NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

TABU SEARCH HEURISTICS FOR HYPERCUBE EMBEDDING

Jiawei Guo

A Thesis

in the Department of

Computer Science

Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

1992

ISBN 0-315-73601-7

Abstract

Tabu Search Heuristics for Hypercube Embedding

Jiawei Guo

The performance of a parallel program on a parallel computer depends heavily on the quality of program mapping: how to map the processes to the processors to minimize the communication overhead. Since all the processors perform communications asynchronously for MIMD parallel machines, the interprocessor communication time can be modeled by the total communication cost. Similarly, since all the processors perform communications synchronously for SIMD parallel machines, the interprocessor communication time can be modeled by the maximal communication cost between any pair of communicating processes. In this thesis we present two new hypercube program embedding strategies for the above two models.

We use tabu search as the main tool to search for optimized hypercube embeddings. Tabu search is a new general technique for solving combinatorial optimization problems. Different from other approaches, tabu search adopts an aggressive, deterministic search strategy. We studied the following issues for our tabu search heuristics: (1) neighborhood design; (2) *gain* functions and their updates; (3) tabu list length control; (4) the choice of tabu list contents; and (5) use of two efficient data structures.

We generate a large set of random, geometric, and perturbed regular graphs and compare the performance of our tabu search heuristics with the most competitive heuristics reported in the literature - simulated annealing and Kernighan Lin heuristics. Experiments show that our new proposed heuristics can significantly improve the solution quality and reduce the execution time.

Acknowledgments

I am grateful to my thesis advisor Dr. Lixin Tao for his support and advice during my study at Concordia University. My thesis benefits greatly from his insights and demand for high quality research. I would also like to express my gratitude to the members of my thesis defense committee Dr. C. Lam, Dr. R. Jayakumar, and especially Dr. Tao Li for their comments and suggestions for my research.

In addition, I would like to thank Prof. Yong Chang Zhao for his suggestions and discussions, Mr. Xin Ming Yu for his help in the Lab and friendship during the past few years. I deeply appreciate my wife Lisha Kang for her love, understanding, and support throughout my study.

This work was partially supported by the Canadian NSERC research grant OGP-0011618 and FCAR research grant 92NC0026.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The performance of a parallel program on a parallel machine with distributed memory depends heavily on the mapping of the processes to the processors. Since the parallel machine usually has a fixed topology and each program usually has different communication patterns, the interprocess messages usually have to go through some intermediate processors before reaching their destinations. To minimize such communication overhead, we should map (load) processes into processors so that each pair of communicating processes are as close as possible, and each pair of communicating processes with heavy communication load are at a distance no longer than that for a pair of communicating processes with light communication load.

We can model the above *program mapping* problem with the following *graph mapping* problem. We abstract the communication pattern of a parallel program into a *task graph* (communication graph) $G = (V, E)$ in which each vertex represents a process, each edge represents a logical communication channel, and the weight on each edge represents the communication load between the two incident processes. We can also abstract the parallel machine into a *system graph* $H = (V_h, E_h)$ in which each vertex represents a processor and each edge represents a physical link. Any function $\pi : V \rightarrow V_h$ represents a program mapping scheme. Given any pair of adjacent vertices $x, y \in V$, we use the term *mapped distance* between x and y to denote the product of the edge weight between x and y and the distance between

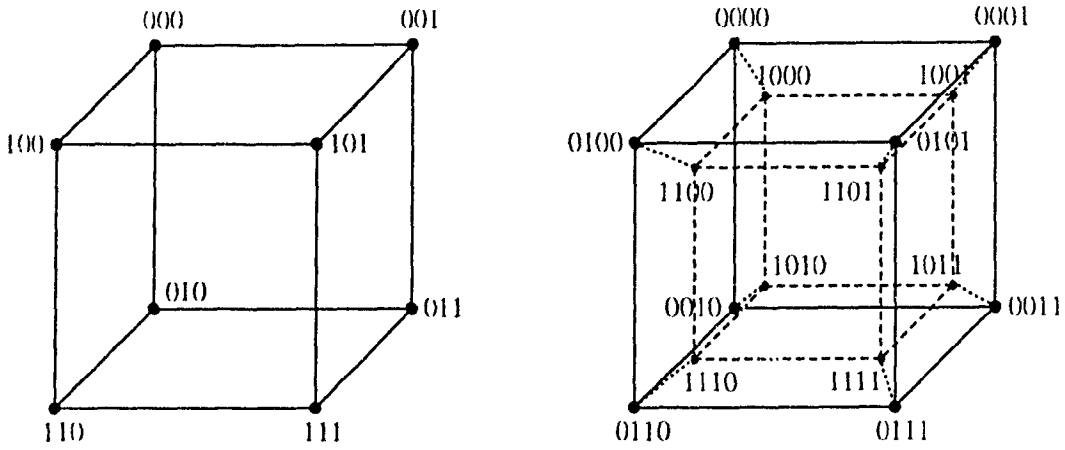
$\pi(x)$ and $\pi(y)$ in H. If the machine is working under the MIMD mode, since the computation and communication of different processes can be conducted in parallel asynchronously, the interprocessor communication time can be modeled by the summation of mapped distances between all pairs of communicating processes [10]. If the machine is working under the SIMD mode, since all the processors perform communications synchronously, the interprocessor communication time can be modeled by the maximal mapped distance between any pair of communicating processes [10, 11].

1.2 The hypercube embedding problem

Due to its high regularity, small diameter, and high efficiency for diverse applications, hypercube interconnection pattern is becoming very popular for commercial and experimental parallel machines. This attracts the study of the *hypercube embedding problem*, which is a restricted version of the above graph mapping problem.

A d-dimensional *hypercube*, also called *d-cube*, is an undirected graph consisting of $n = 2^d$ vertices labeled from 0 to $2^d - 1$ such that there is an edge between any pair of vertices if and only if the binary representations of their labels differ at exactly one bit position. Figure 1.1 shows a 3-cube with 8 vertices and a 4-cube with 16 vertices. In a d-cube, according to the definition, each vertex has exactly d neighbors (degree d); there are a total of $nd/2$ edges; the diameter is d; and the average distance between pairs of vertices is $\frac{dn}{2(n-1)}$. In addition, a d-cube is bipartite; the two bipartitions are the set of vertices whose labels have an even number of 1's and the set of vertices whose labels have an odd number of 1's. A graph G is said to be *d-cubical* if G is isomorphic to a subgraph of a d-cube.

Let $H = (V_h, E_h)$ be a d-cube. Given any two vertices u and v in V_h, we use $\delta(u, v)$ to denote the length of the shortest paths between u and v (Hamming distance). Let $G = (V, E)$ be an undirected graph where $|V| = |V_h|$, and $w : E \to I$ (I is the set of positive integers) a weight function. An embedding of G into H is a bijection $\pi : V \to V_h$. For any pair of vertices $u, v \in V$, $\delta(\pi(u), \pi(v)) \cdot w(u, v)$ is the mapped



(a) 3-cube

(b) 4-cube

Figure 1.1: Hypercubes

distance between u and v.

We use two different objective functions to abstract the program embedding problem under two different computation modes. We define the following *average mapped distance*

$$\mathcal{D}(\pi) = \frac{1}{\sum_{\{u,v\}\in E} w(u,v)} \cdot \sum_{\{u,v\}\in E} \delta(\pi(u),\pi(v)) \cdot w(u,v)$$

to model the program embedding problem on MIMD machines. This objective function gives us a special case of the *Quadratic Assignment Problem* [42, 33]. For any embedding π, $\mathcal{D}(\pi) \geq 1$. If π preserves the neighborship in G, we have $\mathcal{D}(\pi) = 1$. In our performance comparisons in the following chapters, we also use the *total mapped distance* $\mathcal{T}(\pi) = \sum_{\{u,v\}\in E} \delta(\pi(u),\pi(v)) \cdot w(u,v)$ for reference.

We also define the following *maximal mapped distance*

$$\mathcal{D}(\pi) = \max_{\{u,v\}\in E} \delta(\pi(u),\pi(v)) \cdot w(u,v)$$

to model the program embedding problem on SIMD machines. If all edge weights are constant 1, $\mathcal{D}(\pi)$ reduces to the *dilation cost* [44].

The hypercube embedding problem can thus be defined as follows: *Given a task*

graph $G = (V, E)$ *and a d-cube* $H = (V_h, E_h)$ *where* $|V| = |V_h|$, *find a bijection* $\pi : V \rightarrow V_h$ *to minimize the objective function* $\mathcal{D}(\pi)$ *or* $\mathcal{D}(\pi)$.

Unfortunately, the problem of identifying whether a given graph G is a subgraph of a hypercube has been shown to be NP-complete [12]. We can reduce the hypercube subgraph problem into our hypercube embedding problem. Let $w(e) = 1$ for all $e \in E$. If G has less vertices than H, we introduce isolated dummy vertices so that G and H have the same number of vertices. Then G is a subgraph of a hypercube H if and only if G can be embedded into H with $\mathcal{D}(\pi) = 1$ or $\mathcal{D}(\pi) = 1$. Therefore our hypercube embedding problem is NP-hard no matter which objective function is used.

Many heuristics have been proposed for solving the hypercube embedding problem. Lee and Aggarwal [38] proposed a constructive greedy embedding heuristic and an iterative local search embedding heuristic. Simulated annealing heuristics have been studied by Ramamujam, Ercal, and Sadayappan [10], Bollinger and Midkiff [5] and Chen [11, 10]. Chen made a comprehensive comparative study of different hypercube embedding heuristics, proposed a new greedy heuristic, and further improved the local search and Kernighan-Lin heuristics [10]. In this thesis, we propose two new efficient tabu search heuristics for hypercube embedding.

1.2 Tabu search technique

Tabu search, recently introduced by Glover [19], is a new strategy for solving general combinatorial optimization problems. Unlike simulated annealing which is a totally randomized heuristic, tabu search looks more like an intelligent search which may in some sense imitate a human behavior or apply some rules based on artificial intelligence principles. It is an adaptive metaheuristic with the ability to make use of many other methods, such as linear programming and specialized heuristics, which it directs to overcome the limitations of local optimality. Tabu search has been successfully applied to solving many NP-hard problems including character recognition [31], computer channel balancing problem [20], quadratic assignment problems [33,

1. Get a random initial solution π.
2. While stop criterion not met do:
   2.1 Let π' be a neighbor of π maximizing $\Delta = cost(\pi) - cost(\pi')$ and not visited in the last t iterations.
   2.2 Set $\pi = \pi'$.
3. Return the best π visited.

Figure 1.2: Tabu search

42], graph coloring [30], graph partitioning [45], and maximum stable set problems [16, 17].

Tabu search can be viewed as an improved variant of local search. It differs from simulated annealing at two main aspects:

- It is more aggressive. For each iteration the whole neighborhood of the current solution is usually searched exhaustively to find the best candidate *move*.
- It is deterministic. Each iteration repeats the above exhaustive search for best candidate moves. The best candidate *move* which does not cause cycling in the solution space will be used no matter whether it improves on the current solution or not. A *tabu list* is usually used to record the recent move history to avoid solution cycling, so comes the name of the approach.

Figure 1.2 outlines a generic tabu search heuristic using t to represent the length of the tabu list. Given a random initial solution, the heuristic repeats the loop at Step 2 until some stop criterion is met. During each iteration, the heuristic makes an exhaustive search of the solutions in the neighborhood of the current solution which have not been traversed in the last t ($t > 1$) iterations. The neighboring solution with the best cost found in the above process will be used to replace the current solution. The main design issues for a tabu search heuristic include:

1. The design of the neighborhood (moves) of the current solution. Let S be the move-set consisting of all defined moves and π the current solution. The neighborhood of π, denoted by $\mathcal{N}(\pi)$, is the set of solutions which can be reached from π by one move in S. The neighborhood design is basically to make the trade-off between the aggressiveness and the CPU time for the neighborhood search. For a large neighborhood size, the heuristic is more aggressive at each iteration and less efficient. If the neighborhood size is too small, the aggressive property of tabu search will be lost.

2. The design of the contents of the tabu list. Tabu list is basically used to avoid cycling due to uphill moves. The contents of the tabu list can be any attributes of the current solution. If we record a more abstract set of attributes of a solution in the tabu list, more restrictions will be applied to the search at each iteration, and more moves will be tabued. If we use a more detailed set of attributes of a solution in each cell of the tabu list, however, more memory space and checking time will be incurred during the solution space search, and the searches will be less restrictive since less solutions (in addition to the ones visited in the last t iterations) will be tabued.

3. The design of the length t of the tabu list. In general, the longer the tabu list is, the more time tabu status checking takes for each move, and the more restrictive the search process will be. On the other hand, a too short tabu list risks to introduce cycling in the solution space. For many applications, the best tabu list length is both instance-dependent and time-dependent.

4. The design of the aspiration level function. Since most tabu lists only store some attributes (moves, for example) of the solutions, these attributes can tabu too many moves not causing solution cycling. To prevent this case from occurring, an aspiration level function can be defined. For each move s and each solution π, we define an aspiration level $A(s, \pi)$. Let π be the current solution. For any move s which is in the tabu list, we say that the aspiration level is attained if $cost(s(\pi)) < A(s, \pi)$. The role of $A(s, \pi)$ is to provide added flexibility to

choose good moves by allowing the tabu status of a move to be overridden if the aspiration level is attained. The goal is to do this in a manner that retains the ability to avoid solution cycling.

5. The design of the intermediate and long term memory. Intermediate and long term memory functions are employed within tabu search to achieve regional intensification and global diversification of the search. Combined with the short term memory function fulfilled by the tabu list, the intermediate and long term functions work alternatively. Intermediate term memory operates by recording and comparing features of a selected number of best trial solutions generated during a particular period of search. Features that are common to all or a compelling majority of these solutions are taken to be a regional attribute of good solutions. The method then seeks new solutions that exhibit these features, by correspondingly restricting or penalizing available moves during a subsequent period of regional search intensification. The long term memory function, whose goal is to diversify the search, employs principles that are roughly the reverse of those for intermediate term memory. Instead of inducing the search to focus more intensively on regions that contain good solutions found previously, the long term memory function guides the process to regions that markedly contrast with those examined thus far. The approach differs from those methods that seek diversity by generating a series of random starting points, and hence which afford no opportunity to learn from the past. The objective is to create evaluation criteria that can be used by heuristic search process which is specifically designed to produce a new better than random starting point.

1.3 Thesis outline

According to our literature survey on hypercube embedding problems, we found that Chen's recent Ph.D thesis [10] made the most extensive comparative study of various hypercube embedding techniques. Chen claimed that the combination of a

greedy heuristic (FG) and a Kernighan-Lin heuristic with cube neighbor neighborhood (KLC) search strategy gave the best performance over other heuristics (greedy, local search, Kernighan-Lin, and simulated annealing) for various types of task graphs. Chen used unweighted total mapped distance as his objective function [11, 10]. Our main objective in this thesis is to generalize Chen's embedding model and propose new efficient tabu search embedding heuristics to outperform the best heuristics claimed by Chen in both solution quality and running time.

In Chapter 2, we give a concise literature survey on hypercube embedding problem and tabu search technique.

The problem-specific design and implementation issues (heuristic independent) are discussed in Chapter 3. These issues include move-set design, *gain* function and its update, a bucket-list priority queue data structure, benchmark graphs used for performance comparisons, and the test bed.

In Chapter 4, we introduce the established hypercube embedding approaches including greedy, Kernighan-Lin, and simulated annealing. We summarize these approaches in the context of Chen's adaptations of them to hypercube embedding. We also introduce our improvement on Chen's simulated annealing heuristic. The related performance comparisons are reported at the end of this Chapter.

Chapter 5 presents the details of our tabu search heuristic (TSC) with the objective function $\tilde{\mathcal{D}}(\pi)$. We investigate the effects of neighborhood selection on solution quality and CPU efficiency. We successfully introduce an adaptive tabu list to our tabu search heuristic to capture the dynamic properties of the hypercube embedding process. The relevant performance comparisons between tabu search heuristics with static and adaptive tabu lists are presented. The experiments show that the iteration number and running time using adaptive tabu list are 18% and 22% less than those based on static tabu list respectively while the solution quality of the former is better than that of the latter. We also present both extensive and intensive performance comparisons between Chen's most competitive heuristics and our new tabu search heuristic.

A new tabu search heuristic (TSD) for objective function $\mathcal{D}(\pi)$ is introduced in Chapter 6. We discuss the special difficulties in minimizing $\mathcal{D}(\pi)$, and present the *qam* related design based on $\mathcal{D}(\pi)$ and the other implementation issues. We also describe a *heap* priority queue data structure used in TSD to facilitate the evaluation of $\mathcal{D}(\pi)$. The performance comparisons for $\mathcal{D}(\pi)$ are made between TSD and TSC for all of our benchmark graphs. The experiments show that TSD cannot be replaced by TSC.

Chapter 7 concludes the thesis with a summary of our main observations, our contributions to this research, and future work.

Chapter 2

Literature Survey

In this chapter we review the literatures on hypercube embedding and tabu search. Based on our survey, we found that Chen's study [10] on hypercube embedding is most extensive in both embedding approaches and performance comparisons among different approaches; and tabu search is a new promising search technique still under development.

2.1 Hypercube embedding

Many heuristics have been proposed based on different optimization models. In this section, we only survey those previous work related to us.

Bokhari proposed a local search heuristic with pairwise exchange for mapping task graphs to a Finite Element Machine (FEM) ("eight nearest neighbor" interconnection) [4]. Bokhari's objective function maximizes the number of edges in the task graph that are mapped to single edges in FEM, which is also used by the greedy heuristic in [10] (referred to as FG) for hypercube embedding. He tested about 20 structural problems of 9 to 49 vertices for FEM's of size 4×4 to 7×7. To avoid local optima traps, probabilistic jumps were used in the local search to improve performance.

Lee and Aggarwal formulated a set of objective functions which quantify communication overhead for different applications (e.g., asynchronous communication,

synchronous communication, and parallel image-processing model) [38]. The edges in G are divided into some subsets such that the edges in the same subset represent the inter process communications which occur simultaneously and the edges in different subsets represent the inter-process communications which occur sequentially. The two objective functions, $\sum_i(\max_j(\tilde{e}_{ij}))$ and $\max_i(\max_j(\tilde{e}_{ij}))$, were formulated, where i is the subset number, j is edge number in subset i, and $\tilde{e}_{ij}$ is the mapped distance of the jth edge in subset i. A greedy heuristic in combination with a local search (pairwise exchange) was proposed to solve the mapping problem. The heuristic was tested for 9 task graphs on hypercubes of 8 and 16 vertices respectively using the second objective function.

A simulated annealing heuristic was studied and reported by Ramanujam, Ercal, and Sadayappan [40]. To formulate the communication overhead, the total mapped distance $T(\pi)$ was used as the objective function subject to a constraint on a load-imbalance factor. Two strategies, namely simulated annealing with scaling and simulated annealing with exchange, were investigated to prevent the load-imbalance factor from trapping the annealing process at local optima. Experiments were done for 5 structured and 3 random graphs with 141 to 602 vertices.

Ercal, Ramanujam, and Sadayappan further proposed a recursive mapping strategy [13] based on repeated applications of the Kernihan-Lin graph partitioning heuristic [35]. This heuristic was compared with simulated annealing for the same set of test graphs in [40]. Results showed that this heuristic obtained costs slightly worse than those for simulated annealing, but the cost difference was always less than 10% and the computation time of this recursive strategy was much less.

A processor and link assignment heuristic using simulated annealing was developed by Bollinger and Midkiff [5]. Since each link in the multiprocessor may be contended by several processes at the same time (causing traffic congestion on the link), the objective function considers communication costs and the load on each link. The heuristic employs two optimization phases. *Process annealing* is first used to assign processes to processors (processor assignment), and *connection annealing* is

then used to further reduce the communication cost by routing traffic over data paths (link assignment). Two different objective functions were used for the two different phases. The performance of the heuristic was evaluated by mapping hypercubes with 8 to 512 vertices onto themselves and mapping binary trees to hypercubes. Experiments showed that this simulated annealing heuristic was able to consistently map hypercubes of size ≤ 128 into themselves perfectly.

