets from the QAPLIB by reaching the current best known solutions in less computation time or obtaining even better results in the same computing time.

In this chapter, we respectively mention about the components of BLS-OpenMP-QAP algorithm presented in Algorithm 1. To be more precisely, it gets start with generating initial candidate solutions strategy. In order not to start with the same candidate solution twice, the solutions experimented before in each iterations of algorithm are taken into consideration. Later then, it proceeds with the descent procedure which improves local optima. On the purpose of reaching best enhancing neighbouring solution, BLS-OpenMP-QAP explores the whole neighbourhood in the steepest descent process. As soon as the local optima is reached, the number of perturbation moves, jump magnitude, is determined. Eventually, it applies to perturbation stage consisting of tabu search(TS), recency-based and random perturbation methods so that it can escape from the local optima with new starting point.

**Algorithm 1** Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem (BLS-OpenMP-QAP) Algorithm

```
Require:
    Distance and flow matrices $d$ and $f$ of size $n$ x $n$.
    The number of thread $count_{thread}$.
    The size of loop for breakout local search(BLS) processes $size_{bls}$.
    The vector keeping the initial and result solutions $previousSolutions$.
Ensure:
    A permutation $\pi$ over a set of facility locations.
1: $\pi \leftarrow$ random permutation of $\{1, ..., n\}$
2: $\pi_{initDF} \leftarrow$ GenerateSolutionsByDFMatrices() /* $\pi_{initDF}$ is the randomly
   generated initial solution for the first thread by considering distance and
   flow matrices (Subsection 3.1.1) */
3: /* Generate different initial solutions randomly for remaining threads */
4: $\pi_{initRs} \leftarrow$ GenerateDifferentSolutionsRandomly()
5: $\pi_{inits} \leftarrow \pi_{initDF} + \pi_{initRs}$ /* $\pi_{inits}$ is the list of all initial solutions whose size
   is equal to the number of threads $count_{thread}$ */
6: for $i := 1$ to $size_{bls}$ do
7:     if $i > 1$ then
8:         /* Generate different initial solutions randomly for the all threads by
   looking at the previous experimented candidates (Subsection 3.1.2)*/
9:         $\pi_{initRs} \leftarrow$ GenerateDifferentSolutionsRandomly($previousSolutions$)
10:        $\pi_{inits} \leftarrow \pi_{initRs}$
11:    end if
```

**Algorithm 1** Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem (BLS-OpenMP-QAP) Algorithm(continued)

```
12:     Write initial solutions to $previousSolutions$
13:     Send $\pi_{inits}$ to the threads
14:     $c \leftarrow C(\pi)$ /* $c$ is the objective value of the current solution */
15:     Compute the initial $n$ x $n$ matrix $\delta$ of move gains
16:     $\pi_{best} \leftarrow \pi$ /* $\pi_{best}$ is the best solution found so far */
17:     $c_{best} \leftarrow c$ /* $c_{best}$ is the best objective value reached so far */
18:     $c_p \leftarrow c$ /* $c_p$ is the best objective value of the last descent */
19:     $w \leftarrow 0$ /* $w$ is the counter of consecutive non-improving local optima */
20:     $L \leftarrow L_0$ /* set the number of perturb. moves $L$ to its default value $L_0$*/
21:     while stopping condition not reached by all of the threads do
22:         /* Identify the best improving move and apply it to obtain better solution $\pi$ in the whole neighbourhood */
23:         $\pi \leftarrow$ SteepestDescent($\pi, \pi_{best}, c, c_{best}, c_p, H, Iter, \delta, w$) /* Section 3.2 */
24:         /* Determine the perturbation strength $L$ to apply to $\pi$ */
25:         $L \leftarrow$ DetermineJumpMagnitude($c, c_p, w$) /* Section 3.3 */
26:         /* Perturb the current local optimum $\pi$ with $L$ perturb. moves */
27:         $c_p \leftarrow c$ /* Update the objective value of previous local optimum */
28:         $\pi \leftarrow$ Perturbation($\pi, L, H, Iter, \delta, w$) /* Section 3.4 */
29:     end while
30:     Obtain result solutions from threads
31:     Write result solutions to $previousSolutions$
32: end for
```

## 3.1 Generating Candidate Solutions

First of all, BLS-OpenMP-QAP algorithm creates initial solutions based on some background processes. In other words, producing candidates forms the backdrop of BLS-OpenMP-QAP algorithm. Due to the fact that we are not able to study on the whole permutation $\pi$ of $\{1,...,n\}$ solution, the smarter generating initial solution mechanism of the algorithm is, the more effectively we can get the result. Therefore, we first and foremost attach high importance to creating candidate solutions since the most significant thing is that candidates affect the rest of the algorithm.

As presented in Algorithm 1 at lines 2 and 4, producing candidate solutions is composed of two sections which are categorised as smart beginning by considering the distance and flow matrices and smart start by looking at previous experimented candidates and their results. Furthermore, this procedure is executed on the CPU in advance of OpenMP implementations so that these candidate solutions are going to be used by the whole threads via OpenMP. Besides, the number of candidates must be equal to the number of all threads whose size are determined at the beginning of the algorithm.

### 3.1.1 Intelligent Initialisation by Considering Distance and Flow Matrices

The idea of that the more flows there are between the facilities, the closer they must be located leads us to generate a candidate solution according to the distance and flow matrices. In this subsection, we precisely explained our method for producing initial candidate solution considering the distance and flow matrices for the first thread. As a matter of fact, this method is employed only for the first iteration of the algorithm. In this iteration, candidates for remaining threads are produced randomly so that they are different from each other according to the minimum percentage of difference specified at the beginning of the algorithm.

According to the distance matrices, we initially arrange locations in order by putting them on the distance array. That is to say, we enumerate places so that two locations which are the most closer to each other are in the first place of the array and the remainder is arranged appropriately. Afterwards, we build up flow array and bring it into alignment with flow matrices. In other words, facilities which have too much flowing between each other are placed side-by-side and at the first place of the flow array and the rest is adjusted accordingly.

Immediately after we construct the distance and flow arrays, we create first candidate solution. To do so, we take the elements of distance matrices in a sequence as location and we pick the corresponding members of flow matrices in order as facility. Later then, we make use of these location and the facility information so that the facility takes place in the location index of the first candidate solution. In a similar manner, until the last index of the arrays is reached, elements of first candidate are located accordingly. As a matter of fact, this candidate solution is used once by only first thread in OpenMp in the first start of the algorithm. Furthermore, this process is exemplified in Figure 3.1 along with its flows and distances matrices presented in Table 3.1 and 3.2.

Table3.1: Distances between locations for initial solution example

| location | $l_1$ | $l_2$ | $l_3$ | $l_4$ |
|---|---|---|---|---|
| $l_1$ | 0 | 20 | 40 | 60 |
| $l_2$ | 20 | 0 | 50 | 10 |
| $l_3$ | 40 | 50 | 0 | 55 |
| $l_4$ | 60 | 10 | 55 | 0 |

Table3.2: Flows between facilities for initial solution example

| facility | $f_1$ | $f_2$ | $f_3$ | $f_4$ |
|---|---|---|---|---|
| $f_1$ | 0 | 15 | 0 | 1 |
| $f_2$ | 15 | 0 | 10 | 25 |
| $f_3$ | 0 | 10 | 0 | 30 |
| $f_4$ | 1 | 25 | 30 | 0 |

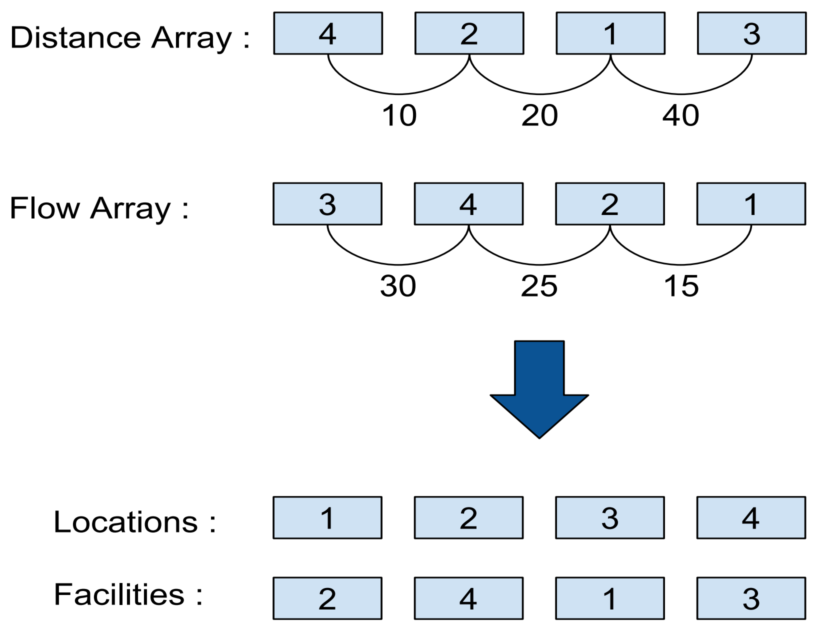


Figure 3.1: Representation of Intelligent Initialisation by Considering Distance and Flow Matrices

### 3.1.2 Intelligent Initialisation by looking at Previous Experimented Candidates

Apart from producing the first candidate solution presented in Section 3.1.1, BLS-OpenMP-QAP algorithm needs much more candidate solutions for the other threads in OpenMP which are responsible for the remainder processes of the algorithm. As presented in Algorithm 2, in order to guarantee that all candidates are different from each other, there is a constraint that randomly generated solutions cannot be the same as previously experimented ones by less than the minimum percentage of difference. According to the previously determined minimum percentage of difference between candidate solutions, randomly generated permutation is assumed as acceptable or not. Unless it is satisfactory, BLS-OpenMP-QAP regenerates random candidate solution until it meets the difference constraint. By the time the algorithm produces adequate number of candidate solutions, that is, as many as the size of threads in OpenMP, BLS-OpenMP-QAP continues with the remaining processes of the algorithm.

**Algorithm 2** Intelligent Initialisation by looking at the Previous Experimented Candidates Algorithm

```
Require:
    The size of thread $count_{thread}$.
    The vector keeping the initial and result solutions $previousSolutions$.
Ensure:
    A permutation $\pi$ over a set of facility locations.
1: for $i := 1$ to $count_{thread}$ do
2:     $\pi \leftarrow$ random permutation of $\{1, ..., n\}$
3:     /* Similarity check for $\pi$ (Algorithm 3)*/
4:     while isSimilarToPreviousSolutions($\pi, previousSolutions$) do
5:         $\pi \leftarrow$ random permutation of $\{1, ..., n\}$
6:     end while
7:     Add $\pi$ to $initialSolutionList$
8: end for
```

As for our similarity check mechanism, it is derived from Minimum Edit Distance algorithm called Levenshtein Distance (LD) devised by Levenshtein. LD is a distance of the similarity between two strings. In other words, it is the minimum edit distance between two strings. This is the minimum amount of editing operations of deletions, insertions, or substitutions needed to convert one string into another. Therefore, the less the LD is, the more similar the strings are.

Indeed, LD algorithm is applied to in several real life applications. For instance, spell correction, speech recognition, computational biology, DNA analysis, machine translation, information extraction and plagiarism detection are some rewarding applications of Levenshtein Distance.

In 2002, Ristad and Yianilos propose a stochastic model for string-edit distance in order to determine the similarity of two strings [72]. They apply this model on the hard problem of speech recognition for pronunciation of words in conversational speech and obtain a remarkable result. Wilbert Heeringa also make use of LD in his thesis in order to measure dialect pronunciation differences [73]. In another study, LD is used for gauging phonetic distances between Norwegian

**Algorithm 3** Similarity Control Mechanism

**Require:**

The size of the problem $n$, minimum percentage of difference $minDifference$, the vector of previous solutions $previousSolutions$.

**Ensure:**

A permutation $\pi$ over a set of facility locations.

```
1:  for $i := 1$ to size of $previousSolutions$ do
2:      for $j := 1$ to size of $n$ do
3:          if $previousSolutions[i][j] \neq \pi[j]$ then
4:              $diffNumber++;$
5:          end if
6:          if $diffNumber > minDifference$ then return $false$
7:          end if
8:      end for
9:      return $true$
10: end for
```

dialects in order to approve Levenshtein Dialect Distance which is applied before in Dutch dialects [74]. In 2003, Bilenko et al. use LD in their study for the comparison and description of methods in textual similarity measures for name matching [75]. This study is based on the idea that recognizing distinct records referencing to the same entities in database is significant for information-integration. Furthermore, string similarity measuring approach is also applied for identifying plagiarism [76]. In 2004, adapted LD is proposed for comparison of signatures depending on an event-string modelling of features by utilising the pen-position and pressure signals of digitizer tablets [77]. In 2012, normalized Levenshtein distance function is proposed for measuring cross-language orthographic similarity [78].

The LD between two strings $a, b$ is calculated according to the following formulation :

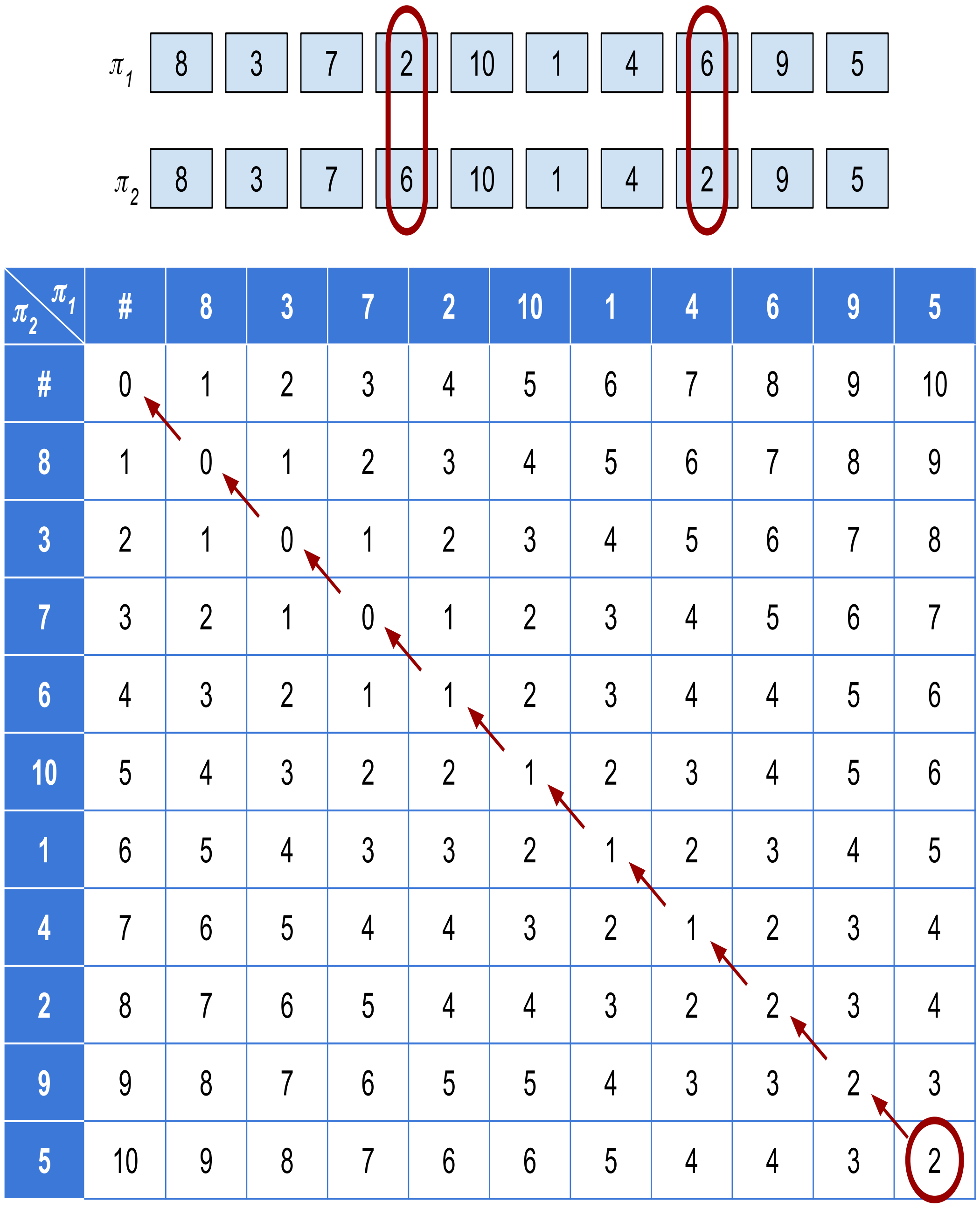
$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases} \quad [79]$$

where $1_{(a_i \neq b_j)}$ is equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

In order to explain LD more precisely, we give two different examples and then we decide whether the second candidate is applicable or not for the remaining procedures of BLS-OpenMP-QAP algorithm. For each of the examples we generate two different candidate solutions. Afterwards, we compute the minimum similarity distance for those solutions based on LD. Finally, according to the similarity threshold value of 30%, we determine that the second candidate is acceptable or not.

