A PRIMAL-DUAL HEURISTIC FOR THE TRAVELING SALESMAN PROBLEM

A PRIMAL-DUAL HEURISTIC FOR THE TRAVELING SALESMAN PROBLEM

By

XIAOXI MA, B.Sc

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

© Copyright by XIAOXI MA, September 2010

MASTER OF SCIENCE (2010)
(Computing And Software)

McMaster University
Hamilton, Ontario

TITLE: A Primal-Dual Heuristic for the Traveling Salesman Problem

AUTHOR: Xiaoxi Ma, B.Sc. (Tsinghua University)

SUPERVISOR: Professor George Karakostas

NUMBER OF PAGES: x, 79

Abstract

In this thesis we provide a Linear Programming (LP) formulation and a heuristic for the symmetric Traveling Salesman Problem (TSP) on certain complete graphs having the triangle inequality.

TSP models cities and their pairwise connections as vertices and edges between them in a graph. The distances are represented by cost values on edges, and the goal is to find a minimum weight tour that visits every vertex exactly once. In symmetric cases all connections are undirected — both directions have the same cost. This problem is NP-Complete, so there is no polynomial time exact algorithm known for it.

We present three major points in this thesis. Inspired by an LP formulation of perfect matching, we develop a relaxation for TSP, and prove that our relaxation is equivalent to the path form of the well-known Held-Karp formulation. Then, based on this relaxation we construct a heuristic, hoping it can approach a constant factor 4/3 of the optimal objective value given by the LP relaxation. At last, we adopt the matroid idea. It's already known that TSP can be modeled as minimum weight intersection of three matroids, but solving that is also NP-Complete. We present in this thesis the attempt to approach it using only two matroids, and analyze the difficulty.

Acknowledgments

First and foremost, my deepest and sincerest gratitude goes to my supervisor, George Karakostas, who has led me through this program with meticulous guidance, profound knowledge and great patience. Our frequent meetings were the strongest motivation of my research. Without his supervision this thesis will never be possible. His enthusiasm and creativity has had a beneficial impact on my life.

I gratefully acknowledge professor Michael Soltys and professor Antoine Deza for teaching outstanding courses. They not only facilitated me with excellent knowledges but also enhanced my interest in computer theory especially in complexity theory.

I also want to thank my friends Jing Wang, Bingzhou Zheng, Mustafa Elsheikh, Volodymyr Babiy and so on. I have had a great time with them, any all their help made my life easier and happier.

It is a pleasure to thank my families: my parents and my grandfather. They have always been so supportive and understanding. Without their love my life will be just meaningless and miserable.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during my research and the completion of this thesis.

Table of Contents

Abstract iii

Acknowledgments v

List of Figures ix

1 Introduction 1
1.1 Linear Programming 3
1.2 Previous work 4
1.3 Our contribution 7
1.4 Thesis outline 8

2 The Primal-Dual Method 11
2.1 Equivalence between TSP paths and TSP tours 11
2.2 The Primal-Dual Method 13
2.3 Edmonds' Blossom Algorithm for Matching 18
2.4 The Sub-tour Constraints (Held-Karp Formulation) of TSP 25
2.5 Our Linear Program Relaxation for TSP path 30
2.6 Test Cases for Our LP Relaxation 33

3 TSP Heuristics 41
3.1 Finding augmenting paths 42
3.2 Solving higher-degree problem nodes 47
3.3 When tight-edge cycles occur 56
3.4 Test Cases on This Heuristic 64
3.5 A Matroid approach 68

4 Conclusions and future work 75

Bibliography **77**

List of Figures

2.1 2-factor 24

2.2 Held-Karp 3/4-factor case 26

2.3 The integrality gap and approximation factor 28

2.4 Test case 1 35

2.5 Test case 2 36

2.6 Test case 3 37

2.7 Test Case 4 38

2.8 Test case 5 39

3.1 grow path 45

3.2 make big group when necessary 47

3.3 higher-degree problem node 48

3.4 Shrinking higher degree moats 50

3.5 Making big group when tight triangle occurs 51

3.6 Making big group when new node is discovered 52

3.7 Solved Higher Degree Problem Nodes 56

3.8 Tight edges close a cycle 57

3.9 Another case of tight edge cycle 59

3.10 Tight edge cycle after solving higher degree problem node 60

3.11 General case of tight edge cycle 61

3.12 four-thirds approximation factor 63
3.13 Running our heuristic on a variation of Held-Karp 4/3 example . . . 65
3.14 Example of the graph translation 72

Chapter 1

Introduction

In the field of combinatorial optimization, the Traveling Salesman Problem (TSP for short) has been very important. Given a set of cities and their pairwise distances we want a shortest possible tour that visits each city exactly once. This problem has been intensively studied since it involves research from many fields like Mathematics, Operation Research, Game Theory and even Artificial Intelligence and Biology. The Traveling Salesman Problem is an NP-Complete problem, so no polynomial-time exact algorithm has been found. Many famous problems can be related to the TSP problem, such as the matching problem, the spanning tree problem and the degree restricted subgraph problem.

The Traveling Salesman Problem can be seen as finding a shortest Hamilton Cycle, since the Hamilton Cycle problem is to find a tour that visit every city exactly once.

We can model the TSP problem as a graph problem. The cities are presented as vertices in a graph, and the connections between cities are edges in the graph, with edge weights representing distances between cities. So, the problem is formulated as: Given a graph $G = (V, E)$ and a weight vector $\bar{c} = \{c_e : e \in E\}$, find a circuit C in

G that covers all the vertices with minimum possible total weight. If the graph G is directed, then we have the Asymmetric TSP, otherwise we have the symmetric TSP (the connection from city A to B has distance equal to the connection from B to A). In this thesis we deal with the symmetric case.

Due to the TSP's comprehensive characteristics, many other problems are closely related to it. We briefly introduce some of them:

Hamilton Path/Cycle Problems

In an undirected connected graph, a Hamilton path is a path that visits every vertex exactly once. A Hamilton cycle is a cycle in an undirected connected graph which visits every vertex exactly once and gets back to the starting point. Deciding whether a graph has a Hamilton path or a Hamilton cycle is NP-Complete.

b-Matchings and b-Factors

In an undirected graph $G = (V, E)$, there is a positive degree constraint vector $\vec{b} = \{b_v : v \in V\} \in Z^V$. Let $\delta(v)$ denote the set of edges incident to v. For any set $S \subseteq E$, let $x(S) = \sum_{e \in S} x_e$. A b-matching of G is a vector $\vec{x} = \{x_e : e \in E\} \in Z^E$ such that $x(\delta(v)) \leq b(v)$ for each node $v \in V$. A b-matching x is called simple if x is a 0,1 vector, i.e., $x_e \in \{0, 1\}$ for every edge $e \in E$, and we call a b-matching perfect if for every node $v \in V$, $x(\delta(v)) = b(v)$. A b-factor is a simple perfect b-matching.

Spanning Tree Problems

In an undirected connected graph $G = (V, E)$, a spanning tree is an acyclic subset $T \subseteq E$ that covers all the vertices. If there is a non-negative weight vector $\vec{c} = \{c_e : e \in E\}$ giving weight value to each edge, a minimum spanning T is a spanning tree with minimal possible total weight $w = \sum_{e \in T} c_e$. Several efficient polynomial time algorithms have been developed to solve this problem. The most

well-known one was given by Kruskal [6] with a running time of $O(|E|\log|V|)$.

