Research Incubator: Combinatorial Optimization

Lixin Tao

School of Computer Science and Information Systems
Pace University

Technical Report 198

February 2004

Lixin Tao is Professor of Computer Science at Pace University.

Dr. Tao holds the Ph.D. in Computer Science from University of Pennsylvania. He has conducted extensive research in parallel and distributed computing, Internet computing, distributed component technologies, software engineering, and operations research. His contribution in operations research focuses on graph embedding, combinatorial optimization and their applications.

Preface

This document is part of the "Research Incubator: Combinatorial Optimization" project designed for Pace CSIS DPS students. In addition to this tutorial, the project has a two-hour intuitive presentation for all the material covered in this tutorial, as well as complete Java implementation of all the meta-heuristics covered in this tutorial in the context of solving the graph bisection problem. The Java implementation can be viewed as a set of software frameworks for the important meta-heuristics and can be easily adapted for solving various combinatorial optimization problems. All the three components of this project can be downloaded from http://csis.pace.edu/~lixin/dps.

Pace CSIS DPS Program is an innovative, rewarding and challenging endeavor. Its success depends on the active contribution and participation of CSIS faculty in the program's student research and supervision process. The accumulation of a set of reusable research methodologies can play critical role in the sustainable growth of this program.

This project is intended as a sample of a series of hands-on research incubators based on our faculty's expertise and research strength. This particular project was motivated by the observation that many of our colleagues have solid mathematical or business backgrounds, and the author believes they can play very important roles in the supervision of DPS students' research in the areas related to system optimization.

It is also the hope of the author that this project can contribute to increased research collaboration across the CSIS School.

Please pitch in!

Research Incubator: Combinatorial Optimization

Dr. Lixin Tao

School of Computer Science and Information Systems
Pace University
Westchester

October 10, 2003

Table of Contents

---

1 Introduction

Combinatorial optimization is a branch of applied mathematics dedicated to the system optimization in real world environments. It represents a huge body of reusable knowledge suitable for graduate students to study, apply, and develop. Many students were scared away by the mathematical notations commonly used in writings in this area, and they thought combinatorial optimization was an area for mathematicians or people who had advanced mathematical knowledge. This project is designed to use intuitive but practical approaches to illustrate that combinatorial optimization is an easy cut-in point for dissertation research, especially for students with limited computing background; and the core knowledge in this area is based on human intuitions that everyone may contribute and further develop.

This project has three components: (1) a personal two-hour presentation based on a PPT file to provide the intuitive and personal introduction to the area; (2) this document that provides comprehensive but concise introduction to the related key concepts and techniques; (3) a sample project of graph bisection integrated into (1) and (2) and implemented in Java for free download and study for all the important meta-heuristics covered in this project. The provided Java implementation is basically a set of software frameworks for multiple meta-heuristics that can be easily adapted to solve any combinatorial optimization problems.

The explanation of concepts in this document is purposely simplified in favor of easy understanding. The students are recommended to do further reading when they start to work on their own dissertation projects.

Please note, graph bisection is only used here as a sample research project for illustration of the general concepts and techniques with hands-on experience. It self is a very mature topic and in general not suitable for a student dissertation.

2 General Nature of Research on Combinatorial Optimization

The main objective of dissertation research is to *develop reusable knowledge in problem solving*.

Dissertation research on combinatorial optimization usually involves the following stages:

1. Identify one small but important problem that has not been resolved properly yet and its solution may bring the society significant benefits.
2. Conduct literature survey to see what is the current state for the resolution of this problem, and what tools are available to address this problem.
3. Abstract the problem into a concise but accurate statement or model, and verify that the abstraction is accurate and any solution based on the abstraction can be applied to any problem that fits this abstraction. This is the process to leave out non-essential elements of the original problem. The less assumption your problem formulation is based on, the harder will be your research, but bigger application domain your research results will have.

4. Design solution strategies and solution algorithms, identify the parameters of your solution algorithm and justify their values.
5. Design experiments to demonstrate the power of your solution algorithms. This usually involves comparing the performance of your algorithm to those claimed best in the literature.

In this project, we use graph bisection as a sample problem abstraction to illustrate the above steps. We recommend that you use this document and our implementation as the base to write a practice research paper following our outline above. This will give you hands-on experience on what is research.

3 Problem Identification and Abstraction

3.1 Real World Problems

We are interested in solving the following real-world problems.

- **VLSI circuit partition**

  A VLSI circuit is made up of a set of connected components. When the circuit grows with sophistication, we cannot implement a circuit on a single silicon chip. We need to partition the circuit into two halves and implement each half on a separate chip. But inter-chip connections can introduce significant signal delays or system vulnerability. The problem is: how to partition the circuit into two halves so that each side contains roughly the same number of components and the number of connections between the two halves is minimized.

- **Load Balance of Software Components on Clustered Servers**

  More clients are depending on computations on remote server computers. Web computing is one example. It is very important for a server to be able to serve a large number of clients concurrently with good response time.

  Server software usually is made up of a set of software components. At run time these components need to communicate with each other to coordinate their computation and exchange data. To provide more raw CPU power, a server system is usually based on a cluster of high-performance server machines. Communications across the server machines are very slow compared to CPU speed.

  The problem is how can we partition (divide) the components of a server software into roughly equal-sized two sets (collections) for running on two server machines, and minimize the inter-component communications between the two sets.

3.2 Problem Abstraction and Formulation

We can abstract the above two problems as a graph bisection problem. But first let us introduce some terminologies.

3.2.1 Basic Mathematical Terminologies and Notations

3.2.1.1 Set

A set is a collection of elements. All the elements in a set must be unique. The elements of a set are usually listed in a pair of curly braces. There is no order for elements in a set. The following are examples of some sets.