As can be seen in Figure 3.2, there are two candidate solutions, $\pi_1$ and $\pi_2$. We assume that $\pi_1$ is used in the first multi-start of BLS-OpenMP-QAP algorithm and we want to generate another candidate in order to execute our algorithm on this solution which is different from the first one by 30% similarity ratio. Therefore, we make use of LD in an attempt to calculate the minimum difference size. As a consequence of this computation, we obtain the distance value $lev(\pi_1, \pi_2)$ $= 2$. This means that two substitutions are sufficient to transform $\pi_1$ into $\pi_2$. Hence, the second candidate, $\pi_2$, violates the similarity ratio constraint by the percentage of 20 (2/10). Furthermore, it is not applicable for our algorithm and it generates another random candidate solution.

In the second example in Figure 3.3, BLS-OpenMP-QAP algorithm randomly generates another candidate solution, $\pi_3$. As for the similarity threshold, we compare $\pi_3$ with the first solution $\pi_1$ which is used in the first multi-start of BLS-OpenMP-QAP algorithm. By this comparison we want to ensure that it is different from the first solution $\pi_1$ by 30% similarity ratio in order to execute



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\pi_1$ | 8 | 3 | 7 | 2 | 10 | 1 | 4 | 6 | 9 | 5 |
| $\pi_2$ | 8 | 3 | 7 | 6 | 10 | 1 | 4 | 2 | 9 | 5 |

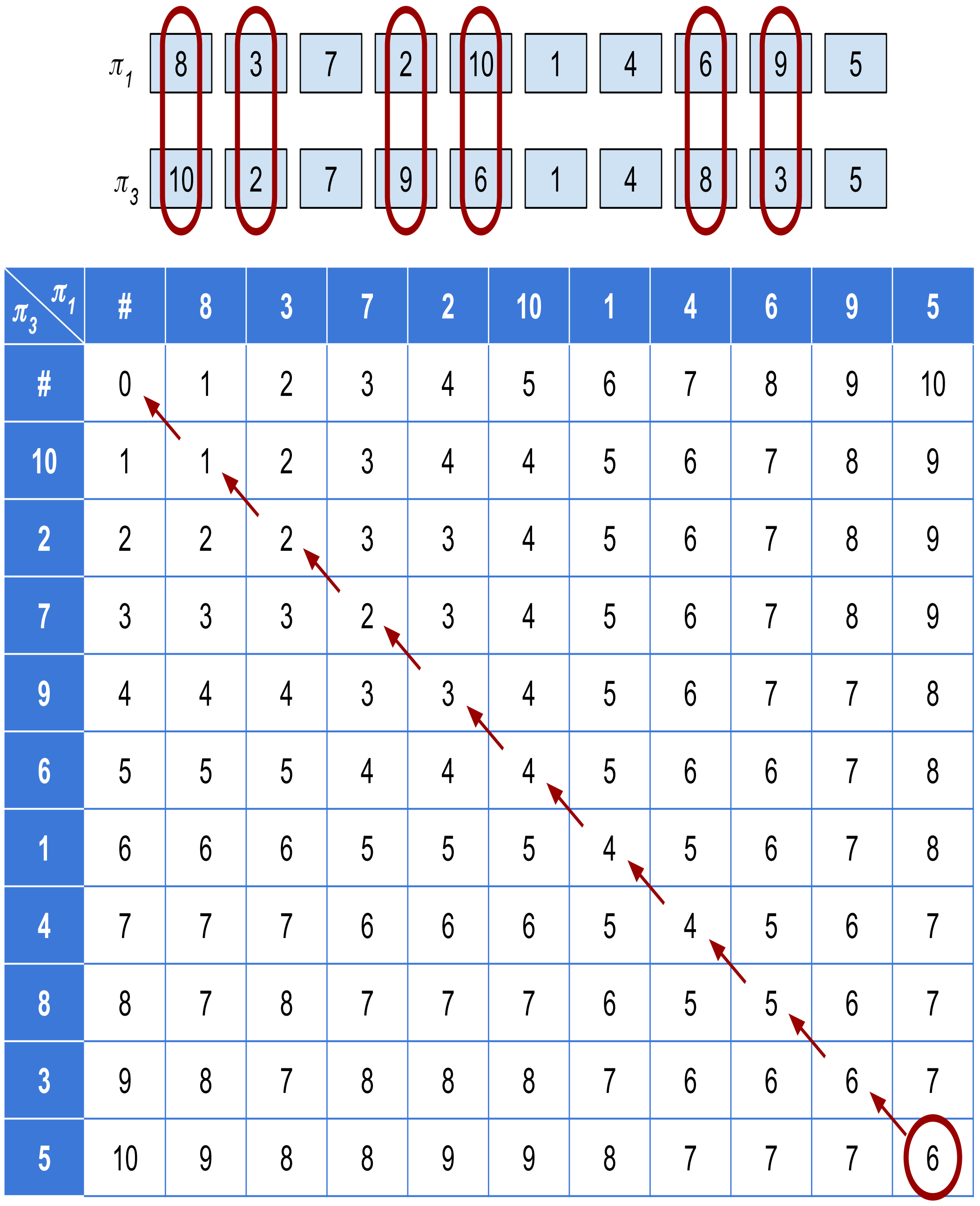
| $\pi_1$ $\pi_2$ | # | 8 | 3 | 7 | 2 | 10 | 1 | 4 | 6 | 9 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 8 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 6 | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 |
| 10 | 5 | 4 | 3 | 2 | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 6 | 5 | 4 | 3 | 3 | 2 | 1 | 2 | 3 | 4 | 5 |
| 4 | 7 | 6 | 5 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 2 | 8 | 7 | 6 | 5 | 4 | 4 | 3 | 2 | 2 | 3 | 4 |
| 9 | 9 | 8 | 7 | 6 | 5 | 5 | 4 | 3 | 3 | 2 | 3 |
| 5 | 10 | 9 | 8 | 7 | 6 | 6 | 5 | 4 | 4 | 3 | 2 |

Figure 3.2: An Example of Violating the Minimum Difference Constraint (30%)

our algorithm on this solution. In an effort to make this comparison, we take advantages of LD with the aim of computing the minimum difference size. As a result of this calculation, we obtain the distance value $lev(\pi_1, \pi_3) = 6$ which means that six substitutions are adequate to transform $\pi_1$ into $\pi_3$. Hence, the second candidate, $\pi_3$, does not violate the similarity ratio constraint by the percentage of 60 (6/10). Consequently, $\pi_3$ can be used by BLS-OpenMP-QAP algorithm.

As for our proposed algorithm, we adapt LD with regard to meet our needs in an effective way. In BLS-OpenMP-QAP, candidate solutions have the same length so that two solutions can be transformed into one another only by the substitution operation. Therefore, we regard the cost of each substitution value to 1. Furthermore, searching for a sequence of edits from the first permutation values to the second permutation values requires huge space and navigating from these sequences of edits is required high afford. As for our algorithm, we make this calculation by comparing all values in the first solution with the corresponding value, at the same index, of the second solution. By this way, we can calculate the minimum difference size in a cost-effective way.

To put this similarity check procedure in a nutshell, the candidate solution $\pi$ is randomly generated for each threads on OpenMP in order to execute the remaining procedures of the algorithm. Therefore, the size of generated candidates must be same as the number of threads specified at the beginning of the algorithm. Afterwards, the algorithm performs the similarity check on these candidates. The similarity check mechanism determines whether solution $\pi$ violates the minimum difference constraint or not. The minimum percentage of difference is also determined at the beginning of the algorithm. In order to provide this control, $\pi$ is compared respectively with the whole solutions in the vector $previousSolutions$ which BLS-OpenMP-QAP algorithm puts the permutations on and keeps track of the previously experimented solutions. Indeed, all values in the $\pi$ is compared one by one with the corresponding value, at the same index, of the solution from the vector $previousSolutions$. If $\pi$ meets the minimum difference constraint, it is put in to the list of the candidates. Otherwise, new candidates are generated randomly and the similarity check mechanism proceeds



|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| $\pi_1$ | 8 | 3 | 7 | 2 | 10 | 1 | 4 | 6 | 9 | 5 |
| $\pi_3$ | 10 | 2 | 7 | 9 | 6 | 1 | 4 | 8 | 3 | 5 |

| $\pi_3$ \ $\pi_1$ | # | 8 | 3 | 7 | 2 | 10 | 1 | 4 | 6 | 9 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 10 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 7 | 3 | 3 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 4 | 4 | 4 | 3 | 3 | 4 | 5 | 6 | 7 | 7 | 8 |
| 6 | 5 | 5 | 5 | 4 | 4 | 4 | 5 | 6 | 6 | 7 | 8 |
| 1 | 6 | 6 | 6 | 5 | 5 | 5 | 4 | 5 | 6 | 7 | 8 |
| 4 | 7 | 7 | 7 | 6 | 6 | 6 | 5 | 4 | 5 | 6 | 7 |
| 8 | 8 | 7 | 8 | 7 | 7 | 7 | 6 | 5 | 5 | 6 | 7 |
| 3 | 9 | 8 | 7 | 8 | 8 | 8 | 7 | 6 | 6 | 6 | 7 |
| 5 | 10 | 9 | 8 | 8 | 9 | 9 | 8 | 7 | 7 | 7 | 6 |

Figure 3.3: An Example of not Violating the Minimum Difference Constraint (30%)

on them until the difference constraint is provided. By the time all candidates are produced for the whole threads on OpenMP, this section of the algorithm terminates.

## 3.2 Improving Local Optima

First of all, in order to take full advantages of parallel programming, this and the following processes of the BLS-OpenMP-QAP algorithm are executed on OpenMP by organizing a parallel computation using the abstractions of threads. In fact, as we will see in the following Chapter 4, in our experiments, generating random initial solutions section of BLS-OpenMP-QAP is also executed with on OpenMP in order to measure the quality of recommended algorithm by making comparisons between them. What is more, a schematic representation of our proposed algorithm, BLS-OpenMP-QAP, in time-line is presented in Figure 3.4.

Previous to expanding on local search procedure, we bring out two different matrices, one is the delta matrices in which we put the result from the computation of cost difference if two elements are swapped in candidate solution $\pi$ and the other one is last swapped matrices which is used for the tabu search mechanism in perturbation strategy.

On the purpose of improving local optima, BLS-OpenMP-QAP makes use of the steepest descent procedure, presented in Algorithm 4. That is to say, the values in candidate permutation $\pi$ are iteratively transposed to minimize the cost and to reach the local optima. As a matter of fact, each iteration of this procedure searches in the whole neighbourhood to discover the best adjacent solution. Furthermore, it proceeds the iteration as long as an improving solution in the neighbouring space is obtained. By the time there is no such better adjacent solution, BLS-OpenMP-QAP continues with the determining Jump Magnitude section of algorithm in an effort to run away from the present local optima.

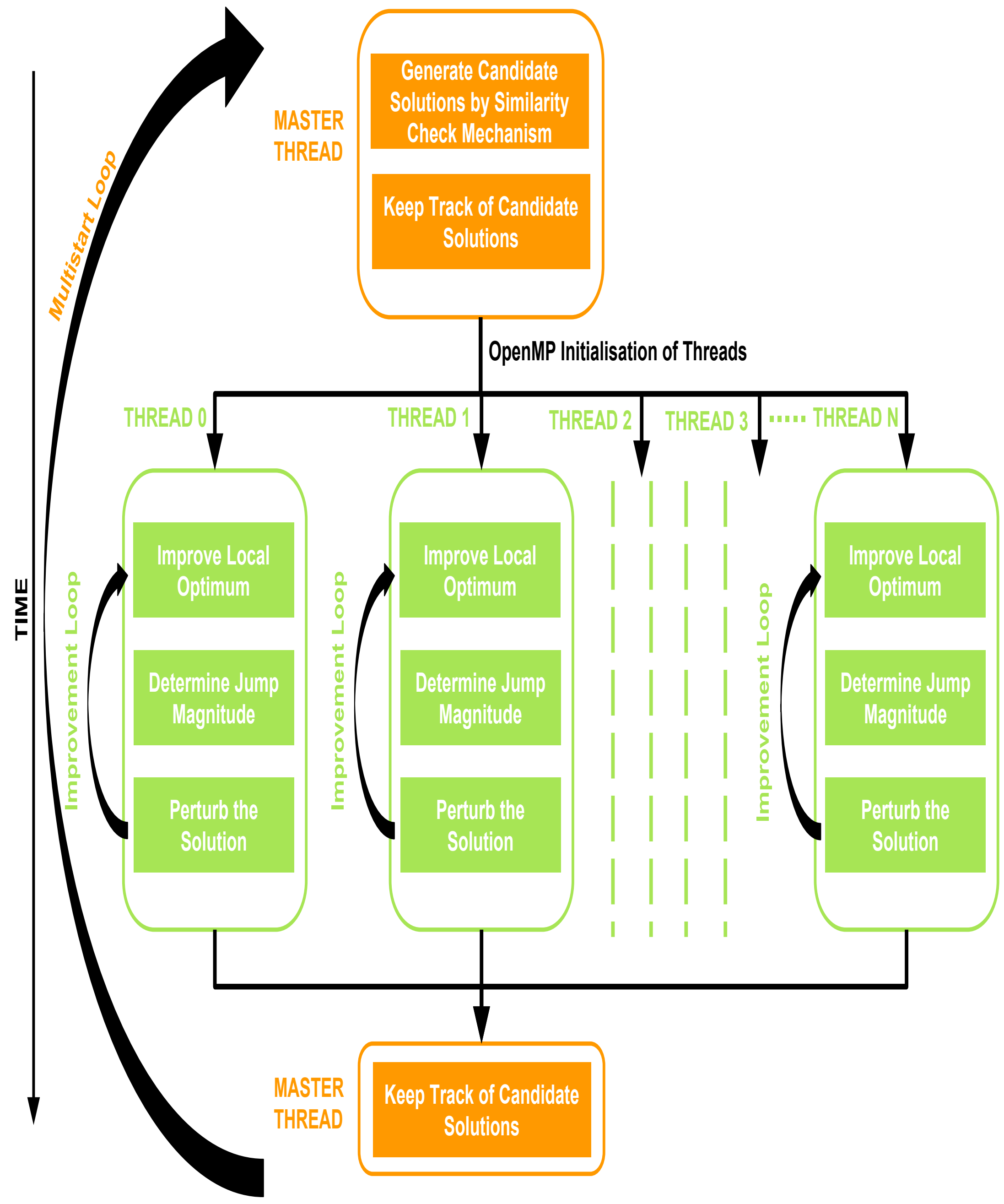


Figure 3.4: A Schematic Representation of BLS-OpenMP-QAP Algorithm in Time-line

**Algorithm 4** Steepest Descent Algorithm

$SteepestDescent(\pi, \pi_{best}, c, c_{best}, c_p, H, Iter, \delta, w)$

**Require:**

Local optimum $\pi$, the best solution found so far $\pi_{best}$, the objective value of the current solution $c$, the best objective value reached $c_{best}$, the best objective value of the last descent $c_p$, matrix $H$, global iteration counter $Iter$, move gain matrix $\delta$, the number of consecutive non-improving local optima visited $w$.