1.1 Linear Programming

In combinatorial optimization, many problems can be modeled as finding the minimum or maximum accumulation of resources, given restrictions on the availability of these resources. If we can model the resources as variables, specify the accumulation as a linear function and the restrictions as linear equalities and inequalities on these variables, then we can formulate this problem as a Linear Programming (or LP for short) problem. Therefore the general form of a Linear Program is :

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to:} \quad & Ax \geq b \\ & x \geq 0 \end{aligned}$$

In this formulation, x is the vector of variables, c and b are vectors of coefficients and A is a given matrix of coefficients. The function $c^T x$ is called the *objective function*, and $Ax \geq b$ are called the *linear constraints*. The program we stated is called a *minimization linear program*, and if we want the maximal possible value of the objective function, it will be called a *maximization linear program*. A *feasible solution* x is a setting of variables that satisfies all the linear constraints. If some constraints are not satisfied, that setting is called an *infeasible solution*. The value of the objective function on feasible solutions is called the *objective value*. Geometrically, the feasible region defined by the linear constraints is called a *convex polytope*.

The linear program we presented above is in *standard form*, which is the most intuitive form to model a problem. An equivalent [6] form, called the *slack form*, is:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to:} \quad & Ax + Ix_s = b \\ & x, x_s \geq 0 \end{aligned}$$

I is the identity matrix, and x_s is a vector of slack variables.

For any given linear program, there is a *dual linear program* that has the same optimal objective value, ensured by the strong duality theorem. The original program is called *primal.* We will state duality theorems in detail in Section 2.2.

There is a discrete version of LP called *Integer Programming*, or *Integer Linear Programming*, where the variables are all required to be integers. Many combinatorial optimization problems are naturally modeled as integer programs. But unlike LPs which can be solved in polynomial time, the integer programs are much more difficult and sometimes NP-hard. So we need to relax the modeled integer programs to continuous linear programs by allowing x to take values in a real range instead of a set of discrete values. The TSP is an example of this phenomenon. More details are provided in Section 2.2. The ratio between the integral optimal solution and the solution of its relaxed linear program is called the *integrality gap.*

1.2 Previous work

The Traveling Salesman Problem has been a very popular subject of research for decades due to its wide range of applications and potential for improvement. Efforts have been made from many different aspects to approach the optimum solution. A good survey of this problem is [16]. We briefly introduce some previous work in this section.

Branch-and-bound

The Traveling Salesman Problem is NP-Complete, and there is no polynomial time exact algorithm known to solve this problem. All the exact algorithms are basically enumerative. So one tries to narrow down the range of enumeration. An idea called branch-and-bound was introduced to solve the TSP problem by Tompkins in [25] and also in [23]. This kind of algorithm finds the exact optimum solution but its running time depends heavily on how to branch and how to bound. Since these are not polynomial time algorithms, several large scale problems cannot be solved efficiently by them.

Construction Heuristics

The **Nearest neighbor heuristic** starts from a random vertex, and moves to its nearest neighbor. From there it again moves to the nearest neighbor which is not visited yet. Repeat this process until all the vertices are visited, and get back to the starting vertex at the end. The approximation factor is defined to be the ratio between the approximate solution and the optimal solution. The nearest neighbor heuristic runs in polynomial time, but there is no approximation factor guaranteed for it. For a detailed description and proofs, readers are referred to [21].

Another kind of heuristic is called the **Insertion Heuristic**. It starts from an initial small tour (possibly one or two vertices), and inserts one unvisited vertex at a time, until all the vertices are included. Different strategies are used to select the candidate inserting vertex, such as the nearest vertex to the tour, the cheapest increase to the tour, or even random. Different strategies result in different running times and approximation factors. These can be found in [21].

Christofides gave a heuristic using minimum spanning trees in [4]. Essentially,

it finds a minimum spanning tree on the given graph, and than collects all the odd degree tree nodes. On this set of nodes it finds a minimum perfect matching. Every matching edge adds one degree to each of its two end points, so all the nodes now have even degrees. Combining the edges of the spanning tree and the perfect matching gives a tour that visits all the nodes and gets back to the starting vertex. He proved that the approximation factor of this heuristic is within 3/2. (We use the word approximation factor to denote the factor between the approximate solution and the optimum solution)

Lower Bounds

G. Reinelt introduced several classes of lower bounds for the traveling salesman problem in [21], including simple lower bounds, lower bounds from linear programming and Lagrangean relaxations. We select two of them which are closely related to our research and give a brief introduction here.

As introduced in previous section, a **2-factor** is a subset of edges that ensures every vertex has exactly 2 incident edges. It consists of a bunch of cycles that cover all the vertices. A TSP tour is a 2-factor that contains only one cycle. So obviously any 2-factor gives a lower bound of the minimum total weight of the TSP tour.

Dantzig, Fulkerson and Johnson [7] gave a linear program relaxation of the TSP, which is called the **sub-tour bound**. It can also be called the **Held-Karp bound** [13], since these two bounds are equal. In Section 2.4 we will discuss this bound in further details.

Intersection of Matroids

A matroid is a structure consisted of a finite universe of elements and a family

of independent subsets of this universe with the hereditary property and the independent property. A precise definition will be given in Section 3.5. The TSP problem can be modeled as intersection of three matroids, described in [15]. First we extend the n nodes graph by adding a new node and n edges that connect this new node to all the original ones. The first matroid is a graphic matroid on this new graph, the second matroid is a partition matroid having independent sets containing at most one edge ingoing to any given node, and the third matroid is similar to the second but contains at most one edge outgoing from any given node. Then the minimum weight intersection of these three matroid will be a solution of the TSP problem.

Unfortunately the intersection of three or more matroids is NP-Complete, which means there exists no polynomial time algorithm for doing this. So, we use another model and try to reduce the number of matroids to two. We will discuss it later in Section 3.5.

1.3 Our contribution

In this thesis we use Junger and Pulleyblank's [12] Linear Program formulation for the metric complete perfect matching problems and develop our own LP formulation for the TSP problems. Classic methods as introduced in the last section find TSP tours that cover all nodes in the graph. But our formulation finds TSP paths that cover all nodes with two given end points. These two forms — TSP tour and TSP path — are equivalent, and we prove it in Section 2.1. Then we prove that our formulation has the same integrality gap as Held-Karp [10] [13], and then run some test cases to see the behavior of this well-known bound.

Then, based on this formulation, we try to design a heuristic for constructing approximate TSP solutions for certain symmetric complete graphs satisfying the triangle inequalities. The integrality gap of the Held-Karp formulation has been proved to be at least $4/3$, so we try to make our heuristic achieve this factor as the approximation guarantee.

On the other hand, as we have already mentioned, the Traveling Salesman Problem can be modeled as the intersection of three matroids. But no polynomial time algorithm is known for solving the intersection of three matroids. So in this thesis we try to model the TSP problem as intersection of two matroids, a 2-factor matroid and a spanning tree matroid. we do not succeed in doing that, but we discuss the difficulties arising from this attempt.