Example: **digits** = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Example: **oddDigits** = {1, 3, 5, 7, 9}
Example: **evenDigits** = {0, 2, 4, 6, 8}

We use $1 \in$ **digits** to mean that 1 is an element of set **digits**, or 1 is *in* set **digits**. We use **oddDigits** $\subseteq$ **digits** to mean all elements in set **oddDigits** are also in set **digits**, or **oddDigits** is a *subset* of **digits**.

Given a set *S*, we use |*S*| to denote the number of elements in *S*, or *cardinality* of |*S*|.

Example: |**digits**| = 10.

Given two sets *X* and *Y*, the *Cartesian product* of X and Y, denoted as $X \times Y$, is a set made up of all pairs like (*x*, *y*) where $x \in X$ and $y \in Y$. Obviously $|X \times Y| = |X| \bullet |Y|$.

Example: Let *X* = {a, b}, *Y* = {1, 2}. Then $X \times Y =$ {(a, 1), (a, 2), (b, 1), (b, 2)}, and $|X \times Y| = |X| \bullet |Y| = 2 \bullet 2 = 4$.

3.2.1.2 Partition of a set

Given a set *S*, a *partition* of *S* is to divide the elements of *S* into two or more non-overlapping subsets. Each element of *S* must be in exactly one of the subsets.

Example: Sets **oddDigits** and **evenDigits** constitute a partition of set **digits**.

Word *partition* has two meanings. One is as an action word for dividing, as seen above. We can also refer to each of the subsets of a partition as a partition too. For example, we can say that **oddDigits** and **evenDigits** are the two partitions of set **digits**.

If we partition set *S* into two subsets *L* and *R*, we also say we *bisect* set *S*, or *L* and *R* make a *bisection* of set *S*. For example, subsets **oddDigits** and **evenDigits** make a bisection of set **digits**.

3.2.1.3 Permutation and combination of a set

Give a set $V = \{0, 1, 2\}$, a *permutation* of set V is an ordering of all the elements of V. If a set has n elements, there will be $n! = n \times (n-1) \times \cdots \times 2 \times 1$ (n factorial) different permutations. Think about n boxes in a line that you need to fill each up with one element of the set. The first box has n choices; the second has n-1 choices; ……; the last box has only one choice. Remember that these choices are independent to each other.

Example: Set V above has $3 \times 2 \times 1 = 6$ permutations:

0, 1, 2 0, 2, 1 1, 0, 2 1, 2, 0 2, 0, 1 2, 1, 0

Given a set $W = \{0, 1, 2, 3\}$, a 2-*combination* of W is a subset of two elements of W. The following are all 2-combinations of set W:

{0, 1} {0, 2} {0, 3} {1, 2} {1, 3} {2, 3}

3.2.1.4 Graph

A *graph* is the abstraction of a set of connected elements. The elements are called *vertices* in a graph. If there is a connection between two elements x and y, we say there is an *edge* between vertex x and vertex y, and denote the edge as $\{x, y\}$. Mathematically, a graph $G = (V, E)$, or a graph G has two components: a set V of all vertices in the graph, and a set E of all edges in the graph. When you draw a visual diagram for a graph, the size or layout of the vertices and edges are not significant.

Example: Let $G = (V, E)$ be a graph where $V = \{0, 1, 2, 3\}$ and $E = \{\{0, 1\}, \{0, 3\}, \{2, 3\}, \{1, 3\}\}$. We can visually draw the graph as

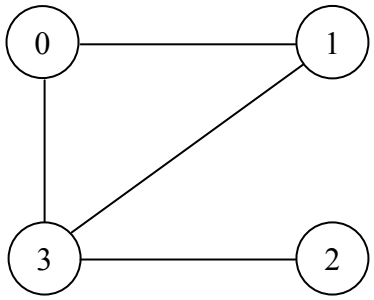

Figure 1 A mathematical graph

3.2.1.5 Adjacency matrix

Given a graph G with n vertices, we can always rename the vertices with integers 0, 1, 2, …, and represent the edges with an $n \times n$ *adjacency matrix adj*. Given any two vertices i and j of graph G, we define $adj(i, j) = 1$ iff (if and only if) there is an edge between vertices i and j. If there is no edge between vertices i and j, we define $adj(i, j) = 0$.

Example: Graph G in the previous diagram can be represented by the following 4×4 adjacency matrix, where each row or each column represents a vertex, and there is a 1 on row i and column j iff there is an edge between vertices i and j. The main diagonal line of a matrix is the line from the upper left corner to the lower right corner. In the following matrix we use a dotted line to show the matrix elements on the main diagonal. Please note that all matrix elements on the main diagonal have value 0 (since we don't have loop edges connecting a vertex back to itself), and the values of the matrix are symmetric to the main diagonal line (since our definition of an edge is symmetric: edge $\{i, j\}$ connects vertex i to vertex j, or from vertex j to vertex i).

	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0

Figure 2 An adjacency matrix

3.2.2 Problem Abstraction

We first consider the VLSI circuit partition problem. We can use non-negative integers 0, 1, 2, … to abstract the circuit components, and use a graph $G = (V, E)$ to abstract a circuit design, where V abstracts the set of circuit components, and E abstracts the connections among the circuit components.

For the second software component load-balancing problem on two server machines, we can also use non-negative integers 0, 1, 2, … to abstract the software components, and use a graph $G = (V, E)$ to abstract a component-based server application, where V abstracts the set of software components of the application, and E abstracts the communication requirements among the software components at run time.

3.2.3 Problem Formulation

Now we can derive our problem formulation for both of the above two problems as follows:

Given a graph $G = (V, E)$ where V has even number of vertices, find a partition of V into two equal-sized subsets so that the number of edges cut by the partition (edges that connect vertices belonging to two different subsets) be minimized.

More formally, we can formulate the problem as follows:

Given a graph $G = (V, E)$ where $|V|$ is an even integer, find a partition of V into subsets L and R that minimizes the *objective function*

$$\text{cutSize}(L, R) = \sum_{(x,y)\in L\times R} adj(x, y)$$

under the *constraint* that

$$|L| = |R|.$$

A partition satisfying the above conditions is called an *optimal* solution to the problem.