**Ensure:**

A permutation $\pi$ over a set of facility locations.

```
1:  while ∃ swap(u, v) such that $(c + \delta(\pi, u, v)) < c$ do
2:      $\pi \leftarrow \pi \oplus swap(u, v)$ /* Perform the best-improving move */
3:      $c \leftarrow c + \delta(\pi, u, v)$
4:      $H_{uv} \leftarrow Iter$ /* Update Iter. number when move $uv$ was last performed*/
5:      Update matrix $\delta$
6:      $Iter \leftarrow Iter + 1$
7:  end while
8:  if $c < c_{best}$ then
9:      $\pi_{best} \leftarrow \pi; c_{best} \leftarrow c$ /* Update the recorded best solution */
10:     $w \leftarrow 0$ /*Reset counter for consecutive non-improv. local optima*/
11: else if $c \neq c_p$ then
12:     $w \leftarrow w + 1$
13: end if
```

## 3.3 Determining the Jump Magnitude

The role of deciding the measure of perturbation moves, jump magnitude, is important for perturbation stage of BLS-OpenMP-QAP. According to the present state of neighbouring exploration, the magnitude of jump is determined. With the benefit of diversifying jump magnitude, the quality of investigation increases.

As declared in Algorithm 5, unless the best recorded solution is improved during the predetermined number of descent phases, since it shows that the search is in stagnating state, strong perturbation mechanism is required in order to get rid of the local optima. Nevertheless, if the candidate solution is enhanced, the number of perturbation moves is determined accordingly so that it can escape from the previous local optimum. However, if the cost of present solution is as the same as of the previous one, exploration returns to the previous local optimum.

By the time BLS-OpenMP-QAP reaches to local optima and then determines the Jump Magnitude, it executes the last part of the algorithm which is the perturbation phase. As a matter of fact, this phase plays a very significant role in terms of diversifying the exploration areas and escaping from the current local optimum.

## 3.4 Diversifying Search Area by Perturbation Strategy

The last and the most significant section of BLS-OpenMP-QAP algorithm is the perturbation phase. The pseudo code of this adaptive diversification exploration mechanism is demonstrated in Algorithm 6 and 7. Immediately after the jump magnitude is assigned, it gets start to escape from the local optimum by selecting among three separate types of moves for perturbation according to the present state of the exploration. In other words, the adaptive perturbation mechanism makes use of tabu search, recency-based and random perturbation methods which are declared in following Subsections 3.4.1, 3.4.2 and 3.4.3.

**Algorithm 5** Determining the Jump Magnitude Algorithm $DetermineJumpMagnitude(c, c_p, w)$

```
Require:
    The objective value of the current solution $c$.
    The best objective value of the last descent $c_p$.
    The number of consecutive non-improving local optima visited $w$.
Ensure:
    A perturbation strength $L$.
 1: if $w > T$ then
 2:     /* Search seems to be stagnating, set $L$ to a large value */
 3:     $L \leftarrow L_{max}$
 4:     $w \leftarrow 0$
 5: else if $c = c_p$ then
 6:     /* Search returned to the previous local optimum, increase jump magnitude by one */
 7:     $L \leftarrow L + 1$
 8: else
 9:     /* Search escaped from the previous local optimum, reinitialize jump magnitude */
10:     $L \leftarrow L_0$
11: end if
```

**Algorithm 6** Adaptive perturbation procedure $perturbation(\pi, L, H, Iter, \delta, w)$

```
Require:
    Local optimum $\pi$, perturbation strength $L$, matrix $H$, global iteration counter $Iter$, move gain matrix $\delta$, the number of consecutive non-improving local optima visited $w$.
Ensure:
    A perturbed solution $\pi$.
1: Determine the probability $P$ according to the Formula $A, B$and $C$ /* Subsection 3.4.1, 3.4.2, 3.4.3 */
2: With probability $P$, $\pi \leftarrow Perturb(\pi, L, H, Iter, \delta, w, A)$ /* Tabu search perturbation */
3: With probability $(1-P).Q$, $\pi \leftarrow Perturb(\pi, L, H, Iter, \delta, w, B)$ /* Recency based perturbation */
4: With probability $(1-P).(1-Q)$, $\pi \leftarrow Perturb(\pi, L, H, Iter, \delta, w, C)$ /* Random perturbation */
5: return $\pi$
```

Above all, perturbation mechanism has an intensive diversification impact on the BLS-OpenMP-QAP algorithm. Hence, it is significant to determine the most suitable perturbation type in order not to deteriorate too much the solution. Indeed, this determination is made probabilistically, and the possibility of determining a specific type is identified dynamically according to the present search state and also to the current number $w$ of iteration which there is not enhancing solution. When the best permutation is obtained through the enhancing exploration or by the time the number of iteration surpasses the specified threshold $T$ when the local optima is not enhanced, $w$ takes value of zero. Moreover, it is declared that as a result of the experimental analysis, it is practical to assure to apply tabu search perturbation type in minimum degree [2]. Consequently, there is a constraint that at least the threshold value $P_0$ must be assigned to the possibility $P$ of the application of Tabu Search Perturbation. $P$ is calculated according to the following formulation :

**Algorithm 7** Perturbation operator $Perturb(\pi, L, H, Iter, \delta, w, M)$

```
Require:
    Identical to $perturbation(\pi, L, H, Iter, \delta, w)$ of Algorithm 6 along with the set of perturbation moves $M$
Ensure:
    A perturbed solution $\pi$.
1:  for $i := 1$ to $L$ do
2:      Take $swap(u, v) \in M$
3:      $\pi \leftarrow \pi \oplus swap(u, v)$
4:      $c \leftarrow c + \delta(\pi, u, v)$
5:      $H_{uv} \leftarrow Iter$
6:      Update the move gain matrix $\delta$ and $M$
7:      $Iter \leftarrow Iter + 1$
8:      if $c < c_{best}$ then
9:          $\pi_{best} \leftarrow \pi; c_{best} \leftarrow c$ /* Update the recorded best solution */
10:         $w \leftarrow 0$ /* Reset counter for consecutive non-improv. local optima */
11:     end if
12: end for
13: return $\pi$
```

$$P = \begin{cases} e^{-w/T}, & \text{if } e^{-w/T} > P_0 \\ P_0, & \text{otherwise} \end{cases} \qquad [2]$$

In this formulation, $T$ is regarded as the maximum number of iteration which there is not enhancing solution any more. In this case, BLS-OpenMP-QAP turns into more stronger perturbation mechanism. If the value of $w$ increases, the possibility of applying tabu search perturbation reduces increasingly whereas the possibility of the application of other two methods grows up in order to obtain more powerful diversification.

Apart from the possibility of applying tabu search perturbation, for recency based perturbation, the probability is calculated as (1-$P$).$Q$ and for the random perturbation, it is identified by (1-$P$).(1-$Q$) where $Q$ is stands for the number between 0 and 1.

### 3.4.1 Tabu Search Perturbation

Tabu Search (TS) has been regarded as an outstandingly efficient approach for solving hard combinatorial problems, for instance, travelling salesman, scheduling, product delivery and routing, transportation systems and manufacturing cell design.

As it is understood by the name, this type of perturbation is based on Tabu Search(TS) fundamentals. A flow chart describing the structure of the tabu search perturbation mechanism is shown in Figure 3.5. As a matter of fact, TS fundamentally looks for whether the move is applied or not throughout the specific number of iterations ($\gamma$). A greater value of $\gamma$ indicates the strength of diversification. Furthermore, it pays attention to the history information about the last time when the move is performed and also it attaches importance not to perturb the solution too much. To put in a nutshell, appropriate moves are chosen according to the constraint as the following formulation presents by the set $A$ :

A = $\{ \ swap(u,v) | min\{\delta(\pi,u,v)\}, (H_{uv} + \gamma) < Iter \ or \ (\delta(\pi,u,v) + c) < c_{best}, u \neq v \ and \ 1 \leq u,v \leq n \ \}$ [2]

In this formulation, $H$ is regarded as the matrix on which the iteration numbers when the move was last executed are placed, $Iter$ is the present iteration number, $c$ is the cost of present permutation and $c_{best}$ is the cost of the best found permutation up to that moment.

### 3.4.2 Recency Based Perturbation

Apart from the Tabu Search Perturbation type, there is also recency based perturbation mechanism which benefits from a part of the history tracks kept on the $H$ matrix. Although the move causes the decrease in the cost, this perturbation type only takes the least recently executed moves in consideration. Indeed, the eligible moves are determined according to the constraint as the following formulation presents by the set $B$ :

B = $\{ \ swap(u,v) | min\{H_{uv}\}, u \neq v \ and \ 1 \leq u,v \leq n \ \}$ [2]

### 3.4.3 Random Perturbation

Finally, the random perturbation method completely randomly determines the moves to be performed. To explain it more precisely, the most suitable relocations are picked out according to the constraint as the following formulation presents by the set $C$ :

C = $\{ \ swap(u,v) | u \neq v \ and \ 1 \leq u,v \leq n \ \}$ [2]

To put this adaptive perturbation strategy in a nutshell, by the time the perturbation genre is found out, corresponding relocations determined by the sets $A$, $B$ and $C$ are appropriately applied. Therefore, at the next iteration of the steepest descent algorithm, this resulting solution is accepted as the new candidate permutation.

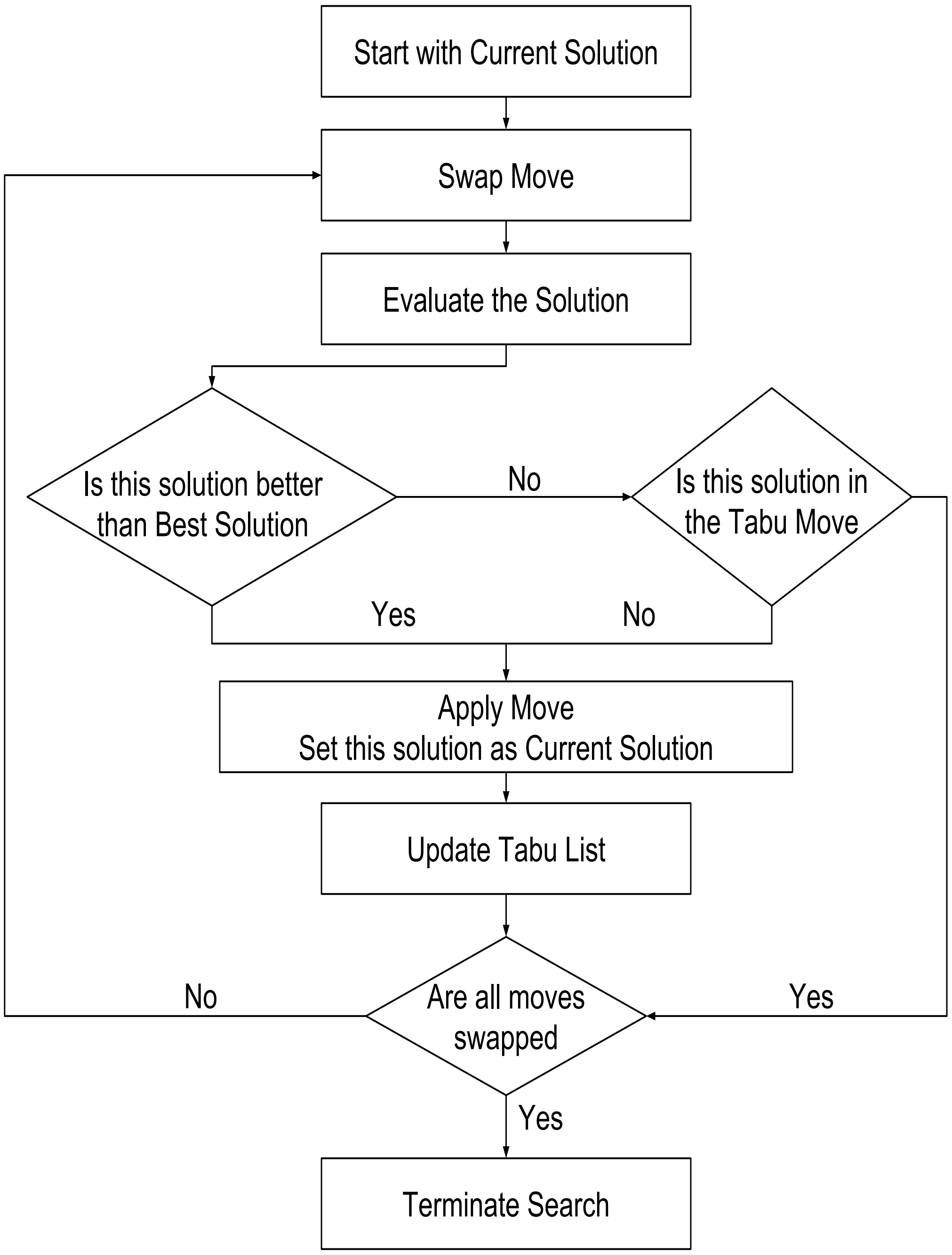

Start with Current Solution
Swap Move
Evaluate the Solution
Is this solution better than Best Solution
No
Is this solution in the Tabu Move
Yes
No
Apply Move
Set this solution as Current Solution
Update Tabu List
No
Are all moves swapped
Yes
Yes
Terminate Search

Figure 3.5: Flow Chart of the Tabu Search Perturbation Mechanism

## 3.5 Update Mechanism for the New Solutions

When the improving solution in the neighbouring space is obtained in the steepest descent or the perturbation procedures of the algorithm, BLS-OpenMP-QAP starts its update mechanism for this improving solution. That is to say, swapping two indexes of a current permutation and creating a different candidate is lead to be the better solution, we update the cost according to the new solution. By means of computing only the difference with the former solution, we can calculate the cost of the improved solution in a very quick way. [15]. In our proposed algorithm, BLS-OpenMP-QAP, this approach is used as a performance improving computation method. Starting from a candidate solution $\beta$, a neighbour solution $\pi$ is taken by changing units $r$ and $s$:

$$\begin{aligned}\pi(k) &= \phi(k)\ \forall k \neq r, s\\ \pi(r) &= \phi(s)\\ \pi(s) &= \phi(r)\end{aligned}$$

If the matrices are symmetrical, the value of a move, $\Delta(\phi, r, s)$ is calculated as :

$$\begin{aligned}\Delta(\phi, r, s) &= \sum_{i=1}^{n}\sum_{j=1}^{n}(a_{ij}b_{\phi(i)\phi(j)} - a_{ij}b_{\pi(i)\pi(j)})\\ &= 2 \cdot \sum_{k\neq r,s}^{n}(a_{sk} - a_{rk})(b_{\phi(s)\phi(k)} - b_{\phi(r)\phi(k)})\end{aligned}$$

To put in a nutshell, the number of computations in the algorithm is an significant criterion while evaluating the performance of heuristics. BLS-OpenMP-QAP algorithm use this fast update mechanism that solely computes the delta part of the novel permutation. Thanks to this technique, we do not need to calculate each novel permutation from scratch which would be a costly process and do its performance would be worse.

# CHAPTER 4

# EXPERIMENTAL RESULTS

In this chapter, we initially explain the environment on which our experiments are performed, then give information about the problem instances used in these experiments. Furthermore, we present the procedure of determining the most efficient parameters for our proposed algorithm BLS-OpenMP-QAP (Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem) in order to obtain an outstanding performance. Consequently, according to these predetermined parameters we carry out our experiments and present the computational results of these experiments. The objective of these experiments is to evaluate the performance of BLS-OpenMP-QAP in the sense of both time efficiency and solution quality. Therefore, we conclude with the general evaluations of the proposed algorithm by these gathered results from the experiments.

## 4.1 Experimental Environment and Setup

In terms of the experimental environment, we arrange our experiments into two categories as CPU and OpenMP implementations since they utilise different hardware capabilities of the personal computer which we use in our experiments.