1.4 Thesis outline

Chapter 2 introduces the Primal-Dual method for linear programming problems. Then in Section 2.3, as an example of the primal-dual method and also a preparation for later sections, we introduce Edmonds' blossom algorithm for minimum weight perfect matching. In Section 2.4 we present the sub-tour or Held-Karp formulation for the TSP. In Section 2.5 we give our own formulation for the TSP and prove that it is equivalent to the Held-Karp one. In the last section we provide some test cases generated using our own formulation.

The first three sections of Chapter 3 present our heuristic for the TSP. In the first section we use the idea of the TSP tour consisting of two alternative matchings and Edmonds' blossom algorithm to build a TSP path. But during this process the

path may grow into a tree or close a cycle which does not cover all nodes, making the solution infeasible. In the following two sections we deal with these problems. In Section 3.4 we show some test cases on this heuristic. Then in Section 3.5 we use matroid ideas to analyze the TSP.

Chapter 2

The Primal-Dual Method

The Primal-Dual Method is a very powerful tool for designing approximation algorithms for combinatorial optimization problems, and especially NP-Hard problems.

In this chapter we first prove a theorem stating that Traveling Salesman Problem (TSP) paths and TSP tours are equivalent, so an α-approximate algorithm focusing on finding TSP paths also guarantees the same approximation factor for TSP tours. Then we introduce some basic concepts of the Duality Theory of Linear Programming based on the Complementary Slackness Conditions and how to get approximation factors from them. We describe Edmond's blossom algorithm for the matching problem as an example of the method. Then we introduce the Sub-Tour Constraints formulation for the TSP. In the last section we present our own linear program relaxation for the TSP.

2.1 Equivalence between TSP paths and TSP tours

In this thesis there are two kinds of TSP solutions involved: TSP tours that in a given graph visit all nodes exactly once and get back to the starting point; and TSP

paths that visit every node exactly once, starting with and ending in two different given nodes.

Suppose that we have an algorithm that finds a TSP path within an approximation factor α. We run this algorithm on every pair of nodes s and t, so we can get the α guaranteed paths P_{st} for all those pairs. Then we add edge (s,t) with cost $c_{(s,t)}$ to each path P_{st} respectively, and get TSP tours. We select among all these tours the one with the minimal cost.

Theorem 2.1.1. *If the approximate TSP paths are bounded by an approximation factor α, then the minimal cost TSP tour selected by the above selection criteria is bounded by the same approximation factor α.*

Proof. We know that for any node u, there must exist a node v such that edge (u,v) is on the optimum TSP tour OPT_T. Since we run the algorithm on every pair of end points, there must be a pair of nodes s and t such that edge $(s,t) \in OPT_T$. The cost of an approximate tour APP_T is the summation of the edge cost $c_{(s,t)}$ and the cost of an approximate path APP_P, which is bounded by a factor $\alpha \geq 1$ times the cost of the optimal path OPT_P, so

$$cost(APP_T) = c_{(s,t)} + cost(APP_P) \tag{2.1.1}$$

$$cost(APP_P) \leq \alpha cost(OPT_P) \tag{2.1.2}$$

$$\begin{aligned}(2.1.1),(2.1.2) \Longrightarrow cost(APP_T) &\leq c_{(s,t)} + \alpha cost(OPT_P)\\ &\leq \alpha(c_{(s,t)} + cost(OPT_P))\\ &= \alpha OPT_T\end{aligned}$$

The selected TSP tour has the minimal cost, i.e., $cost(T) = min\{cost(APP_T)\}$, so $cost(T) \leq \alpha OPT_T$. Thus we know that if the TSP paths are bounded by an approximation factor α, then the selected TSP tour is within the same approximation factor. □

2.2 The Primal-Dual Method

To explain the Primal-Dual method in detail, we need some concepts from Linear Programming.

Suppose that n and m are two positive integers, A is a given coefficients matrix and c, b are coefficients vectors having dimensions of n and m respectively. The primal variable vector has dimension of $|x| = |c| = n$, and the dual variable vector has dimension of $|u| = |b| = m$. For any given linear programming problem, there exists a dual problem corresponding to it:

$$\begin{array}{cc} \text{Primal problem(LP):} & \text{Dual problem(DLP):} \\ minimize \quad cx & maximize \quad ub \\ subject \quad to: & subject \quad to: \\ Ax \geq b & uA \leq c \\ x \geq 0 & u \geq 0 \end{array} \tag{2.2.1}$$

Theorem 2.2.1. *(Weak Duality) Let $\bar{x}$ be any feasible solution for the primal problem, and $\bar{u}$ be any feasible solution for the corresponding dual problem. Then we have*

$$c\bar{x} \geq \bar{u}b$$

Proof. From the dual problem we know that $\bar{u}A \leq c$, so we got $\bar{u}A\bar{x} \leq c\bar{x}$, and on the other hand, from the primal problem we got $A\bar{x} \geq b$ and $\bar{u}A\bar{x} \geq \bar{u}b$. So $c\bar{x} \geq \bar{u}A\bar{x} \geq \bar{u}b$, which means $c\bar{x} \geq \bar{u}b$. □

Theorem 2.2.2. *(Strong Duality) If the primal problem (or the dual problem) has unbounded solutions, then the corresponding dual (or the primal respectively) has no feasible solution. If either of them has a finite optimal solution, then so does the other, and their optimal objective values are equal.*

For the detailed proof of the Strong Duality Theorem, readers are referred to Linear Programming textbooks like [28].

Theorem 2.2.3. *(Complementary Slackness Conditions) Let $\bar{x}$ be a feasible solution for the primal problem and $\bar{u}$ be a feasible solution for the corresponding dual problem. Then both $\bar{x}$ and $\bar{u}$ are optimal if and only if:*

1. (Primal Complementary Slackness Conditions)

$$\bar{x}_j > 0 \Longrightarrow \sum_{i=1}^{m} \bar{u}_i a_{ij} = c_j, \qquad \textit{for any} \quad j \in [1, n]$$

2. (Dual Complementary Slackness Conditions)

$$\bar{u}_i > 0 \Longrightarrow \sum_{j=1}^{n} a_{ij} \bar{x}_j = b_i, \qquad \textit{for any} \quad i \in [1, m]$$

The Complementary Slackness Conditions (or CS Conditions for short) can be written in other forms as well, e.g., see [28, 19, 14] or any Linear Programming textbook.

The classic Primal-Dual method [11] is a technique to solve linear programming problems based on the CS conditions. Consider the linear programs (2.1.1). The main idea of this method is to start from an initial feasible dual solution u (typically an all-zero solution, if feasible) and check to see if there exists a feasible primal solution satisfying the CS Conditions with respect to u. While there is no such primal solution, we need to construct a new pair of linear programs. The objective function of the new primal program characterizes the "violation" of the original primal constraints and the CS conditions,

$$\begin{aligned}
& Minimize \quad \sum_{i \notin I} s_i + \sum_{j \notin J} x_j \\
subject \quad to: \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \qquad \forall i \in I \\
& \sum_{j=1}^{n} a_{ij} x_j - s_i = b_i, \qquad \forall i \notin I \\
& x_j \geq 0, \qquad \forall j \in E \\
& s_i \geq 0, \qquad \forall i \notin I
\end{aligned}$$

where $I = \{i | u_i = 0\}$ and $J = \{j | \sum_{i=1}^{m} u_i a_{ij} = c_j\}$. If the optimum objective value of this program is 0, then x is a feasible primal solution satisfying the CS conditions with respect to u and we are done. Otherwise, the new dual program will also have an optimum objective value greater than 0, i.e., $\sum_{i=1}^{m} u'_i b_i > 0$. This solution u' provides a way to improve the original dual solution u. We can find an $\epsilon > 0$ such that $u + \epsilon u'$ is feasible to the original dual program with increased optimum objective value. This improvement of dual solution brings the corresponding primal solution "closer" to feasibility. We always keep the dual solution feasible and repeat this "check and improve" procedure until the primal solution is also feasible. At this point we've

found a pair of feasible solutions for both the primal and dual programs, and by Theorem 2.1.3 these are optimal solutions to both the primal and the dual problems.