A greedy heuristic embedding strategy was proposed by Chen and Gehringer in [9]. They chose the average distance (the total mapped distance divided by number of edges in G) without edge weight as the objective function. The performance of embedding fully connected task graphs, almost fully connected task graphs, hypercubes, and rings into hypercube were presented.

The performance of several different mapping heuristics for the hypercube embedding problem was compared by André, Pazat, and Priol [2]. They adopted the quadratic assignment model for their objective function and compared four different heuristics including Bokhari's local search heuristic [4], Chen's greedy heuristic [9], a simulated annealing heuristic, and Pazat's "friendly greedy" heuristic [39]. The comparison was based on mapping 4×4 meshes to 4-cubes and mapping a parallel ray-tracing algorithm to 4- and 5-cubes.

Based on the same objective function as used in [9], Chen's recent thesis studied three greedy heuristics, a local search heuristic with flat moves, a Kernighan Lin heuristic with uphill moves, and a simulated annealing heuristic [10]. An efficient *bucket-list* data structure (see Section 3.3) proposed by Fiduccia and Mattheyses [15] was employed to implement a priority queue. We will describe these heuristics in Chapter 4.

2.2 Tabu search

Tabu search was first proposed for the integer programming problem [19] and then developed in a general framework by Glover. Independently, Hansen proposed a

Steepest Ascent/Mildest Descent technique based on similar ideas and used it to solve maximum satisfiability problem [28]. Research on tabu search has been focused on the following issues [30, 6, 16, 50, 48, 29, 42, 32, 18, 17, 33, 43, 41].

1. **Neighborhood design**

   Given a solution π, its neighborhood $\mathcal{N}(\pi)$ is defined as the set of all possible solutions which can be found from π by one *move* or modification. A large neighborhood size will result in a long computation time for neighborhood search. To reduce the time for neighborhood search, Hartz and Werra [30] proposed to stop the search as soon as a better neighbor solution π' is found in the neighborhood of the current solution π, and the move is performed directly from π to π'. In [10] Chen chose the cube-neighbor neighborhood for hypercube embedding and demonstrated that the heuristics with cube-neighbor neighborhood performs better than those with all-swap neighborhood.

2. **Tabu list size**

   The choice of tabu list size is critical for many applications. If tabu list is too short, cycling may occur in the search process; if tabu list is too long, appealing moves may be forbidden and leading to the exploration of lower quality solutions. It is, however, difficult to give a general rule for finding the best tabu list size. In most references, the tabu list size was fixed. Taillard proposed a scheme to change the tabu list size randomly during the search for Quadratic Assignment Problem [42]. The tabu list size k will be chosen between $k_{\min}$ and $k_{\max} = k_{\min} + \Delta$, where both $k_{\min}$ and Δ are given positive integers. For every $2 \cdot k_{\max}$ iterations, a random number r uniformly distributed over interval $[0,\Delta]$ is used to change the tabu list size to $k_{\min}+r$. In this way Taillard claimed that one can find better solutions in a more reliable way and reduce 30% iterations than for a fixed tabu list size.

3. **Aspiration level function**

(a) Skorin-Kapov used the best cost, *best_cost*, obtained thus far as the aspiration level function [33]. If a move transforming solution from π to π' is in tabu status and $cost(\pi') < best\_cost$, then the tabu status can be overridden. While this aspiration level function is simple, its ability to override the tabu status is very limited.

(b) Hertz et al. proposed that the tabu status of a move from π to π' be dropped if $cost(\pi')$ is smaller than the best cost obtained earlier by leaving a solution π'' where $cost(\pi'') = cost(\pi)$ [30, 29, 32].

1. **Long term memory**

Skorin-Kapov incorporated long term memory in her tabu search heuristic for the Quadratic Assignment Problem (QAP) [33]. The heuristic has two phases — *construction* and *improvement*. The former provides an initial solution based on the flow and distance matrices, and the latter improves the initial solution using tabu search. An $n \times n$ matrix LTM is used to record the move statistics of each facility for the QAP of size n. At the end of the improvement phase, the recent move history stored in LTM is used to modify the distance matrix and the above process is repeated with a new initial solution. The experiments show that tabu search with LTM takes much more time than that without LTM and the improvement on solution quality is very limited.

2.3 Summary

For most research on hypercube embedding, the total communication cost (with or without edge weight) is employed as the objective function. The techniques that are used to solve the hypercube embedding problem can be classified into two categories: constructive approach and iterative improvement approach. The former includes the variants of various greedy heuristics, and the latter includes local search, Kernighan Lin, and simulated annealing heuristics. Chen made extensive performance comparisons for most of these heuristics [10].

Tabu search is a new general search strategy still under development. It has been successfully applied to a wide range of problem domains with superior performance. The major design issues are focused on neighborhood, tabu list, aspiration level function, and intermediate/long term memory functions.

Chapter 3

Problem-Specific Design Issues

Let X be the set of all bijections $\pi : V \rightarrow V_h$, and $cost(\pi)$ denote either $\mathcal{P}(\pi)$ or $\mathcal{D}(\pi)$. Our hypercube embedding problem can be presented in the following form

$$\text{Minimize } cost(\pi) : \pi \in X.$$

We call X the *solution space*.

A wide range of heuristics for solving problems capable of being written in the above form can be characterized conveniently by reference to sequences of *moves* that lead from one trial solution (selected $\pi \in X$) to another. A *move* s is a mapping from a subset $X(s)$ of X to X. Let S be the move-set containing all defined moves. We use $S(\pi)$ ($\pi \in X$) to denote the subset of moves in S applicable to π, and $S(\pi, Y)$ ($\pi \in X$, $Y \subseteq V$) the subset of *moves* in $S(\pi)$ that only redefines $\pi(v)$ for $v \in Y$. For any $s \in S(\pi)$, $s(\pi)$, the new solution obtained by applying move s to π, is called a *neighbor* of π. We use $\mathcal{N}(\pi)$, called *neighborhood* of π, to denote the set of all neighbors of π, and $|\mathcal{N}(\pi)|$ the *neighborhood size* of π. Since all of the moves which we use are bijections, we have $|S(\pi)| = |\mathcal{N}(\pi)|$.

3.1 Move-set and neighborhood design

From an abstract point of view, all the heuristics (except greedy heuristics) in this thesis perform a series of iterations. At each iteration, a subset of the neighborhood

$\mathcal{N}(\pi)$ of the current solution π in the solution space X is investigated and the current solution π is updated accordingly (making a *move* $s \in S(\pi)$). To make these heuristics efficient and effective, the move-set S should be defined with the following properties:

- *Reachability:* Given any two solutions π and π' in X, it should be possible to apply a sequence of moves in S to reach π' from π. This property will greatly increase the probability for an heuristic to converge to the global optimum.
- *Efficiency:* Given any solution $\pi \in X$ and $s \in S(\pi)$, the cost of $s(\pi)$ can be easily evaluated by incrementally updating the cost of π. This will allow us to avoid evaluating the cost function during each iteration.
- *Injectiveness:* For any $\pi \in X$ and any two different *moves* $s, s' \in S(\pi)$, $s(\pi) \neq s'(\pi)$. This will make sure that each neighbor of the current solution will be checked only once for the current neighborhood search.

Let $\pi \in X$ be the current solution. Given a vertex $u \in V$, we use $\eta(\pi, u, i)$ to denote a vertex $v \in V$ such that $\pi(v)$ is the ith $(1 \leq i \leq d)$ dimensional neighbor of $\pi(u)$ in the d-cube. The following are two popular move-set designs for hypercube embedding:

Cube-neighbor move-set : $S_1(\pi) = \{\{u, v\} | u, v \in V, \delta(\pi(u), \pi(v)) = 1\}$;

All-swap move-set : $S_2(\pi) = \{\{u, v\} | u, v \in V; u \neq v;\}$.

For a move $s = \{u, v\}$, $s(\pi)$ is obtained by exchanging the values of $\pi(u)$ and $\pi(v)$. We let $\mathcal{N}_1(\pi) = \{s(\pi) | s \in S_1\}$ and $\mathcal{N}_2(\pi) = \{s(\pi) | s \in S_2\}$. We call $\mathcal{N}_1(\pi)$ the *cube-neighbor neighborhood*, and $\mathcal{N}_2(\pi)$ the *all-swap neighborhood*. For all-swap neighborhood, the neighborhood size $|\mathcal{N}_2(\pi)| = |S_2(\pi)| = n(n-1)/2$. For cube-neighbor neighborhood, however, the neighborhood size is reduced to $|\mathcal{N}_1(\pi)| = |S_1(\pi)| = (n \log n)/2$ by allowing swaps only between vertices in G that are mapped to adjacent hypercube vertices.

For both objective functions $\mathcal{D}(\pi)$ and $D(\pi)$, both S_1 and S_2 enjoy the reachability and the injectiveness properties. For objective function $\mathcal{D}(\pi)$, the efficiency is acquired for both of the two move-sets by incrementally updating the current solution cost $\mathcal{D}(\pi)$ to get its neighbor's solution cost. To be more specific, for any $\pi \in \Lambda$ and $s \in S$, we define a *gain* $gain(\pi, s)$ such that $\bar{\mathcal{D}}(s(\pi)) = \mathcal{D}(\pi) + gain(\pi, s)$. We will introduce the gain function for $\bar{\mathcal{D}}(\pi)$ in the next section. For objective function $D(\pi)$, however, the property of efficiency is not easily available. We will describe the gain function for $D(\pi)$ in Chapter 6.

3.2 Gain function and its update

In this section, we only discuss the gain function and its update for objective function $\mathcal{D}(\pi)$. For objective function $D(\pi)$, the corresponding discussion will be made in Chapter 6.

A gain function should be defined to measure the effectiveness of each potential move, and this gain function should be easily updated following each move to minimize the computational overhead introduced by reevaluating the gain function. We assume that a data structure is used to store the gain value for each move; and for any move s, it takes constant time to find the gain associated with it.

Given any current solution $\pi \in X$ and any move $s \in S(\pi)$, we define the gain of s related to π as $gain(s) = \bar{\mathcal{D}}(\pi) - \bar{\mathcal{D}}(s(\pi))$. Because a move s (swapping two vertices) only affects part of the total mapped distance, and our neighborhoods have special properties, the gain function can be refined. In the following, for any vertex $u \in V$, we use $\beta(u)$ to denote the set of all the neighboring vertices of u in G.

3.2.1 Gain function for all-swap neighborhood

For any vertices $u, v \in V$, the exchange of u and v will only affect the mapped distances between vertex u and its neighbors and between vertex v and its neighbors.

The $gain(u, v)$ can thus be refined as

$$gain(u,v) = \sum_{x \in \beta(u)} \delta(\pi(u), \pi(x)) \cdot w(u,x) + \sum_{y \in \beta(v)} \delta(\pi(v), \pi(y)) \cdot w(v,y) -$$

$$\sum_{x \in \beta(u)} \delta(\pi(v), \pi(x)) \cdot w(u,x) - \sum_{y \in \beta(v)} \delta(\pi(u), \pi(y)) \cdot w(v,y).$$

The time complexity of function $gain(u, v)$ is $O(deg(u) + deg(v)) = O(\hat{d})$, where $\hat{d} = (2m)/n$ is the average degree of the vertices in G, and $m = |E|$ is the number of edges in G.

Given a specific embedding, the values of $gain(u, v)$ for all combinations of u and v can be initialized in time $O(n^2 \cdot \hat{d}) = O(n^2 \cdot \frac{2m}{n}) = O(2mn) = O(mn)$. Given any $u, v \in V$, whenever a swap $\{u, v\}$ is performed, it is sufficient to update the gains for the following pairs:

- $\{u, x\}$ and $\{v, x\}$ for all $x \in V$: a reevaluation of the $gain$ for each of these swaps can be done in time $O(\hat{d})$; there are $O(n)$ such pairs, the total time for the reevaluation for all these pairs is thus $O(n\hat{d}) = O(n(2m/n)) = O(m)$;
- $\{x, y\}$ for $x \in \beta(u) \cup \beta(v)$, all $y \in V - \{u, v\}$: these gains need only be adjusted according to the new distances between x and u and between x and v. The gain can be updated by the following formula: $gain(x, y) = gain(x, y) + \Delta(x, y) \cdot w(x, y)$, where

$$\Delta(x,y) = \begin{cases} (\delta(x,u) - \delta(y,u)) - (\delta(x,v) - \delta(y,v)), & \text{if } x \in \beta(u) \\ (\delta(x,v) - \delta(y,v)) - (\delta(x,u) - \delta(y,u)), & \text{if } x \in \beta(v). \end{cases}$$

  The time complexity is $O((deg(u) + deg(v))n) = O((\hat{d} + \hat{d})n) = O(m)$.

The time complexity for updating the gain function for each move is thus $O(m)$.

3.2.2 Gain function for cube-neighbor neighborhood

Let $\gamma(u,i)$, $1 \leq i \leq d$, be the decrease in cost that would result from moving u to the ith dimensional neighbor of $\pi(u)$ (without moving any other vertices). We have

$$\gamma(u,i) = \sum_{x \in \beta(u)} \alpha(x,u,i) \cdot w(u,x)$$

where $\alpha(x,u,d)$ equals 1 if the ith bit of $\pi(x)$ in cube is the same as the ith bit of $\pi(u)$, -1 otherwise. The $gain(u,v)$ for a move $s = \{u,v\}$, $v = \eta(\pi,u,i)$, $1 \leq i \leq d$, is thus

$$gain(u,v) = \gamma(u,i) + \gamma(v,i) - 2 \times adj(u,v),$$

where $adj(u,v)$ is 1 if u and v are adjacent in G, 0 otherwise. The time complexity of function $gain(u,v)$ is thus $O(deg(u) + deg(v)) = O(\tilde{d})$.

Given a specific embedding, the values of $gain(u,v)$ for all combinations of u and v can be initialized in time $O(n \cdot \log n \cdot \tilde{d}) = O(n \cdot \log n \cdot \frac{2m}{n}) = O(2m \log n) = O(m \log n)$. Given any $u \in V$ and $1 \leq i \leq d$, let $v = \eta(\pi,u,i)$. Assume that move $s = \{u,v\}$ has just been performed.

- For each dimension j, $1 \leq j \leq d$, let $x = \eta(s(\pi),u,j)$ and $y = \eta(s(\pi),v,j)$, the values of $gain(u,x)$ and $gain(v,y)$ are reevaluated. The corresponding time complexity is thus $O(\tilde{d} \log n)$;

- For all $x \in \beta(u) \cup \beta(v)$, let $y = \eta(s(\pi),x,i)$. Since the swap of vertices u and v only affects the value of gain related to the ith dimensional cube neighbor of $\pi(x)$, only $gain(x,y)$ needs to be reevaluated. The corresponding time complexity is thus $O(\tilde{d}^2)$.

The total time complexity of gain update for each move is thus $O(\tilde{d}(\tilde{d} + \log n))$.

We implemented each heuristic in this thesis for both all-swap and cube neighbor neighborhoods. For each heuristic, the all-swap version always has inferior performance to the cube-neighbor version since the former can only afford very limited number of iterations due to its large neighborhood size. This confirms a similar claim

made by Chen in [10]. Therefore, this thesis will focus on the performance comparisons of heuristics based on the cube-neighbor neighborhood design.

3.3 Bucket-list priority queue data structure

For most heuristics, we need a priority queue to support our implementations. The general setting for this queue is as follows. There is a finite set Y of items, an integer $gmax > 0$, and an integer function $g : Y \rightarrow \{-gmax, \cdots, gmax\}$. The priority queue should support the following operations:

$Insert(Q, y, g)$	insert item y with gain g into Q
$Delete(Q, y)$	remove item y from the queue Q
$Max(Q)$	return y, the item with the largest gain in Q
$Move(Q, y, g)$	revise item y's position in the queue Q in accordance with its new gain g

Fiduccia and Mattheyses introduced a *bucket-list* data structure to implement this priority queue in their Kernighan-Lin heuristic for graph partition [15]. This data structure was also employed by Chen in his implementation of hypercube embedding [10]. The implementation is based on an array $bucket[-gmax, ..., gmax]$, whose kth entry is a pointer to a doubly-linked list, called a *bucket*, of cells each containing an item with gain currently equal to k. Each cell has a field to keep its current bucket number. A separate array of pointers, indexed by moves, allow us to have direct access to each item in the buckets. An integer typed variable *best_gain* always points to the first (from top) non-empty bucket. Figure 3.1 shows the bucket-list implementation of the priority queue used in our heuristics.

The following lemma is a restatement of the time bound found in [15], presented in more general terms.

Lemma 1 *Let Q_I be the number of times that the priority-queue operations $Insert()$, $Delete()$, and $Move()$ are called, Q_M the number of calls to $Max()$, M the maximum gain of any item, and Δ_{max} the maximum amount by which the gain of any item may*

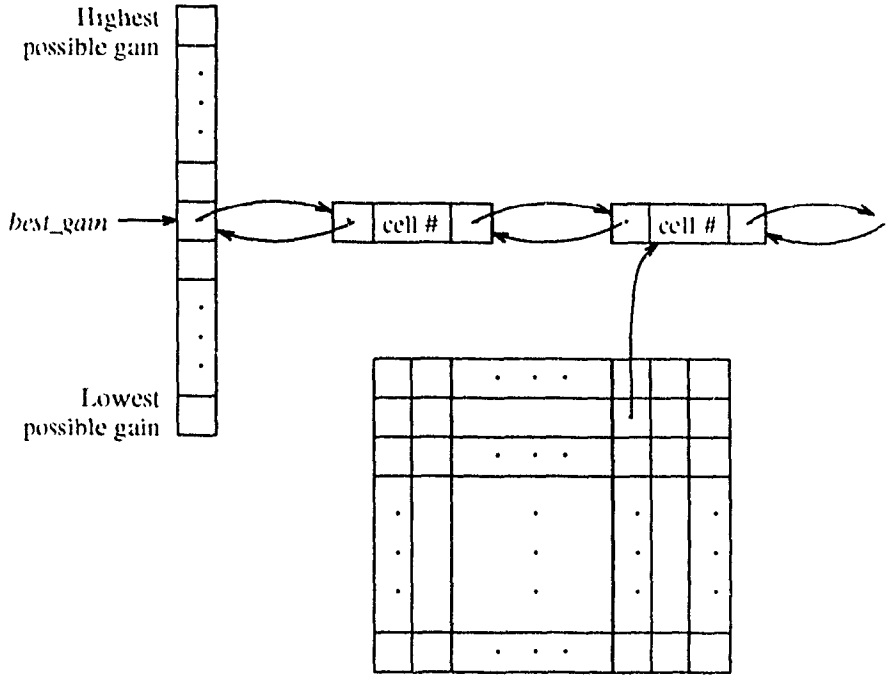


Figure 3.1: "Bucket-list" priority queue

increase between two successive $Max()$ operations. Then, assuming all insertions occur at the beginning, the priority queue can be implemented so that the total time for all operations is $O(Q_I + Q_M \Delta_{max} + M)$. □

Proof: The operation $Move()$ is simply a matter of moving a cell from one doubly linked list to another and can be done in constant time. $Insert()$ and $Delete()$ can also be done in constant time. The auxiliary variable *best_gain* is used to facilitate $Max()$. Variable *best_gain* is updated whenever the gain of an item becomes larger than the current value of *best_gain* as the result of $Insert$ or $Move()$. After a $Delete()$, *best_gain* may have to be decreased (if the bucket containing the item of highest gain was emptied) by scanning for the next non-empty bucket. However, the scan for the next non-empty bucket can be deferred until the next $Max()$ operation. If the time spent on scanning is ignored, $Max()$ can be done in constant time.