Main properties of our personal computer are

- Intel Core i7-6700 CPU 3.40 GHz with 4 cores
- 16 GB Memory (RAM)
- Windows 8.1 64-bit Operating System

For the CPU implementations, we solely make use of these main specifications. As for the OpenMP implementations, we additionally take advantages of

- 8 Logical Processors
- OpenMP Library

## 4.2 Problem Instances

By the nature of heuristic algorithms, it is very significant to analyse the performance of recently proposed algorithm by experimenting on well established benchmark instances of the problem and evaluating the results with its corresponding state-of-the-art QAP algorithms in the literature. Therefore, four sets of the problem instances given in the QAP benchmark, QAPLIB, are solved during our experiments [4] in an attempt to measure the performance of our proposed algorithm BLS-OpenMP-QAP. QAPLIB provides a wide range of problem instances that are classified into four classes by Stützle, covering real applications and random problems [25]. These categorized instances indicated by Stützle can be shown as :

- Type 1. *Unstructured, randomly generated instances* have the distance and the flow matrices that are randomly generated based on a uniform distribution.
- Type 2. *Instances with Grid-based distances* contain instances in which the distances are the Manhattan distance between points on a grid, whereas flows are randomly generated.

- Type 3. *Real-life instances* are produced from real-life practical QAP applications.
- Type 4. *Real-life-like instances* are generated instances that are resembled to the real-life QAP problems.

In our experiments, we make use of the problem instances belonging to these four problem types. Namely, for $Type1$, tai20a, tai25a, tai30a, tai35a, tai40a, tai50a, tai60a, tai80a and tai100a; for $Type2$, sko42, sko49, sko56, sko64, sko72, sko81, sko90, sko100a, sko100b, sko100c, sko100d, sko100e and sko100f; for $Type3$, kra30a, kra30b, kra32, ste36a, ste36b, ste36c, esc32b, esc32c, esc32d, esc32e, esc32g, esc32h, esc64a and esc128; for $Type4$, tai20b, tai25b, tai30b, tai35b, tai40b, tai50b, tai60b, tai80b and tai100b are used. (the numbers indicate the problem size)

## 4.3 Setting the Parameters of BLS-OpenMP-QAP Algorithm

In this section, we present the procedure of defining the most efficient parameters for our proposed algorithm BLS-OpenMP-QAP so as to obtain an outstanding performance. Specifically, the parameters that mostly affect our proposed algorithm are listed as the similarity ratio, the number of threads, the number of iterations and the number of multi-starts. Immediately after we define these parameters, we realise other experiments that verifies the quality and the performance of BLS-OpenMP-QAP algorithm by comparing with state-of-the-art algorithms in the literature.

### 4.3.1 Setting the Similarity Ratios of the New Exploration Areas

In order to provide good quality results, we make experiments for finding the most effective similarity ratio on the problem set, $Tai60a$. While realising these experiments we keep the other parameters like the number of threads, the number of multi-starts and the number of iterations at the same degree against the alteration on the value of the similarity ratio. That is to say, while we increase the similarity ratio from the percentage of 10 to 90, the other parameters remain the same. These experiments are initialised with 16 threads and the number of iterations is set to 10000. In addition, we arrange the algorithm to restart 100 times for each experiment.

The computational results of these tests are presented in Table 4.1 and in Figure 4.1. In the plot, *x-axis* represents the similarity ratio value as percentage, whereas *y-axis* represents the average percentage deviation (APD) from the best known solution (BKS) and it is calculated according to the minimum cost reached when the experiment is terminated.

As a consequence of these experiments, we determine that the most appropriate and optimum similarity ratio which makes BLS-OpenMP-QAP algorithm effective is the percentage of thirty. Furthermore, the similarity ratio experiment on the percentage of 30 with tai60a problem instance has reached to the best minimum solution founded so far in the literature.

Table4.1: Similarity Ratio Analysis on Tai60a Problem Instance. APD is the average percentage deviation from the BKS. The times are given in minutes. 30% similarity ratio does not provide any deviation.

| **Ratio (%)** | **# of Threads** | **# of Iterations** | **# of Multi-starts** | **Time (min)** | **APD** |
|---|---|---|---|---|---|
| 10 | 16 | 10000 | 100 | 108.429 | 0.236 |
| 20 | 16 | 10000 | 100 | 107.785 | 0.260 |
| 30 | 16 | 10000 | 100 | 107.891 | 0.000 |
| 40 | 16 | 10000 | 100 | 108.165 | 0.352 |
| 50 | 16 | 10000 | 100 | 107.502 | 0.491 |
| 60 | 16 | 10000 | 100 | 108.017 | 0.369 |
| 70 | 16 | 10000 | 100 | 106.855 | 0.328 |
| 80 | 16 | 10000 | 100 | 106.614 | 0.379 |
| 90 | 16 | 10000 | 100 | 106.700 | 0.417 |

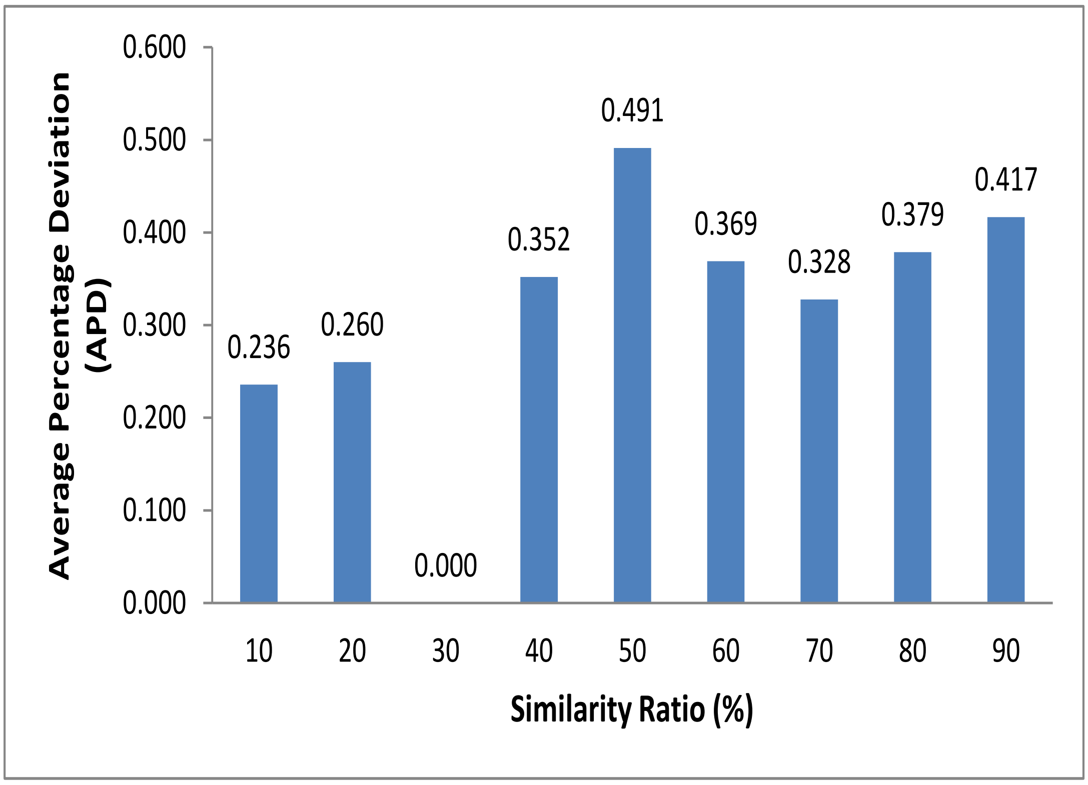


Figure 4.1: Similarity Ratio Analysis on Tai60a Problem Instance. APD is the average percentage deviation from the BKS. 30% similarity ratio does not provide any deviation.

### 4.3.2 Setting the Number of Threads

After we determine the similarity ratio value, we make experiments on the problem set $Tai60a$ in order to ensure the most efficient performance of the threads. While realising these experiments we keep the other parameters like similarity ratio, the number of multi-starts and the number of iterations at the same degree against the alteration on the number of threads. That is to say, while we increase the number of threads from 1 to 32, the other parameters remain the same. These experiments are initialised with thirty percent of the similarity ratio and the number of iterations is set to 10000. In addition, we arrange the algorithm to restart 100 times for each experiment.

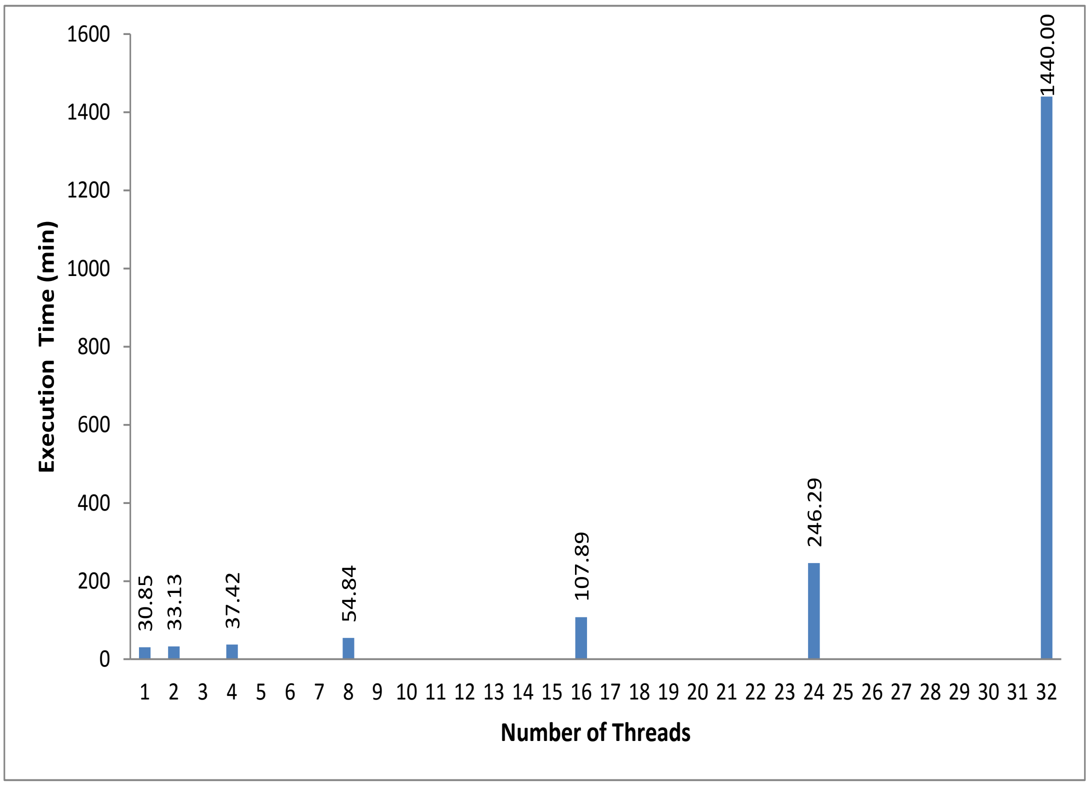


Figure 4.2: Execution Time Analysis of the Number of Threads on Tai60a Problem Instance. With more than the 4 threads, time increases accordingly.

The computational results of these experiments are presented in Table 4.2 and in Figure 4.2. In the plot, *x-axis* represents the number of threads, whereas *y-axis* represents the execution time in minutes for realising each of the experiments.

Table4.2: Execution Time Analysis of the Number of Threads on Tai60a Problem Instance. The times are given in minutes. APD is the average percentage deviation from the BKS. With more than the 4 threads, time increases accordingly.

| **Ratio (%)** | **# of Threads** | **# of Iterations** | **# of Multi-starts** | **Time (min)** | **APD** |
|---|---|---|---|---|---|
| 30 | 1 | 10000 | 100 | 30.85 | 0.686 |
| 30 | 2 | 10000 | 100 | 33.13 | 0.279 |
| 30 | 4 | 10000 | 100 | 37.42 | 0.298 |
| 30 | 8 | 10000 | 100 | 54.84 | 0.331 |
| 30 | 16 | 10000 | 100 | 107.89 | 0.000 |
| 30 | 24 | 10000 | 100 | 246.29 | 0.257 |
| 30 | 32 | 10000 | 100 | $>$ 1440.00 | - |

As a result of these experiments, we determine that the most appropriate and optimum number of threads which makes BLS-OpenMP-QAP algorithm efficient is provided with 16 threads. Furthermore, we have observed that the initialisation of the algorithm with 16 threads can be realised almost at the same time with 8 threads and two times more multi-starts. In this way, the same number of the solutions can be produced and executed by the BLS-OpenMP-QAP algorithm. Therefore, we can also obtain better results from the initialisation of the algorithm with 8 threads like with 16 threads.

Meanwhile, these experiments validate the fact that executing the algorithm on more threads than the computer has would not increase the performance since the logical cores operating on the same physical core share resources with other logical cores. To put in a nutshell, as our personal computer has 4 physical and 8 logical cores, the most efficient way for BLS-OpenMP-QAP algorithm is to work on 8 threads as observed in the experiments. On the other hand, we have preferred to execute our algorithm on 16 threads since it takes a little bit less time when compared to the initialisation with 8 threads and double multi-starts.

### 4.3.3 Setting the Number of Iterations for the BLS

In an effort to obtain best results from BLS-OpenMP-QAP algorithm, we realise experiments for finding the most effective number of iterations on the problem set, $Tai60a$. While making these experiments we keep the other parameters like similarity ratio, the number of threads and the number of multi-starts at the same degree against the alteration on the value of the number of iterations. Namely, as we increase the number of iterations from 100 to 50000, the other parameters remain the same. These experiments are initialised with thirty percent of the similarity ratio and 16 threads. Moreover, we arrange the algorithm to restart 100 times for each experiment.

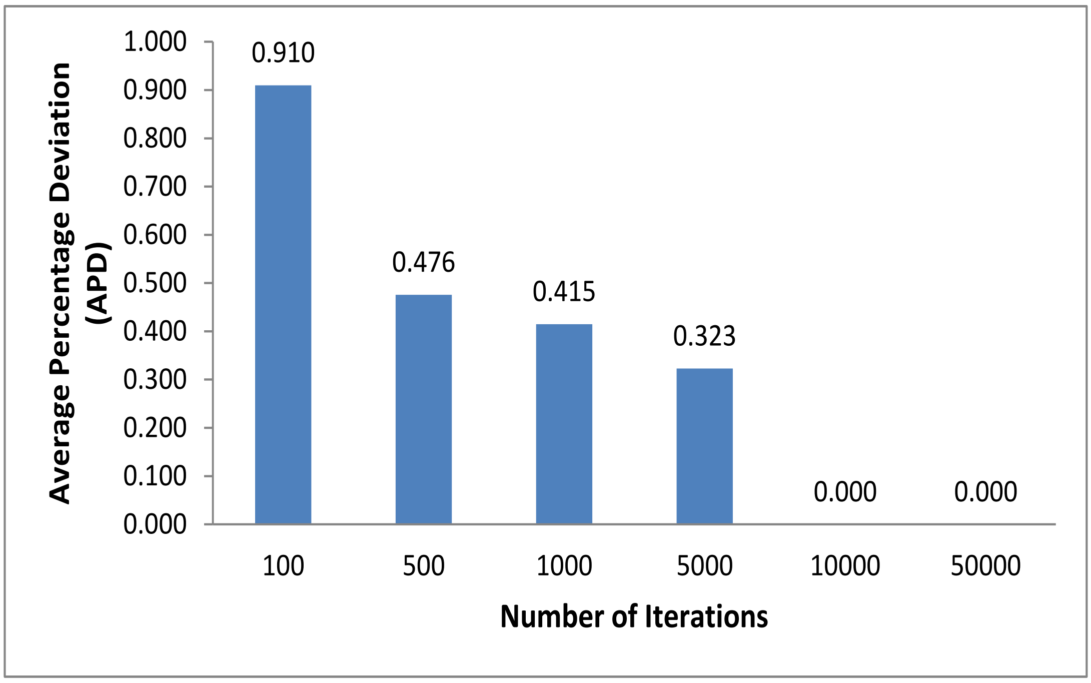


Figure 4.3: APD Analysis with the Number of Iterations on Tai60a Problem Instance. APD is the average percentage deviation from the BKS. Execution with 10000 and 50000 iterations does not provide any deviation.