In combinatorial optimization, many problems can be naturally formulated as linear programs. So the Primal-Dual method has played an important role in the design of many algorithms, as we will see in the following sections. In these problems, what we want are usually integral solutions. But most of the times the optimal solution of the formulated linear programs are fractional. So we can't use the classic Primal-Dual method directly. We need to modify it by relaxing the CS conditions, and then we can get approximate integral solutions.

In the classic Primal-Dual method, when we construct the new pair of LPs we take into consideration both the primal and the dual CS conditions, so when terminate with both primal and dual feasible solutions, Theorem 2.1.3 ensures that they are both optimal. In the design of approximation algorithms, we relax one of the CS conditions, say the dual. So when we improve the dual solution in each iteration, we not only keep the dual solution feasible, but also keep the primal solution integral. During each iteration we try to bring the primal solution closer to feasibility, i.e., the primal CS condition closer to satisfaction. Due to the integrality gap, we can not model this process as the classic Primal-Dual method. As an example of the primal-dual scheme, see, e.g., its application to hitting set problems surveyed in [11].

The relation between primal and dual solutions is the Complementary Slackness Conditions, as described in Theorem (2.2.3). During the whole procedure, a very important thing to keep in mind is the approximation factor. If both the CS Conditions are satisfied, then after termination we've found the exact optimum solution. If we force only one condition, say the dual CS Condition, and relax the primal one

by an approximation factor α, then what we get will be an approximation algorithm bounded by α. Consider the pair of primal and dual programs (2.1.1). The primal and dual CS conditions are:

$$x_j > 0 \Longrightarrow \sum_{i=1}^{m} u_i a_{ij} = c_j, \qquad \text{for any} \quad j \in [1, n], \text{ and } x \geq 0$$

$$u_i > 0 \Longrightarrow \sum_{j=1}^{n} a_{ij} x_j = b_i, \qquad \text{for any} \quad i \in [1, m], \text{ and } u \geq 0$$

We maintain all the dual conditions and $x \geq 0$, $u \geq 0$, and relax the primal ones as follows for some $0 < \beta < 1$:

$$x_j > 0 \Longrightarrow \beta c_j \leq \sum_{i=1}^{m} u_i a_{ij} \leq c_j, \qquad \text{for any} \quad j \in [1, n]$$

Proposition 2.2.4. *For any feasible primal solution x and dual solution u, if both the relaxed primal CS condition and the dual CS condition are satisfied, then* $\sum_{j=1}^{n} c_j x_j \leq \frac{1}{\beta} \sum_{i=1}^{m} u_i b_i$.

Proof. From the relaxed primal CS condition we know that $c_j \leq \frac{1}{\beta} \sum_{i=1}^{m} u_i a_{ij}$, so:

$$\begin{aligned} \sum_{j=1}^{n} c_j x_j &\leq \sum_{j=1}^{n} \left(\frac{1}{\beta} \sum_{i=1}^{m} u_i a_{ij}\right) x_j \\ &= \frac{1}{\beta} \sum_{i=1}^{m} \sum_{j=1}^{n} u_i a_{ij} x_j \\ &= \frac{1}{\beta} \sum_{i=1}^{m} u_i b_i \qquad (\text{ using the dual CS condition}) \end{aligned}$$

□

Let $\alpha = \frac{1}{\beta}$, so $\alpha \geq 1$. Suppose that we have a primal feasible solution $\bar{x}$ and a dual optimal solution $\bar{y}$ such that the relaxed CS conditions hold. Then from Proposition 2.2.4 we know that $c\bar{x} \leq \alpha\bar{y}b$, so the approximation factor for our solution is no greater than α, because the objective value cx for our integral primal solution x is bounded by $[\bar{y}b, \alpha\bar{y}b]$ — $\bar{y}b$ being the optimal objective value.

2.3 Edmonds' Blossom Algorithm for Matching

In this section we focus on the Matching problem, see how the Primal-Dual method helps in designing a combinatorial optimization algorithm for it, and how the Matching algorithm may help to build our own TSP heuristic.

Given a graph $G = (V, E)$, a matching M is a set of edges such that no node is incident to more than one edge in the set M. A node v is *covered* by M if some edge e_{vw} in M is incident to v, otherwise v is *M-exposed.* A *maximum matching* is one that has the fewest M-exposed nodes, or equivalently, one with the maximum cardinality. A *perfect matching* is one that covers all the nodes. Given a graph $G = (V, E)$ and a matching M of it, a path P is an *M – augmenting* path if the edges on P are alternatively in M and not, and both the end points of P are M-exposed.

Theorem 2.3.1. *(Augmenting Path Theorem) Given a graph $G = (V, E)$, a matching M is maximum if and only if there do not exist any M – augmenting paths.*

For the proof of this theorem, readers are referred to [1] .

We can assign a weight value c_e to each edge e, and then finding a perfect matching with the lowest total weight value is called the Minimum-Weight Perfect

Matching problem. Edmonds gave to this problem an integer programming formulation in [5]:

$$\begin{aligned} & Minimize \quad \sum_{e\in E} c_e x_e \\ subject \quad to: \quad & x(\delta(v)) = 1, \qquad \forall v \in V \\ & x_e \in \{0,1\}, \qquad \forall e \in E. \end{aligned}$$

In this formulation $\delta(v)$ is the set of edges incident to node v, and

$$x(\delta(v)) = \sum_{i\in\delta(v)} x_i$$

We relax the integral constrains, replacing them by nonnegative constraints

$$x_e \geq 0, \quad \forall e \in E$$

and get a linear program. From this linear program we can expect a lower bound for the optimum solution. For bipartite graphs, Birkhoff in [2] and Edmonds in [8] proved that the relaxed linear program is a sufficient characterization of the perfect matching polytope whose extreme points correspond to the matchings in graph G. (See also [22]).

For the general case, Edmonds [8] gave a new formulation which outputs the optimum solution. He added to the relaxed linear program a new kind of constrains, which are called the blossom inequalities.

Given a graph $G = (V, E)$, a cut $D = \delta(S)$ is called an odd cut if the set S of nodes has an odd cardinality and $\delta(S)$ is the set of edges who have one end point in S and the other outside S. If a matching M is a perfect matching of G, then M must contain at least one edge from D. Hence, we get the blossom inequalities:

$$x(D) \geq 1, \qquad \forall D \in \mathcal{C}$$

where $\mathcal{C}$ denotes the set of all possible odd cuts in G which are not of the form $\delta(v)$ for any single node v.