The total time spent scanning for the next non-empty bucket during $Max()$ is proportional to the number of buckets, plus the sum over all $Max()$ operations of

the increase in *best_gain* since the previous $Max()$. Except the increases due to the initial insertions, which account for a total of at most M, *best_gain* cannot increase in value unless the *gain* of an individual item increases by at least the same amount. Thus the time spent on scanning is $O(Q_M \Delta_{max} + M)$. □

In the implementation of our heuristics with bucket-list data structure, we keep all possible swaps formed (u, v) in the bucket-list together with their associated gains. Let d_{max} be the maximal vertex degree in G, $0 \leq d_{max} \leq (n-1)$.

- For move-set $S_2(\pi)$, whenever two vertices u and v are swapped, the mapped distance between u and one of its neighbors in G can chane by at most $\log n - 1$; similarly, the mapped distance between v and one of its neighbors in G can change by at most $\log n - 1$. Therefore the maximum gain (M) of any single swap is in the range $\pm 2 d_{max}(\log n - 1) = O(d_{max} \log n)$.

- For move-set $S_1(\pi)$, whenever vertex $u \in V$ is swapped with vertex $v = \eta(\pi, u, i), 1 \leq i \leq d$, the mapped distance between $\pi(u)$ (or $\pi(v)$) and one of its neighbors in G can change by at most 1 because $\pi(u)$ and $\pi(v)$ are cube neighbors. Therefore the maximum gain (M) of any single swap is in the range $\pm 2 d_{max} = O(d_{max})$.

3.4 Benchmark graphs

Three classes of task graphs are generated and used to compare the solution quality of various heuristics. They are random graphs, geometric graphs, and perturbed regular graphs (cubes and meshes). The former two classes of graphs are mainly characterized by two parameters: n, the vertex number; and d, the expected degree for each vertex.

Generation of random graphs:

Given positive integers n and d, define $p = d/(n-1)$. Value p specifies the probability that any given pair of vertices constitutes an edge. The edge weight is generated

randomly in some specific integer ranges.

Generation of geometric graphs:

Given positive integers n and d, define $k = \sqrt{d/(n\pi)}$. The coordinates of n vertices are first generated randomly on a unit square plane. Two vertices share a connecting edge if and only if the Euclidean distance between them is k or less. The weight for any edge is the ceiling integer of the product of a scale factor S and the ratio of the distance between the vertices incident to the edge over k. Figure 3.2 shows a

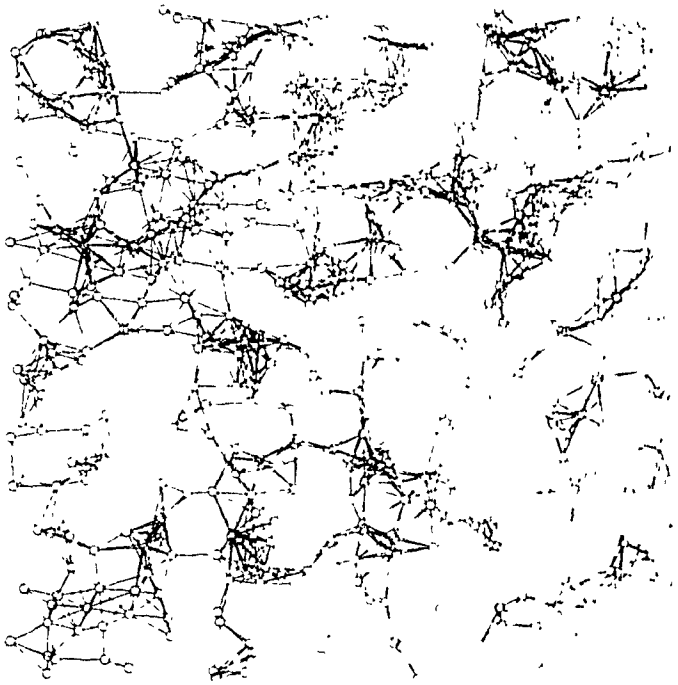


Figure 3.2: Geometric graph G512

geometric graph G512 generated with $n = 512$ and $d = 9$.

Generation of perturbed regular graphs:

Given positive integers n and k, where $n = 2^d$, we can generate a perturbed cube by first generating a d-cube, then randomly delete (add) k edges from (to) this d-cube.

name	n	d	w	#	d_{ave}	d_{min}	d_{max}	$\|E\|$
R128_3	128	3	1	10	3.036	0	8	194.3
R128_7	128	7	1	10	6.952	0	12	444.9
R128_10	128	10	1	10	9.978	1	18	638.6
R512	512	9	1	1	9.188	1	18	2352.0
R128W_3	128	3	1-5	10	3.036	0	8	194.3
R128W_7	128	7	1-5	10	6.952	0	12	444.9
R128W_10	128	10	1-5	10	9.978	1	18	638.6
R512W	512	9	1-5	1	9.188	1	18	2352.0

Table 3.1: Characteristics of the random benchmark graphs

Given positive integers n_1, n_2, and k, where $n_1 \cdot n_2 = 2^d$, we can generate a perturbed 2-dimensional mesh by first generating an $n_1 \times n_2$ mesh, and then randomly delete (add) k edges from (to) this mesh. Since the optimal embeddings of this class of graphs into hypercubes of the same size are easier to be obtained, we can use them to get more objective evaluations for our heuristics.

name	n	d	$\mathcal{S}$	#	d_{ave}	d_{min}	d_{max}	$\|E\|$
G128_3	128	3	1	10	2.778	0	9	177.8
G128_7	128	7	1	10	6.166	0	14	394.6
G128_10	128	10	1	10	8.605	0	17	550.7
G512	512	9	1	1	8.133	0	16	2082.0
G128W_3	128	3	5	10	2.778	0	9	177.8
G128W_7	128	7	5	10	6.166	0	14	394.6
G128W_10	128	10	5	10	8.605	0	17	550.7
G512W	512	9	5	1	8.133	0	16	2082.0

Table 3.2: Characteristics of the geometric benchmark graphs

All of our sets of benchmark graphs are specified in Tables 3.1, 3.2, and 3.3. The first letter of a graph-set name designates the graph class: R for random graph, G for geometric graph, C for cube, and M for mesh.

- For each set of the random and geometric graphs, we specify its vertex number n, expected degree d, range for w (for random graphs), and scale-factor $\mathcal{S}$ (for

name	n	#	d_{ave}	d_{min}	d_{max}	$\lvert E\rvert$
C7	128	1	7	7	7	448
C7_P3	128	10	7.017	7	8	451
C7_M3	128	10	6.953	6	7	445
C7_P7	128	10	7.109	7	8	455
C7_M7	128	10	6.891	6	7	441
M8_16	128	1	3.625	2	4	232
M8_16_P3	128	10	3.672	2	5	235
M8_16_M3	128	10	3.578	2	4	229
M8_16_P7	128	10	3.734	2	5	239
M8_16_M7	128	10	3.516	2	4	225

Table 3.3: Characteristics of the perturbed regular benchmark graphs

geometric graphs). We also list their average degree d_{ave}, minimum degree d_{min}, maximum degree d_{max}, average edge number $|E|$, and number of graphs #. The last four entries are averages over all the graphs in the set. We in general choose small d as most interesting applications involve graphs with a low average degree, and because such graphs are better for distinguishing the performance of different heuristics than denser ones [31].

- For each set of the perturbed regular graphs, we specify its vertex number n. For each name for perturbed cubes, "C" is followed by a number d specifying the dimension of the graphs. For each name for perturbed meshes, "M" is followed by two numbers $n_1\_n_2$ specifying a 2-dimensional $n_1 \times n_2$ mesh. At the end of each graph-set name, "Pk" stands for graphs with k edges randomly added; and "Mk" stands for graphs with k edges randomly deleted. We use an 8×16 mesh to generate our perturbed meshes. We also list their average degree d_{ave}, minimum degree d_{min}, maximum degree d_{max}, average edge number $|E|$, and number of graphs #.

Although neither of these three classes is likely to arise in a typical application, they provide the basis for repeatable experiments, and, it is hoped, constitute a broad enough spectrum to yield insights into the general performance of the heuristics.

3.5 Test bed

All of our experiments are carried out on a Sun-Sparc 1+ workstation under SunOS Release 4.01. Since the heuristics are randomized, one run on one graph may not be conclusive. Ideally, the most accurate results are obtained with a huge number of graphs and a huge number of iterations per graph. However, we cannot afford excessive computation time for each experiment, since many experiments are to be conducted. In this research, most running times and solution costs reported are based on running 10 times for each of the 10 problem instances generated randomly for a specific set of parameters. This choice is a compromise between the reliability of the data we obtained and the time we can afford.

In our experiments, all heuristics are run on the same set of benchmark graphs. Thus the average solution cost of each heuristic reflects its relative solution quality. Also, all the iterative improvement heuristics are given the same set of initial solutions for each graph, so they begin at the same starting point. In case that greedy heuristics are used as front ends of local search variants, the solutions of the former are fed directly to the latter.

3.6 Summary

In this Chapter, we described some problem-specific design issues which are foundations for all of our heuristics and implementations for hypercube embedding. We described the criteria for neighborhood design. We introduced the gain function and its update for objective function $\mathcal{D}(\pi)$. A bucket-list data structure employed in both Chen's and our implementations of embedding heuristics, the benchmark graphs used in our experiments, and the test environment were also presented.

Chapter 4

Established Hypercube Embedding Approaches

Based on our literature survey, we found that the major approaches applied to hypercube embedding problems are greedy, local search, Kernighan-Lin, and simulated annealing. Chen studied the most hypercube embedding approaches and made extensive comparisons for minimizing $\bar{\mathcal{D}}(\pi)$ (without edge weights) [10]. To conduct performance comparisons between Chen's heuristics and our new proposed tabu search heuristics, we summarize in this chapter the most competitive approaches in the literature in the contexts of their adaptations to the hypercube embedding problem reported in [10]. We also improve Chen's simulated annealing heuristic and make the relevant performance comparisons.

4.1 Greedy approach

Greedy approach is a kind of constructive technique. It is very efficient to be used to generate better than random solutions. The generic form of a greedy heuristic for hypercube embedding is given in Figure 4.1. The heuristic gradually decreases the number of unmapped vertices until every vertex has been mapped. During each iteration, v, a vertex in V having most neighbors already mapped, is chosen greedily as the next vertex to be mapped. Given v, the heuristic then greedily chooses h, the

While not all vertices in V have been mapped to H **do**:

Choose any vertex $v \in V$ which has most neighbors already mapped to H.

Choose any $h \in V_h$ such that the total distance between v and its mapped neighbors is minimized if v is mapped to h.

Map v to h.

Report the embedding and its cost.

Figure 4.1: Greedy approach for hypercube embedding

hypercube vertex to which v is mapped, such that the total distance between v and its mapped neighbors is minimized if v is mapped to h.

Chen implemented three greedy heuristics for hypercube embedding based on the different methods of choosing h and v in the generic greedy heuristic shown in Figure 4.1. The *simple greedy* (SG) chooses vertex h in a predetermined sequence the binary represented gray-code sequence. The *fast gre* (FG) chooses vertex h according to the principle that the maximal number of edges in G incident to v should be mapped preserving neighborship. The third one was proposed by Chen (referred to as G). It is slightly different from the variants of the generic greedy heuristic (SG and FG). Basically, the constraint of v being chosen before h is eliminated; the (v, h) pair is chosen from all possible candidates in $V \times H$ to minimize the summation of mapped distances between v and all its neighbors in G. Chen's G is more powerful than SG and FG for random graphs but less efficient.

The time complexities of SG, FG, and G are $O(m + n)$, $O(m \log n)$, and $O(mn)$ respectively, where m is the number of edges in graph G, i.e., $m = |E|$ [10].

Our experiments confirmed Chen's claim: SG is the fastest heuristic, but also gives worst solution quality for random and geometric graphs. However SG can embed meshes and cubes optimally. G generates better solution than SG and FG for random and geometric graphs, but is less efficient. While the solution quality and running

1. Get a random initial solution π.
2. Let $g_0 = 0$.
3. **Repeat**
   - 3.1 Let $\hat{\pi}_0 = \pi$, $T = \emptyset$, $i = 1$.
   - 3.2 **While** $S(\hat{\pi}_{i-1}, V - T) \neq \emptyset$ **do:**
     - 3.2.1 Find a move s_i in $S(\hat{\pi}_{i-1}, V - T)$ such that $\Delta = \mathcal{D}(\hat{\pi}_{i-1}) - \mathcal{D}(s_i(\hat{\pi}_{i-1}))$ is maximized.
     - 3.2.2 Let $\hat{\pi}_i = s_i(\hat{\pi}_{i-1})$.
     - 3.2.3 Let $\hat{g}_i = \hat{g}_{i-1} + \Delta$, $T = T \cup V(s_i)$, $i = i + 1$
   - 3.3 Let $g_k = \max\{g_1, g_2, \ldots, g_{i-1}\}$.
   - 3.4 Let $\pi = \hat{\pi}_k$.

   Until $g_k \leq 0$.
4. Return π.

Figure 4.2: Kernighan-Lin approach

time of FG are between those for SG and G, it can embed optimally for meshes, cubes, and meshes and cubes with some edges deleted. Due to their efficiency and ability to embed regular graphs, FG and SG are good front end heuristics for other iterative heuristics.

4.2 Kernighan-Lin approach

The Kernighan-Lin approach differs from the local search at two key aspects: (1) It is more aggressive during each iteration by always using the move among the current candidates that can maximize the gain; (2) It gives each vertex a chance to move, and the "uphill" moves will be accepted as long as the compound gain of the sequence of moves including these "uphill" ones is positive.

Let $V(s)$ be the subset of vertices in V involved in the move s. We can present the Kernighan-Lin approach in Figure 4.2. Given a random initial solution (π), the heuristic executes a main loop (step 3) which will not stop until no positive compound gain can be found. During each iteration of this main loop, an inner loop (step 3.2) is used to move each vertex exactly once, and T is used to maintain the

vertices already involved in some previous moves in this main iteration. The ith iteration of the inner loop makes one move (s_i), updates the current solution ($\tilde{\pi}_i$), and calculates the compound gain up to that stage ($\tilde{g}_i$). The chosen move (s_i) must maximize the gain (Δ) and involve no vertices already moved in the current main iteration ($s_i \in S(\tilde{\pi}_{i-1}, V - T)$). After this inner loop the solution corresponding to the maximum compound gain ($\tilde{g}_k$) is used as the initial solution for the next iteration of the main loop.

As an implementation detail, Chen made the following adaptations for Kernighan-Lin heuristic in [10].

1. Move-set $S_1(\pi)$ is used to define the neighborhood of solution π.

2. The termination condition of the main loop (step 3) is relaxed to allow ρ consecutive uphill moves, where $\rho \geq 0$ is an integer typed parameter. The step 3.4 is changed to "if $\tilde{g}_k > 0$, then $\pi = \tilde{\pi}_k$; otherwise, $\pi = \tilde{\pi}_r$ where $0 < r < i$ is a random integer.

3. Each vertex can be swapped more than once. In the inner loop (step 3.2), suppose a move formed (u, v), $u, v \in V$ and $v = \eta(\pi, u, i), 1 \leq i \leq d$, gives the maximal Δ, then both vertices u and v are allowed to be swapped again with other vertices in later iterations if they only occur as the cube neighbors of other vertices formed $(*, u)$ or $(*, v)$.

We call the Kernighan-Lin heuristic with all-swap move-set KL, the Kernighan-Lin heuristic with cube-neighbor move-set KLC. For every inner loop iteration, there are $O(\log n)$ $Delete()$ operations and $O(\tilde{d} \log n)$ $Move()$ operations. For every outer loop iteration, there are $O(n)$ inner loop iterations, hence we have $Q_M = O(n)$ $Max()$ operations. Hence $Q_I = O(n(\log n + \tilde{d} \log n)) = O(m \log n)$. We have known that the maximum amount by which the gain of any swap may increase is $\Delta_{max} = O(d_{max})$, and the maximum gain of any swap is $M = 2\Delta_{max} = O(d_{max})$ (see page 23 in Section 3.3). According to Lemma 1, the time complexity for one outer loop iteration

of KLC is $O(m \log n + nd_{max} + d_{max}) = O(m \log n + nd_{max})$. The total time complexity of KLC is thus $O(l(m \log n + nd_{max}))$ where l is the total outer loop iteration number.

The experimental results showed: (1) Without uphill moves, although significantly slower, KL does give results much better than those for KLC without uphill moves. (2) With increasing ρ, however, KLC does so well that it outperforms KL. Chen claimed that KLC outperforms simulated annealing (especially for random graphs) [10].

4.3 Simulated annealing approach

Simulated Annealing (SA), developed by Kirkpatrick et al. [36], can be viewed as an enhanced version of the local search. It attempts to avoid being trapped in poor local optima by allowing occasional uphill moves. This is done under the influence of a random number generator and a control parameter T called the *temperature*. As typically implemented [31], the SA approach involves a pair of nested loops and three parameters: the initial temperature T_0; the *cooling ratio* T_r, $0 < T_r < 1$, and the integer typed *temperature length* L (see the generic simulated annealing heuristic in Figure 4.3). At the end of step 3, the term *frozen* refers to a state in which no further improvement on $cost(\pi)$ seems likely.

The heart of this procedure is the loop at Step 3.1. Note that $e^{-\Delta/T}$ will be a number in the interval $(0, 1)$ when $T > 0$ and $\Delta > 0$, and rightfully can be interpreted as a probability that depends on Δ and T. The probability that an uphill move will be accepted diminishes as the temperature declines, and, for a fixed temperature T, small uphill moves have higher probabilities of acceptance than larger ones. This particular method of operation is motivated by a physical analogy, best described in terms of the physics of crystal growth [36]. It has been proven that the heuristic will converge to a global optimum if the temperature is lowered in exponential time and the initial temperature is chosen sufficiently high [27].

We use CSAC to denote Chen's simulated annealing heuristic. Chen made the

1. Generate an initial embedding π.
2. Let $T = T_0$, an initial temperature.
3. **Repeat**
   - 3.1 **For** L iterations **do**
     - 3.1.1 Choose a random move $s \in S(\pi)$.
     - 3.1.2 Let $\Delta = \bar{\mathcal{D}}(s(\pi)) - \bar{\mathcal{D}}(\pi)$.
     - 3.1.3 **If** $\Delta \leq 0$ **then**
       $\pi = s(\pi)$.
       else
       $\pi = s(\pi)$ with probability $e^{-\Delta/T}$.
   - 3.2 $T = T_r \times T$ (reduce temperature).

   Until frozen condition is met.
4. Return the best π visited.

Figure 1.3: Simulated annealing approach

following adaptations for hypercube embedding:

1. Move-set $S_1(\pi)$ is used to define the neighborhood of solution π.
2. Let $L = |S_1| \cdot$ SIZE_FACTOR, where SIZE_FACTOR is a parameter.
3. Frozen condition: The acceptance rate of the proposed moves is measured for each temperature. The heuristic stops when for five temperatures the acceptance rate is lower than MIN_PERCENT and the best visited solution is not improved in that period of time. Here MIN_PERCENT is another parameter in the range $(0, 1)$.

Several parameters that affect running time and solution quality must be adjusted for CSAC implementation. Since all these parameters are not independent, for each of our benchmark graphs, we tune parameters one at a time, and repeat the process until no perturbation of the parameters can improve the performance.

1. Generate an initial embedding π.
2. Let $T = T_0$, an initial temperature.
3. **Repeat**
   - 3.1 **While** $|chain\_accept\_ratio - epoch\_accept\_ratio| > \epsilon$ do
     - 3.1.1 **For** c iterations **do**
       - Choose a random move $s \in S_1(\pi)$.
       - Let $\Delta = \mathcal{D}(s(\pi)) - \mathcal{D}(\pi)$.
       - **If** $\Delta \leq 0$ **then**
         - $\pi = s(\pi)$.
       - **else**
         - $\pi = s(\pi)$ with probability $e^{-\Delta/T}$
     - 3.1.2 Calculate $epoch\_accept\_ratio$ and $chain\_accept\_ratio$.
   - 3.2 $T = T_r \times T$ (reduce temperature).