The computational results of these experiments are presented in Table 4.3 and in Figure 4.3. In the plot, *x-axis* represents the number of iterations, while *y-axis* represents the average percentage deviation (APD) from the BKS and it is calculated according to the minimum cost reached when the experiment is terminated.

Table4.3: APD Analysis with the Number of Iterations on Tai60a Problem Instance. APD is the average percentage deviation from the best known solution. The times are given in minutes. Execution with 10000 and 50000 iterations does not provide any deviation.

| **Ratio (%)** | **# of Threads** | **# of Iterations** | **# of Multi-starts** | **Time (min)** | **APD** |
|---|---|---|---|---|---|
| 30 | 16 | 100 | 100 | 1.097 | 0.910 |
| 30 | 16 | 500 | 100 | 5.116 | 0.476 |
| 30 | 16 | 1000 | 100 | 10.873 | 0.415 |
| 30 | 16 | 5000 | 100 | 53.919 | 0.323 |
| 30 | 16 | 10000 | 100 | 107.891 | 0.000 |
| 30 | 16 | 50000 | 100 | 529.432 | 0.000 |

As a result of these experiments, we define that the most appropriate and optimum number of iterations which makes BLS-OpenMP-QAP algorithm effective is provided with the value of 10000. In spite of the fact that the greater the number of iterations is, the better the results are, we have preferred to initialise the algorithm with the value of 10000 iterations by the reason of ensuring the time efficiency. That is to say, the most significant thing is to reach the solution in the least time as possible and it is observed that better solutions can be obtained with the number of 10000 iterations. Therefore, we determine to set the value of the number of iterations to 10000.

### 4.3.4 Setting the Number of Multi-Starts

With the aim of increasing the performance and obtaining best results, we make experiments to find the most effective number of multi-starts on the problem set, $Tai60a$. As we run these tests we keep the other parameters like the similarity ratio, the number of threads and the number of iterations at the same degree against the alteration on the number of restarts of the algorithm. Therefore, as we increase the number of multi-starts from 10 to 500, the other parameters remain the same. These tests are initialised with 16 threads and the similarity ratio value of thirty percent and the number of iterations is set to 10000.

Table4.4: APD Analysis with the Number of Multi-starts with Tai60a Problem Instance. APD is the average percentage deviation from the best known solution. The times are given in minutes. Execution with 100 multi-starts does not provide any deviation.

| **Ratio (%)** | **# of Threads** | **# of Iterations** | **# of Multi-starts** | **Time (min)** | **APD** |
|---|---|---|---|---|---|
| 30 | 16 | 10000 | 10 | 10.682 | 0.573 |
| 30 | 16 | 10000 | 20 | 21.519 | 0.452 |
| 30 | 16 | 10000 | 40 | 42.942 | 0.369 |
| 30 | 16 | 10000 | 60 | 64.413 | 0.036 |
| 30 | 16 | 10000 | 80 | 86.077 | 0.036 |
| 30 | 16 | 10000 | 100 | 107.891 | 0.000 |

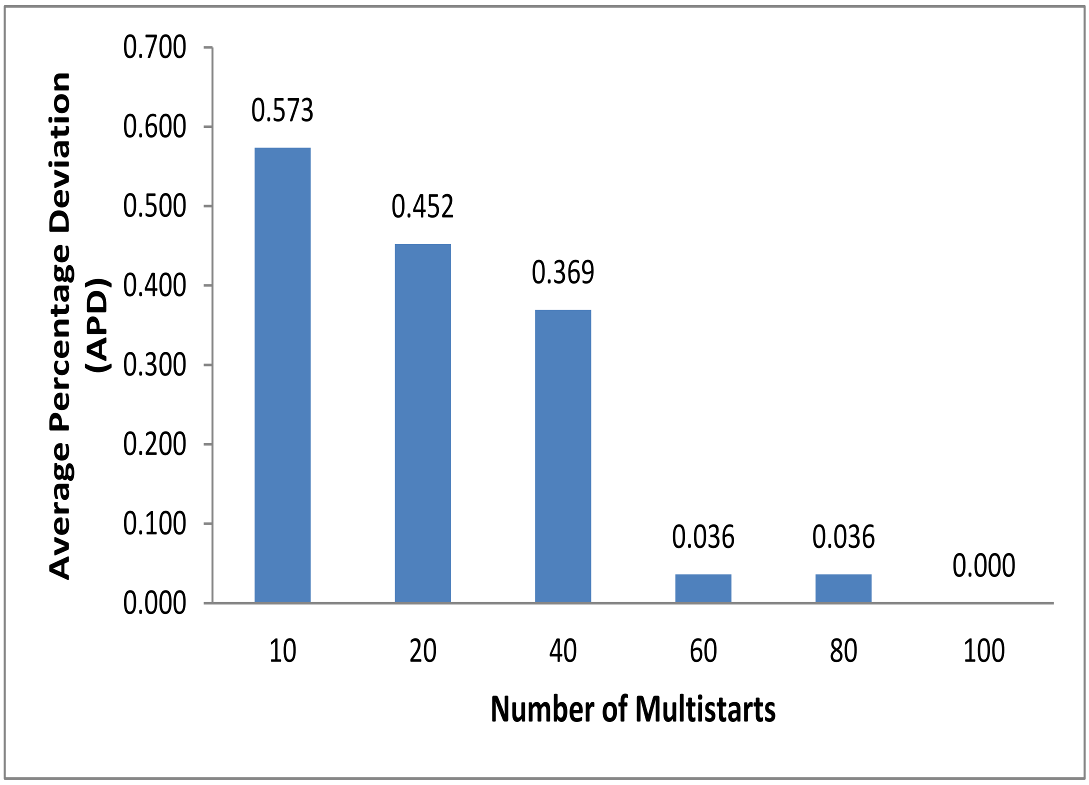


Figure 4.4: APD Analysis with the Number of Multi-starts on Tai60a Problem Instance. APD is the average percentage deviation from the best known solution. Execution with 100 multi-starts does not provide any deviation.

The computational results of these experiments are presented in Table 4.4 and in Figure 4.4. In the plot, *x-axis* represents the number of multi-starts, whereas *y-axis* represents the average percentage deviation (APD) from the BKS and it is calculated according to the minimum cost reached when the experiment is finished.

As a consequence of these experiments, we observe that as the number of multi-starts increases, better results are obtained. Nevertheless, each restart multiply the execution time respectively. Therefore, we determine to assign the number of multi-starts according to the size of the problem. That is to say, we execute BLS-OpenMP-QAP algorithm in less multi-starts for the smaller problems, whereas we increase the number of restarts for the greater ones as the algorithm can reach the optimal solution on the smaller problems sooner than the problems with greater size.

## 4.4 Speed Up Performance of BLS-OpenMP-QAP Algorithm

After we determine all parameters of BLS-OpenMP-QAP algorithm including, similarity ratio, the number of threads, the number of iterations and the number of multi-starts, we measure the speed up performance of BLS-OpenMP-QAP algorithm. To do so, we make experiments on Tai60a and Tai100a problem instances. Therefore, we compare BLS-OpenMP-QAP algorithm with CPU implementations counterpart respectively. In these experiments, we pay attention to keep the execution time until the algorithm is terminated same for each of the implementations.

### 4.4.1 Comparison of OpenMP and CPU Versions of the Proposed Algorithms with Tai60a Problem Instance

In this section, we separate our experiments into four groups according to the algorithm type including the implementations of CPU, Multi-start CPU, Multi-start with Similarity Check CPU and Multi-start with Similarity Check OpenMP. In order to observe the performance difference between these implementations, we make experiments on the same problem set, $Tai60a$. The objective of these experiments is to observe the quality of the solutions obtained from the implementations terminated at the same time.

Initially, we make the experiments for the CPU implementation. In this test, we start algorithm once and we limit the time to approximately 107 minutes as the other following tests. In this CPU implementation, we get the worst result as for the cost of the produced solution. This is because of the fact that we start the algorithm once and it runs in the excessive iteration number, 3255266, that is almost 325 times much more than the following tests; however the algorithm turns into the stagnated situation and cannot escape from this condition in spite of the perturbation mechanism of the algorithm.

Secondly, we realise the experiments for the multi-start CPU implementation. In this experiment, we set the number of iterations parameter to 10000 as same as the following tests and we again limit the time to approximately 107 minutes.

In this multi-start CPU implementation, we get the better result as for the cost of the produced solution than the CPU implementation. That is the result of escaping from the stagnation status thanks to the restart mechanism of the algorithm. In this time limit, algorithm is restarted 331 times; hence, it searches on 331 permutations. We observe that the significant thing is that instead of increasing the number of iterations, searching on different solutions by multi-start mechanism is more effective as better solutions can be obtained from several solutions.

As for the multi-start similarity check CPU implementation, we use a log-based approach for the previously searched permutations of the QAP problem instances in order not to be explored again. Therefore, we set the similarity ratio to the value of 30 percent which is determined before according to the previous experiments. Furthermore, we set the number of iterations parameter to 10000 as same as the other multi-start tests and we again limit the time to approximately 107 minutes. As a consequence, we obtain better result as for the cost of the produced solution than the multi-start CPU without similarity check implementation. That is the result of that we do not search the same solution again if it is explored before thanks to the similarity check mechanism. At the determined time, the algorithm restarts 347 times; therefore, it explores on 347 candidate solutions.

Finally, we make experiments for the multi-start similarity check OpenMP implementation of our proposed algorithm, BLS-OpenMP-QAP. This experiment is initialised with 16 threads, thirty percent of the similarity ratio and the number of iterations is set to 10000. In addition, we arrange the algorithm to restart 100 times for each experiment. This experiment with these predetermined parameters lasts 107 minutes. In an attempt to compare the results at the equal conditions we use this time limit value in previous experiments. Moreover, we also use other parameter values in this experiment like similarity ratio and the number of iterations in previous multi-start experiments. The best result is obtained from this experiment. That is because of the fact that we search on 1600 different solutions (16 threads $X$ 100 multi-starts) that is approximately 5 times more than the others thanks to the initialisations of threads by OpenMP.

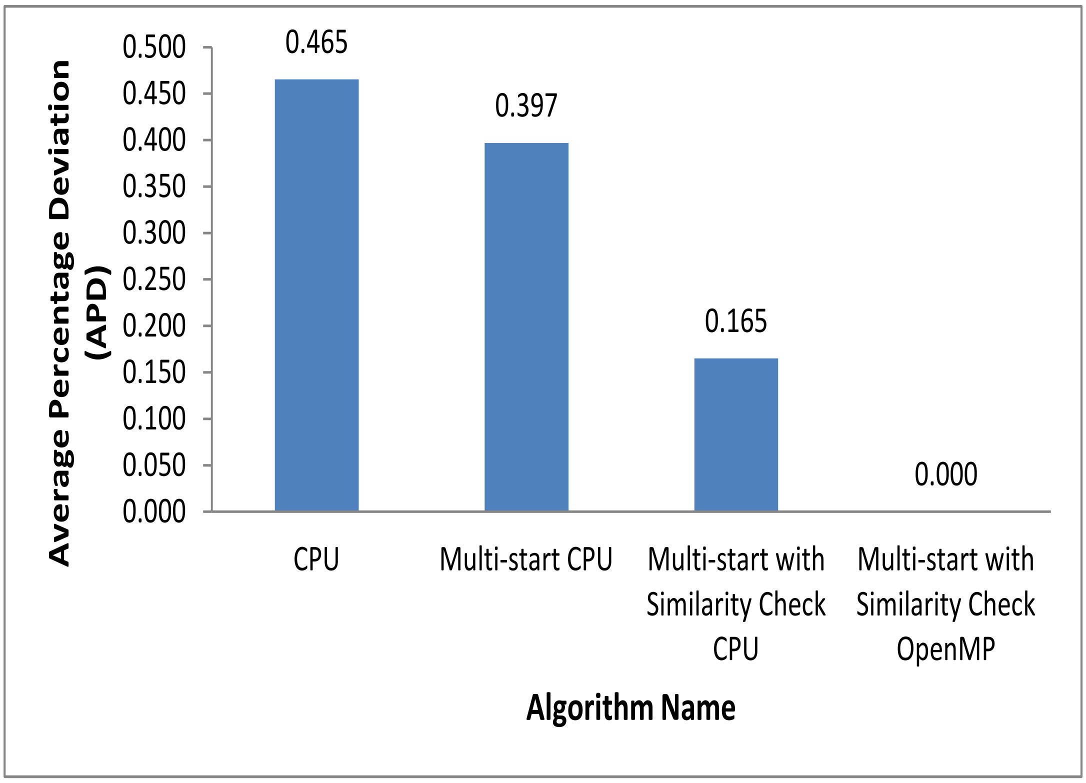


Figure 4.5: Comparison of OpenMP and CPU Versions of the Proposed Algorithms on Tai60a Problem Instance. APD is the average percentage deviation from the best known solution. Implementation of multi-start similarity check OpenMP does not provide any deviation.

Table4.5: Comparison of OpenMP and CPU Versions of the Proposed Algorithms on Tai60a Problem Instance. APD is the average percentage deviation from the best known solution. The times are given in minutes. Implementation of multi-start similarity check OpenMP does not provide any deviation.

| **Algorithm Name** | **Ratio (%)** | **# of Threads** | **# of Iterations** | **# of Multi-starts** | **Time (min)** | **APD** |
|---|---|---|---|---|---|---|
| CPU | - | 1 | 3255266 | 1 | 107.883 | 0.465 |
| Multi-start CPU | - | 1 | 10000 | 331 | 107.884 | 0.397 |
| Multi-start with Similarity Check CPU | 30 | 1 | 10000 | 347 | 108.347 | 0.165 |
| Multi-start with Similarity Check OpenMP | 30 | 16 | 10000 | 100 | 107.891 | 0.000 |

The computational results of these experiments are presented in Table 4.5 and in Figure 4.5. In the plot, *x-axis* represents the type of the algorithm, whereas *y-axis* represents the average percentage deviation (APD) from the BKS and it is calculated according to the minimum cost reached when the experiment is finished.

To put in a nutshell, by taking advantages of threads via OpenMP and applying multi-start mechanism along with using log-based approach for the previously searched permutations of the QAP problem instances in order not to be explored again, we obtain outstandingly best results.

### 4.4.2 Comparison of OpenMP and CPU Versions of the Proposed Algorithms with Tai100a Problem Instance

As the previous section, we again separate our experiments into four groups according to the algorithm type including the implementations of CPU, Multi-start CPU, Multi-start with Similarity Check CPU and Multi-start with Similarity Check OpenMP. In order to observe the quality of these implementations, we make experiments on the same problem set, $Tai100a$ and these executions are terminated at the same time.

First of all, we make the experiments for the CPU implementation. In this test, we start algorithm once and we limit the time to approximately 448 minutes as the following tests. In this CPU implementation, we get the worst result as for the cost of the produced solution. This is because of the fact that we start the algorithm once and it runs in the excessive iteration number, 2736285, that is almost 273 times much more than the following tests; however the algorithm turns into the stagnated situation and cannot escape from this condition in spite of the perturbation mechanism of the algorithm.

As for the multi-start CPU implementation, we set the number of iterations parameter to 10000 as same as the following tests and we again limit the time to approximately 448 minutes. In this multi-start CPU implementation, we get the better result as for the cost of the produced solution than the CPU implementation. That is the result of escaping from the stagnation status thanks to the restart mechanism of the algorithm. In this time limit, algorithm is restarted 287 times; hence, it searches on 287 permutations. We observe that the significant thing is that instead of increasing the number of iterations, searching on different solutions by multi-start mechanism is more effective as better solutions can be obtained from several solutions.

As for the multi-start similarity check CPU implementation, we use a log-based approach for the previously searched permutations of the QAP problem instances in order not to be explored again. Therefore, we set the similarity ratio to the value of 30 percent which is determined before according to the previous experiments. Furthermore, we set the number of iterations parameter to 10000 as same as the other multi-start tests and we again limit the time to approximately 448 minutes. As a consequence, we obtain better result as for the cost of the produced solution than the multi-start CPU without similarity check implementation. That is the result of that we do not search the same solution again if it is explored before thanks to the similarity check mechanism. At the determined time, the algorithm restarts 278 times; therefore, it explores on 347 candidate solutions.