Including the blossom inequalities to the linear program, we get

$$\begin{aligned} & Minimize \quad \sum_{e\in E} c_e x_e \\ subject \quad to: \quad & x(\delta(v)) = 1, \qquad \forall v \in V \\ & x(D) \geq 1, \qquad \forall D \in \mathcal{C} \\ & x_e \geq 0, \qquad \forall e \in E. \end{aligned}$$

Theorem 2.3.2. *(Edmonds' Matching Theorem) Given a graph $G = (V, E)$ and a real vector $c \in \mathbf{R}^E$, G has a perfect matching if and only if the blossom relaxed linear program has a feasible solution, and the optimal objective value equals to the minimum total weight of a perfect matching.*

To prove this theorem, we will construct a perfect matching M using the optimal solution of the linear program. First we need the dual problem,

$$\begin{aligned} & Maximize \quad \sum_{v\in V} y_v + \sum_{D\in \mathcal{C}} Y_D \\ subject \quad to: \quad & y_v + y_w + \sum_{e\in D} Y_D \leq c_e, \qquad \forall e = vw \in E \\ & Y_D \geq 0, \qquad \forall D \in \mathcal{C}. \end{aligned}$$

The Complementary Slackness Conditions for this pair of programs are

$$x_e > 0 \Longrightarrow y_v + y_w + \sum_{e\in D} Y_D = c_e, \quad \forall e \in E \qquad (Primal)$$
$$Y_D > 0 \Longrightarrow x(D) = 1, \quad \forall D \in \mathcal{C} \qquad (Dual)$$

In the remainder of this thesis we use the word *moats* to denote dual variables. If a moat is of the form y_v, or Y_D and $v \in S, D = \delta(S)$, we say that this moat is *around* node v and v is *in* this moat. Especially, we call Y_D a *blossom moat*, and it's

around the node set S and any node $v \in S$. A moat *cuts* an edge if one end point of it is inside this moat and the other is outside. We say that an edge is *paid* by some moat by an amount equal to this moat's dual variable if this moat cuts the edge. An edge is *fully paid* if the total value of all moats cutting this edge adds up to the edge weight value exactly. We also call an edge *tight* if it is fully paid.

The primal CS condition means that for an edge to be in the matching M, it has to be fully paid by all the dual moats cutting it. And the dual CS condition means that for any odd cut $D = \delta(S)$ of G, if there exists a non-zero blossom moat Y_D around S then exactly one edge of M must be in this cut. We ensure the primal CS Condition by picking matching edges only from fully paid ones. In the following algorithm we describe how the blossom moats are grown, and that ensures the dual condition.

During the process of this blossom algorithm we keep a structure T which we call an *augmenting tree.* B and A denote two disjoint sets of nodes, and $T = (B, A)$ is the union of B and A. Initially $T = \emptyset$, and an invariant property of the augmenting tree is: along any branch of this tree, matching edges and unmatched edges appear alternatively. The node set B contains tree nodes who have *even distance* from the root (meaning the number of edges from the root to it is even), and node set A contains *odd distance* tree nodes. Recall that a node is called M-exposed if it is not incident to any edges in the matching M.

Now we can state the blossom algorithm for finding a minimum weight perfect matching:

Initial: Set $y_v = 0$ for any $v \in V$, $Y_D = 0$ for any $D \in \mathcal{C}$. $M = \emptyset$. Select an M-exposed node r as the current root for the augmenting tree, and $T = (B, A) = (\{r\}, \emptyset)$

Loop:

Case 1: If there exists some tight edge e_{uv} with one end point $u \in B(T)$ and the other an M-exposed node $v \notin T$, then use this edge e_{uv} to augment the matching M: $M = M \cup P - (M \cap P)$, where P is the path from root r to u together with edge e_{uv}.
If after augmenting there is no more M-exposed nodes, then terminate and we have found the perfect matching.
Otherwise, move to a new M-exposed node r' and replace T by $T = (\{r'\}, \emptyset)$. Back to Loop.

Case 2: If there exists some tight edge e_{uv} with one end point $u \in B(T)$ and the other an M-covered node $v \notin T$, then use this edge e_{uv} to extend T by making $E(T) = E(T) \cup \{e_{uv}, e_{vw}\}$ where w is the other end of the matching edge covering v. Back to Loop.

Case 3: If there exists some tight edge e_{uv} with both end points $u, v \in B(T)$, which means the tight edges have closed an odd cycle C, then we can shrink this tight edge odd cycle to a supernode s', and update M and T: $M = M - \{e_{xy} : e_{xy}$ is matched and $e_{xy} \in C\}$, $T = T - (C - \{e_{uv}\})$. Back to Loop.

Case 4: If there is a supernode $w \in A(T)$ with $Y_w = 0$, which means we cannot shrink it any more, then expand it and update M and T: assume that C is the cycle shrunk to w, $M = M \cup \{e_{xy} :$ e_{xy} is matched and $e_{xy} \in C\}$, let e_{mw} and e_{nw} denote the two edges incident to the supernode w in the augmenting tree T, and let u, v denote the end points of e_{mw}, e_{nw} on cycle C. P is the even-length path on C from u to v. $T = T \cup P$. Back to Loop.

Case 5:	If every $e_{uv} \in E$ incident to $v \in B(T)$ has its other end in $A(T)$ and $A(T)$ contains no supernode, then terminate with graph $G = (V, E)$ having no perfect matching.
Case 6:	Otherwise we need to change the y values. $\bar{c_e}$ denotes the remaining unpaid weight of edge e. Let $\epsilon = min\{\epsilon_1, \epsilon_2, \epsilon_3\}$, where $\epsilon_1 = min\{\bar{c_e} : e = uv, v \in B(T) \text{ and } u \notin T\}$, $\epsilon_2 = \frac{1}{2}min\{\bar{c_e} : e = uv, u \in B(T), v \in B(T)\}$, $\epsilon_3 = min\{Y_w : w \text{ is a supernode and } w \in A(T)\}$ Grow all the moats around B nodes by amount of ϵ and shrink all the moats around A nodes by the same amount. Back to Loop.

Edmonds' blossom algorithm is closely related to our problem. We are looking for a path in the TSP problem, and it can be deconstructed into two alternative matchings. On the other hand, if we can find two disjoint maximal matchings, we can combine them together and end up with a path. During the process of the blossom algorithm we always keep an augmenting tree consisting of alternating matched and un-matched edges along any path from a tree node to the root. Since it is a tree rather than a path, some internal nodes may have degree more than 2. At these nodes two or more un-matched edges are incident, which makes it impossible for the current unmatched edges to form a new matching. If we can restrict the augmenting tree to a path, or find out two maximal matchings who share no edges at all, then we get a TSP path.

The minimum weight perfect matching problems is a special case of the general matching problem, usually called the b-matching problem [23].