   Until the best π is not improved for λ consecutive T's.
4. Return the best π visited.

Figure 4.4: New simulated annealing for hypercube embedding

4.4 An adaptive version of simulated annealing

In CSAC, the length of each Markov chain (iterations under the same temperature) is a constant. It means that the same effort is made for each temperature no matter what acceptance ratio is. Experiments show that this implementation is very sensitive to parameter tuning and inefficient if we use conservative size factors. Therefore we improve CSAC heuristic by introducing adaptive Markov chain lengths.

4.4.1 New SA Heuristic

Our new annealing heuristic with cube-neighbor neighborhood (SAC), outlined in Figure 4.4, is a refinement of the scheme used in [49]. We use the statistics of acceptance ratio for each Markov chain to determine if the relative equilibrium has been reached. If this is the case, the chain can be terminated. In this way, instead of blindly wasting time in continuing a stabilized chain, we can spend more time to

intensify the solution search in more promising chains.

Five parameters are involved in our SAC: the initial temperature T_0, the cooling ratio T_r, the integer typed *epoch* length ϵ, the float typed *tolerance* ϵ, and the integer typed termination parameter λ. To be more specific, we define an *epoch* to be ϵ consecutive iterations. After the execution of an *epoch*, the acceptance ratio during this epoch is compared with the total acceptance ratio since the beginning of this temperature. If the difference between these two acceptance ratios is larger than a given tolerance ϵ, another *epoch* is given at the same temperature. Otherwise, the chain is assumed to be in equilibrium at that temperature, and the temperature is reduced by a cooling ratio T_r. This process is repeated until the best solution is not improved for λ consecutive temperatures. In case that the value of tolerance ϵ is too small to terminate the iterations at one temperature, a counter will be used to terminate the loop when the pre-defined maximal chain length $(100 \cdot |S_1(\pi)|)$ is reached. The comparison of the two acceptance ratios is started after the first two epochs to avoid terminating the chain right after the execution of the first epoch.

4.4.2 Parameter tuning for SAC

The performance of any SA heuristic depends heavily on the parameter settings. In attempting to optimize the performance of our SAC heuristic, we faced the same kinds of questions that any potential designer of SA heuristics must address. Since there are too many related parameters for us to investigate, we study one or two of them at a time in hope of isolating their effects. This process is repeated for all the parameters until their values stabilize. In the following we show the final round of parameter tuning to get our standard parameter settings for random graph R512W: $T_0 = 5, T_r = 0.97, \epsilon = 1000, \epsilon = 0.001$, and $\lambda = 8$.

Initial temperature T_0

Let us first concentrate on the effect of initial temperature T_0, and do so by taking a more detailed look at the operation of the heuristic. In Figure 4.5, curve (a) presents a

time exposure of an annealing run on random graph R512W. The standard parameters were used with the exception that the initial temperature T_0 was increased from 5 to 50 so that the initial acceptance ratio α is increased from 20% to 90%. During the run, the solution cost was sampled every 1000 iterations, and these values were plotted as a function of time at which they were encountered.

We can observe from curve (a) that slow progress is made at the beginning of the schedule. For the first 100 or so samples (100,000 iterations, or 10 seconds) the costs can barely be distinguished from those of totally random partitions (the costs for 1,000 randomly generated partitions for this graph range from 4.400 to 4.625 with a mean of 4.507). But there still remains the question of whether the time spent at high temperatures might somehow be necessary, but not be seen yet. Curves (b) (c)

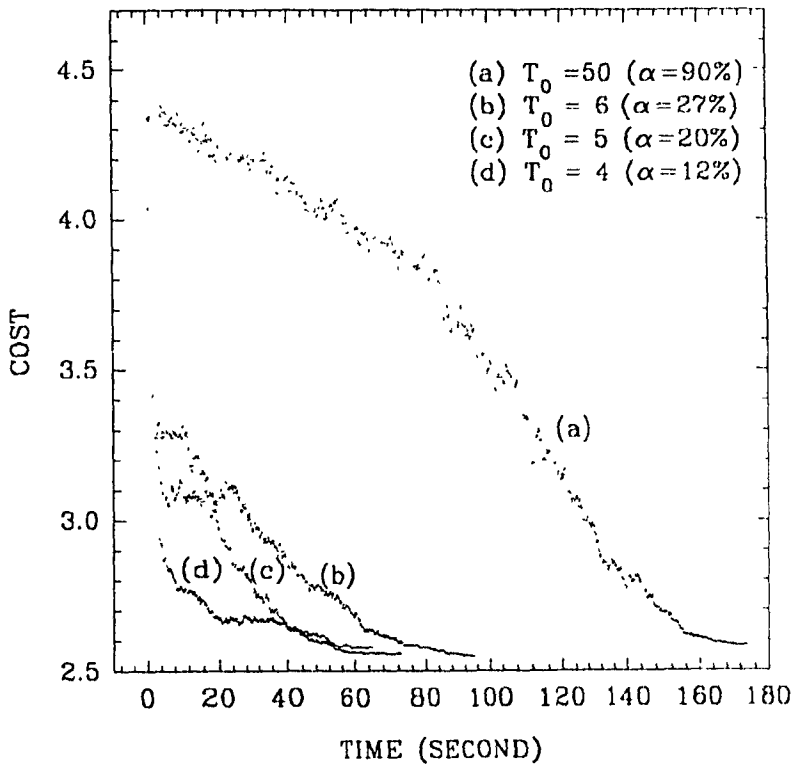


Figure 4.5: The effect of different initial temperatures

and (d) in Figure 4.5 address this issue by presenting time exposures of three shorter annealing runs with initial acceptance ratio 27%, 20%, and 12% (using T_0 = 6, 5, and

1) respectively. Experiments show that a T_0 greater than 5 does not help in solution quality but increases the running time. When $T_0 < 5$, the performance is worse than that when $T_0 = 5$. With $T_0 = 5$, we got the better solution in shorter time than those with other initial temperatures. Therefore we choose $T_0 = 5$.

Epoch length e and Tolerance ϵ

Parameters e and ϵ determine the length of each Markov chain. Let e and ϵ take on various combinations of values from {50, 100, 200, 500, 800, 1000, 2000} and {0.1, 0.01, 0.005, 0.002, 0.001, 0.0008} respectively. We keep all the other parameters at their standard settings. From Table 4.1 we can see that when $e < 500$ and $\epsilon < 0.005$ the solution costs are improved but the speed of improvement is much slower than that with $\epsilon > 500$. In general, the better solution quality can be obtained by either increasing epoch length e or decreasing tolerance ϵ. When doing so, the running time is increased too. As a compromise between solution quality and running time, we use $e = 1000$ and $\epsilon = 0.001$.

Cooling ratio T_r and termination parameter λ

Parameters T_r and λ together control how many times the temperature is reduced in the cooling process. Table 4.2 illustrates the effects of these parameters for random graph R512W. We fix all parameters except the two in question at their standard settings. We let T_r and λ take on various combinations of values from {0.8, 0.85, 0.9, 0.95, 0.96, 0.97, 0.98} and {1, 3, 5, 8, 10}, respectively. We see from Table 1.2 that the solution quality is getting better (and the running time is also getting longer) as T_r is getting larger. Also we can see that the solution costs are almost not changed when λ is larger than 8. Therefore we choose the combination of $T_r = 0.97$ and $\lambda = 8$ which gives the best compromise between solution quality and running time.

4.4.3 Performance comparisons between SAC and CSAC

We designed experiments to demonstrate the performance of our SAC heuristic relative to Chen's CSAC heuristic. For CSAC, we used the same method to tune param-

c	ϵ					
	0.1	0.01	0.005	0.002	**0.001**	0.0008
50	3.50	4.33	18.51	201.20	129.60	310.70
100	4.37	4.90	6.48	109.00	166.50	192.30
200	5.91	8.10	9.74	14.79	111.40	107.00
500	7.15	14.12	17.96	25.09	34.21	38.75
800	8.98	19.48	24.38	29.50	44.28	52.11
1000	9.84	22.37	27.47	37.13	**52.63**	58.62
2000	15.25	36.12	46.07	60.62	83.31	88.28
3000	20.55	48.65	60.30	81.45	109.50	119.00

(a) Running Time

c	ϵ					
	0.1	0.01	0.005	0.002	**0.001**	0.0008
50	2.913	2.880	2.792	2.614	2.562	2.530
100	2.878	2.836	2.791	2.663	2.579	2.575
200	2.840	2.761	2.737	2.681	2.617	2.605
500	2.806	2.687	2.664	2.635	2.607	2.602
800	2.772	2.650	2.637	2.609	2.586	2.580
1000	2.763	2.645	2.628	2.607	**2.581**	2.571
2000	2.699	2.605	2.599	2.573	2.555	2.550
3000	2.673	2.581	2.576	2.561	2.541	2.537

(b) Average cost

Table 4.1: Dependence of running time and average cost on c and ϵ for R512W

λ	T_r						
	0.8	0.85	0.9	0.95	0.96	**0.97**	0.98
1	25.23	27.38	22.95	22.82	22.01	20.80	18.36
3	27.34	30.33	32.41	40.47	46.66	48.99	44.27
5	28.71	30.62	33.71	42.79	49.22	58.08	64.40
8	29.53	31.03	33.77	44.28	52.41	**61.76**	72.49
10	29.74	31.57	34.48	45.28	53.10	62.05	73.34

(a) Running Time

λ	T_r						
	0.8	0.85	0.9	0.95	0.96	**0.97**	0.98
1	2.648	2.636	2.720	2.747	2.774	2.799	2.827
3	2.646	2.633	2.621	2.588	2.584	2.594	2.637
5	2.646	2.633	2.620	2.586	2.582	2.569	2.555
8	2.646	2.633	2.620	2.586	2.581	**2.568**	2.550
10	2.646	2.633	2.620	2.586	2.581	2.568	2.549

(b) Average Cost

Table 4.2: Dependence of running time and average cost on T_r and λ for R512W

eters, and found the best parameter settings for each graph. The results are reported in Table 4.3 for random graphs and Table 4.4 for geometric graphs. We also report the total mapped distance $\mathcal{T}(\pi)$ and maximal mapped distance $\mathcal{D}(\pi)$ in the tables for reference.

The results show that SAC outperforms CSAC in both solution quality and running time. For all of our random graphs, for example, SAC spends on the average 75% of time which CSAC takes while improves on the average cost of CSAC by 3%.

Graph	$\mathcal{T}(\pi)$		$\hat{\mathcal{D}}(\pi)$		$\mathcal{D}(\pi)$		Time	
	CSAC	SAC	CSAC	SAC	CSAC	SAC	CSAC	SAC
R128_3	345.4	280.4	1.776	1.439	1.8	1.0	12.22	10.08
R128_7	935.0	926.2	2.101	2.081	5.6	5.6	12.22	8.61
R128_10	1464.5	1460.2	2.293	2.286	6.0	6.1	18.13	16.08
R512	6383.0	6375.0	2.711	2.710	8.0	8.0	48.55	15.01
R128W_3	820.4	797.1	1.410	1.372	13.5	12.4	11.87	7.92
R128W_7	2557.1	2551.0	1.925	1.924	19.8	19.1	18.22	17.65
R128W_10	4193.2	4176.8	2.195	2.187	23.0	23.0	18.20	12.12
R512W	18580.0	18567.0	2.613	2.611	30.0	30.0	51.42	20.00

Table 4.3: Comparisons between two SA heuristics for random graphs

Graph	$\mathcal{T}(\pi)$		$\hat{\mathcal{D}}(\pi)$		$\mathcal{D}(\pi)$		Time	
	CSAC	SAC	CSAC	SAC	CSAC	SAC	CSAC	SAC
G128_3	276.2	247.9	1.556	1.396	3.8	3.0	11.82	10.61
G128_7	676.6	675.2	1.718	1.715	4.0	3.8	11.78	10.43
G128_10	1034.4	1026.0	1.878	1.863	4.4	3.8	13.17	12.73
G512	4087.0	4016.0	1.715	1.929	6.0	6.0	15.55	11.33
G128W_3	813.4	796.8	1.362	1.331	12.1	12.1	11.82	7.60
G128W_7	3516.3	3329.8	1.845	1.749	26.8	25.3	12.91	7.92
G128W_10	6154.1	6088.8	1.977	1.956	32.8	32.6	16.91	7.18
G512W	12315.0	12251.0	1.976	1.966	20.0	20.0	72.57	67.95

Table 4.4: Comparisons between two SA heuristics for geometric graphs

4.5 Summary

This Chapter summarized the main ideas of previously established optimization approaches including greedy, Kernighan-Lin, and simulated annealing, as well as Chen's adaptations of them for hypercube embedding. We introduced our new adaptive simulated annealing heuristic SAC and made the performance comparisons with Chen's annealing heuristic CSAC. The experiments show that SAC outperforms CSAC in both solution quality and running time. In the following chapters, we will use SAC to represent simulated annealing heuristic in performance comparisons.

Chapter 5

Tabu Search Heuristic with Objective Function $\bar{D}(\pi)$

In this chapter, our tabu search heuristic TSC with objective function $\bar{D}(\pi)$ is described. Intensive and extensive performance comparisons are made for the most competitive heuristic FG+KLC claimed by Chen, our new proposed TSC, and our improved simulated annealing heuristic SAC. Experiments show that TSC outperforms SAC, and both of them outperform Chen's FG+KLC with a big margin in both solution quality and running time.

5.1 Tabu search heuristic and its characteristics

In this section we describe our tabu search heuristic and its characteristics. Figure 5.1 presents our tabu search heuristic for hypercube embedding with objective function $D(\pi)$. Given an (random or greedy generated) initial solution, it repeats the loop at step 1 until the best solution π visited is not improved for ρ consecutive iterations. At each iteration, our tabu search heuristic always chooses the move s which has the best cost improvement over all possible candidates from the move-set $S(\pi)$ of the current solution π and has not been used in the last t iterations.

In the following we discuss the main design issues that directly affect the perfor-

1. Initialize the tabu list $T = \phi$.
2. Get an initial solution π.
3. Initialize all possible gains according to π.
4. **Repeat**
   - 4.1 Find a move $s \in S(\pi) - T$ that maximizes $\mathcal{D}(\pi) - \mathcal{D}(s(\pi))$.
   - 4.2 Set $\pi = s(\pi)$ and insert s into T.
   - 4.3 Update gains.

   Until best π visited is not improved for ρ consecutive iterations.
5. Return the best π visited.

Figure 5.1: Tabu search heuristic

mance of our tabu search heuristic.

5.1.1 Move-set design

From Figure 5.1 we can see that there is a trade-off between the number of iterations and the aggressiveness of each iteration. With large move-set size, each iteration is more aggressive but more time consuming; with small move-set size, however, the search can afford more iterations but each iteration is less aggressive.

To reduce the search time for the best neighbor, we employ the bucket-list data structure to implement a priority queue. We keep all possible moves associated with their gains in the bucket-list. The best candidate move can thus be found in constant time. Whenever a move is done, we update the affected gains. By using the bucket list, the major time for each iteration is spent on gain update.

We implemented tabu search heuristics with both cube-neighbor move set and all swap move-set, and call them TSC and TS respectively. According to Lemma 1, we have the following time complexities for our tabu search heuristic.

- Move-set $S_2(\pi)$: We have known that the time complexity of gain initialization is $O(mn)$ and the time complexity of gain update for each move is $O(n)$ (see

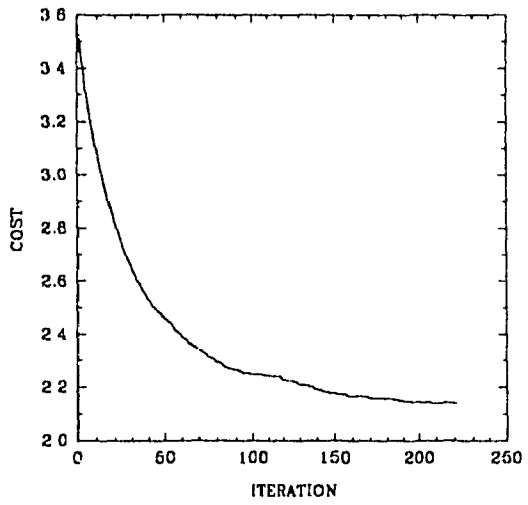
Subsection 3.2.1). Let l be the total iteration number. The number of calls to $Max()$ is $Q_M = l$; Q_I is dominated by $O(lm)$ $Move()$ operations. $M = O(d_{max} \log n)$ (see page 23 in Section 3.3. We have $\Delta_{max} = O(d_{max} \log n)$. The overall running time of TS is therefore $O(mn + l(m + d_{max} \log n))$.

- Move-set $S_1(\pi)$: Similarly, the time complexity of gain initialization is $O(m \log n)$ and the time complexity of gain update for each move is $O(\check{d}(\hat{d} + \log n))$ (see Subsection 3.2.2). The number of calls to $Max()$ is $Q_M = l$; Q_I is dominated by $O(l\hat{d}(\check{d} + \log n))$ $Move()$ operations. $M = O(d_{max})$ (see page 23 in Section 3.3. We have $\Delta_{max} = O(d_{max})$. The overall running time of TSC is therefore $O(m \log n + l\hat{d}(d_{max} + \log n))$.

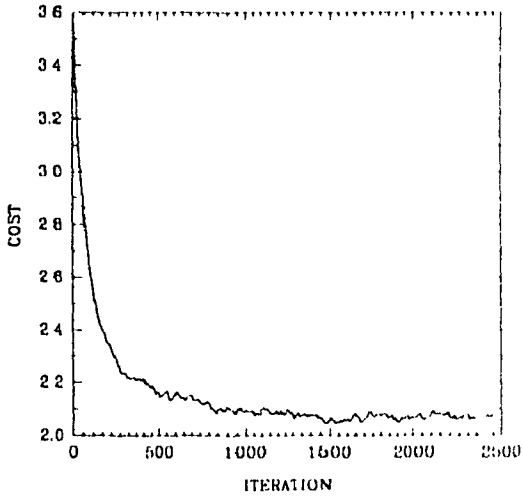
As shown in Figure 5.1, in each iteration, our tabu search heuristic always chooses the move s with the best cost improvement over all eligible moves from the move-set $S(\pi)$. For TS, whenever a move s is applied to the current solution π, updating affected gains will take time $O(m)$ (see Subsection 3.2.1). For TSC, however, it will only need time $O(\hat{d}(\hat{d} + \log n))$ to update the affected gains (see Subsection 3.2.2). For the same period of time, TSC can perform more iterations than TS. In general TSC has more chances to get to and jump out of a local optimum than TS. Figure 5.2 shows the progress of a TS run and a TSC run for the first random graph of R128_7. We see that the number of iterations for TS is just 10% of that for TSC; TS spends CPU time more than 50 times of that for TSC; and the solution quality of TS is worse than that of TSC.

5.1.2 The contents of the tabu list

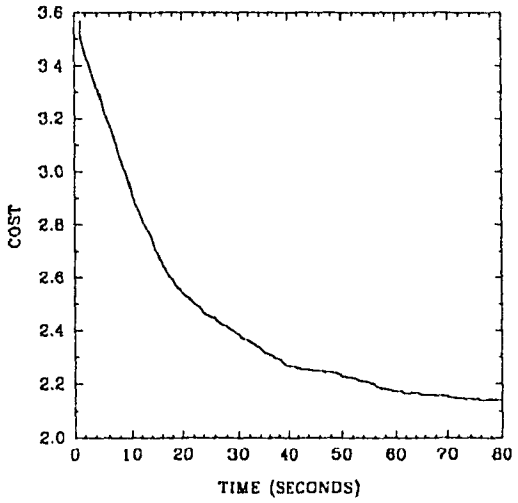
If move s is used to transform the current solution π to $s(\pi)$, the corresponding cell of the tabu list should store some attributes of $s(\pi)$ so that $s(\pi)$ will not be traversed again in the next t iterations. At one extreme, we can store solution π directly in the tabu list. But in practice, to save memory space and checking time, some attributes of $s(\pi)$ will be stored in the tabu list to prevent s or s^{-1} from being used in the next



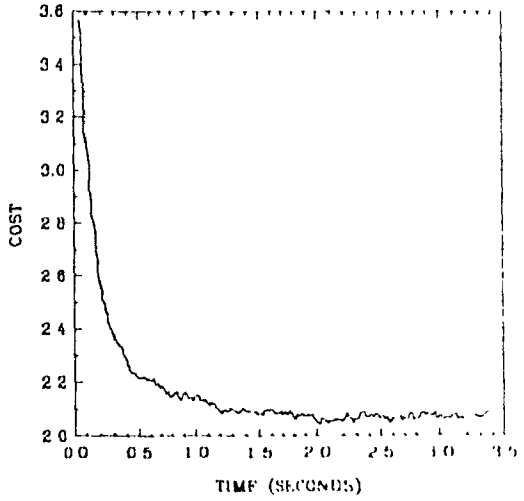
(a) Cost vs. iteration for TS



(c) Cost vs. iteration for TSC



(b) Cost vs. time for TS



(d) Cost vs. time for TSC

Figure 5.2: The progress of TS and TSC

l iterations.