Example: The following graph has been partitioned into two equal-sized subsets by a dotted line, and the partition has a cut size of 2, which is the minimal cut size possible. This partition is therefore called optimal.

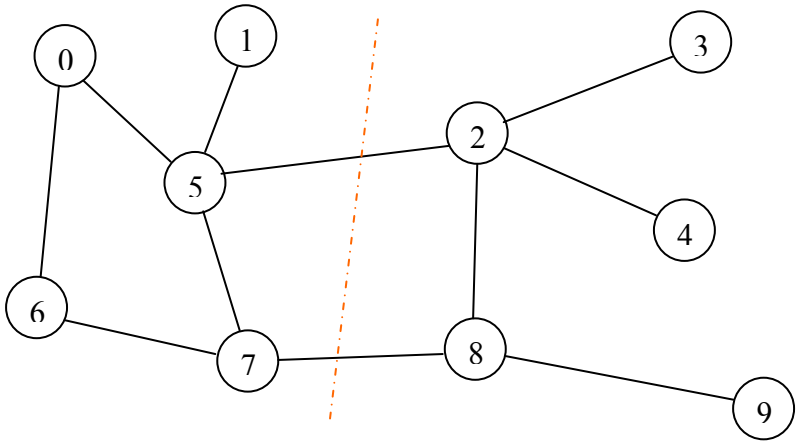

Figure 3 An optimal graph bisection

If in our research we could design an algorithm (problem solving procedure that can be implemented by a computer program) that can find an optimal solution for any given graph G, we could use this algorithm to solve both of the two real-world problems outlined in our Section 1, and many more real-world problems that also satisfy our mathematical problem formulation.

Exercise: Try to come up with two more applications of our mathematical problem formulation.

A solution satisfying the constraint is called a *feasible solution*. For graph bisection, a feasible solution is a partition of set V into two equal-sized subsets. In the following discussion we will call these two subsets the *left partition* and *right partition* respectively. Our objective is to find

one of the feasible solutions that minimize the objective function cutSize(). The objective function value of a feasible solution is also called the *cost* of the solution.

4 Problem Properties and Solution Space Design

After we have formulated a problem, before we start to think about the solution techniques for the problem, we should first think about the properties of the problem itself. Is it possible to compute an optimal solution for large problem instances? Can we simplify the problem first? The answers to these problems are problem specific.

This section also tries to set up the common base for all the solution techniques covered in the later sections.

4.1 Basic Terminologies for Computing Complexity

4.1.1 Problem vs. Problem Instances

A *problem* is a general problem formulation. A *problem instance* is a concrete case of the problem formulation.

Example:

Problem:	Sort *n* integers into non-decreasing order
Instance:	Sort 5 integers 2, 1, 4, 3, 6 into non-decreasing order

Given a problem, it represents infinite number of problem instances. Each problem instance can be characterized by a *problem instance size*. For the sorting example, the number of integers to be sorted, *n*, is the problem instance size. For our graph bisection problem, the problem instance size is the number of vertices in the graph. We can expect that the larger the problem instance size, the more CPU time will be required to solve a problem instance.

Our research aims to designing efficient general solutions to a problem, not specific solutions for a particular problem instance.

4.1.2 Algorithm

An algorithm is a sequence of well-defined steps for solving a problem. An algorithm must terminate execution for any problem instance within finite amount of CPU time. An algorithm is usually implemented as a software program. For time-critical real-time systems, we can also use hardware to implement an algorithm to speed up execution.

4.1.3 Time Complexity of an Algorithm

Given a problem, we may have many algorithms to solve it. We need a framework to compare the merits of the algorithms. Two of the main performance merits are *solution quality* and algorithm *running time*, and the latter is also called *time complexity*.

But we cannot just compare the actual running time of two algorithm implementations, since the actual running time also depends on hardware speed, compiler quality, and runtime environment (Is the system running many programs concurrently? How much memory a program can use?). Therefore we only define algorithm time complexity as a function of problem instance size and the logic of the algorithm itself. We usually surround the time complexity function with $O()$ (called big-O) to mean we don't care about constant coefficients or non-dominant terms of the function.

The time complexity of an algorithm is in general represented as a function of the problem instance size n. The most popular time complexity functions are $O(1)$ for constant time, $O(\log_2(n))$ for logarithmic time, $O(n \times \log_2(n))$ (popular for sorting), $O(n)$ for linear time, $O(n^2)$ or $O(n^3)$ for polynomial time, and $O(2^n)$ for exponential time.

We don't care about algorithm time complexity much when the problem instance is small since computers can finish computation fast any way. We focus on comparing algorithm time complexities when the problem instance is big.

4.2 NP-Hardness of a Problem

If an algorithm has an exponential time complexity $O(2^n)$, the algorithm cannot be used when a problem instance is larger than 100 or so. Since each decimal digit is represented roughly by three binary digits, 2^{100} is roughly equal to 10^{33}. The fastest supercomputer today can complete less than 10^{15} floating-point operations per second. Therefore, 10^{33} operations will take $10^{33}/10^{15} = 10^{18}$ seconds, or at least 10^{14} hours, or at least 10^{10} years. Even supercomputers in the future cannot help.

Over the last 40 yeas computer scientists have identified a set of problems and proved that, with very high probability, no algorithms can solve these problems with a time complexity fundamentally different from $O(2^n)$ [1]. This is an unusual achievement in science since we can make a claim that applies to the intelligence of future human beings. These problems are called *NP-hard* or *intractable* since there is no hope to come up with efficient algorithms to solve them for practical problem instances.

Unfortunately, many interesting problems in science and engineering belong to this intractable category. Our graph bisection is also intractable.

Even though we cannot have efficient algorithms to solve these intractable problems, the industries need good solutions to these problems any way. Any small improvement in the solution quality may imply significant benefits. So the problem becomes: within practical time limits, how can we find optimized, instead of optimal, solutions?