Given a graph $G = (V, E)$ and a positive degree constraint vector $b \in N^V$, let

$\delta(v)$ denote the set of edges incident to v. A *b-matching* of G is a vector $x \in N^E$ such that $x(\delta(v)) \leq b(v)$ for each node $v \in V$. A b-matching x is called *simple* if x is a 0,1 vector, i.e., $x_e \in \{0, 1\}$ for every edge $e \in E$. A *b-factor* is a simple *perfect* b-matching, i.e., $x(\delta(v)) = b(v)$ for each node $v \in V$. A generalized blossom algorithm for general b-matching was described in detail by William R. Pulleyblank in [20], and also by R. J. Urquhart in [27].

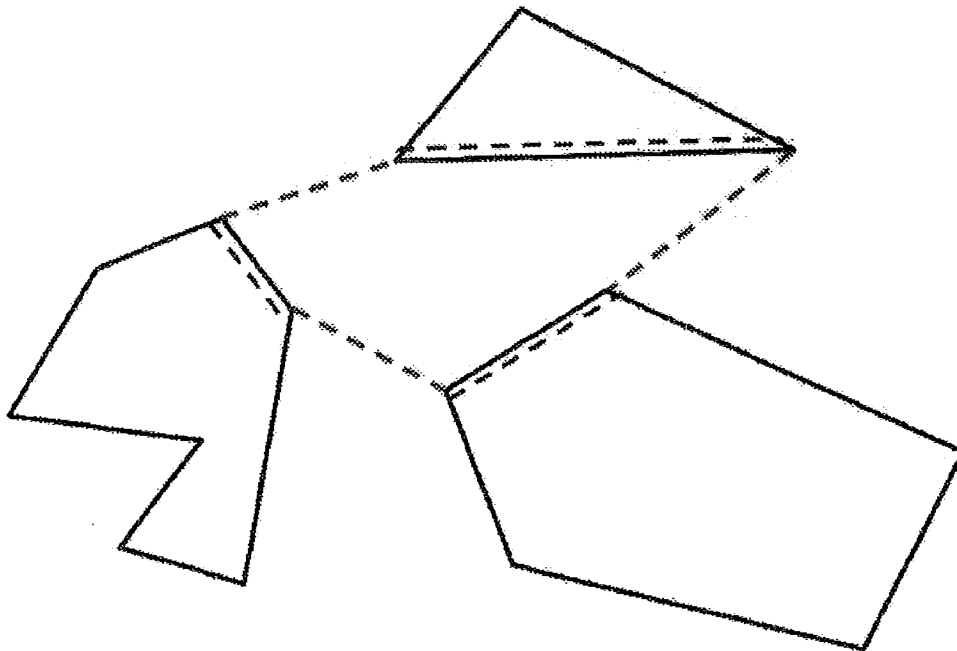


Figure 2.1: 2-factor

In the 1-factor problem, every node is incident to exactly one edge, and this is the perfect matching problem. The 2-factor problem, in which every node is incident to exactly two edges, is a relaxation of the TSP problem. As we can see, for the solid edges in Figure 2.1, every node has two incident edges that make it a 2-factor solution. But for it to be a TSP feasible solution, there must be a single connected component. For example, if in Figure 2.1 we eliminate the dash-solid edges from the 2-factor solution, and add to it the dash ones, a TSP tour will be formed.

2.4 The Sub-tour Constraints (Held-Karp Formulation) of TSP

In this section we introduce a linear programming formulation called Sub-tour constraints given by Dantzig, Fulkerson and Johnson [7]. It is also known as Held-Karp formulation by Held and Karp [10] [13] since the two provide the same lower bound for the TSP problem. (A proof of the equivalence of these two formulations can be found in [5]). The ratio between the objective value of the integral optimal TSP solution and the optimal objective value of the linear program is called the integrality gap. First we state the Held-Karp LP relaxation:

$$\begin{aligned} & Minimum \quad \sum_{e \in E} c_e x_e \\ subject \quad to: \quad & x(\delta(v)) = 2, \qquad \forall v \in V \\ & x(\delta(S)) \geq 2, \qquad \forall S \subset V, S \neq \emptyset \\ & x_e \geq 0, \qquad \forall e \in E. \end{aligned} \qquad \text{(HK)}$$

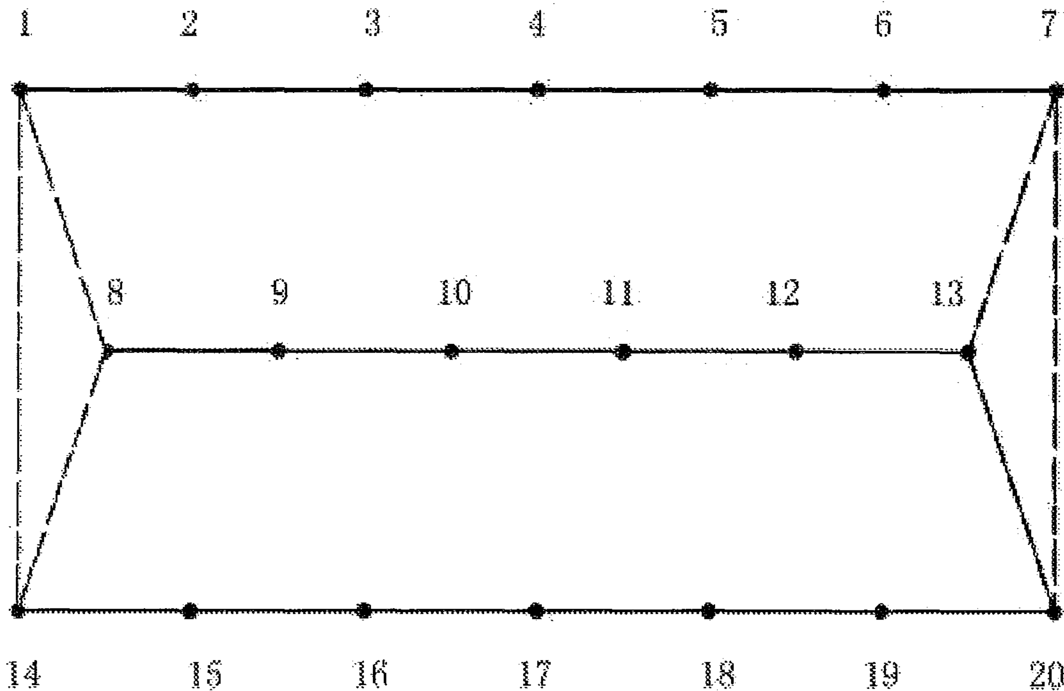


Figure 2.2: Held-Karp 3/4-factor case

Goemans and Bertsimas establish a property in [9] called the *parsimonious property*. In the Held-Karp relaxation, we say that a feasible solution x is *parsimonious* at node v if $x(\delta(v)) = 2$. The constraints $x(\delta(v)) = 2, \forall v \in V$ are called the *parsimonious constraints*. The parsimonious property states that if the cost vector $\vec{c}$ satisfies the triangle inequality, then introducing the parsimonious constraints doesn't affect the solution of the LP relaxation. Note that if we restrict the nonnegative constraints $x_e \geq 0, \forall e \in E$ and let the vector $\vec{x}$ take only 0 or 1 as its value, i.e., $x_e \in \{0, 1\}, \forall e \in E$, then in any optimal solution every node has exactly two incident edges. We can see them as one incoming edge and one outgoing edge. Plus we know this solution has only one connected component, then every node is visited exactly