In our implementation of tabu list, we tried the following two sets of attribute of $s(\pi)$. (a) Whenever a vertex u is swapped with another vertex v where $v = \eta(\pi, u, i), 1 \leq i \leq d$, we put pair (u, i) into tabu list. (b) Whenever a vertex u is swapped with another vertex v, we put pair $\{u, v\}$ into tabu list. We can see that the pair (u, i) prohibits vertex u from being swapped again with its ith cube neighbor in the following l iterations no matter what its ith cube neighbor is, while the pair $\{u, v\}$ only prohibits vertex u from being swapped again with vertex v in the

following t iterations. Therefore scheme (a) is more prohibitive than scheme (b). The experiments showed that the performance of scheme (a) is much poorer than that of scheme (b). Therefore we adopted scheme (b) in our final implementation.

5.1.3 The design of the adaptive tabu list

One of the major design issues for the tabu list is to decide its length. In general, the longer the tabu list is, the more time tabu status checking takes for each move, and the more restrictive the search process will be. On the other hand, a too short tabu list risks to introduce cycling in the solution space. For most tabu search heuristics in the literature, the tabu lists are of fixed length. Ideally, the tabu list length should be changed dynamically during the search process. For different problem instances, the best tabu list length may not be the same. It is usually hard to determine the best tabu list length for different problem instances. Even for the same problem instance, the best tabu list length may be different from time to time. For most applications, the tabu list length should be short at the beginning to allow most flexible searches, and be increased whenever the search is trapped in local optima.

In our heuristic, we propose a dynamic tabu list mechanism to capture the dynamic nature of the search process. Our motivation is to let the length of the tabu list change according to the variation of the recent solution quality. According to our experiments, the solution can always be improved easily until a local optimum is reached during the first certain iterations. During this period, we should reduce the size of tabu list to allow the search to reach a local optimum quickly. To help the search get out of the local optimum afterwards, the length of the tabu list should be increased until the solution cost is decreasing again. To prevent the tabu list length from being increased too much thus losing its aggressiveness, or decreased too much thus introducing cycles, we set up a range in which the length of the tabu list can be updated.

We implement the tabu list T as a circular list with variable size t. Let q be a variable indicating the position in T where the last move is recorded. When $q = t - 1$,

the next move will be stored in location 0 to replace its old contents.

Let t_b denote the base length of the tabu list; t_s be an integer, called *span*, defining the range of length variation, $t_s < t_b$; and t be the current tabu list length. Three adaptive tabu list strategies are tested. In the following strategies, the tabu list length will be updated only if the specified condition is met.

Strategy 1 $t_s = 0$ for all the time.

Strategy 2 After each t iterations, we check if the best solution cost is improved. If it is, the value of t is decreased by one if $t > t_b - t_s$; otherwise, the value of t is increased by one if $t < t_b + t_s$. Repeat this procedure until the termination condition is met.

Strategy 3 If the best solution cost is improved in the past t iterations, the value of t will be decreased by one if $t > t_b - t_s$; otherwise, the value of t will be increased by one if $t < t_b + t_s$ for each of the following iterations until an improvement on the best solution cost is found. Then after the next t iterations, we repeat the above procedure.

Strategy 4 This strategy is proposed by Taillard in [12]. Let span t_s be any integer. For every $2 \cdot (t_b + t_s)$ iterations, a random number r uniformly distributed over $[0, t_s]$ is used to change the tabu list length to $t_b + r$.

For the first strategy, the tabu list keep the same length all the time. For the second strategy, the update of the tabu list only occurs after every t iterations. For the third strategy, if the improvement on the best solution cost can be found for each t iterations, it is the same as the first strategy; otherwise, the tabu list length will be increased continuously until the improvement on the best solution cost is found. We see that the interval of checking whether the tabu list length should be updated is variable. For the fourth strategy, however, the length of tabu list is randomly updated after every constant number of iterations.

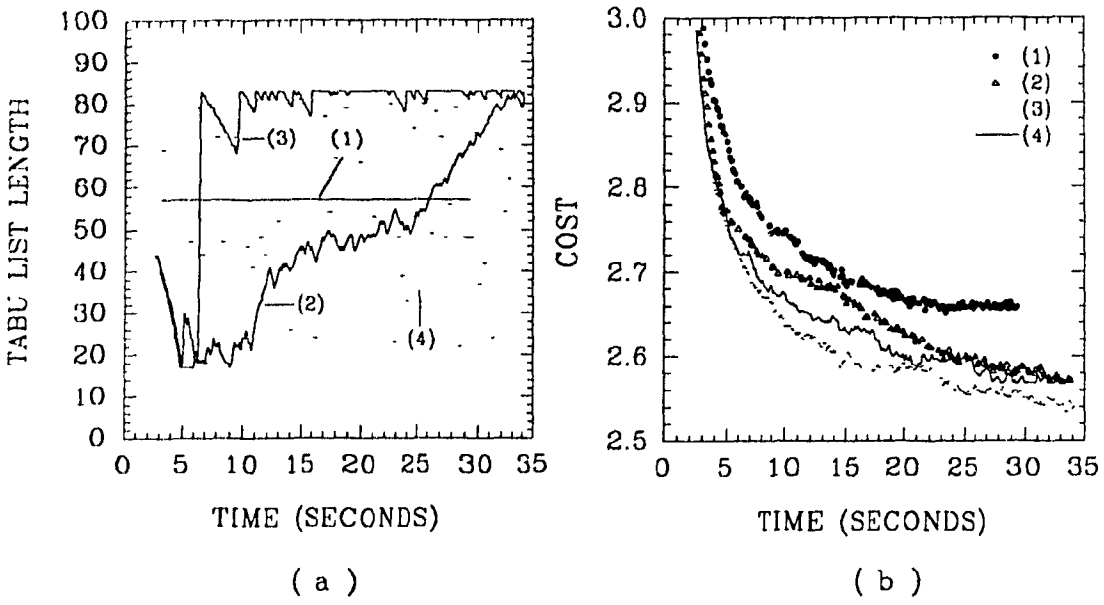


Figure 5.3: The effect of tabu list length on solution cost for R512W.

Figure 5.3 shows an example of the variation of tabu list length and solution costs during four runs of tabu search heuristic on graph R512W with the four different tabu list implementations. In the figure, (a) shows the variation of the tabu list length during the search, and (b) shows the solution cost comparisons for TSC with different adaptive tabu list strategies. Curve (1) is obtained by TSC with static tabu list; curve (2) is obtained by TSC with strategy 2; curve (3) is obtained by TSC with strategy 3; and curve (1) is obtained by TSC with strategy 1. Experiments show that TSC with strategy 3 outperforms the rest strategies. Therefore we adopt strategy 3 in our final version of TSC.

To further demonstrate the effects of t_b and t_s on solution quality, we conduct the following experiment. We let $\rho = 2000$, and let t_b and t_s take on values from {10, 20, 30, 40, 45, 50, 55, 60, 70, 80} and {0, 3, 7, 13, 18, 23, 28, 33, 38} respectively. We run our TSC for graph R512W. The results are shown in Table 5.1. From the column for $t_s = 0$ in Table 5.1, we can see that for $t_b \leq 50$, the average cost can be improved by increasing t_b. For most rows, however, we see that the solution cost can be further improved by choosing $t_s > 0$. We can see that the best solution cost in Table 5.1 (a) is obtained when $t_b = 50$ and $t_s = 7$, a combination taking the longest CPU time in

l_b	l_s								
	0	3	**7**	13	18	23	28	33	38
10	2.783	2.771	2.732	-	-	-	-	-	-
20	2.673	2.663	2.633	2.581	2.559	-	-	-	-
30	2.605	2.566	2.570	2.561	2.526	2.533	2.533	-	
10	2.558	2.552	2.556	2.519	2.529	2.519	2.513	2.539	2.530
15	2.555	2.557	2.559	2.516	2.513	2.530	2.546	2.512	2.519
50	2.536	2.514	**2.525**	2.537	2.547	2.540	2.510	2.528	2.515
55	2.552	2.539	2.558	2.531	2.566	2.556	2.511	2.517	2.562
60	2.532	2.510	2.538	2.511	2.511	2.550	2.535	2.517	2.532
70	2.553	2.514	2.531	2.517	2.542	2.550	2.553	-	-
80	2.512	2.529	2.555	2.514	2.510	-	-		

(a) Cost

l_b	l_s								
	0	3	**7**	13	18	23	28	33	38
10	9.71	11.86	12.55	-	-	-	-	-	
20	22.99	25.97	33.87	47.67	47.04	-	-	-	-
30	46.76	52.16	52.49	44.41	60.73	46.88	44.55	-	-
10	43.59	41.16	11.87	42.59	44.15	36.68	35.16	36.74	13.15
15	14.41	40.79	35.44	38.26	40.06	52.10	37.67	41.61	42.72
50	48.78	10.67	**51.23**	43.01	37.52	41.01	39.32	13.15	18.97
55	39.06	48.11	35.61	48.39	29.17	32.39	35.65	37.21	30.80
60	46.25	43.52	35.92	39.40	39.72	35.04	47.15	35.44	16.00
70	36.06	36.55	38.58	33.01	40.00	33.82	31.58	-	
80	37.42	44.87	29.30	31.68	40.62	-	-		

(b) Running time

Table 5.1: Dependence of average cost on l_b and l_s

Table 5.1 (b). Here the reader may ask whether we can get similar solution cost for $t_s = 0$ by increasing the value of ρ to increase the running time. Table 5.2 shows the comparisons of running time for TSC with adaptive tabu list and TSC with static tabu list to reach similar solution cost. Experiments show that the iteration number

t_b	t_s	ρ	iterations	time	$\bar{\mathcal{D}}(\pi)$
50	7	2000	21871	51.23	2.525
50	0	2500	22168	59.50	2.526
43	0	2800	28953	68.21	2.531
57	0	3300	28578	69.17	2.528

Table 5.2: Comparisons between static and adaptive tabu lists

for reaching a specific solution cost can be greatly reduced by using the adaptive tabu list strategy. According to Table 5.2, the iteration number with parameter setting $t_b = 50$ and $t_s = 7$ is 18% less than the average iteration number with parameter settings $t_b \in \{50, 43, 57\}$ and $t_s = 0$; the running time with the former parameter setting is 22% less than the average running time with the latter parameter settings; while the solution quality of the former is better than that of the latter. Therefore we decide to use $t_b = 50$ and $t_s = 7$ as our standard parameter setting for graph R512W.

5.1.4 Aspiration level

By experiments we observed that our heuristic has the tendency of intensifying searches in small solution subspaces. To explore the solution space more evenly, a long tabu list is usually necessary. Since the main role of aspiration level is to further intensify the solution search in local area, we conjecture that aspiration function may not be helpful for our heuristic. We tried a simple and popular aspiration function — the best cost found so far. Whenever a proposed move s is in the tabu list but the new cost $\mathcal{D}(s(\pi))$ is smaller than the best cost found so far, the tabu status of s will be overridden. Table 5.3 shows the performance comparisons between TSC without aspiration function and TSC with the aspiration function (TSCA) for our

heuristics	$\mathcal{D}(\pi)$	time
TSCA	2.530	51.19
TSC	2.525	51.23

Table 5.3: Performance comparisons between TSC and TSCA

standard graph R512W. Experiments show that TSCA does not perform better than TSC. For example, in about the same CPU time, the average solution cost of TSCA is worse than that of TSC. It confirmed our conjecture about the aspiration function. Therefore, for all of our reported experimental results in the following sections, we did not use the aspiration function.

5.1.5 Application of tabu status array

In general, the tabu list is implemented as a circular list. To check if a move is in the tabu list, we have to go through each cell in the list. Hence the checking time is proportional to the length of the tabu list. When the tabu list length is large, the time spent for checking tabu status at each iteration can be significant. To make our TSC more efficient, an $n \times n$ array TABUSTATE is employed to accelerate the checking of tabu status. For any $u, v \in V$, TABUSTATE(u, v) has a positive value if vertices u and v are forbidden from being swapped, and zero otherwise. The positive value in TABUSTATE(u, v) specifies the position in the tabu list where the attribute $\{u, v\}$ is recorded.

In our tabu search heuristic, there are two cases where the TABUSTATE needs to be updated. Let variable q denote the current position in tabu list where the last move was recorded. Whenever two vertices u and v are swapped, we reset the TABUSTATE cell corresponding to the pair at tabu list position $q + 1$, insert the pair $\{u, v\}$ into tabu list cell $q + 1$, and set TABUSTATE(u, v) to $q + 1$. When the length of tabu list will be reduced by one, the TABUSTATE cell corresponding to the pair at tabu list position $l - 1$ is reset.

All of our experiments use array TABUSTATE. To see if the array TABUSTATE gives us real help, we repeat the previous experiment on R512W for showing the effects of adaptive tabu list on solution costs reported in Table 5.2 without incorporating the array TABUSTATE. The performance of TSC without using array TABUSTATE is

t_b	t_s	time	$\bar{\mathcal{D}}(\pi)$
50	7	57.14	2.525
50	0	61.86	2.526
43	0	71.95	2.531
57	0	74.74	2.528

Table 5.4: Performance of TSC without using array TABUSTATE

reported in Table 5.4. We see that TSC with array TABUSTATE runs on the average 7.7% faster than that without this array for the same solution costs. It confirmed that this array is very helpful when the tabu list becomes long.

5.1.6 Accelerating the neighborhood search

In this subsection we demonstrate the effectiveness of the bucket-list data structure.

For the initial version of our TSC, we did not use the bucket-list data structure. We used an $n \times \log n$ two dimensional array to store all the possible gains. For each iteration, the time complexity of finding the maximum gain is $O(n \log n)$ and the time complexity of updating the affected gains is $O(\tilde{d} + \log n)$ (see Subsection 3.2.2). Therefore the major time is spent on finding the maximum gain. To reduce the search time for the maximum gain, we introduce the bucket-list priority queue to store the gains. This allows us to find the maximum gain in constant time, and spend $O(\tilde{d} + \log n)$ CPU time to update the positions of the cells in the bucket-list corresponding to those affected gains after a move.

Table 5.5 shows the time and solution cost comparisons between TSC with bucket-list and TSC without bucket-list for our random graph R512. We can see that TSC

TSC	$\mathcal{T}(\pi)$	$\mathcal{D}(\pi)$	time
with b.l.	6271.2	2.666	83.10
without b.l.	6111.2	2.739	326.56

b.l. - bucket-list

Table 5.5: Comparisons between TSC with and without bucket list

with bucket-list spends time 74.5% less than that for TSC without bucket list while improving the solution cost by 2.6%.

5.2 Performance comparisons

In this section, we compare the performances of our TSC and SAC with that of FG+KLC which is the most competitive heuristic claimed by Chen in [10]. The performance comparisons are divided into three categories. Based on the standard parameter settings, we show the performances of the heuristics under test with various limits on running time in Subsection 5.2.1. Subsection 5.2.2 reports the result of stability test with 1,000 runs on our standard random graph R512W for each of the competing heuristics. Subsection 5.2.3 presents the extensive performance comparisons for graphs with 128 vertices and various structures to show the consistency in performance of our heuristics for different graph types. Some observations of our experimental results are discussed in Section 5.3.

5.2.1 Comparisons with various time limits

For iterative heuristics, there is a trade-off between solution quality and running time. For most of them, the solution cost is a nonincreasing function of running time. The relative performance of different heuristics may change with different time limits. One approach for performance comparison is to compare the solution cost of

$(T_0\ T_r\ e\ c\ \lambda)$	$\bar{\mathcal{D}}(\pi)$	Time
(5 0.90 1000 0.005 8)	2.670	19.52
(5 0.95 1000 0.002 8)	2.609	35.18
(5 0.96 1000 0.001 8)	2.581	52.73
(5 0.97 1000 0.001 8)	2.568	60.96
(5 0.98 1000 0.001 8)	2.550	72.75
(5 0.99 1000 0.001 8)	2.533	93.25
(5 0.99 1000 0.0008 8)	2.525	111.84
(5.5 0.995 1500 0.0008 15)	2.495	223.39
(5.5 0.995 1500 0.0005 20)	2.479	390.32
(5.5 0.993 1500 0.0001 25)	2.458	756.61
(5.5 0.995 1500 0.0003 30)	2.441	1202.40

Table 5.6: Cost vs. time for SAC

different heuristics with the same running time limit. In this subsection, we show the performance of each heuristic with various time limits up to 1300 CPU seconds. For each time limit, we tune the parameters for each heuristic to optimize its performance. We focus our tests on our standard random graph R512W. Results for the other graphs are similar.

SAC

For SAC, the solution cost can be improved by adjusting the value of any one of the five parameters. For the experiments in this subsection, we allow more time than the standard parameter setting. Based on our standard setting, we tune the parameters to minimize the cost under different time limits. The resulting parameter settings and the corresponding costs and times are reported in Table 5.6. We can see from Table 5.6 that better solution cost can be obtained by increasing the length of each Markov chain (decreasing the value of c), or increasing the epoch length e, or increasing the cooling ratio T_r. The price of such adjustments is the increase in running time.

ρ	$\hat{\mathcal{D}}(\pi)$	Time
0	2.714	62.15
1	2.661	147.43
2	2.640	194.63
5	2.628	283.50
10	2.584	486.72
16	2.545	761.93
32	2.537	962.64
50	2.523	1333.73

Table 5.7: Cost vs. time for FG+KLC

FG+KLC

For FG+KLC, the parameter setting is simple. We can obtain better solutions by increasing the value of ρ, the number of consecutive uphill moves. Table 5.7 presents the cost and time as functions of ρ for FG+KLC. We can see that FG+KLC performs worse than SAC under various time limits. The running time grows dramatically when we increase ρ.

TSC

As explained in Subsection 5.1.3, the best parameter setting for TSC is $l_t = 50$ and $l_s = 7$. To increase the running time, we should only adjust the value of ρ, the number of consecutive uphill moves. The effects of various values for ρ are reported in Table 5.8. Compared with Table 5.6 and Table 5.7, the results for TSC are clearly the best.

Based on our above parameter tuning, we compare the solution qualities of FG+KLC, SAC, and TSC under various time limits in Figure 5.4. It is clear that TSC is the best for any time limit. SAC performs worse than TSC with small differences in solution costs. Both TSC and SAC outperform FG+KLC with large performance margins.

ρ	$\tilde{\mathcal{D}}(\pi)$	Time
500	2.607	16.44
1000	2.573	26.03
3000	2.524	66.94
5000	2.511	80.57
10000	2.492	129.33
15000	2.478	195.46
20000	2.471	245.04
40000	2.458	388.64
70000	2.444	700.46
100000	2.423	1211.00

Table 5.8: Cost vs. time for TSC

5.2.2 Stability evaluation

For most iterative improvement heuristics, we note that the solution quality can be improved by either increasing the running time or performing multiple runs. Since all of our competing heuristics use random initial solutions, we should not conclude the relative performance of the heuristics based on only limited number of try runs. A good heuristic should be stable in the sense that it can provide uniform good performance in huge number of runs.