4.3 Solution Representation

Starting from this Sub-section we provide a common base for the solution techniques discussed in the next section.

For our graph bisection problem with instance size n, we can represent a solution with a one-dimensional array of size n. The vertices will be represented by the indices of the array. Let us call the array p. We define for each vertex i in {0, 1, 2, …, n-1},

p[i] = 0 if vertex i belongs to the left partition (first subset)
p[i] = 1 if vertex i belongs to the right partition (second subset)

Example: The optimal graph bisection in Figure 3 can be represented as

0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	0	0	0	1	1

Figure 4 Solution representation

4.4 Solution Space

We can view each partition p of set V as a point in an n-dimensional space. We call this space the *solution space* for the graph bisection problem.

Example: We can visualize the solution space as the following:

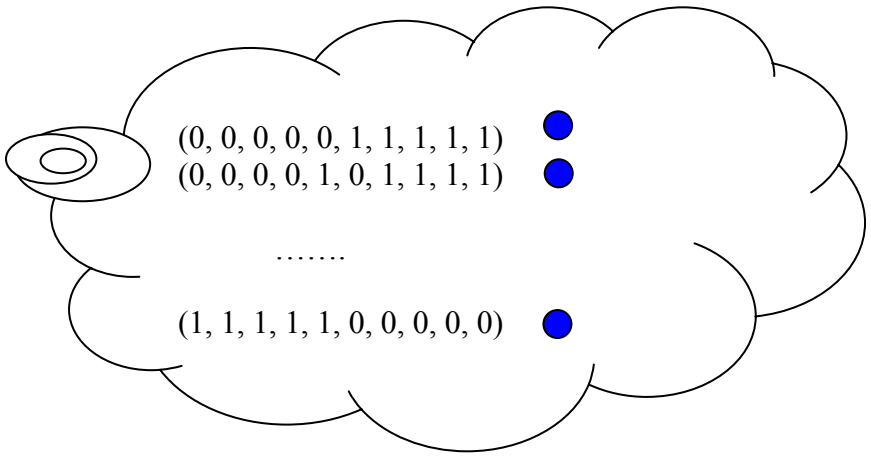

Figure 5 Solution space

Not all points in the solution space represent a feasible solution. For example, point (1, 0, 0, 1, 1, 0, 1, 1, 0, 1) does not represent a feasible solution since it does not satisfy the problem constraint that the left and right partitions must have equal number of vertices. We allow infeasible solutions in a solution space because for some algorithms a more flexible solution space may improve their freedom in exploring the feasible solutions (an important idea for research).

4.5 Solution Moves and Neighborhood Design

Most algorithms for solving NP-hard problems are based on a loop and a current solution. During each iteration of the loop, we perturb the current solution a little bit to get a neighbor of it, and decide whether to accept the neighbor as the new current solution based on some criteria. Therefore it is important to design a solution neighborhood for a current solution.

First we need to decide what *moves* (perturbations) are suitable for the problem solutions at hand. The moves are problem specific. For graph bisection, the common moves include swapping two vertices belonging to the two partitions, and move a vertex from its current partition to the other partition. Since moving a vertex will make a solution infeasible, we decide to use vertex swapping as our only move.

There are some general guidelines for choosing moves for a problem [9]:

- The reachability property. The moves should allow the algorithm to visit any feasible solution in the solution space, through a series of moves, starting from any current solution
- Ideally, the moves should support the incremental update of the objective function. Evaluating the objective function is in general a time consuming process. Suppose we know the cost of the current solution. After we apply a local move or perturbation to the current solution, hopefully we can derive the cost of the resulting solution by some simple constant-time modifications to the current cost, instead of evaluating the objective function entirely. Such incremental cost update can benefit any solution techniques, and based on the author's experience, is one of the key factors for success.

Exercise: For easy understanding, we don't use incremental update of the cut size in our algorithm implementations. We challenge you to derive formula for the incremental update of the current cut size cost after the application of a swap move, implement it in our algorithm implementation, and observe the improvement in algorithm execution time for large problem instances.

Give a solution in the solution space, a *neighbor* of the solution is another solution in the solution space that can be reached through the application of one of the defined moves. The *neighborhood* of a solution is made up of all neighbors of the solution. The neighborhood structure is derived from the move design. A more flexible move set will lead to a larger solution neighborhood.

5 Solution Meta-Heuristics and Their Application

For NP-hard problems, like graph bisection, we can only obtain optimal solutions for small problem instances. For practical problem instance sizes, heuristics must be used to find optimized solutions within reasonable time frame. A *heuristic* is an algorithm that tries to find good solutions to a problem but it cannot guarantee its success. Most heuristics are not based on rigid mathematical analysis, but on human intuitions, understanding of the properties of the problem at hand, and experiments. The value of a heuristic must be based on performance

comparisons among competing heuristics. The most important performance metrics are solution quality and running time.

Over the last half a century people have studied combinatorial optimization heuristics in solving many practical NP-hard problems, and some common problem-solving strategies underlying these heuristics emerged as meta-heuristics. A meta-heuristic is basically a pattern or main idea for a class of heuristics. Meta-heuristics represent reusable knowledge in heuristic design, and they can provide valuable starting points for us to design effective new heuristics in addressing new NP-hard problems. But meta-heuristics are not based on theory. We should not let them limit our own creativity. Meta-heuristics are not algorithms. To effectively solve a problem based on a meta-heuristic, we need to have deep understanding of the characteristics of the problem, and creatively design and implement the major components of the meta-heuristic. Therefore, using a meta-heuristic to propose an effective heuristic to solve an NP-hard problem is an action of research.

5.1 Exhaustive Search

For small problem instances of an NP-hard problem, we may still want to design algorithms to evaluate optimal solutions. The optimal solutions found by such algorithms can serve as comparison reference points for evaluating the effectiveness of heuristics.