once. So from the definition we know that any optimal solution for this integer program is a tour. And since this tour is of the minimal total weight, we know it is a TSP tour. Readers are referred to [18] for further details of this integer program. When it's relaxed to the linear program above, the optimum objective value is a lower bound for the TSP tour. Consider the graph in Figure 2.2. Solid lines indicate edges with $c_e = 1$ and $x_e = 1$, and dashed lines indicate edges with $c_e = 2$ and $x_e = 1/2$. The graph is complete, and for all edges that are not shown we have $x_e = 0$. The cost for edges $(u, v) \in E$ is $c_{uv} = \sum_{e \in \hat{p}_{uv}} c_e$, where $\hat{p}_{uv}$ is the shortest path between node u and v. The optimum objective value given by the (HK) relaxation is $\sum_{e \in E} c_e x_e = 23$, and a TSP tour, which is an integral optimal solution, has an objective value of 27. Figure 2.2 can be generalized to a family of graphs which have $3n + 2, n \in \mathbf{Z}^+$ nodes in total: $n + 1$ nodes instead of 7 nodes on path $p_{1,2,3,4,5,6,7}$ and path $p_{14,15,16,17,18,19,20}$, and n nodes instead of 6 nodes on path $p_{8,9,10,11,12,13}$. Edge weights follow the same rule as the example in Figure 2.2. For a graph of this family having $3n+2$ nodes, the optimum objective value given by the (HK) relaxation is $2n + (n - 1) + 6 = 3n + 5$, and the objective value of a TSP tour solution is $2n+2(n-1)+2*3 = 4n+4$. When n goes to infinitely large, the limit value of the integrality gap is

$$\alpha = \lim_{n \to +\infty} \frac{4n + 4}{3n + 5} = \frac{4}{3}$$

The integrality gap α is the ratio between the integral optimal solution and the fractional optimal solution of the relaxed linear program. For an LP relaxation and the rounding algorithm designed based on it, the approximation factor f is the ratio between the solution given by the algorithm and the LP objective values. The

relationship among the integral optimal solution, the fractional optimal solution and solution given by the rounding algorithm is illustrated in Figure 2.3. It is true that $f \geq \alpha$ all the time, since the solution given by the algorithm is also feasible solution to the integer program and always greater than or equal to the optimum. So the integrality gap proves a lower bound for the approximation factor.

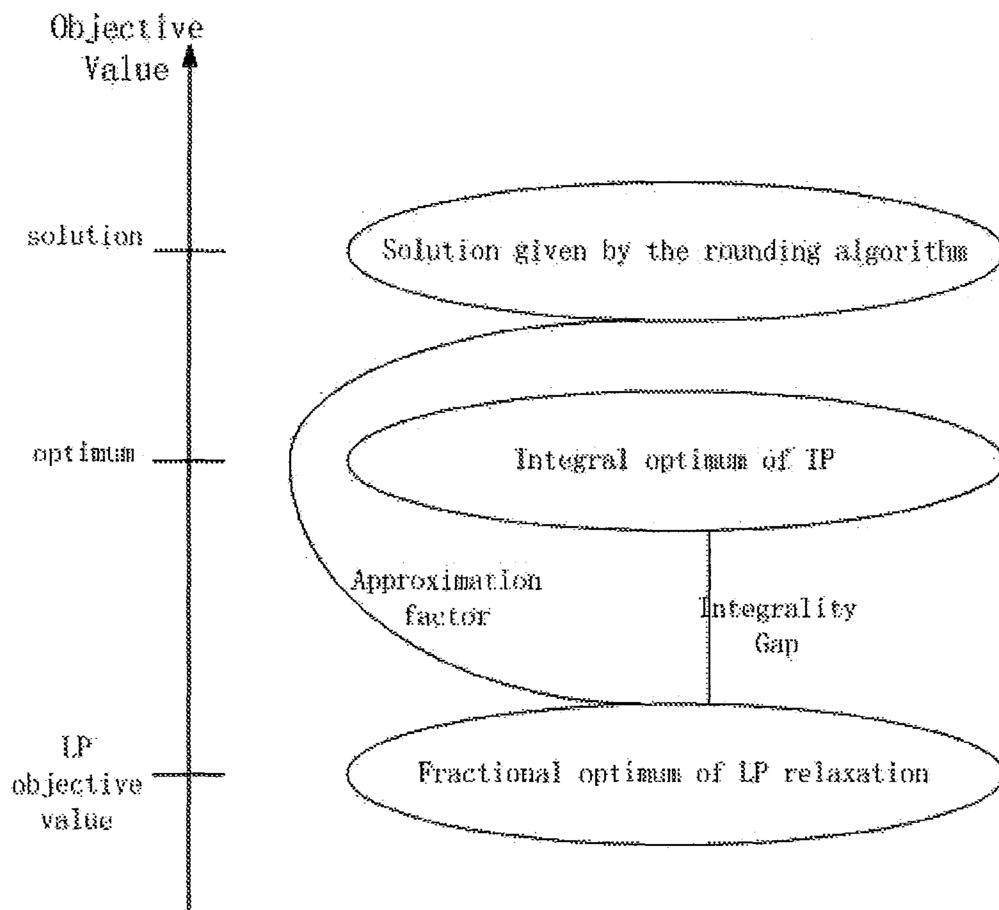


Figure 2.3: The integrality gap and approximation factor

The dual of (HK) is:

$$\begin{aligned} & Maximize \quad 2\sum_{S\subset V} w_s \\ subject \quad to: \quad & \sum_{e\in\delta(S)} w_s \leq c_e, \qquad \forall e \in E \qquad\qquad \text{(DHK)} \\ & w_s \geq 0, \qquad \forall S \subset V, S \neq \emptyset. \end{aligned}$$

Usually we want both the primal and the dual Complementary Slackness Conditions to be satisfied so we can get the exact optimum solution. But for some problems like this, it could be impossible to satisfy both of them at the same time, since the optimal solutions of the linear programs are fractional but what we want are integral solutions. If we can relax one of these two conditions, then we will get an integral solution within a guaranteed approximation factor.

We develop a path form of the Held-Karp formulation. On a graph $G = (V, E)$ with cost vector $\vec{c}$ and two given end points s and t, the path form of Held-Karp formulation (pHK) is:

$$\begin{aligned} & Minimum \quad \sum_{e\in E} c_e x_e \\ subject \quad to: \quad & x(\delta(v)) \geq 1, \qquad \forall v \in \{s,t\} \qquad\qquad \text{(pHK)} \\ & x(\delta(v)) \geq 2, \qquad \forall v \in V - \{s,t\} \\ & x(\delta(S)) \geq 1, \qquad \text{if } |S \cap \{s,t\}| = 1, \forall S \subset V, S \neq \emptyset \\ & x(\delta(S)) \geq 2, \qquad \text{if } |S \cap \{s,t\}| \neq 1, \forall S \subset V, S \neq \emptyset \\ & x_e \geq 0, \qquad \forall e \in E. \end{aligned}$$

If we restrict this linear program to a (0,1)-integer program, i.e., $x_e \in \{0, 1\}, \forall e \in E$, then any optimum solution is a TSP path between nodes s and t, since every node other than s and t has exactly two incident edges, both s and t have only one incident edge, and the solution has only one connected component. So, as a relaxed (0,1)-integer program, the linear program (pHK) is a relaxation of the TSP path.