In this experiment, we run each heuristic 1000 times on the same graph R512W using their standard parameter settings. We report the best value, worst value, average value, and standard deviation of cost and running time for each heuristic in Table 5.9, the comparisons of the histograms of cost in Figure 5.5, and the cost/time information for every run in Figure 5.6.

According to Table 5.9, the average results for the above three heuristics show that (1) TSC spends time 61.96% less than FG+KLC and achieves average cost 1.16% lower than that for FG+KLC; (2) TSC spends time 28.8% less than SAC and achieves average cost 1.62% lower than that for SAC; (3) SAC spends time 50.53% less than FG+KLC and achieves average cost 3.29% lower than that for FG+KLC.

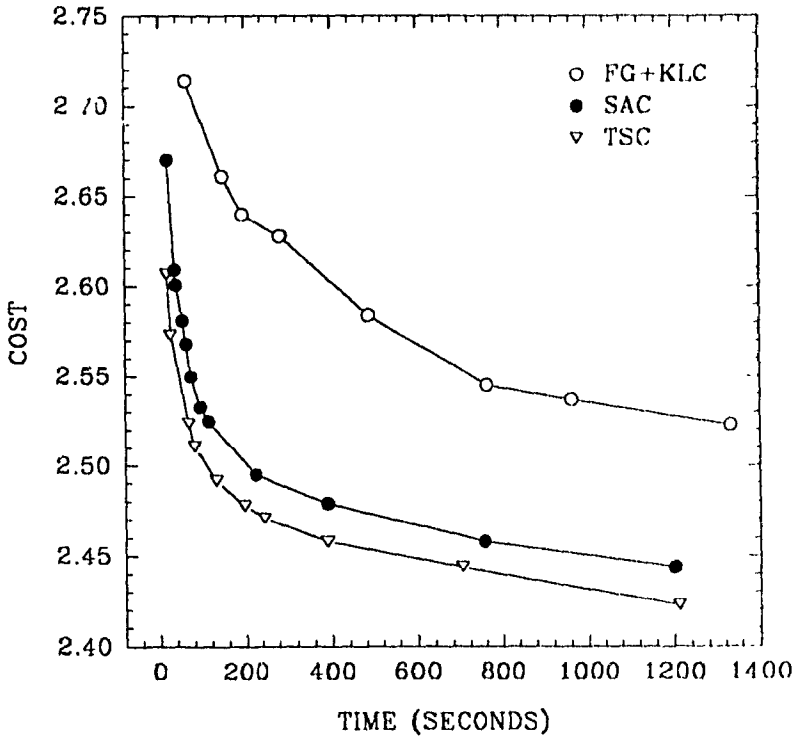


Figure 5.4: Comparison with various time limits for random graphs

The standard deviation values show that SAC is the most stable heuristic for both cost distribution and time distribution. Therefore multiple runs do not help much in solution quality for SAC. FG+KLC is the least stable heuristic.

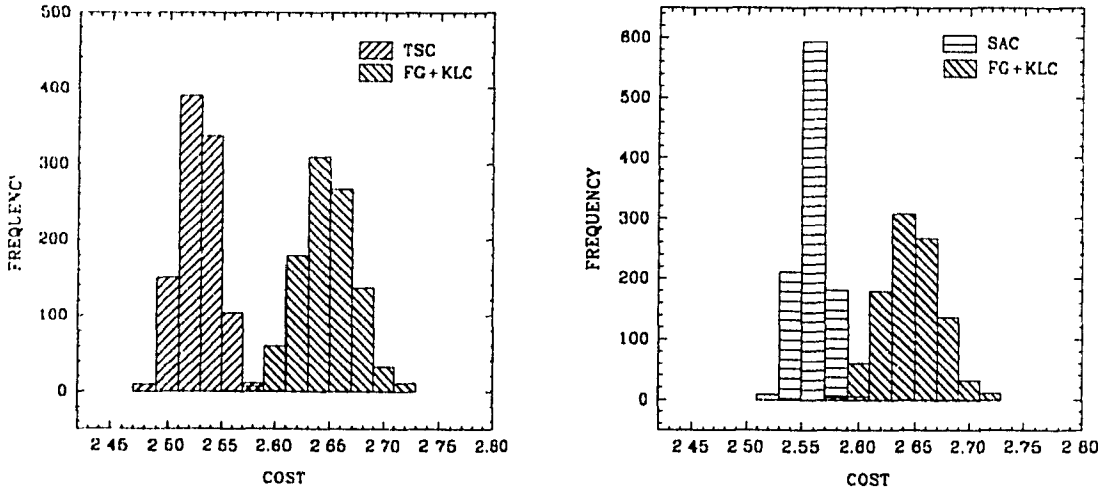
Figure 5.5 shows that the histograms of cost for both FG+KLC and either TSC or SAC can be displayed on the same axis with little overlap between the worst cost found by TSC (or SAC) and the best cost found by FG+KLC.

Figure 5.6 illustrates the solution cost distribution according to the time spent for each solution. From the cost-time graph we can see that the solutions obtained by TSC and SAC are concentrated in two small areas close to the lower left corner while the solutions obtained by FG+KLC are in a relatively larger area on or above the main diagonal. It implies that SAC and TSC are more stable than FG+KLC and have superior performances.

Table 5.10 shows our estimates for the expected best of k runs of FG+KLC, k runs

	TSC			SAC			FG+KLC		
	$\mathcal{T}(\pi)$	$\mathcal{D}(\pi)$	Time	$\mathcal{T}(\pi)$	$\mathcal{D}(\pi)$	Time	$\mathcal{T}(\pi)$	$\mathcal{D}(\pi)$	Time
Best	17596	2.474	118.21	17936	2.522	103.88	18277	2.570	317.70
Worst	18129	2.592	30.90	18473	2.598	67.22	19375	2.725	77.17
Average	17981	2.529	60.96	18202	2.560	85.59	18825	2.617	173.00
StdDev.	126.962	0.018	11.427	86.838	0.012	9.160	178.477	0.025	51.286

Table 5.9: Statistics for 1000 runs of TSC, SAC, and FG+KLC



(a) TSC and FG+KLC (b) SAC and FG+KLC

Figure 5.5: Histograms of solution costs for 1,000 runs on R512W using TSC, SAC, and FG+KLC

of SAC, and k runs of TSC for graph R512W and various values of k, based on the 1,000 runs of FG+KLC, SAC, and TSC (see Appendix A). It is interesting to note that the solution cost by one run of TSC is equal to the best cost of 100 runs of SAC; and the solution cost by one run of SAC is better than the best cost of 100 runs of FG+KLC. Therefore TSC clearly outperforms both SAC and FG+KLC.

5.2.3 Extensive comparisons

In the last two subsections we made intensive performance comparisons for our standard random graph R512W. For graphs with different structures, however, the heuris-

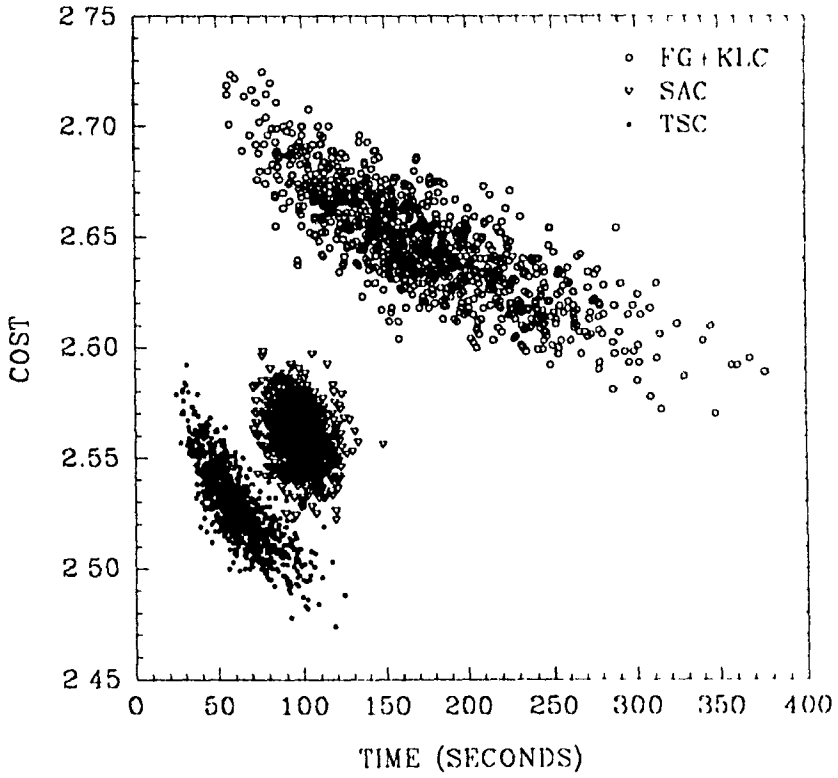


Figure 5.6: Cost-time graph for 1000 runs on R512W using TSC, SAC, and FG+KLC

tics normally have different performances. To see whether a heuristic performs consistently better than others, we conduct the following experiments to test our heuristics on different types of benchmark graphs.

In this subsection, we compare the performances for TSC, SAC, FG+KLC, SG and FG. TSC and SAC are our two most competitive heuristics proposed in this thesis. FG+KLC is the most competitive heuristic claimed by Chen in [10]. SG and FG are two very efficient greedy heuristics which will be used to generate better than random initial solutions.

To compromise solution quality and running time, we limit the running time of each heuristic to be within 70 CPU seconds. We will test our heuristics on random graphs, geometric graphs, and perturbed regular graphs. Each graph has 128 vertices. For random and geometric graphs, three different average vertex degree (3, 7, and 10 respectively) are used. For perturbed regular graphs, 3 or 7 edges are randomly deleted from or added to 7-cube and 8×16 mesh. For each set of 10 graphs, we run

k	TSC	SAC	FG+KLC
1	2.529	2.560	2.617
2	2.519	2.553	2.633
5	2.508	2.516	2.618
10	2.502	2.541	2.609
25	2.496	2.536	2.599
50	2.491	2.532	2.593
100	2.487	2.529	2.587

Table 5.10: Comparisons of TSC, SAC, and FG+KLC on R512W

10 times on each graph, and take the average cost and average running time over the 100 runs. In this way we see a general trend of performance for each heuristic. All results shown in this subsection are based on our standard parameter settings introduced in subsection 4.4.2 (on page 35) and in subsection 5.1.3 (on page 17).

We will see that TSC is the most powerful heuristic for most graphs. By using greedy initial solutions for geometric graphs and perturbed regular graphs, TSC can be further enhanced so that it can outperform all the other heuristics for all the graph types we tested. By using a greedy initial solution, TSC can significantly reduce its iteration number and improve its solution quality within specified time limits.

We also report the solution cost generated by random solution (RAND) to provide an intuitive feeling of how much improvement is achieved by applying the heuristics.

Random graphs

Shown in Table 5.11 are the performance comparisons for embedding random graphs. As the table shows, TSC beats all the other heuristics in both solution quality and running time by a wide margin. We do not use greedy heuristic as the front end of TSC because it does not help much in solution quality. SAC performs better than FG+KLC in both solution quality and running time for dense random graphs. By experiments we found that the performance of FG+KLC is sensitive to the average vertex degree $\bar{d}$. FG+KLC does not perform well when $\bar{d}$ is getting large. From

Heuristics	R128_3		R128W_3	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
TSC	1.356	21.96	1.311	31.68
SAC	1.432	30.23	1.355	31.76
FG+KLC	1.431	30.62	1.351	36.71
FG	1.955	0.01	1.975	0.01
SG	2.272	0.01	2.264	0.01
RAND	3.528	0.00	3.526	0.00

Heuristics	R128_7		R128W_7	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
TSC	1.987	32.16	1.872	13.33
SAC	2.046	31.30	1.922	38.10
FG+KLC	2.061	51.15	1.951	13.36
FG	2.651	0.08	2.656	0.08
SG	2.851	0.02	2.848	0.02
RAND	3.521	0.01	3.523	0.01

Heuristics	R128_10		R128W_10	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
TSC	2.255	16.08	2.138	57.38
SAC	2.290	17.76	2.161	57.17
FG+KLC	2.313	62.73	2.198	62.15
FG	2.815	0.11	2.813	0.11
SG	3.010	0.02	3.009	0.02
RAND	3.526	0.01	3.528	0.01

Table 5.11: Comparisons for embedding random graphs to minimize $\mathcal{D}(\pi)$

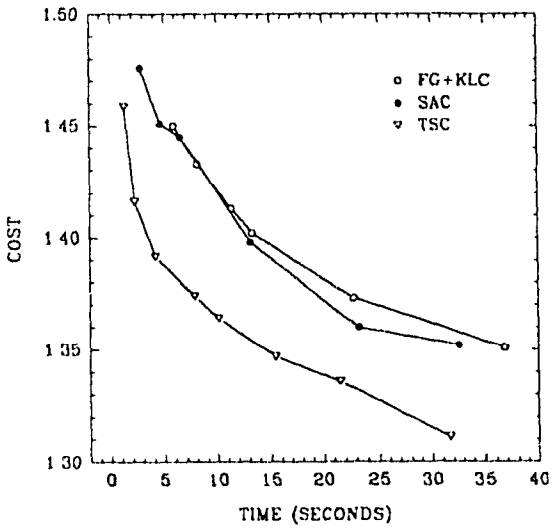
Table 5.11 we can also see that the relative performances of our heuristics are not sensible to the addition of edge weights to the graphs.

Figure 5.7 gives a closer look at TSC, SAC, and FG+KLC with different parameter settings. We use the same method introduced previously to determine the parameter settings. Each curve in the figure represents the trade-offs between running time and solution quality made by a particular heuristic using different parameter settings. From the figure we see that TSC obtains better solutions than all the other heuristics in the same or less running time. When $\dot{d} = 3$ as in Figure 5.7 (a), SAC has basically the same performance as FG+KLC. When $\dot{d}$ is getting large, however, SAC apparently outperforms FG+KLC with big margin as shown in Figure 5.7 (b) and Figure 5.7 (c). We did not give the curves for graphs without edge weights because the corresponding performance comparisons are very similar. From Figure 5.7 we can see that for all the three heuristics, both the solution cost and the running time increase with the average vertex degree $\dot{d}$.

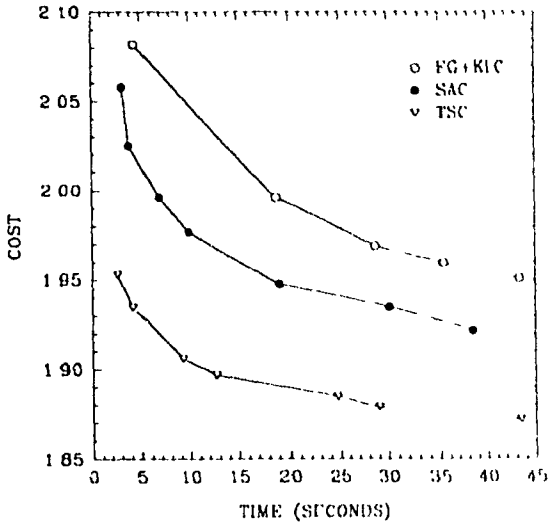
Geometric graphs

The performance comparisons for our geometric graphs are shown in Table 5.12. The general trends are similar to those of random graphs but there are some differences. One of the major differences is that the greedy heuristic SG improves the performance of TSC significantly. TSC with SG front end beats all the other heuristics. FG+KLC performs better than it does for random graphs. As for random graphs, the relative performance of our heuristics are not sensible to the addition of edge weights to the geometric graphs.

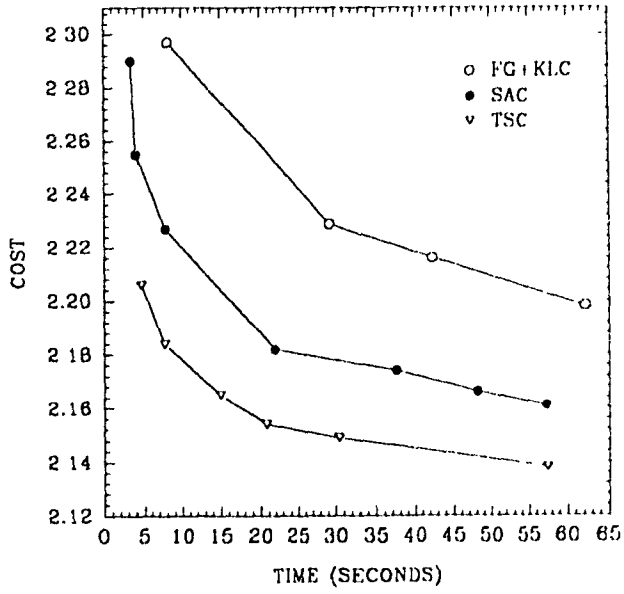
Figure 5.8 gives a closer look at the heuristics that are competitive in solution quality and running time. As shown in Figure 5.8 (a) for graphs with $\dot{d} = 3$, SG+TSC, TSC, and FG+KLC have basically the same performance and SAC has the worst performance. As shown in Figure 5.8 (b) and Figure 5.8 (c) for graphs with $\dot{d} = 7$ and 10, however, SAC starts to stand on the second place and SG+TSC beats all the other heuristics with big margin. Like for random graphs, both the solution cost and



(a) R128W_3



(b) R128W_7



(c) R128W_10

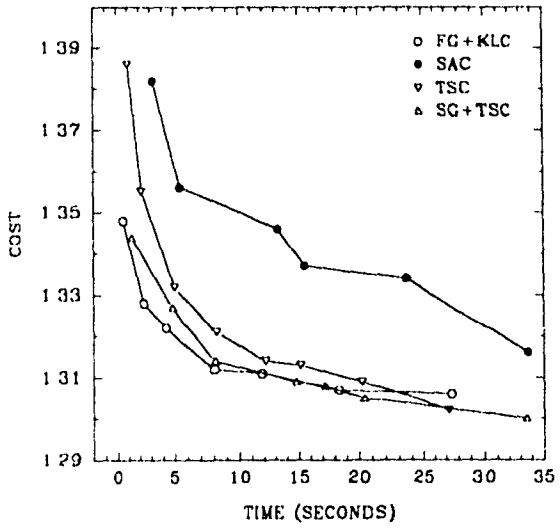
Figure 5.7: Cost/time trade-offs for random graphs

Heuristics	G128_3		G128W_3	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
SG+TSC	1.358	16.49	1.300	33.50
TSC	1.364	15.85	1.302	27.20
FG+KLC	1.365	16.39	1.307	27.81
SAC	1.402	25.79	1.316	33.64
FG	1.471	0.05	1.459	0.05
SG	1.575	0.01	1.600	0.01
RAND	3.530	0.01	3.529	0.01

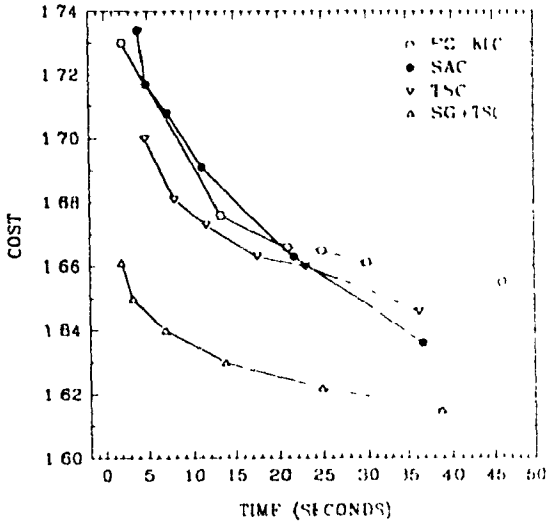
Heuristics	G128_7		G128W_7	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
SG+TSC	1.653	26.69	1.615	37.76
TSC	1.690	34.61	1.646	36.40
SAC	1.688	39.89	1.636	36.78
FG+KLC	1.688	36.38	1.655	45.98
FG	1.895	0.08	1.924	0.08
SG	1.991	0.01	2.065	0.01
RAND	3.528	0.01	3.527	0.01

Heuristics	G128_10		G128W_10	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
SG+TSC	1.820	36.91	1.797	34.92
SAC	1.840	38.10	1.816	39.83
TSC	1.862	36.17	1.836	42.20
FG+KLC	1.864	39.18	1.832	45.50
FG	2.080	0.11	2.123	0.11
SG	2.188	0.02	2.279	0.02
RAND	3.530	0.01	3.531	0.01

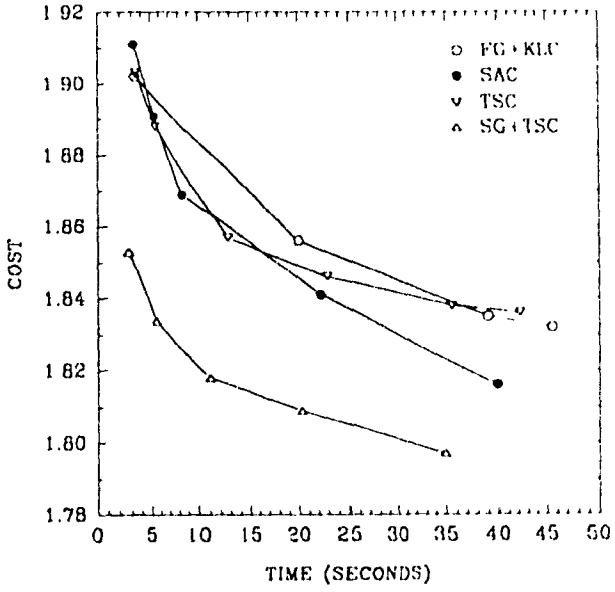
Table 5.12: Comparisons for embedding geometric graphs to minimize $\mathcal{D}(\pi)$



(a) G128W_3



(b) G128W_7



(c) G128W_10

Figure 5.8: Cost/time trade-offs for geometric graphs

the running time increase with the average vertex degree $\bar{d}$.