The basic approach for finding optimal solutions is *exhaustive search*. This is a brute force approach. We systematically enumerate all feasible solutions, evaluate the objective function value (cost) for each of them, and report the ones with the minimal (maximal) costs.

The more advanced approach for finding optimal solutions is *branch-and-bound*. It is similar to exhaustive search. It incrementally constructs all feasible solutions. It uses a *bound function* to find the best cost that may be produced from a partial solution. When the bound function finds that a partial solution cannot produce any solutions better than the best solution seen so far, that partial solution will not be further expanded and explored, thus saving time in generating and evaluating all solutions based on that partial solution. The bound function is problem specific and it is the key for the performance improvement of branch-and-bound over exhaustive search.

For the graph bisection problem, our exhaustive search algorithm is based on the following pseudo-code.

```
For each (n/2)-combination of set V
    Let vertices in the (n/2)-combination go to the left partition.
    Let other vertices go to the right partition.
    Evaluate the cost of the resulting partition.
    If the cost improves the best one seen so far, record it.
End For.
Return the best partition visited.
```

Figure 6 Exhaustive search

5.2 Repeated Random Solutions

For larger problem instances, the exhaustive search cannot produce optimal solutions as reference points for performance evaluation. But we should expect any reasonable heuristic perform better than randomly generated solutions since randomly generated solutions are not exploring any property or structure of the problem, it is mindless.

One possible basic performance evaluation for a heuristic could be repeatedly generating random solutions for as long as the heuristic does, and see which produce better solutions in the same amount of time.

Our repeated random solution algorithm is based on the following pseudo-code. The constant 100 in it should be adjusted during implementation so the algorithm will run the same amount of time as a heuristic under evaluation.

```
Repeat 100 times
    Generate a random partition.
    Evaluate the cost of the resulting partition.
    If the cost improves the best one seen so far, record it.
End Repeat.
Return the best partition visited.
```

Figure 7 Repeated random solutions

5.3 Local Optimization

Local optimization is also called *greedy algorithm* or *hill-climbing*. Starting from a random initial partition, the algorithm keeps migrating to better neighbors in the solution space. If all neighbors of the current partition are worse, then the algorithm terminates. This scheme can only find *local optimal* partitions that are better than all of their neighbors; the found partitions may not be the *global optimal* ones. Figure 8 visually shows the difference between local and global solutions.

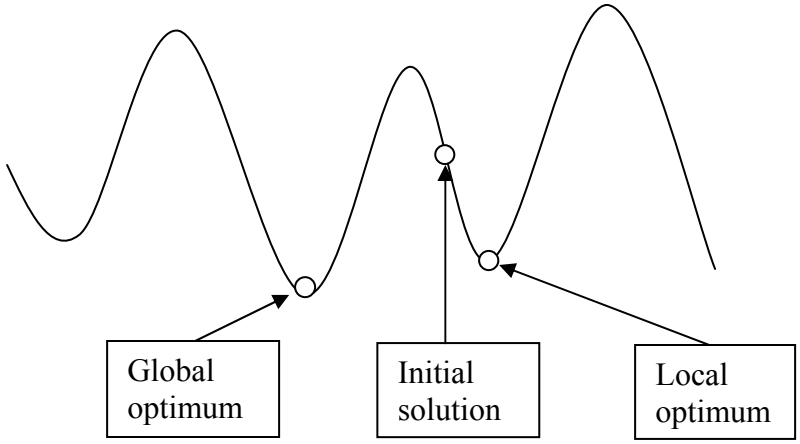

Figure 8 Local vs. global optimal solutions

By the original local optimization algorithm, from a current partition, we may need to check out all of its neighbors before we can terminate the algorithm. It is a time consuming process to maintain the visited neighbors of the current solution. Therefore we customized our local optimization algorithm for graph bisection as follows. The value 100 is a parameter for adjustment.

```
Get a random initial partition as the current partition.
While there is any improvement to the best cost seen so far in the last 100 iterations
    Generate a random neighbor of the current partition.
    Evaluate the neighbor's cost.
    If the neighbor's cost improves the current cost
        Let the neighbor be the new current partition.
        If the neighbor's cost improves the best one seen so far, record it.
    End If.
End While.
Return the best partition visited.
```

Figure 9 Improved local optimization

5.4 Simulated Annealing

In 1983 Kirkpatrick and his coauthors proposed to use the analogy of metal annealing process to design combinatorial optimization heuristics [22].

The atoms in metal have their natural home positions. When they are away from their natural positions, they hold energy to pull them back. Metal will be in its softest state when most of its atoms are in their natural home positions, and in its hardest state when most of its atoms are far away from their natural home positions. To make a sword with steel, we need first to put the

steel in very high temperature so the atoms can randomly move around instead of getting stuck in some foreign positions. The natural force will pull the atoms back to their home positions. Little by little, we lower the temperature until most of the atoms are frozen in their home positions. If the temperature is lowered too fast, some atoms may get stuck in foreign positions. This is the annealing process.

Kirkpatrick viewed the combinatorial optimization process analogous to the metal annealing process, and the optimal solutions analogous to metal in its softest state. Such analogies are not logically justifiable. Basically the physical metal annealing process gave Kirkpatrick some fresh ideas on combinatorial optimization.

The simulated annealing heuristic for graph bisection starts with a random initial partition as its current partition and a high temperature t. The heuristic then goes through loop iterations for the same temperature. During each iteration, a random neighbor of the current partition is generated. If the neighbor improves the current cost, then the neighbor becomes the new current partition for the next iteration. If the neighbor worsens the current cost, it will be accepted as the new current partition with a probability. When the temperature is high, the probability is not sensitive to how bad the neighbor is. But when the temperature is low, the probability to accept a worsening neighbor will diminish with the extent of the worsening. After L iterations at the same temperature, the temperature will be reduced by a very small amount, and the above looping repeats. The process will terminate when some termination criteria is met.