In conclusion, we make experiments for the multi-start similarity check OpenMP implementation of our proposed algorithm, BLS-OpenMP-QAP. This experiment is initialised with 16 threads, thirty percent of the similarity ratio and the number of iterations is set to 10000. In addition, we arrange the algorithm to restart 100 times for each experiment. This experiment with these predetermined parameters lasts 448 minutes. In order to compare the results at the equal conditions we use this time limit value in previous experiments. Moreover, we use other parameter values like similarity ratio and the number of iterations of this experiment in previous multi-start experiments. The best result is obtained from this experiment. That is because of the fact that we search on 1600 different solutions (16 threads $X$ 100 multi-start) that is approximately 5 times more than the others thanks to the initialisations of threads by OpenMP.

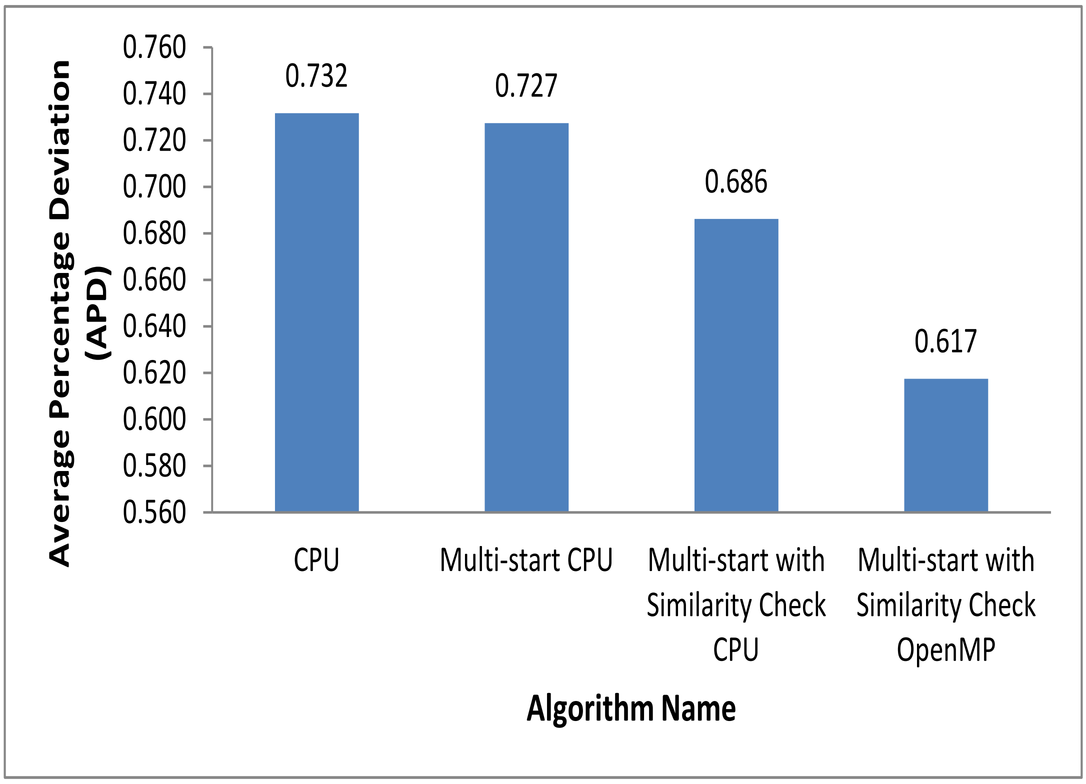


Figure 4.6: Comparison between OpenMP and CPU versions on Tai100a Problem Instance. APD is the average percentage deviation from the best known solution. Implementation of multi-start similarity check OpenMP provides the minimum deviation among them.

Table4.6: Comparison between OpenMP and CPU versions on Tai100a Problem Instance. APD is the average percentage deviation from the best known solution. The times are given in minutes. Implementation of multi-start similarity check OpenMP provides the minimum deviation among them.

| **Algorithm Name** | **Ratio (%)** | **# of Threads** | **# of Iterations** | **# of Multi-starts** | **Time (min)** | **APD** |
|---|---|---|---|---|---|---|
| CPU | - | 1 | 2736285 | 1 | 448.517 | 0.732 |
| Multi-start CPU | - | 1 | 10000 | 287 | 448.520 | 0.727 |
| Multi-start with Similarity Check CPU | 30 | 1 | 10000 | 278 | 451.170 | 0.686 |
| Multi-start with Similarity Check OpenMP | 30 | 16 | 10000 | 100 | 448.529 | 0.617 |

The computational results of these experiments are presented in Table 4.6 and in Figure 4.6. In the plot, *x-axis* represents the type of the algorithm, whereas *y-axis* represents the average percentage deviation (APD) from the BKS and it is calculated according to the minimum cost reached when the experiment is finished.

As a consequence of taking advantages of threads via OpenMP and applying multi-start mechanism along with using log-based approach for the previously searched permutations of the QAP problem instances in order not to be explored again, we obtain outstandingly best results.

## 4.5 Overhead of Similarity Checking Procedure

Whereas similarity checking mechanism makes a great contribution to find the best solution by using a log-based approach for the previously searched permutations of the QAP problem instances in order not to be explored again, it has a trade-of as for the execution time for controlling the similarity check constraint. In order to observe this overhead, we make experiments on the problem set, $Tai60a$. As we run these tests we keep all the parameters like the similarity ratio, the number of threads, the number of iterations and the number of multi-starts at the same degree. Therefore, these tests are initialised with 16 threads, the similarity ratio value of thirty percent and 100 times multi-starts and the number of iterations is set to 10000.

In these experiments, we observe that the time while controlling the similarity check constraint is approximately 1 minute. Indeed, it corresponds to the % 0.8 of all execution time. However, this time overhead extends as the problem size gets larger and the number of multi-start increases.

## 4.6 Comparison of State-of-the-art Algorithms with BLS-OpenMP-QAP

In this section, we compare proposed BLS-OpenMP-QAP algorithm with the recent state-of-the-art algorithms counterpart in terms of the solution quality and the computational efficiency. As a matter of fact they provide good results by using classical meta-heuristics that is why we choose them to compare with our BLS-OpenMP-QAP algorithm.

In these experiments for comparison with the state-of-the-art algorithms, 59 problem instances given in the QAP benchmark, QAPLIB, are solved during the experiments [53]. Most of the state-of-the-art QAP algorithms make use of QAPLIB in order to evaluate the quality of their algorithms and execute them on these problem instances. Hence, QAPLIB provides a fair ground in order to make comparisons with the other algorithms in the literature.

The state-of-the-art algorithms in the literature we compare with our BLS-OpenMP-QAP algorithm are Multi-Start TS Algorithm JRG-DivTS by James et al. [12], Iterated Tabu Search (ITS) by Misevicius [61], Self Controlling Tabu Search (SC-Tabu) by Fescioglu-Unver and Kokar [62], Ant Colony Optimization GA/Local Search Hybrid ACO/GA/LS by Tseng and Liang [64], GA Hybrid with Concentric TS Operator GA/C-TS and GA Hybrid with a Strict Descent Operator GA/SD by Drezner [60], Parallel Hybrid Algorithm (PHA) by Tosun [71], Memetic search for the QAP (BMA) by Benlic and Hao [67], Great Deluge and Tabu Search (GDA) by Acan and Unveren [63] are selected to be compared with BLS-OpenMP-QAP algorithm during our experiments.

The most significant impact on the performance of a heuristic algorithm is to set the right parameters in order to obtain results efficiently with the best quality. Therefore, we make experiments for the parameter settings and we use these settings while realising our experiments. Table 4.7 presents the parameters used during the tests.

Table4.7: Parameter Settings for the BLS-OpenMP-QAP Algorithm

| **Problem Instances** | **Parameter** | **Setting** |
|---|---|---|
| All | Similarity Ratio | 30% |
| Type 1 | Number of Threads | 16 |
| Type 2-4 and Nug | Number of Threads | 8 |
| All | Number of Iterations | 10000 |
| All | Number of Multi-starts | 100 |

Table4.8: Optimal Solutions Found by the BLS-OpenMP-QAP Algorithm on Nug Problem Instances. APD is the average percentage deviation from the best known solution. BPD is the best percentage deviation from the best known solution. The times are given in seconds. All of the Nug problem instances are solved exactly.

| Instance | BKS | APD | BPD | Time(sec.) |
|---|---|---|---|---|
| nug14 | 1014 | 0 | 0 | 0.016 |
| nug15 | 1150 | 0 | 0 | 0.016 |
| nug16a | 1610 | 0 | 0 | 0.610 |
| nug16b | 1240 | 0 | 0 | 0.015 |
| nug17 | 1732 | 0 | 0 | 1.093 |
| nug18 | 1930 | 0 | 0 | 0.687 |
| nug20 | 2570 | 0 | 0 | 0.046 |
| nug21 | 2438 | 0 | 0 | 0.484 |
| nug22 | 3596 | 0 | 0 | 0.015 |
| nug24 | 3488 | 0 | 0 | 0.046 |
| nug25 | 3744 | 0 | 0 | 1.640 |
| nug27 | 5234 | 0 | 0 | 0.016 |
| nug28 | 5166 | 0 | 0 | 1.031 |
| nug30 | 6124 | 0 | 0 | 1.156 |

Tables from 4.8 to 4.12 present the results of our experiments with respect to the four categories specified by Stützle respectively [25]. Moreover, the best performing first three results are given in bold face.

BLS-OpenMP-QAP algorithm is executed on 59 problem instances of the QAPLIB benchmark and optimal results are obtained for 57 of the instances. The overall deviation for the problem instances is obtained as 0.019% on the average. The results are given with respect to the Best Known Solution (BKS) of the problem instances in the QAPLIB.

Table4.9: Comparison of the BLS-OpenMP-QAP Algorithm with State-of-the-art Algorithms on Type-1 Problem Instances. APD is the average percentage deviation from the best known solution. The times are given in minutes. 7 of the Type 1 problem instances are solved exactly by BLS-OpenMP-QAP.

| | | BLS-OpenMP-QAP | | JRG-DivTS | | ITS | | SC-Tabu | | ACO/GA/LS | | GDA | | BMA | | PHA | | CPTS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | BKS | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time |
| tai20a | 703482 | **0** | 0.09 | **0** | 0.2 | **0** | 0.0 | 0.246 | 0.01 | - | - | **0** | 2.10 | **0** | 0.0 | **0** | 0.37 | **0** | 0.1 |
| tai25a | 1167256 | **0** | 0.13 | **0** | 0.6 | **0** | 0.1 | 0.239 | 0.03 | - | - | **0** | 15.82 | **0** | 0.0 | **0** | 0.55 | **0** | 0.3 |
| tai30a | 1818146 | **0** | 0.20 | **0** | 1.3 | **0** | 0.2 | 0.154 | 0.07 | 0.341 | 1.4 | 0.091 | 20.29 | **0** | 0.0 | **0** | 0.97 | **0** | 1.6 |
| tai35a | 2422002 | **0** | 0.32 | **0** | 4.4 | **0** | 0.5 | 0.280 | 0.18 | 0.487 | 3.5 | 0.153 | 24.99 | **0** | 0.0 | **0** | 1.28 | **0** | 2.3 |
| tai40a | 3139370 | **0** | 32.16 | 0.222 | 5.2 | 0.22 | 1.3 | 0.561 | 0.20 | 0.593 | 13.1 | 0.261 | 27.78 | **0.059** | 8.1 | **0** | 10.60 | 0.148 | 3.5 |
| tai50a | 4938796 | **0** | 68.16 | 0.725 | 10.2 | 0.41 | 5.5 | 0.889 | 0.23 | 0.901 | 29.7 | 0.276 | 41.14 | **0.131** | 42.0 | **0** | 12.74 | 0.440 | 10.3 |
| tai60a | 7205962 | **0** | 107.89 | 0.718 | 25.7 | 0.45 | 12.5 | 0.940 | 0.41 | 1.068 | 58.5 | 0.448 | 78.86 | **0.144** | 67.5 | **0** | 19.58 | 0.476 | 26.4 |
| tai80a | 13499184 | **0.504** | 235.95 | 0.753 | 52.7 | **0.36** | 60.0 | 0.648 | 1.01 | 1.178 | 152.2 | 0.832 | 111.34 | **0.426** | 65.8 | 0.644 | 39.97 | 0.570 | 94.8 |
| tai100a | 21052466 | 0.617 | 448.53 | 0.825 | 142.1 | **0.30** | 200.0 | 0.977 | 1.99 | 1.115 | 335.6 | 0.874 | 138.32 | **0.405** | 44.1 | **0.537** | 71.89 | 0.558 | 261.2 |
| Average | | **0.125** | 99.27 | 0.360 | 26.9 | 0.19 | 31.1 | 0.548 | 0.46 | 0.812 | 84.9 | 0.326 | 51.18 | **0.129** | 25.3 | **0.131** | 17.55 | 0.244 | 44.5 |

Table4.10: Comparison of the BLS-OpenMP-QAP Algorithm with State-of-the-art Algorithms on Type-2 Problem Instances. APD is the average percentage deviation from the best known solution. The times are given in minutes. All of the Type 2 problem instances are solved exactly by BLS-OpenMP-QAP.

| | | BLS-OpenMP-QAP | | JRG-DivTS | | SC-Tabu | | ACO/GA/LS | | GA/SD | | GA/C-TS | | GDA | | BMA | | PHA | | CPTS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | BKS | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time |
| sko42 | 15812 | **0** | 0.37 | **0** | 4.0 | 0.016 | 0.14 | **0** | 0.7 | 0.014 | 0.16 | **0** | 1.2 | **0** | 2.24 | **0** | 0.002 | **0** | 1.60 | **0** | 5.3 |
| sko49 | 23386 | **0** | 0.56 | 0.008 | 9.6 | 0.085 | 0.22 | 0.056 | 7.6 | 0.107 | 0.28 | 0.009 | 2.1 | 0.005 | 3.85 | **0** | 0.01 | **0** | 4.03 | **0** | 11.4 |
| sko56 | 34458 | **0** | 0.81 | 0.002 | 13.2 | 0.069 | 0.34 | 0.012 | 9.1 | 0.054 | 0.42 | 0.001 | 3.2 | 0.001 | 14.72 | **0** | 0.02 | **0** | 16.24 | **0** | 21.0 |
| sko64 | 48498 | **0** | 1.20 | **0** | 22.0 | 0.074 | 0.51 | 0.004 | 17.4 | 0.051 | 0.73 | **0** | 5.9 | **0** | 29.39 | **0** | 0.03 | **0** | 23.10 | **0** | 42.9 |
| sko72 | 66256 | **0** | 1.82 | 0.006 | 38.0 | 0.159 | 0.73 | 0.018 | 70.8 | 0.112 | 0.93 | 0.014 | 8.4 | 0.007 | 37.99 | **0** | 3.50 | **0** | 33.63 | **0** | 69.6 |
| sko81 | 90998 | **0** | 2.40 | 0.016 | 56.4 | 0.076 | 1.05 | 0.025 | 112.3 | 0.087 | 1.44 | 0.014 | 13.3 | 0.019 | 57.14 | **0** | 4.30 | **0** | 39.87 | **0** | 121.4 |
| sko90 | 115534 | **0** | 3.25 | 0.026 | 89.6 | 0.134 | 1.44 | 0.042 | 92.1 | 0.139 | 2.31 | 0.011 | 22.4 | 0.031 | 93.83 | **0** | 15.30 | **0** | 40.53 | **0** | 193.7 |
| sko100a | 152002 | **0** | 29.80 | 0.027 | 129.2 | 0.094 | 1.99 | 0.021 | 171.0 | 0.114 | 3.42 | 0.018 | 33.6 | 0.029 | 153.17 | **0** | 22.30 | **0** | 41.72 | **0** | 304.8 |
| sko100b | 153890 | **0** | 8.51 | 0.008 | 106.6 | 0.059 | 1.99 | 0.012 | 192.4 | 0.096 | 3.47 | 0.011 | 34.1 | 0.015 | 164.27 | **0** | 6.50 | **0** | 42.32 | **0** | 309.6 |
| sko100c | 147862 | **0** | 4.28 | 0.006 | 126.7 | 0.039 | 1.99 | 0.005 | 220.6 | 0.075 | 3.22 | 0.003 | 33.8 | 0.013 | 154.51 | **0** | 12.00 | **0** | 42.19 | **0** | 316.1 |
| sko100d | 149576 | **0** | 12.87 | 0.027 | 123.5 | 0.091 | 1.99 | 0.029 | 209.2 | 0.137 | 3.45 | 0.049 | 33.9 | 0.017 | 148.86 | 0.006 | 20.90 | **0** | 41.91 | **0** | 309.8 |
| sko100e | 149150 | **0** | 4.33 | 0.009 | 108.8 | 0.037 | 1.99 | 0.002 | 208.1 | 0.071 | 3.31 | 0.002 | 30.7 | 0.016 | 146.15 | **0** | 11.90 | **0** | 42.48 | **0** | 309.1 |
| sko100f | 149036 | **0** | 17.11 | 0.023 | 110.3 | 0.122 | 1.99 | 0.034 | 210.9 | 0.148 | 3.55 | 0.032 | 35.7 | 0.013 | 153.38 | **0** | 23.00 | **0** | 41.98 | 0.003 | 310.3 |
| Average | | **0** | 6.72 | 0.012 | 72.1 | 0.081 | 1.26 | 0.020 | 117.1 | 0.093 | 2.05 | 0.013 | 19.9 | 0.013 | 89.19 | **0** | 9.21 | **0** | 31.66 | **0** | 178.8 |