Perturbed regular graphs

Table 5.13 and Table 5.14 present the performance comparisons among the heuristics for perturbed 7-cubes and perturbed 8×16 meshes.

We can see that SG+TSC gives the best solutions than all the others for perturbed regular graphs. FG and SG can embed cubes and meshes without edges deleted or added optimally. FG and SG+TSC can embed cubes with certain edges deleted optimally. Since it is hard for most heuristics to embed graphs C7_P7 and M8_16_P7 optimally, we choose these two graphs as examples to show the relative performance of the heuristics in Figure 5.9.

For graph C7_P7, FG+KLC can acquire the same performance as SG+TSC after 10 seconds. We conjecture that both FG+KLC and SG+TSC have embedded graph C7_P7 optimally. SAC and TSC seem worse than FG+KLC and SG+TSC. For graph M8_16_P7, it is apparent that SG+TSC beats all the other heuristics. TSC, FG+KLC, and SAC are on the second, third, and fourth places respectively.

5.3 Summary

This Chapter described the design of our tabu search heuristic and made the various experiments. The major characteristics of our tabu search heuristic based on objective function $D(\pi)$ for hypercube embedding were discussed. Cube-neighbor neighborship was shown to be a better compromise between the solution search and the running time of each iteration. We implemented an adaptive tabu list scheme to reduce the iteration number and improve the solution quality. We used a tabu status array to accelerate the checking of whether a move is in tabu list. Experiments showed that aspiration function is not helpful for this particular hypercube embedding model.

By performing the various experiments, we conclude that SG+TSC is the most

Heuristics	C7	
	$\mathcal{D}(\pi)$	time
FG	1.000	0.015
SG	1.000	0.015
SG+TSC	1.000	0.063
FG+KLC	1.000	0.110
TSC	1.029	5.110
SAC	1.085	9.020
RAND	3.517	0.002

Heuristics	C7_M3		C7_M7	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
FG	1.000	0.06	1.000	0.07
SG+TSC	1.000	0.31	1.000	0.35
FG+KLC	1.000	0.38	1.000	0.39
TSC	1.021	5.26	1.055	1.70
SAC	1.014	7.52	1.021	8.22
SG	1.329	0.02	1.653	0.02
RAND	3.515	0.01	3.517	0.01

Heuristics	C7_P3		C7_P7	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
SG+TSC	1.066	1.38	1.010	6.25
TSC	1.081	5.36	1.067	8.96
FG+KLC	1.091	6.68	1.010	10.52
SAC	1.066	8.14	1.071	13.15
FG	1.110	0.06	1.285	0.06
SG	1.697	0.01	2.066	0.01
RAND	3.515	0.01	3.515	0.01

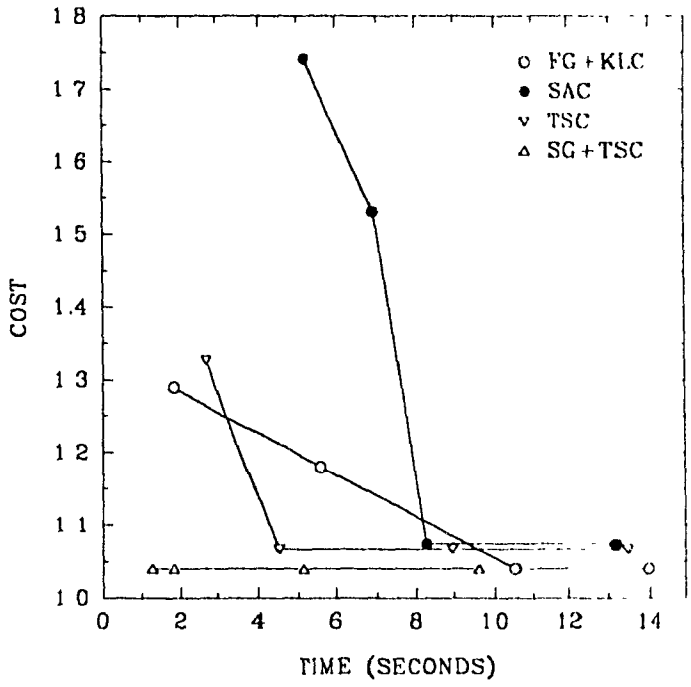
Table 5.13: Comparisons for embedding perturbed cubes to minimize $D(\pi)$

Heuristics	M8_16	
	$\mathcal{D}(\pi)$	time
SG	1.000	0.001
FG	1.000	0.033
SG+TSC	1.000	0.033
FG+KLC	1.000	0.183
TSC	1.310	9.417
SAC	1.401	25.720
RAND	3.173	0.007

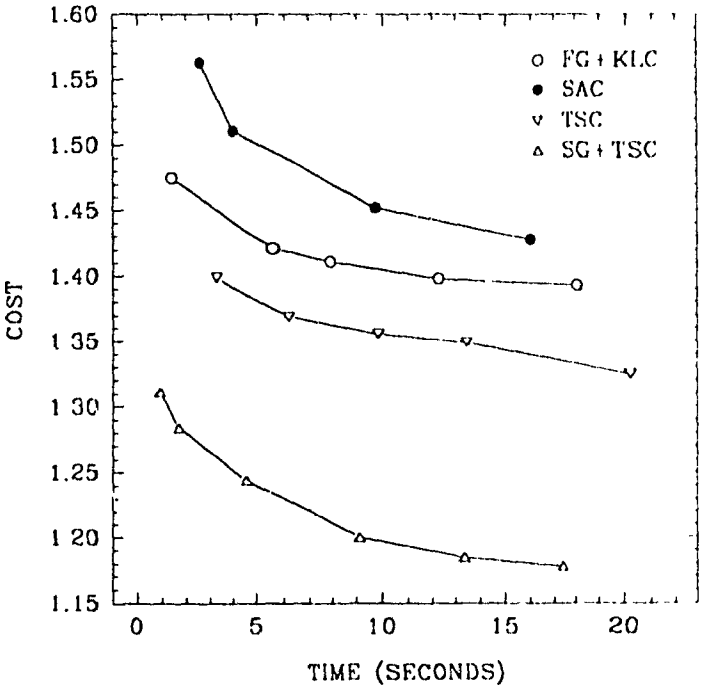
Heuristics	M8_16_M3		M8_16_M7	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
SG+TSC	1.052	6.21	1.120	8.80
FG+KLC	1.148	6.27	1.323	10.10
TSC	1.311	9.88	1.287	12.48
SAC	1.398	21.46	1.106	22.41
FG	1.290	0.03	1.567	0.03
SG	1.307	0.01	1.687	0.01
RAND	3.416	0.01	3.416	0.01

Heuristics	M8_16_P3		M8_16_P7	
	$\mathcal{D}(\pi)$	time	$\mathcal{D}(\pi)$	time
SG+TSC	1.106	7.47	1.200	9.97
TSC	1.309	9.28	1.356	9.82
FG+KLC	1.309	9.86	1.398	12.29
SAC	1.426	18.81	1.428	15.97
FG	1.478	0.04	1.663	0.04
SG	1.618	0.01	1.907	0.01
RAND	3.416	0.01	3.417	0.01

Table 5.11: Comparisons for embedding perturbed meshes to minimize $\mathcal{D}(\pi)$



(a) C7_P7



(b) M8_16_P7

Figure 5.9: Cost/time trade-offs for perturbed regular graphs

successful and efficient heuristic for all types of graphs. TSC is the best choice for all random graphs. When running time is not strictly limited, SAC is the second good choice for large random graphs. When the problem size is relatively small, FG+KLC is also a good choice for sparse geometric graphs. For regular graphs, SG and FG are good choices. SG and FG are also useful for situations where better than random solution is desired.

For task graphs whose structure is close to cube, we can get better embeddings than for random graphs. When the average vertex degree $\tilde{d}$ is getting large, the solution cost is getting worse and more running time is acquired.

As a final observation, the parameter tuning for TSC is much easier than that for SAC since TSC has less parameters than SAC.

Chapter 6

Tabu Search Heuristic with Objective Function $\mathcal{D}(\pi)$

For synchronous SIMD parallel machines, the objective function $D(\pi)$ is more appropriate to be used to minimize the maximal mapped distance of embeddings. Graph embedding with objective function $\mathcal{D}(\pi)$ is much harder than that with objective function $\bar{\mathcal{D}}(\pi)$ because of the former's special difficulties for cost evaluation and gain update. Based on our literature survey on hypercube embedding, no paper has seriously worked on this model. In this chapter, we propose another tabu search heuristic to solve the hypercube embedding problem with objective function $\mathcal{D}(\pi)$ and refer to this new heuristic as TSD.

We first discuss some special properties of the objective function $D(\pi)$ and design a special gain function for $\mathcal{D}(\pi)$. To facilitate the evaluation of the cost function, an efficient heap data structure is introduced. After we present the heuristic TSD, the performance comparisons between TSD and TSC is conducted to demonstrate TSD's special role in minimizing $\mathcal{D}(\pi)$.

6.1 Gain function and its update

For heuristics with objective function $\mathcal{D}(\pi)$, it is straightforward to define the *gain* of swapping vertices u and v relative to the current solution as the decrease in total cost after the swapping. For heuristics with objective function $\mathcal{D}(\pi)$, however, the same definition would not work because changing the mapped distance for some $e \in E$ does not imply changing the global embedding cost. The straightforward definition of gain function for $\mathcal{D}(\pi)$ does not encourage local improvements which is critical to the reduction of the embedding cost.

A good definition for the gain function should encourage the gradual reduction in the number of edges with maximal mapped distance. In our heuristic, we use the following special definition for the gain function to encourage any swap of vertices u and v that can reduce the maximal mapped distance of the edges $e \in E$ adjacent to u or v. Given the current embedding π and any swap of vertices u and v, we define the gain of the swap $\{u, v\}$ as

$$gain(u,v) = \max_{x \in \beta(u), y \in \beta(v)} \{\delta(\pi(u), \pi(x)) \cdot w(u,x),\ \delta(\pi(v), \pi(y)) \cdot w(v,y)\} -$$

$$\max_{x \in \beta(u), y \in \beta(v)} \{\delta(\pi(v), \pi(x)) \cdot w(u,x),\ \delta(\pi(u), \pi(y)) \cdot w(v,y)\}$$

where $\pi(v)$ may be $\pi(u)$'s cube-neighbor. The time complexity for evaluating function $gain(u,v)$ is thus $O(\hat{d})$.

To minimize the computational overhead introduced by reevaluating the *gain* function, we also use a bucket-list priority queue data structure to store all possible gains. Whenever a swap is done, only affected *gains* are updated.

For all swap neighborhood, the time complexity for gain initialization is $O(n \cdot n \cdot \hat{d}) = O(mn)$; and the time complexity for gain update after a swap is $O(\hat{d} \cdot n\hat{d}) = O(\hat{d}m)$.

For cube-neighbor neighborhood, the gains can be initialized for each swap in time $O(n \log n \cdot \hat{d}) = O(m \log n)$. Whenever vertices u and v are swapped where $\pi(v)$ is cube neighbor of $\pi(u)$, the affected gains will be recomputed for all the pairs $\{x, y\}$ where $x \in \beta(u) \cup \beta(v) \cup \{u, v\}$ and $\pi(y)$ is cube-neighbor of $\pi(x)$. The time spent on

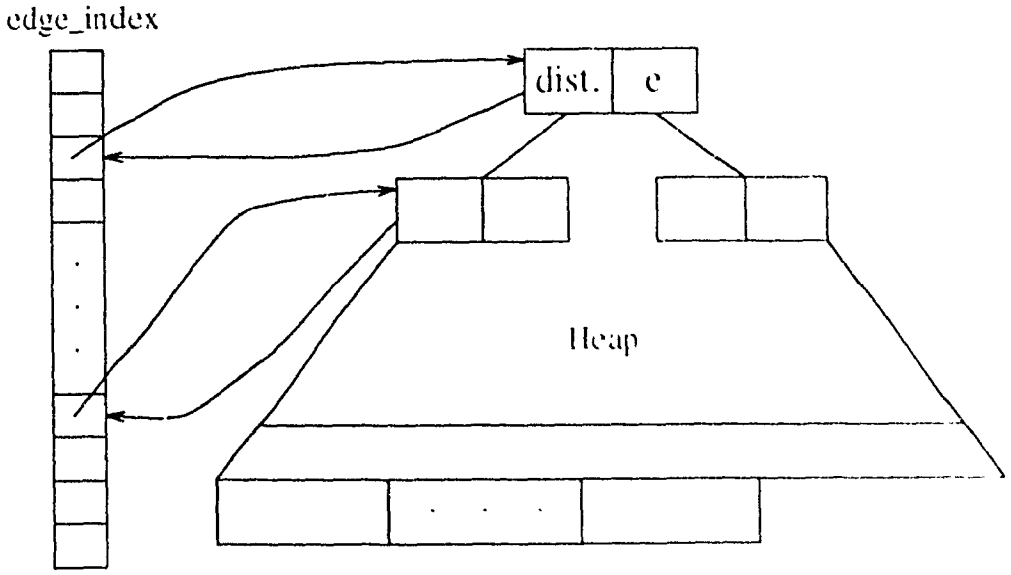


Figure 6.1: Heap structure used to report the maximal mapped distance

the updating of gain after a swap is thus $O(\hat{d}(deg(a) + deg(c)) \log n)$ $O(\hat{d} \log n)$

6.2 Cost function evaluation

For objective function $D(\pi)$, one important issue is to reduce the time for finding the maximal mapped distance after each move. If we use an array to store the mapped distances for all edges in E, we need time $O(|E|)$ to find the maximal mapped distance. To make our heuristic efficient, we use a priority queue Q to store the mapped distances so that the maximal mapped distance can be found in constant time.

Our priority queue is implemented by a heap, as shown in Figure 6.1. For each edge $e \in E$ we store a pair $(dist, e)$ in the heap where $dist$ is the key specifying the mapped distance of edge e. A linear array $edge\_index$ indexed by edges is used. For each $e \in E$, $edge\_index[e]$ is a pointer to the corresponding node in the heap specifying the mapped distance of edge e. Following three operations are supported on the heap:

1. Initialize the tabu list $T = \phi$.
2. Get an initial solution π.
3. Initialize all possible gains according to π.
4. **Repeat** ITER_MAX times:
   - 4.1 Find a move $s \in S(\pi) - T$ that maximizes $gain(s)$.
   - 4.2 Set $\pi = s(\pi)$ and insert s into T.
   - 4.3 Update gains.
5. Return the best π visited.

Figure 6.2: Heuristic TSD

$Max(Q_h)$	return the maximal mapped distance c in Q_h;
$Insert(Q_h, dist, c)$	insert a node $(dist, c)$ into Q_h;
$Delete(Q_h, index)$	delete a heap node pointed to by $index$ from Q_h.

The algorithms for these three procedures are given in Appendix B. $Max()$ has constant time complexity. The time complexities for $Insert()$ and $Delete()$ are both $O(\log |E|) = O(\log m)$.

This heap can be initialized in time $O(m \log m)$. Whenever two vertices u and v are swapped, only the mapped distances for edges incident to u or v in G are changed. Therefore the update of heap after a swap can be done in time $O(d \log m)$.

6.3 Tabu search heuristic

We implemented two versions of tabu search heuristic with objective function $D(\pi)$, one for all swap neighborhood, the other for cube-neighbor neighborhood. The experimental results show that TSD with cube-neighbor neighborhood outperforms TSD with all swap neighborhood. Therefore we only discuss TSD with cube-neighbor neighborhood. Like TSC, TSD also uses our adaptive tabu list design and tabu status array implementation. Figure 6.2 shows the heuristic TSD. We can see from Figure 6.2 that TSD is basically the same as TSC except that the objective function,

the *gain* function, and the termination condition are different. TSD first initializes all possible gains and put all moves associated with these gains into the bucket list. TSD also initializes the heap to store all mapped distances of edges in G. $IFER \times MAX$ iterations of the loop at step 4 are then executed. TSD transforms the current solution to one of its neighbors by performing the best move which is not in the tabu list. Whenever a move s is performed, s is inserted into the tabu list and the affected gains are updated as explained in Section 6.1. Different from TSC, TSD also spends $O(\dot{d}\log m)$ CPU time to update the affected mapped distances stored in the *heap* for some edges.

The diameter of a d-cube is d, the maximum gain (M) of any single swap for objective function $\mathcal{D}(\pi)$ is thus in range $\pm(d-1)\cdot 2(\log n - 1) = O(\log n)$. Then we have $\Delta_{max} = O(\log n)$. Let I be the total number of iterations. The number of calls to $Max()$ is $Q_M = I$, Q_I is dominated by $O(Id^2 \log n)$. According to Lemma 1, the overall running time of TSD is therefore the summation of gain initialization and update for both bucket-list and heap operations, that is, $O(m\log n)$ for gain initialization, $O(m\log m)$ for heap initialization, $O(Id\log m)$ for heap update, and $O(Id^2\log n \cdot (I\log n + \log n)$ for gain update. The total time complexity is thus $O(m\log m + Id(\dot{d}\log n + \log m))$.

6.4 Performance comparisons

As for TSC in Section 5.2, the random graphs, the geometric graphs and the perturbed regular graphs specified in Section 3.1 are used as benchmark graphs. The comparisons are made (1) between heuristics TSD and TSC for random graphs and geometric graphs; (2) among heuristics FG+TSD, TSD, and TSC for perturbed regular graphs. The performance comparisons reported in this section are obtained by making the trade-offs between solution quality and execution time. For each set of 10 graphs, we run 10 times on each graph, and take the average cost and average running time over the 100 runs.