Simulated annealing has been widely applied to solve many combinatorial optimization problems. Simulated annealing is unique among all the other meta-heuristics for combinatorial optimization in that it has been mathematically proven to converge to the global optimum if the temperature is reduced sufficiently slowly. But this theoretical result is not very interesting to practitioners since very few real world problems will be able to afford such excessive execution time.

Our simulated annealing heuristic for graph bisection is based on the following pseudo-code.

```
Get a random initial partition π as the current partition.
Get an initial temperature t > 0.
While stop criteria not met do
    Perform the following loop L times.
        Let π' be a random neighbor of π.
        Let Δ = cost(π') – cost(π).
        If  Δ ≤ 0  (downhill move), set  π = π'.
        If  Δ > 0  (uphill move), set  π = π' with probability e^(-Δ/t).
    Set t = 0.95t (reduce temperature).
End While.
Return the best π visited.
```

Figure 10 Simulated Annealing

Let us have a look of the function of Δ and t in the probability function $e^{-\Delta/t}$. Since $\Delta > 0$ and $t > 0$, $e^{-\Delta/t} = \frac{1}{e^{\Delta/t}}$. When t is much larger than Δ, Δ/t approaches 0, $e^{\Delta/t}$ approaches to 1, and $e^{-\Delta/t}$ approaches to 1. This indicates that when the temperature is high, the heuristic has very high chance to accept worsening neighbors. But when t is much smaller than Δ, Δ/t approaches ∞, $e^{\Delta/t}$ approaches to ∞, and $e^{-\Delta/t}$ approaches to 0. This indicates that when the temperature is very small, the heuristic has very small chance to accept worsening neighbors.

When we compare local optimization and simulated annealing, we find they mainly differ in whether to accept worsening neighbors. For simulated annealing, it starts with random walk in the solution space. When a random neighbor is better, it always takes it. But if the neighbor is worsening, its chance of accepting it is reduced slowly. Simulated annealing is reduced to local optimization when the temperature is very low.

The heuristic above is purposely simplified to improve its readability. The readers are encouraged to improve it using the ideas outlined below [2, 6].

- The initial temperature is a parameter for adjustment. A too high temperature will waste CPU time. A too low temperature risks entrapment in local optimum.
- The stop criteria for the while loop should be refined. A typical implementation is to stop the algorithm when no solution improvement happens for some consecutive number of iterations of the while loop.
- The top criteria for the inner loop should be refined. The parameter could be adjusted. More sophisticated stop criteria may improve the performance.
- The way the temperature is reduced should be refined. The constant 0.95 should be replaced with a value based on experiments. Or a more sophisticated cooling schedule could be used to improve the performance.

5.5 Tabu Search

Tabu search is another meta-heuristic for combinatorial optimization becoming very active in late 1980s [5, 6]. The proponents of tabu search disagree with the analogy of optimization process to metal annealing process. They argued that when a hunter was put in an unfamiliar environment, he will not walk randomly first but zero in to the area that appears most promising in finding games. This is similar to the greedy local optimization process. Only when neighboring areas are all worse than the current area will the hunter be willing to walk through worsening neighboring areas in hope of finding a better local optimum. To avoid being trapped in a loop in the solution space (for example, accepting a worsening neighbor, then returning back to the better starting solution right away), tabu search uses a tabu list to remember the recent moves and avoid repeating them. Tabu means prohibition here.

Tabu search differs from simulated annealing in that it is more aggressive and deterministic. Our tabu search heuristic for graph bisection starts by generating a random partition as the current

partition. It then executes a loop until some stopping criteria are met. During each iteration, the current partition is replaced with its best neighbor that is not tabued on the tabu list.

Theoretically the tabu list should record the solutions visited recently so we can avoid repeating them. But checking each neighbor against each of the recorded solutions will take up too much execution time. Therefore in practice we usually only record some features of the recent solutions, or some features leading to these recent solutions. We use the vertices involved in the last swap generating a solution to characterize that solution. Our tabu list is implemented as a circular queue with 10 cells. If the best adopted neighbor is generated by swapping vertices x and y, both x and y will be inserted in the circular queue replacing the ones that entered the queue the earliest. A neighbor of the current partition will be tabued if one of the vertices involved in the swap to generate this neighbor is on the tabu list.

Our tabu search heuristic for graph bisection is based on the following pseudo-code, in which the length of the tabu list is $2t$.

```
Get a random initial partition π as the current partition.
While stop criteria not met do
    Let π' be a neighbor of π minimizing Δ = cost(π') – cost(π)
       and not visited in the last t iterations.
    Set π = π'.
End While.
Return the best π visited.
```

Figure 11 Tabu Search

The heuristic above is purposely simplified to improve its readability. The readers are encouraged to improve it using the ideas outlined below [5, 6].

- The stopping criteria for the while loop should be refined with a better scheme.
- The tabu list should be refined. Better characteristics of a recent solution should be used in the tabu list so tabu list checking could be efficient and the minimal number of unvisited solutions will be tabued unnecessarily.
- The length of tabu list should be studied and adjusted. The literature reported a length of five to be appropriate. But this number is problem specific.
- The concept of *aspiration level* should be studied and applied. Aspiration level of a move is used to partially correct the over-tabu problem caused by the approximate implementation of a tabu list. If a swap achieves a particular aspiration level, it can be applied even if one of its vertices is tabued.
- More advanced concepts like short-term memory and long-term memory could be applied to improve the performance.

5.6 Genetic Algorithm

Genetic algorithm is based the analogy of combinatorial optimization to the process of natural selection and natural genetics. Its application in combinatorial optimization area started back in early 1960s [7].

In a genetic algorithm, a solution is represented by a *code*. The algorithm starts with the generation of a pool of codes for random solutions. This pool is called *generation* 0. To generate the next generation, parents are randomly selected from the previous generation according to some *selection* criteria. Every pair of such parent codes could be randomly *crossovered* (mixed) to generate a new child code in the new generation. Each such parent could also be randomly *mutated* (modified) to generate a new child code. Hopefully the advantages of the parents could be combined to generate a better child, and a mutated parent could lead to unexplored area of the solution space. This generation production process will be repeated until some stopping criteria are met.