Table4.11: Comparison of the BLS-OpenMP-QAP Algorithm with State-of-the-art Algorithms on Type-3 Problem Instances. APD is the average percentage deviation from the best known solution. The times are given in minutes. All of the Type 3 problem instances are solved exactly by BLS-OpenMP-QAP.

| | | BLS-OpenMP-QAP | | ACO/GA/LS | | ITS | | SC-TABU | | GDA | | BMA | | PHA | | CPTS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | BKS | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time |
| kra30a | 88900 | **0** | 0.0037 | **0** | 0.1208 | 0.01 | 0.025 | 0.714 | 0.048 | **0** | 0.947 | **0** | 0.040 | - | - | - | - |
| kra30b | 91420 | **0** | 0.0330 | **0** | 0.6703 | **0** | 0.025 | 0.178 | 0.048 | **0** | 1.392 | **0** | 0.020 | - | - | - | - |
| kra32 | 88700 | **0** | 0.0122 | - | - | - | - | - | - | **0** | 0.626 | **0** | 0.018 | - | - | - | - |
| ste36a | 9526 | **0** | 0.1990 | **0** | 0.6247 | 0.04 | 0.093 | 0.761 | 0.085 | **0** | 2.173 | **0** | 0.082 | **0** | 1.37 | - | - |
| ste36b | 15852 | **0** | 0.1170 | **0** | 0.1062 | **0** | 0.092 | 0.761 | 0.085 | **0** | 0.406 | **0** | 0.022 | - | - | - | - |
| ste36c | 8239110 | **0** | 0.1821 | **0** | 0.4547 | **0** | 0.092 | 0.761 | 0.085 | **0** | 1.395 | **0** | 0.043 | **0** | 1.37 | **0** | 2.5 |
| esc32b | 168 | **0** | 0.0004 | **0** | 0.0070 | **0** | 0.032 | - | - | **0** | 0.598 | **0** | 0.002 | - | - | - | - |
| esc32c | 642 | **0** | 0.0003 | **0** | 0.0005 | **0** | 0.035 | - | - | **0** | 0.005 | **0** | 0.002 | - | - | - | - |
| esc32d | 200 | **0** | 0.0003 | **0** | 0.0028 | **0** | 0.033 | - | - | **0** | 0.026 | **0** | 0.002 | - | - | - | - |
| esc32e | 2 | **0** | 0.0003 | **0** | 0.0003 | **0** | 0.032 | - | - | **0** | 0.004 | **0** | 0.002 | - | - | - | - |
| esc32g | 6 | **0** | 0.0002 | **0** | 0.0003 | **0** | 0.032 | - | - | **0** | 0.005 | **0** | 0.000 | - | - | - | - |
| esc32h | 438 | **0** | 0.0024 | **0** | 0.0048 | **0** | 0.035 | - | - | **0** | 0.005 | **0** | 0.002 | - | - | - | - |
| esc64a | 116 | **0** | 0.0012 | **0** | 0.0043 | **0** | 0.183 | 1.207 | 0.001 | **0** | 0.038 | **0** | 0.003 | - | - | - | - |
| esc128 | 64 | **0** | 0.0052 | **0** | 0.0378 | 0.01 | 1.000 | 14.271 | 0.124 | **0** | 2.135 | **0** | 0.022 | - | - | - | - |
| Average | | **0** | 0.0398 | **0** | 0.1565 | **0** | 0.131 | 2.665 | 0.068 | **0** | 0.697 | **0** | 0.018 | **0** | 1.37 | **0** | 2.5 |

Table4.12: Comparison of the BLS-OpenMP-QAP Algorithm with State-of-the-art Algorithms on Type-4 Problem Instances. APD is the average percentage deviation from the best known solution. The times are given in minutes. All of the Type 4 problem instances are solved exactly by BLS-OpenMP-QAP.

| | | BLS-OpenMP-QAP | | JRG-DivTS | | ITS | | ACO/GA/LS | | SC-TABU | | GDA | | BMA | | PHA | | CPTS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | BKS | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time | APD | time |
| tai20b | 122455319 | **0** | 0.072 | **0** | 0.2 | **0** | 0.01 | - | - | 0.335 | 0.003 | **0** | 0.074 | **0** | 0.000 | **0** | 0.37 | **0** | 0.1 |
| tai25b | 344355646 | **0** | 0.023 | **0** | 0.5 | **0** | 0.02 | - | - | 0.702 | 0.010 | **0** | 0.327 | **0** | 0.000 | **0** | 0.57 | **0** | 0.4 |
| tai30b | 637117113 | **0** | 0.221 | **0** | 1.3 | 0.010 | 0.04 | **0** | 0.3 | 0.313 | 0.035 | **0** | 1.148 | **0** | 0.000 | **0** | 0.81 | **0** | 1.2 |
| tai35b | 283315445 | **0** | 0.295 | **0** | 2.4 | 0.020 | 0.08 | **0** | 0.3 | - | - | **0** | 6.387 | **0** | 0.000 | **0** | 1.11 | **0** | 2.4 |
| tai40b | 637250948 | **0** | 0.303 | **0** | 3.2 | 0.010 | 0.21 | **0** | 0.6 | 0.219 | 0.092 | **0** | 4.877 | **0** | 0.003 | **0** | 1.57 | **0** | 4.5 |
| tai50b | 458821517 | **0** | 0.707 | **0** | 8.8 | 0.020 | 0.55 | **0** | 2.9 | 0.281 | 0.235 | 0.005 | 10.249 | **0** | 1.200 | **0** | 5.82 | **0** | 13.8 |
| tai60b | 608215054 | **0** | 18.622 | **0** | 17.1 | 0.040 | 1.07 | **0** | 2.8 | 0.886 | 0.415 | **0** | 33.639 | **0** | 5.200 | **0** | 9.49 | **0** | 30.4 |
| tai80b | 818415043 | **0** | 218.053 | 0.006 | 58.2 | 0.230 | 3.00 | **0** | 60.3 | 0.798 | 1.004 | 0.025 | 0.005 | **0** | 31.300 | **0** | 27.70 | **0** | 110.9 |
| tai100b | 1185996137 | **0** | 160.797 | 0.056 | 118.9 | 0.140 | 6.67 | 0.010 | 698.9 | 0.553 | 1.988 | 0.028 | 72.601 | **0** | 13.600 | **0** | 42.50 | 0.001 | 241.0 |
| Average | | **0** | 44.344 | 0.007 | 23.4 | 0.052 | 1.29 | 0.001 | 109.4 | 0.511 | 0.473 | 0.006 | 14.367 | **0** | 5.700 | **0** | 9.99 | **0** | 45.0 |

The overall deviation percentage of the BLS-OpenMP-QAP algorithm for the problems classified as Type 1, 2, 3 and 4 by Stützle is 0.025%. With this result, BLS-OpenMP-QAP algorithm is among the best three algorithms that are presented in this study. That is to say, we obtain an outstandingly high performance result and this also proves the robustness of the BLS-OpenMP-QAP algorithm when compared with the other state-of-the-art algorithms in the literature. As a result, we observe that the BLS-OpenMP-QAP algorithm is capable of handling larger number of heuristics and can improve its solution quality by this way.

As for Type 1, BLS-OpenMP-QAP is the best algorithm with respect to the performance of the other algorithms. It has only 0.125% deviation from the BKS; whereas, other two algorithms (BMA and PHA) have 0.129% and 0.131% respectively. The deviations of BLS-OpenMP-QAP for the well-known problem instances $tai60a$ and $tai80a$ are among the best three performing algorithms by the 0.0% and 0.504% deviations respectively. As for the remaining problems, Type 2, 3 and 4, BLS-OpenMP-QAP is among the best performing algorithms in the literature having 0.0% deviation with respect to the BKS results reported in the QAPLIB. In these problems, the algorithms BMA, PHA and CPTS also perform 0.0% deviation.

As a consequence of these experiments, we also verify the fact that multi-starting a heuristic from a different exploration point is an efficient technique. Indeed, it has been used for most of the state-of-the-art algorithms [11] [12]. In our implementation, it helps BLS-OpenMP-QAP algorithm escape from stucking into local optima by restarting the exploration with a new candidate solution by smart beginning mechanism. As a matter of fact, we can observe its positive effect on the solution quality when the number of multi-starts is increased for $tai60a$ problem instance in Figure 4.4.

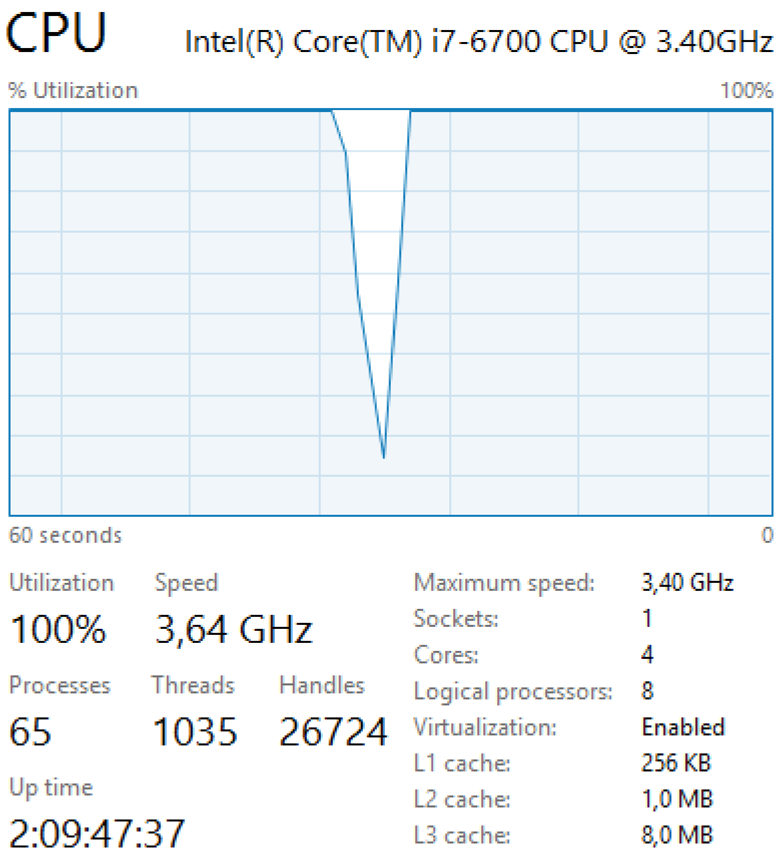
Another significant aspect about the heuristics is the speed-up and scalability obtained by the proposed algorithm. BLS-OpenMP-QAP algorithm takes advantages of multi-core architectures. That is to say, it makes use of the threads by the help of OpenMP library. Therefore, the workload is shared among these threads so that high performance is obtained. Moreover, the total execution time of the algorithm is reduced proportionally with the predetermined number of threads. The more number of threads is initialised, the more the execution time decreases with regards to the number of threads. The performance of the algorithm can be further improved by initialising larger number of threads. This will reduce the execution time and give a higher chance to improve the number of multi-starts and search the landscape of the QAP more effectively.

Consequently, the greatest contribution of BLS-OpenMP-QAP algorithm is provided by multi-starts with similarity check mechanism. In this way, we improve the candidate solutions by using a log-based similarity checking approach for the previously searched permutations of the QAP problem instances. In addition to this, BLS-OpenMP-QAP makes use of the parallel computation paradigm of the contemporary multi-core architectures using OpenMP programming paradigm. Thanks to the parallel computation of the BLS-OpenMP-QAP, we obtain the results in less time and cost effective way.

## 4.7 CPU Utilisation

While we execute our experiments, we also follow the changes in CPU utilization in the meantime. The result from this observation demonstrates that the CPU utilization raises by increasing the number of threads working simultaneously. As a matter of fact, CPU utilization values reach its maximum value - 100% when the threads are all in execution process and work in parallel. In other words, all processes of BLS-OpenMP-QAP algorithm apart from generating candidate solutions, improving local optima, determining jump magnitude and perturbation, are executed concurrently on the threads via OpenMP.

The graph in Figure 4.7 follows a particular trend in the whole experiments. That is to say, CPU utilisation raises when the parallel computation is initialised and the number of threads working simultaneously is increased.



| | | |
|---|---|---|
| Utilization | Speed | |
| 100% | 3,64 GHz | |
| Processes | Threads | Handles |
| 65 | 1035 | 26724 |
| Up time | | |
| 2:09:47:37 | | |

| | |
|---|---|
| Maximum speed: | 3,40 GHz |
| Sockets: | 1 |
| Cores: | 4 |
| Logical processors: | 8 |
| Virtualization: | Enabled |
| L1 cache: | 256 KB |
| L2 cache: | 1,0 MB |
| L3 cache: | 8,0 MB |

Figure 4.7: CPU Utilization

# CHAPTER 5

# CONCLUSION

QAP is one of the most challenging NP-Hard combinatorial optimization problems with its several real life applications. The paradigm of parallel computing is used to speed up the scientific, engineering, consumer, and similar computation intensive applications. Our proposed algorithm named BLS-OpenMP-QAP (Breakout Local Search Algorithm with Open Multi-Processing for Quadratic Assignment Problem) takes advantage of parallel programming by OpenMP. This provides a more efficient use of the available computing power for the search. Furthermore, the BLS-OpenMP-QAP algorithm has a similarity check mechanism as a diversification technique by making use of the adapted Levenshtein Distance (LD). Therefore, we ensure not to search again on the previously explored permutations of the QAP problem instances.

Although there are several sophisticated techniques, the time for obtaining good quality results diversifies from several minutes for small or medium size problem sets, to a few hours for the larger ones. When the results are compared with the existing state-of-the-art heuristics, the proposed BLS-OpenMP-QAP algorithm can be considered as one of the best algorithms in the literature in terms of the computation time and the solution quality. The results showed that parallel computation with OpenMP has an outstanding performance due to speeding-up the fitness evaluations as many as the number of the processors/threads. In terms of the quality of the solutions, BLS-OpenMP-QAP is capable of delivering good quality solutions by reaching often optimal or the best known solutions. The BLS-OpenMP-QAP algorithm is experimented on 59 problem instances of the QAP library benchmark and shown to be able to solve 57 of the instances exactly and the overall deviation for the algorithm is obtained as 0.019% on the average. As a matter of fact, this rewarding performance is derived from the simultaneous exploration of the different search area which is selected intelligently based on the similarity checking mechanism provided by the adapted LD.