Graphs	TSD		TSC	
	$\mathcal{D}(\pi)$	Time	$\mathcal{D}(\pi)$	Time
R128W_3	10.00	8.21	11.96	8.80
R128W_7	11.70	11.95	19.92	11.90
R128W_10	16.30	24.73	21.40	23.31
R512W	20.00	66.68	28.80	66.91
R128_3	3.00	6.33	3.83	6.35
R128_7	4.00	16.61	5.16	16.61
R128_10	5.00	10.51	6.12	17.02
R512	6.00	42.23	7.10	71.81

Table 6.1: Comparisons for embedding random graphs to minimize $\mathcal{D}(\pi)$

As described in Subsection 4.4.2 and Subsection 5.4.3, for each of our benchmark graphs, we tune the parameters one at a time, and repeat the process until no perturbation for the parameters can improve the performance. The following reports are obtained based on the best parameter settings.

6.4.1 Random graphs

Table 6.1 shows the performance comparisons between TSD and TSC on random graphs. For all random graphs with edge weights, TSD on the average can improve the cost $\mathcal{D}(\pi)$ of TSC by 25.9% in 86% of the running time of TSC. For all random graphs without edge weights, TSD on the average can improve the cost $\mathcal{D}(\pi)$ of TSC by 21.4% in 76% of the running time of TSC.

6.4.2 Geometric graphs

Table 6.2 shows the performance comparisons on geometric graphs. For geometric graphs, the performance od TSD depends on the attributes of the graphs. TSD still outperforms TSC for geometric graphs with edge weights and various average vertex degrees. For geometric graphs without edge weights, TSD gives low maximal

Graphs	TSD		TSC	
	$D(\pi)$	Time	$D(\pi)$	Time
G128W_3	10.10	13.15	12.38	21.12
G128W_7	15.20	21.11	17.38	25.82
G128W_10	18.60	31.59	19.98	33.29
G512W	24.00	52.70	27.00	54.51
G128_3	3.30	13.03	3.10	13.21
G128_7	1.00	28.75	1.00	36.36
G128_10	1.10	31.27	1.16	36.17
G512	6.00	14.89	6.20	17.53

Table 6.2: Comparisons for embedding geometric graphs to minimize $D(\pi)$

mapped distances for graphs with either $d = 10$ or $n = 512$. For graphs G128_3 however, TSD performs worse than TSC.

To be more specific, for geometric graphs with edge weight, TSD on the average can improve the cost $D(\pi)$ of TSC by 14.5% in 85.7% of the running time of TSC. For all geometric graphs without edge weights, TSD on the average can improve the cost $D(\pi)$ of TSC by 1.8% in 86% of the running time of TSC.

6.4.3 Perturbed regular graphs

Table 6.3 presents the performance comparisons for perturbed regular graphs. We can see from Table 6.3 that TSD performs worse than TSC for 7-cube and 7-cube with 3 or 7 edges randomly deleted. By experimental results reported in Subsection 5.2.3 we know that greedy heuristics SG and FG can embed a d-cube and d-cube with certain edges deleted to a d-cube optimally. Therefore, by using FG as the front end of TSD, we can significantly improve the solution quality. For perturbed cube FG+TSD outperforms both TSD and TSC in both solution quality and running time. For perturbed meshes, FG+TSD also outperforms all the other heuristics in both solution quality and running time, and TSD performs better than TSC.

Graphs	FG+TSD		TSD		TSC	
	$\mathcal{D}(\pi)$	Time	$\mathcal{D}(\pi)$	Time	$\mathcal{D}(\pi)$	Time
C7	1.0	0.240	4.0	8.896	1.1	5.410
C7_M3	1.0	0.267	1.0	8.795	1.2	5.260
C7_M7	1.0	0.233	4.0	8.689	1.6	1.699
C7_P3	3.0	1.095	1.0	8.913	4.2	5.365
C7_P7	3.1	1.148	1.0	9.011	5.2	8.959
M8_16	1.0	0.133	3.0	9.888	3.2	9.117
M8_16_M3	2.5	1.622	3.0	9.612	3.5	9.876
M8_16_M7	3.0	1.927	3.0	9.475	3.1	12.180
M8_16_P3	3.0	2.081	3.1	11.051	3.7	9.281
M8_16_P7	3.0	2.781	3.2	11.381	1.1	7.757

Table 6.3: Comparisons for embedding perturbed regular graphs to minimize $\mathcal{D}(\pi)$

To be more specific, for all perturbed cubes, FG+TSD on the average can improve the cost $\mathcal{D}(\pi)$ of TSC by 27.8% in 29.1% of the running time of TSC. For all perturbed meshes, FG+TSD on the average can improve the cost $\mathcal{D}(\pi)$ of TSC by 31% in 22% of the running time of TSC.

6.5 Summary

In this chapter, we described the design of another tabu search heuristic to minimize objective function $\mathcal{D}(\pi)$. We explained the difficulty in decreasing $\mathcal{D}(\pi)$ and defined the special gain function to measure the incremental improvement during the search. We also introduced another efficient priority queue implemented by a heap data structure. For SIMD hypercube machines, the experiments showed that the FG+TSD cannot be replaced by TSC because FG+TSD performs better for minimizing the maximal mapped distance than TSC.

Chapter 7

Conclusion

In this thesis we proposed three new efficient heuristics TSC, SAC, and TSD. They are used to solve the hypercube embedding problem with objective functions either minimizing average mapped distance or minimizing the maximal mapped distance. The most successful heuristic for minimizing the average mapped distance is TSC. The efficient heuristic for minimizing the maximal mapped distance is FG+TSD.

In the following we make some observations based on our experimental results, summarize our major contributions to this research, and point out several future research directions.

7.1 Observations on experimental results

A variety of conclusions can be drawn from the results which we reported. Each of the following items highlights one of our major observations.

1. For most iterative improvement heuristics, their is a trade off between solution quality and running time. We can usually adjust this trade off by varying the parameter settings for each heuristic.

2. Iterative improvement heuristics perform better when the starting solution is better than random (SG and FG are good methods for generating starting

solutions). Our observation on this point is true especially for geometric and regular graphs when the expected running time is limited. When the running time is long enough, however, this feature cannot be seen explicitly.

3. The structure of the task graph must be taken into account when choosing a heuristic. For completely random graphs, the best choices tend to be iterative improvement heuristics. For graphs that are cubes or almost cubes, FG and SG are the best choices. FG and SG can, for example, efficiently map a cube or mesh to hypercube with the same number of vertices optimally.

4. Experimental results show that simulated annealing heuristic with adaptive Markov chains outperforms that with fixed-length Markov chains.

5. For MIMD hypercube machines, SG+TSC is the most effective and efficient heuristic for all types of graphs. TSC is the best choice for all random graphs. When running time is not strictly limited, SAC has a performance only second to TSC for large random graphs. When the problem size is relatively small, FG+KLC is also a good choice for sparse geometric graphs. For regular graphs, SG and FG are good choices. SG and FG are also useful for situations where better than random solution is desired.

6. For SIMD hypercube machines, the experiments showed that the FG+TSD cannot be replaced by TSC because FG+TSD performs better for minimizing the maximal mapped distance than TSC does.

7.2 Major contributions

The followings are our major contributions to this research:

1. Generalization of Chen's mapping model. First, we introduce the edge weight function to the objective function to model the potentially different communication loads between different pairs of processes. Second, in addition to the

objective function $\mathcal{D}(\pi)$ which minimizes the average mapped distance, we also use the maximal mapped distance $D(\pi)$ as our objective function. Therefore, we design embedding heuristics for program mapping on both synchronous and asynchronous parallel systems.

2. Introduction of adaptive tabu list to tabu search heuristics for hypercube embedding. For both of our two tabu search heuristics, the length of tabu list can be updated according to the recent improvement history of the best solution found so far. The purpose of designing an adaptive tabu list is to avoid introducing cycling in the solution space and to help the search to get out of the local optimum quickly. The experimental results showed that the tabu search heuristic with adaptive tabu list outperforms that with static tabu list. As shown in Table 5.2, for example, tabu search heuristic with adaptive tabu list improves on the solution quality of that with static tabu list in 9% of CPU time of the latter for graph R512W.

3. Design and implementation of two tabu search heuristics with objective functions $D(\pi)$ and $\mathcal{D}(\pi)$ respectively. For the heuristic to minimize $D(\pi)$, we designed a special *gain* function. Our special combination of the cube neighborhood, the adaptive tabu list, the application of tabu status array, and our efficient data structures make our tabu search heuristics very efficient. Compared with Chen's most competitive heuristic FG+KLC for objective function $\mathcal{D}(\pi)$, for example, TSC spends only 35% of time which FG+KLC takes to reduce the average cost of FG+KLC for random graph R512W by 5%.

4. The improvement on Chen's simulated annealing heuristic. To optimize the simulated annealing heuristic, we introduced the adaptive Markov chains search strategy. The experimental results show that our new annealing heuristic outperforms Chen's in both solution quality and running time. For all of our random graphs, for example, our new annealing heuristic spends on the average 75% of time which Chen's annealing heuristic takes to reduce the average cost of Chen's by 3%.

7.3 Future work

For tabu search heuristics implemented in this thesis, further research can be conducted in the following directions.

1. Incorporation of intermediate and long term memory functions. The intermediate memory is used to record the recent *move* history to intensify the search to reach the local optimum faster. The long term memory is used to record the *move* history since the start of the search to diversify the search to new areas and try to avoid being trapped in local optima.

2. Determination of the values of the parameters t_b and t_s by the statistics of the moves. For different periods of the search process, the best range for the tabu list length may be different. Therefore the value of parameters t_b and t_s may need to be changed during the search process. To reduce the effort on parameter tuning and to improve the solution quality, t_b and t_s may be periodically adjusted according to the statistics of the moves.

3. Parallelization of our sequential heuristics. Our experimental studies are based on sequential machines. To speed up the solution search and make our heuristics more efficient, we can parallellize the hypercube embedding heuristics. Initial investigations have been conducted.

Bibliography

[1] E.H.L. Aarts, F.M.J. de Bont, J.H.A. Habers, and P.J.M. van Laarhoven. "A parallel statistical cooling algorithm". In *Lecture notes in computer science*, '10, pages 87–97, 1986.

[2] F. André, J.L. Pazat, and T. Priol. "Experiments with mapping algorithms on a hypercube". In *Proc. of the forth conference on hypercubes, concurrent computers, and applications*, pages 39–46, 1989.

[3] Francine Berman and Lawrence Snyder. "On Mapping Parallel Algorithms into Parallel Architectures". *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

[4] Shahid H. Bokhari. "On the Mapping Problem". *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[5] S. Wayne Bollinger and Scott F. Midkiff. "Processor and Link Assignment in Multicomputers Using Simulated Annealing". In *Proc. of the 1988 International Conference on Parallel Processing*, volume 1, pages 1–7, 1988.

[6] J. Bovet, C. Constantin, and D. de Werra. "A Convoy Scheduling Problem" Technical Report ORWP 87/23, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, 1988.

[7] A. Casotto, F. Romeo, and A.L. Sangiovanni-Vincentelli. "A parallel simulated annealing algorithm for the Placement of Macro-Cells". In *Proc. of international conference on computer design*, pages 30–33, 1986.

[8] Mee Yee Chan. "Dilation-2 Embeddings of Grids into Hypercubes". In *Proc. of International Conference on parallel processing*, pages 295–298, 1988.

[9] W.-K. Chen and E.F. Gehringer. "A graph-oriented mapping strategy for a hypercube". In *Proc. of the third conference on hypercube concurrent computers and applications*, pages 200–209, 1988.

[10] Wocl Kae Chen. *Theoretical and experimental approaches for the hypercube embedding problem*. PhD thesis, North Carolina State University, 1991.

[11] Wocl-Kae Chen, Matthias F.M. Stallmann, and Edward F. Gehringer. "Hypercube Embedding Heuristics: An Evaluation". *International Journal of Parallel Programming*, 18(6):505–549, 1989.

[12] Georgr Cybenko, David W. Krumme, and K.N. Venkataraman. "Fixed Hypercube Embedding". *Information Processing Letters*, 25:35–39, 1987.

[13] F. Ercal, J. Ramanujam, and P. Sadayappan. "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning". *ACM*, pages 210–221, 1988.

[14] E. Felten, S. Karlin, and S.W. Otto. "The Traveling Salesman Problem on a hypercubic, MIMD Computer". In *Proc. of 1985 international conference on parallel processing*, pages 6–10, 1985.

[15] C.M. Fiduccia and R.M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In *IEEE 19th Design Automation Conference*, pages 175–181, 1982.

[16] C. Friden, A. Hertz, and D. de Werra. "Stabulus: A technique for finding stable sets in large graphs with Tabu search". *Computing*, 42:35–44, 1989.

[17] C. Friden, A. Hertz, and D. de Werra. "TABARIS: An exact algorithm based on tabu search for finding a maximum independent set in a graph". *Computers and Operations Research*, 17(5):437–445, 1990.

[18] Michel Gendreau, Louis Salvail, and Patrick Soriano. "Solving the Maximum Clique Problem Using a Tabu Search Approach". Technical Report Technical Report CRT-675, Publication #675, Centre de recherche sur les transports, 1990.

[19] Fred Glover. "Future paths for integer programming and links to artificial intelligence". *Computers and Operations Research*, 1(3):533–549, 1986.

[20] Fred Glover. "Tabu Search – Part I". *ORSA Journal on Computing*, 1:190–206, 1989.

[21] Fred Glover. "Tabu Search – Part II". *ORSA Journal on Computing*, 2(1):4–32, 1990.

[22] Fred Glover. "TABU Search: A Tutorial". Technical Report Technical Report, Center for Applied Artificial Intelligence, University of Colorado, 1990.

[23] Fred Glover and Harvey J. Greenberg. "New approaches for heuristic search: A bilateral linkage with artificial intelligence". *European Journal of Operational Research*, 39:119–130, 1989.

[24] Fred Glover and Claude McMillan. "The general employee scheduling problem: an integration of management science and artificial intelligence". *Computers and Operations Research*, 13(5):563–573, 1986.

[25] B.L. Golden and C.C. Skiscim. "Using simulated annealing to solve routing and location problems". *Naval Research Logistics*, 33:261–279, 1986.

[26] D.R. Greening. "Parallel simulated annealing techniques". *Physica D*, 42(1–3):293–306, 1990.

[27] B. Hajek. "Cooling schedules for optimal annealing". *Mathematics of Operation Research*, 13(2):311–329, 1988.

[28] P. Jaumard B. Hansen. "Algorithms for the Maximum Stability Problem". Technical Report RUTCOR Research Report 43-87, Ruters University, New Brunswick, NJ, 1987.

[29] A. Hertz. "Tabu Search for Large Scale Timetabling Problems". Technical Report Technical Report ORWP 89/4, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, 1989.

[30] A. Hertz and D. de Werra. "Using Tabu search techniques for graph coloring" *Computing*, 29:345 351, 1987.

[31] A. Hertz and D. de Werra. "The Tabu search metaheuristic: How we used it" *Annals of Mathematics and Artificial Intelligence*, 1:111 121, 1990

[32] Alain Hertz. "Finding a Feasible Course Schedule Using Tabu Search". Technical Report Technical Report, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, 1990.

[33] Skorin-Kapov Jadranka. "Tabu Search Applied to the Quadratic Assignment Problem". *ORSA Journal on Computing*, 2(1):33 45, 1990.

[34] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Scheron. "Optimization by simulated annealing: An experimental evaluation; Part 1, Graph partitioning". *Operations Research*, 37(6):865 892, 1989.

[35] B.W. Kernighan and S. Lin. "An efficient Heuristic procedure for partitioning graphs". *The Bell System Technical Journal*, pages 291 307, 1970.

[36] S. Kirkpatrick, C.D.Jr. Gelatt, and M.P. Vecchi. "Optimization by Simulated Annealing". *Science*, 220(1598):671-680, May 1983.

[37] C-H Lee, C-L Park, and M. Kim. "Efficient algorithm for graph partitioning problem using a problem transformation method". *Computer-Aided Design*, 21(10):611 618, 1989.

[38] Soo-Young Lee and J.K. Aggarwal. "A Mapping Strategy for Parallel Processing". *IEEE Transactions on Computers*, C-36(4):433 442, 1987.

[39] Jean Louis Pazat. "A friendly-greedy algorithm for process assignment". In M.Cosnard et al., editor, *Parallel and distributed algorithms*, pages 321-327. Elsevier Science Publishers B.V. (North-Holland), 1989.

[40] J. Ramanujam, F. Ercal, and P. Sadayappan. "Task allocation by simulated annealing". In *Proc of international conference on Supercomputing*. 1988.

[41] Frederic Semet and Eric Taillard. "Solving Real-Life Vehicle Routing Problems Efficiently Using Taboo Search". Technical Report ORWP 91/03, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, 1991.

[42] E. Taillard. "Robust Taboo Search for the Quadratic Assignment Problem". Technical Report ORWP 90/10, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, 1990.

[43] E. Taillard. "Some efficient heuristic methods for the flow shop sequencing problem". *European Journal of Operational Research*, 47:65-74, 1990.

[44] Lixin Tao. *Mapping parallel programs onto parallel systems with torus and mesh based communication structures*. PhD thesis, University of Pennsylvania, 1988.

[45] Lixin Tao, Yongchang Zhao, and Jiawei Guo. "An Efficient Tabu Search Algorithm for m-way Graph Partition". In *Supercomputing Symposium'91*, pages 263-270, Fredericton, NB, 1991.

[46] Jia wei Hong, Kurt Mehlhorn, and Arnold L. Rosenberg. "Cost Trade-offs in Graph Embeddings, with Applications". *Journal of the ACM*, 30(4):709-728, 1983.

[47] D.de Werra and A. Hertz. "Tabu Search Techniques: A Tutorial and an Application to Neural Networks". *OR Spektrum*, 11:131-141, 1989.

[48] M. Widmer. "Job shop scheduling with tooling constraints: a tabu search approach". Technical Report ORWP 89/22, Département de Mathématiques, Ecole Polytechnique Fédérale de Lausanne, 1989.

[19] Mickey R. Wilhelm and Thomas L. Ward. "Solving Quadratic Assignment Problems by 'Simulated Annealing"'. *IIE Transactions*, 19(1):107–119, 1987.

[50] M. Windmer and A. Hertz. "A new approach for solving the flow shop sequencing problem". *European Journal of Operational Research*, 11(2):186–193, 1989

Appendix A: Derivation of the Expected Minimal Cost for Multiple Runs

If we already have the costs for s runs of an heuristic with random initial solutions, we can easily derive from these costs the expected minimal cost for k ($k \ll s$) runs of the heuristic with random initial solutions.

Let $L = (c_1, c_2, \ldots, c_s)$ be the list of given costs in nondecreasing order of their value. The expected minimal cost for k runs of the heuristic is

$$\sum_{i=1}^{s-k+1} p_i \cdot c_i$$

where p_i is the probability that c_i is the minimal for k costs randomly chosen from L. We can decompose p_i as $p_i = p_i^1 \cdot p_i^2$ where p_i^1 is the probability that none of the first $i-1$ costs in L is among the k chosen costs, and p_i^2 is the probability that c_i is among the k chosen costs. It can be verified that

$$p_i^1 = \prod_{j=0}^{k-1} \frac{(s-i+1)-j}{s-j}$$

and

$$p_i^2 = 1 - \prod_{j=0}^{k-1} \left(1 - \frac{1}{(s-i+1)-j}\right) = \frac{k}{s-i+1}.$$

The cost derived above is more reliable than the one obtained by simply running the heuristic k times because it is based on the information for a much larger population of costs.

Appendix B: Heap Operations

```
Algorithm: Max(Q_h)
  begin
      return heap[1].key;
  end

Algorithm: Insert_Heap(Q_h,key,code)
  begin
      insert new_node(key, code) to last position of heap;
      while ( new_node.key > parent.key )
          swap( new_node, parent );
          update array edge_index according to parent code;
      end while
      return final position of new_node;
  end

Algorithm: Delete_Heap(Q_h,index)
  begin
      move last node to heap[index];
      update array edge_index according to heap[index] code;
      while (heap[index].key > parent.key)
          swap( heap[index], parent );
          update two edge_index values according to
          heap[index].code and parent.code;
      end while
      while (heap[index].key < children.key)
          swap( heap[index], children);
          update two edge_index values according to
          heap[index].code and children.code;
      end while
  end
```