For our genetic algorithm for graph bisection, we represent a partition with an array of n real numbers between 0 and 1, where n is the size of the vertex set. To convert such a code to a partition, we sort the real numbers with their corresponding indexes or vertex numbers. Those vertices in the left half of the sorted list belong to the left partition, the others in the right partition.

Example: The following code c can be converted to partition p.

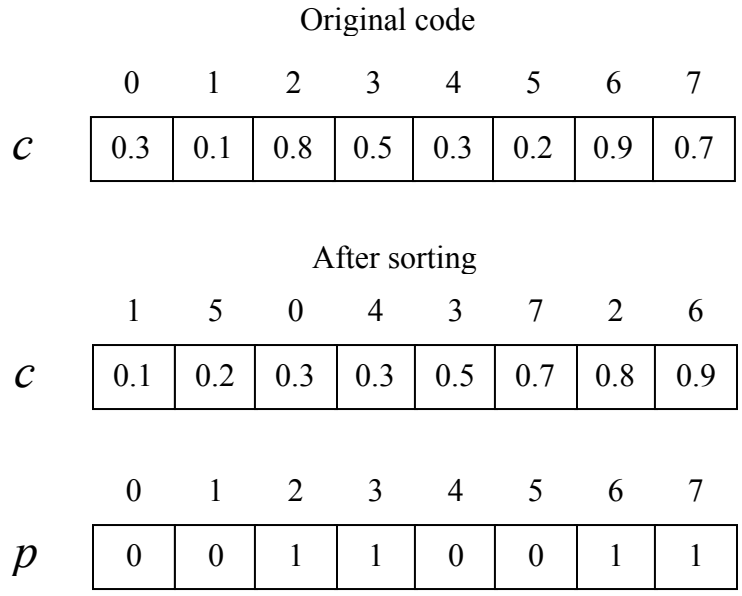

Figure 12 Convert a code to a partition

Given the codes of two parents, we conduct crossover by let the child code inherit half or more values from parent 1, and the rest from parent 2. The exact number of values to be inherited from parent 2 as well as the exact cells that will take corresponding values in parent 2 is randomly selected.

Given the code of a parent, its mutation child code will first inherit its values from the parent, then randomly modify up to ¼ of its own values. The exact number of modifications as well as the exact cells that will be modified is randomly selected.

Now we can describe our genetic algorithm for graph bisection with the following pseudo-code.

```
Generate 50 codes of random partitions, and sort them in the generation table
   according to their costs.
While there are improvements to the best cost seen so far in the last 50 iterations do
      Use crossover to generate 40 children with randomly chosen parents,
         and insert them into the generation table according to their costs.
      Use mutation to generate 10 children with randomly chosen parents,
         and insert them into the generation table according to their costs.
      The generation table only keeps the best 50 codings.
      If the best code improves the best cost seen so far, record it.
End While.
Return the best partition visited.
```

Figure 13 Genetic Algorithm

The heuristic above is purposely simplified to improve its readability. The readers are encouraged to improve it using the ideas outlined below [7, 6].

- Find better encoding for the partitions that can be easily converted to partitions.
- Adopt a better selection scheme for parents to generate the following generation.
- Find better schemes to conduct crossover and mutation.
- Find the best generation table size.
- Find the best ratio for children generated by crossovers and mutations.
- Adopt more effective stopping criteria.

6 Experiment Design for Performance Evaluation

To compare the relative performance of the various heuristics, we should either collect problem instance data from real-world applications, or generate random problem instances of various sizes and connection density. We should evaluate performance in terms of both partition quality and running time. The following are some possible experiments to conduct:

- For the same problem instance, let all the heuristics run to their natural conclusion, and compare their partition quality and running time.

- For the same problem instance and a good partition achievable by previous experiments, see which heuristic can reach this partition with the shortest running time.
- For problem instances of diverse sizes and connection densities, see which heuristic can provide consistent good performance without adjusting its parameters.

All data reported should be based on statistics over multiple runs for the same problem instance and multiple runs for different problem instances of the same size and connection density. The commonly reported data include the best value, the worst value, the average value, and the standard deviation.

7 Common Research Pitfalls

- Work on a problem without practical significance

  Research, especially practical research, should aim at solving problems that are important in improving the solutions to real world problems.

- Strictly follow book descriptions of a meta-heuristic

  Meta-heuristics are only problem-solving ideas. They must be adapted for solving a particular problem. Meta-heuristics are not based on rigid logic reasoning. You should treat them as suggestions only. You should feel free to modify the meta-heuristics and come up with your own approaches. Remember, when you are solving a very hard NP-hard problem, a mathematician may not be in a better position than you are.

- Solve any given problem with meta-heuristics

  First, meta-heuristics are only useful for solving NP-hard problems. If a problem is not NP-hard, you should try to design algorithms for finding optimal solutions.

  Second, even if you are working on an NP-hard problem, if the solution quality is really critical, you should try to study the special properties of the problem and come up with a problem-specific heuristic. You should treat meta-heuristics as your last resort in solving NP-hard problems.

- Solve a problem without proper problem abstraction and formulation

  Best chance of outperforming existing heuristics for a problem is to study the special properties of the problem. Problem abstraction will help you identify the key components and structure of the problem, and problem formulation will help you see opportunities of simplifying the problem. The result of these two steps can benefit any solution strategy that you decide to adopt later.

- Formulate a problem based on a meta-heuristic

By doing so you are adding conditions and views of the solution strategy to the problem formulation. It may lead you to an unnecessarily complex problem formulation. When you later decide to switch to other solution strategies, those extra conditions and views embedded in the problem formulation may cause serious inefficiency.

- Parameters need be adjusted for different problem instances

  Research is for discovering reusable knowledge. If you need to adjust parameters for different problem instances, the knowledge represented by your heuristic is not generally applicable. It is not practical to make such frequent parameter adjustments in real-world environment.