The similarity checking mechanism makes a great contribution to find the best solution by using a log-based approach for the previously searched permutations of the QAP problem instances in order not to be explored again. However, it has a trade-of as for the execution time for controlling the similarity check threshold. As a future work, we intend to build the hash map from the previously explored permutations. Therefore, the explored solutions will be stored in this hash map data structure which is very efficient in storing and retrieving the data. Furthermore, we plan to execute BLS-OpenMP-QAP algorithm on hundreds of processors with several multi-starts in order to obtain higher quality solutions in shorter time.

# REFERENCES

[1] R. Battiti and G. Tecchiolli, "The Reactive Tabu Search," *INFORMS Journal on Computing*, vol. 6, no. 2, pp. 126–140, 1994.

[2] U. Benlic and J.-k. Hao, "Breakout local search for the quadratic assignment problem," *Applied Mathematics and Computation*, vol. 219, no. 9, pp. 4800–4815, 2013.

[3] M. R. Wilhelm and T. L. Ward, "Solving Quadratic Assignment Problems by 'Simulated Annealing'," *IIE Transactions*, vol. 19, no. 1, pp. 107–119, 1987.

[4] R. Burkard, S. Karisch, and F. Rendl, "QAPLIB - a Quadratic Assignment Problem Library," *European Journal of Operational Research*, vol. 55, no. 1, pp. 115–119, 1991.

[5] T. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica: Journal of the Econometric ...*, vol. 25, no. 1, pp. 53–76, 1957.

[6] T. U. of Wisconsin, "Neos." Website, 2016. `http://www.neos-guide.org/content/quadratic-assignment-problem`.

[7] T. Dokeroglu and A. Cosar, "A novel multistart hyper-heuristic algorithm on the grid for the quadratic assignment problem," *Engineering Applications of Artificial Intelligence*, vol. 52, pp. 10–25, 2016.

[8] D. T. Connolly, "An improved annealing scheme for the QAP," *European Journal of Operational Research*, vol. 46, no. 1, pp. 93–100, 1990.

[9] J.-C. Wang, "A multistart simulated annealing algorithm for the quadratic assignment problem," in *Innovations in Bio-Inspired Computing and Applications (IBICA), 2012 Third International Conference on*, pp. 19–23, IEEE, 2012.

[10] C. Rego, T. James, and F. Glover, "An ejection chain algorithm for the quadratic assignment problem," *Netw.*, vol. 56, pp. 188–206, Oct. 2010.

[11] T. James, C. Rego, and F. Glover, "A cooperative parallel tabu search algorithm for the quadratic assignment problem," *European Journal of Operational Research*, vol. 195, no. 3, pp. 810–826, 2009.

[12] T. James, C. Rego, and F. Glover, "Multistart tabu search and diversification strategies for the quadratic assignment problem," *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, vol. 39, no. 3, pp. 579–596, 2009.

[13] A. Misevicius, "A tabu search algorithm for the quadratic assignment problem," *Computational Optimization and Applications*, vol. 30, no. 1, pp. 95–111, 2005.

[14] J. Skorin-Kapov, "Tabu Search Applied to the Quadratic Assignment Problem," *ORSA Journal on Computing*, vol. 2, no. 1, pp. 33–45, 1990.

[15] E. Taillard, "Robust tabu search for the quadratic assignment problem," *Parallel computing*, vol. 17, pp. 443–455, 1991.

[16] N. Ç. Demirel and M. D. Toksarı, "Optimization of the quadratic assignment problem using an ant colony algorithm," *Applied Mathematics and Computation*, vol. 183, no. 1, pp. 427–435, 2006.

[17] L. M. Gambardella, É. Taillard, and M. Dorigo, "Ant Colonies for the Quadratic Assignment Problem," *The Journal of the Operational Research Society*, vol. 50, no. 2, p. 167, 1999.

[18] T. Stützle and M. Dorigo, "New ideas in optimization," ch. ACO Algorithms for the Quadratic Assignment Problem, pp. 33–50, Maidenhead, UK, England: McGraw-Hill Ltd., UK, 1999.

[19] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard, "Parallel Ant Colonies for the quadratic assignment problem," *Future Generation Computer Systems*, vol. 17, no. 4, pp. 441–449, 2001.

[20] Z. Drezner, "A New Genetic Algorithm for the Quadratic Assignment Problem," *INFORMS Journal on Computing*, vol. 15, no. 3, pp. 320–330, 2003.

[21] M. Inostroza-Ponta, R. Berretta, and P. Moscato, "Qapgrid: A two level qap-based approach for large-scale data analysis and visualization," *PloS one*, vol. 6, no. 1, p. e14468, 2011.

[22] J. Carrizo, O. Tinetti, P. Moscato, and L. Plata, "A computational ecology for the quadratic assignment problem," in *in Proceedings of the 21st Meeting on Informatics and Operations Research, (Buenos Aires), SADIO*, 1992.

[23] H. Meneses and M. Inostroza-Ponta, "Evaluating memory schemas in a memetic algorithm for the quadratic assignment problem," in *Chilean Computer Science Society (SCCC), 2011 30th International Conference of the*, pp. 14–18, Nov 2011.

[24] V.-D. C. V.-D. Cung, T. Mautor, P. Michelon, and a. Tavares, “A scatter search based approach for the quadratic assignment\nproblem,” *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC ’97)*, pp. 165–169, 1997.

[25] T. Stützle, “Iterated local search for the quadratic assignment problem,” *European Journal of Operational Research*, no. C, pp. 2005–2008, 2006.

[26] L. Steinberg, “The Backboard Wiring Problem: A Placement Algorithm,” *1SIAM Review*, vol. 22, no. 4, pp. 37–50, 1961.

[27] H. A. Eiselt and G. Laporte, “A combinatorial optimization problem arising in dartboard design,” *Journal of the Operational Research Society*, pp. 113–118, 1991.

[28] M. Dell’Amico, J. C. D. Díaz, M. Iori, and R. Montanari, “The single-finger keyboard layout problem,” *Computers and Operations Research*, vol. 36, no. 11, pp. 3002–3012, 2009.

[29] M. S. Bazaraa and A. N. Elshafei, “An exact branch-and-bound procedure for the quadratic-assignment problem,” *Naval Research Logistics Quarterly*, vol. 26, pp. 109–121, 1979.

[30] J. W. Dickey and J. W. Hopkins, “Campus building arrangement using topaz,” *Transportation Research*, vol. 6, pp. 59–68, 1972.

[31] M. Inostroza-Ponta, R. Berretta, A. Mendes, and P. Moscato, *Artificial Intelligence in Theory and Practice: IFIP 19th World Computer Congress, TC 12: IFIP AI 2006 Stream, August 21–24, 2006, Santiago, Chile*, ch. An automatic graph layout procedure to visualize correlated data, pp. 179–188. Boston, MA: Springer US, 2006.

[32] S. A. De Carvalho and S. Rahmann, “Microarray layout as quadratic assignment problem.,” in *German Conference on Bioinformatics*, pp. 11–20, 2006.

[33] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido, “A survey for the quadratic assignment problem,” *European Journal of Operational Research*, vol. 176, no. 2, pp. 657–690, 2007.

[34] S. Sahni and T. Gonzalez, “P-Complete Approximation Problems,” *Journal of the ACM*, vol. 23, no. 3, pp. 555–565, 1976.

[35] M. Lstibůrek, J. Stejskal, A. Misevicius, J. Korecky, and Y. A. El-Kassaby, “Expansion of the minimum-inbreeding seed orchard design to operational scale,” *Tree genetics & genomes*, vol. 11, no. 1, pp. 1–8, 2015.

[36] D. F. Rossin, M. C. Springer, and B. D. Klein, “New complexity measures for the facility layout problem: an empirical study using traditional and neural network analysis,” *Computers & Industrial Engineering*, vol. 36, no. 3, pp. 585–602, 1999.

[37] G. F. Pfister, *In Search of Clusters (2Nd Ed.).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[38] T. Dokeroglu, “Hybrid teaching-learning-based optimization algorithms for the Quadratic Assignment Problem,” *Computers and Industrial Engineering*, vol. 85, pp. 86–101, 2015.

[39] D. R. Heffley, “THE QUADRATIC ASSIGNMENT PROBLEM: A NOTE,” *Econometrica*, vol. 40, no. 6, pp. 1155–1163, 1972.

[40] D. R. Heffley, “Decomposition of the Koopmans-Beckmann Problem,” *Regional Science and Urban Economics*, vol. 10, no. 4, pp. 571–580, 1980.

[41] A. M. Geoffrion and G. W. Graves, “Scheduling Parallel Production Lines with Changeover Costs: Practical Application of a Quadratic Assignment/ LP Approach,” *Operations Research*, vol. 24, no. 4, pp. 595–610, 1976.

[42] J. Krarup and P. M. Pruzan, “Computer-aided layout design,” in *Mathematical programming in use*, pp. 75–94, Springer, 1978.

[43] L. Hubert, *Assignment methods in combinational data analysis*, vol. 73. CRC Press, 1986.

[44] J. H. Forsberg, R. M. Delaney, Q. Zhao, G. Harakas, and R. Chandran, “Analyzing lanthanide-induced shifts in the nmr spectra of lanthanide (iii) complexes derived from 1, 4, 7, 10-tetrakis (n, n-diethylacetamido)-1, 4, 7, 10-tetraazacyclododecane,” *Inorganic Chemistry*, vol. 34, no. 14, pp. 3705–3715, 1995.

[45] M. J. Brusco and S. Stahl, “Using quadratic assignment methods to generate initial permutations for least-squares unidimensional scaling of symmetric proximity matrices,” *Journal of Classification*, vol. 17, no. 2, pp. 197–223, 2000.

[46] J. Bos, “Zoning in forest management: a quadratic assignment problem solved by simulated annealing,” *Journal of Environmental Management*, vol. 37, pp. 127–145, 1993.

[47] S. Benjaafar, “Modeling and Analysis of Congestion in the Design of Facility Layouts,” *Management Science*, vol. 48, no. 5, pp. 679–704, 2002.

[48] C. S. Rabak and J. S. Sichman, “Using A-Teams to optimize automatic insertion of electronic components,” *Advanced Engineering Informatics*, vol. 17, no. 2, pp. 95–106, 2003.

[49] G. Miranda, H. P. L. Luna, G. R. Mateus, and R. P. M. Ferreira, "A performance guarantee heuristic for electronic components placement problems including thermal effects," *Computers and Operations Research*, vol. 32, no. 11, pp. 2937–2957, 2005.

[50] E. Duman and I. Or, "The quadratic assignment problem in the context of the printed circuit board assembly process," *Computers and Operations Research*, vol. 34, no. 1, pp. 163–179, 2007.

[51] G. Ben-David and D. Malah, "Bounds on the performance of vector-quantizers under channel errors," *IEEE Transactions on Information Theory*, vol. 51, no. 6, pp. 2227–2235, 2005.

[52] K. Anstreicher, "Recent advances in the solution of quadratic assignment problems," *Math. Program, Ser. B*, vol. vol, pp. 97pp27–42, 2003.

[53] R. E. Burkard, S. E. Karisch, and F. Rendl, "Qaplib–a quadratic assignment problem library," *Journal of Global optimization*, vol. 10, no. 4, pp. 391–403, 1997.

[54] Y. Li and P. M. Pardalos, "Generating quadratic assignment test problems with known optimal permutations," *Computational Optimization and Applications*, vol. 1, no. 2, pp. 163–184, 1992.

[55] S. K. R.E. BURKARD, E. ÇELA and F. RENDL, "Qaplib." Website, 2016. `http://anjos.mgi.polymtl.ca/qaplib/`.

[56] G. Palubeckis, "Generating hard test instances with known optimal solution for the rectilinear quadratic assignment problem," *J. of Global Optimization*, vol. 15, pp. 127–156, Sept. 1999.

[57] G. Palubeckis, "An algorithm for construction of test cases for the quadratic assignment problem," *Informatica*, pp. 281–296, 2000.

[58] Z. Drezner, P. M. Hahn, and É. D. Taillard, "Recent Advances for the Quadratica Assignment Problem with Sepcial Emphasis on Instances that are Difficult for Meta-Heuristic Methods," *Annals of Operation Research*, vol. 139, no. 1, pp. 65–94, 2005.

[59] T. Stützle, S. Fernandas, T. Stutzle, and S. Fernandes, "New benchmark instances for the QAP and the experimental analysis of algorithms," in *Evolutionary Computation in Combinatorial Optimization, Proceedings*, vol. 3004, pp. 199–209, 2004.

[60] Z. Drezner, "The extended concentric tabu for the quadratic assignment problem," *European Journal of Operational Research*, vol. 160, no. 2, pp. 416 – 422, 2005. Decision Support Systems in the Internet Age.

[61] A. Misevicius, “An implementation of the iterated tabu search algorithm for the quadratic assignment problem,” *OR Spectrum*, vol. 34, no. 3, pp. 665–690, 2012.

[62] N. Fescioglu-Unver and M. M. Kokar, “Self controlling tabu search algorithm for the quadratic assignment problem,” *Computers & Industrial Engineering*, vol. 60, no. 2, pp. 310 – 319, 2011.

[63] A. Acan and A. Ünveren, “A great deluge and tabu search hybrid with two-stage memory support for quadratic assignment problem,” *Applied Soft Computing*, vol. 36, pp. 185 – 203, 2015.

[64] L.-Y. Tseng and S.-C. Liang, “A hybrid metaheuristic for the quadratic assignment problem,” *Computational Optimization and Applications*, vol. 34, no. 1, pp. 85–113, 2006.

[65] U. Benlic and J. K. Hao, “Breakout Local Search for maximum clique problems,” *Computers and Operations Research*, vol. 40, no. 1, pp. 192–206, 2013.

[66] U. Benlic and J. K. Hao, “Breakout local search for the max-cutproblem,” *Engineering Applications of Artificial Intelligence*, vol. 26, no. 3, pp. 1162–1173, 2013.

[67] U. Benlic and J. K. Hao, “Memetic search for the quadratic assignment problem,” *Expert Systems with Applications*, vol. 42, no. 1, pp. 584–595, 2015.

[68] S. Tsutsui and N. Fujimoto, “ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment,” pp. 1547–1554, 2011.

[69] M. Harris, R. Berretta, M. Inostroza-Ponta, and P. Moscato, “A memetic algorithm for the quadratic assignment problem with parallel local search,” in *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pp. 838–845, May 2015.

[70] M. Czapiński, “An effective Parallel Multistart Tabu Search for Quadratic Assignment Problem on CUDA platform,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1461–1468, 2013.

[71] U. Tosun, “On the performance of parallel hybrid algorithms for the solution of the quadratic assignment problem,” *Engineering Applications of Artificial Intelligence*, vol. 39, pp. 267 – 278, 2015.

[72] E. S. Ristad and P. N. Yianilos, “Learning string-edit distance,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 522–532, May 1998.

[73] W. J. Heeringa, *Measuring dialect pronunciation differences using Levenshtein distance.* PhD thesis, Citeseer, 2004.

[74] C. Gooskens and W. Heeringa, "Perceptive evaluation of levenshtein dialect distance measurements using norwegian dialect data," *Language Variation and Change*, vol. 16, pp. 189–207, 10 2004.

[75] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg, "Adaptive name matching in information integration," *IEEE Intelligent Systems*, vol. 18, pp. 16–23, Sept. 2003.

[76] Z. Su, B. R. Ahn, K. Y. Eom, M. K. Kang, J. P. Kim, and M. K. Kim, "Plagiarism detection using the levenshtein distance and smith-waterman algorithm," in *Innovative Computing Information and Control, 2008. ICICIC '08. 3rd International Conference on*, pp. 569–569, June 2008.

[77] S. Schimke, C. Vielhauer, and J. Dittmann, "Using adapted levenshtein distance for on-line signature authentication," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 2, pp. 931–934 Vol.2, Aug 2004.

[78] J. SCHEPENS, T. DIJKSTRA, and F. GROOTJEN, "Distributions of cognates in europe as based on levenshtein distance," *Bilingualism: Language and Cognition*, vol. 15, pp. 157–166, 1 2012.

[79] Wikipedia, "Levenshtein distance — wikipedia, the free encyclopedia," 2016. [Online; accessed 22-June-2016].