- Performance evaluation does not cover running time

  We use heuristics because we cannot afford the running time of algorithms for optimal solutions, and we need to find good solutions fast to serve the real-world needs. Combinatorial optimization is a practical problem, not only a theoretical problem.

- Performance evaluation only for trivial problem instances

  We adopt heuristics because we need to find good solutions fast for practical large problem instances. For trivial problem instances, we can just use exhaustive search. This is a common problem for combinatorial optimization heuristics based on neural networks because these heuristics are intrinsically slow.

- Methodology evaluation based on user survey only

  The effectiveness of a solution methodology depends on many factors, including the properties of the problem, how hard is the problem, whether the methodology has been implemented properly for the problem, and whether the users know how to use the methodology implementation effectively. Users' feedback may not reflect the effectiveness of the methodology itself.

  User surveys should always be used as a last resort to analyze the properties of complex systems whose internal structure or underlying logic is beyond our understanding. A good example is to study the social impacts of some technologies.

- Human help during heuristic execution

  While human beings can help heuristics in solving small problem instances based on their intuitions and logic reasoning, human beings cannot help for large problem instances. More importantly, when human beings and the heuristic have different opinions, which should have more weight? Unless this weighting can be automatically derived with a well-defined algorithm, this effort has no value for research.

8 References

1. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco. 1979.
2. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. "Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research*, vol. 37, issue 6 (Nov.-Dec.), 1989, pp.865-892.
3. S. Kirkpatrick, C. D. Celatt Jr., and M. P. Vecchi. "Optimization by Simulated Annealing," Science 220, May 13, 1983, pp.671-680.
4. B. W. Kernighan and S. Lin. "An Efficient Heuristic procedure for partitioning Graphs," Bell Systems Technical Journal, 49, 1970, pp.291-307.
5. F. Glover. "Tabu Search – Part 1," *Operations Research Society of America Journal on Computing*, vol. 1, no. 3, summer 1989, pp.190-206.
6. F. Glover and G. A. Kochenberger. *Handbook of Metaheuristics*. Kluwer Academic Publishers, 2003.
7. D. E. Goldberg. Genetic Algorithms in Search, Optimization & machine Learning. Addison-Wesley, 1989.
8. T. N. Bui and B. Moon. "Genetic Algorithm and Graph partitioning," IEEE Transactions on Computers, vol. 45, no. 7, July 1996, pp. 841-855.
9. L. Tao and Y. Zhao. "Multiway Graph Partition by Stochastic Probe," Computers & Operations Research, vol. 20, no.3, 1993, pp.321-347.

9 Appendix A: A Sample Dissertation Outline for Combinatorial Optimization

Solving a Class of Time-Dependent Combinatorial Optimization Problems with Abstraction, Transformation and Simulated Annealing

Chapter 1: Introduction

1.1 Problem Statement and Solution Strategies
1.1.1 Problem Statement
1.1.2 Solution Strategies
1.2 Research Contributions
1.3 Dissertation Outline

Chapter 2: Combinatorial Optimization Strategies

2.1 Fundamental Concepts
2.1.1 Generic Formulation of Combinatorial Optimization
2.1.2 NP-hardness of a Problem
2.1.3 Solution Space, Moves and Neighborhood
2.1.4 Local vs. Global Optimal Solutions
2.2 Dominant Solution Meta-Heuristics
2.2.1 Local Optimization
2.2.2 Genetic Algorithm
2.2.3 Simulated Annealing
2.2.4 Tabu Search

Chapter 3: Abstraction and Transformation of a Class of Time-Dependent Optimization Problems

3.1 Real World Time Dependent Optimization Problems
3.1.1 Minimal Cost Satellite Receiver Placement
3.1.2 Highway Minimum Bidding
3.2 Abstraction of a Class of Time-Dependent Optimization Problems
3.2.1 Problem Formulation A
3.2.2 Application of Problem Formulation A
3.3 Problem Transformation
3.3.1 Critical Observation of Problem Formulation A
3.3.2 Simplified Problem Formulation B
3.3.3 Problem Transformation Theorem
3.3.4 A Transformation Example
3.4 Benefit of Problem Transformation
3.4.1 Exhaustive Search
3.4.2 Heuristic Solution Searches
3.5 Common Foundations for Solution Searches

10 Appendix B: How to Run the Sample Heuristic Implementations

Before you can compile and run my java implementations, you must install a J2SE full version on you PC, and put its bin directory on your PATH.

Assume you installed your version of J2SE in c:\j2se1.4. You can use Start|Run|cmd to launch a terminal window, and type

```
set PATH=c:\j2se1.4\bin;%PATH%
```

to set up your PATH.

Download my file graphBisection.zip from URL

http://csis.pace.edu/~lixin/dps/

and unzip the file on your hard disk. Inside a terminal window, you change directory to the directory containing my Java source code.

To compile the source files, type

```
javac  GenData.java
javac  PartGraph.java
```

To generate a new problem instance with 20 vertices into a file named graph20.txt, type

```
java  GenData  20  graph20.txt
```

To run all heuristics on the same problem instance file graph20.txt, type

```
java  PartGraph  graph20.txt
```

To run Exhaustive Search only on problem instance file graph20.txt, type

```
java  ExhaustiveSearch  graph20.txt
```

To run Hill Climbing (local optimization) only on problem instance file graph20.txt, type

```
java  HillClimbing  graph20.txt
```

To run Repeated Random only on problem instance file graph20.txt, type

```
java  RepeatedRandom  graph20.txt
```

To run Simulated Annealing only on problem instance file graph20.txt, type

```
java  SimulatedAnnealing graph20.txt
```

To run Tabu Search only on problem instance file graph20.txt, type

```
java  TabuSearch  graph20.txt
```

To run Genetic Algorithm only on problem instance file graph20.txt, type

```
java  GeneticAlgorithm  graph20.txt
```