2.5 Our Linear Program Relaxation for TSP path

We develop our own linear program formulation for the TSP path from node s to node t:

$$
\begin{aligned}
& Minimize \quad \sum_{e \in E} c_e x_e \\
\text{s.t. :}\quad & x(\delta(S)) + y(S) \geq \begin{cases} 2, & \text{if } s \notin S, \forall S \subseteq V,\ S \neq \emptyset \\ 0, & \text{if } s \in S, \forall S \subseteq V,\ S \neq \emptyset \end{cases} \\
& x(\delta(S)) - y(S) \geq \begin{cases} 2, & \text{if } t \notin S, \forall S \subseteq V,\ S \neq \emptyset \\ 0, & \text{if } t \in S, \forall S \subseteq V,\ S \neq \emptyset \end{cases} \\
& y(V) = 0 \\
& x_e \geq 0, \quad \forall e \in E \\
& y(v) \text{ free}, \quad \forall v \in V
\end{aligned}
\tag{ST}
$$

$y(v)$ is a value assigned to node v, and $y(S) = \sum_{v \in S} y(v)$.

The corresponding dual program has a dual variable w_C for each $C \subseteq V$:

$$
\begin{aligned}
& Maximize \quad \sum_{s \notin C} 2w_C + \sum_{t \notin D} 2u_D \\
\text{s.t. :}\quad & \sum_{e \in \delta(C), C \subseteq V} w_C + \sum_{e \in \delta(D), D \subseteq V} u_D \leq c_e, \quad \forall e \in E \\
& \sum_{v \in C, C \subseteq V} w_C - \sum_{v \in D, D \subseteq V} u_D = \alpha, \quad \forall v \in V \\
& w_C \geq 0, \quad \forall C \subseteq V \\
& u_D \geq 0, \quad \forall D \subseteq V \\
& \alpha \quad free
\end{aligned}
$$

In the remainder of this section we will give a proof that our formulation is equivalent to the path form of the Held-Karp formulation, so it is also a relaxation

of the TSP path. And in the next chapter we will develop a heuristic for the TSP problems using this linear program.

Theorem 2.5.1. *Our linear program formulation (ST) is a relaxation of the TSP path.*

The proof of this theorem comes out immediately from the next theorem, since (pHK) is a relaxation of the TSP path.

Theorem 2.5.2. *Our linear program formulation (ST) of the TSP path is equivalent to the path form of the Held-Karp formulation (pHK).*

Proof. We break this theorem down into the following two propositions.

Proposition 2.5.3. *A feasible solution for our formulation is also feasible for the path form of Held-Karp formulation.*

Proof. Given a feasible solution for our formulation, we can eliminate all the y values by adding two constraints together, and prove that it's feasible for the path form of Held-Karp formulation.

1. $\forall v \in V \backslash \{s, t\}$:

$$\begin{cases} x(\delta(v)) + y(v) \geq 2 \\ x(\delta(v)) - y(v) \geq 2 \end{cases}$$

imply that $x(\delta(v)) \geq 2$

2. If $v = s$ (or $v = t$, similarly) :

$$\begin{cases} x(\delta(s)) + y(s) \geq 0 \\ x(\delta(s)) - y(s) \geq 2 \end{cases}$$

imply that $\quad x(\delta(s)) \geq 1$

3. $\quad \forall S \subset V, S \neq \emptyset, s,t \notin S:$

$$\begin{cases} x(\delta(S)) + y(S) \geq 2 \\ x(\delta(S)) - y(S) \geq 2 \end{cases}$$

imply that $\quad x(\delta(S)) \geq 2$

4. $\quad \forall S \subset V, \{s,t\} \subseteq S:$

Let $\bar{S} = V - S$. We know that $x(\delta(\bar{S})) = x(\delta(S))$. From case 3 we know that for $s,t \notin \bar{S}$, $x(\delta(\bar{S})) \geq 2$, so $x(\delta(S)) \geq 2$.

5. $\quad \forall S \subset V, S \neq \emptyset, |S \cap \{s,t\}| = 1: \qquad x(\delta(S)) \geq 1$

Combining the above five cases we've proved that any feasible solution for our relaxation is also feasible for the path form of Held-Karp formulation. □

Proposition 2.5.4. *A feasible solution for the path form of Held-Karp formulation is also feasible for ours.*

Proof. Given a feasible solution for the (pHK) formulation, we set y values to each node as follows:

$$y(s) = -1, \qquad y(t) = 1, \qquad \text{and } y(v) = 0 \text{ for all other nodes } v$$

If $s \notin S$ but $t \in S$, we know that $x(\delta(S)) \geq 1$ and $y(S) = 1$, so $x(\delta(S)) + y(S) \geq 2$; If $s \notin S$ and $t \notin S$, we know that $x(\delta(S)) \geq 2$ and $y(S) = 0$, so $x(\delta(S)) + y(S) \geq 2$. Thus $x(\delta(S)) + y(S) \geq 2$, $\forall S \subseteq V$, $S \neq \emptyset$, $s \notin S$.

If $t \in S$ and $s \in S$, we know that $x(\delta(S)) \geq 2$ and $y(S) = 0$, so $x(\delta(S)) - y(S) \geq 2$; If $t \in S$ but $s \notin S$, we know that $x(\delta(S)) \geq 1$ and $y(S) = 1$, so $x(\delta(S)) - y(S) \geq 0$. Thus $x(\delta(S)) - y(S) \geq 0$, $\forall S \subseteq V$, $S \neq \emptyset$, $t \in S$.

It's easy to check that all the other constraints in our formulation hold, using the same technique we used for proving the above two constraints. Thus any feasible solution for the (pHK) is also feasible for our formulation (ST). □

Propositions 2.5.3 and 2.5.4 prove Theorem 2.5.2. □

2.6 Test Cases for Our LP Relaxation

In this section, we show some simple examples of running our TSP relaxation on some selected graphs and feed those programs to a linear program solver called QSopt (http://www2.isye.gatech.edu/~wcook/qsopt/index.html). We stated in last section that our LP formulation targets to finding TSP paths instead of TSP tours, so we need to assign a starting node s and a terminal node t. After solving (ST) we need to add the edge weights $c_{(s,t)}$ to the objective value. All these graphs obey the triangle inequality, and they are all complete graphs.

Figure 2.4 shows our result on a 14-node graph. All the solid lines indicate edges with $c_e = 1$ and dashed lines indicate ones with $c_e = 2$. Unlike the sub-tour formulation, all the edges in our solution have $x_e = 1$, which means that it finds an integral solution of the optimum weight on this graph. But it is an integral TSP path. The edge (s, t) is not fully paid by the current set of dual variables.

All the nodes are indexed with integers from 1 and indicated as circled numbers. We present moats of dual variables w_C and u_D as polygons, with real numbers on border indicating the value of the dual variables. All the edges not displayed have weight values equal to the shortest paths between their two end points.

Figure 2.5 to 2.8 show four more examples, all with the same naming and presenting rules. In addition, underlined numbers indicate the weights of edges beside them. All the four TSP paths are integral solutions of the optimum objective value. And all the edges (s,t) in these four examples can be fully paid by the current sets of dual variables.

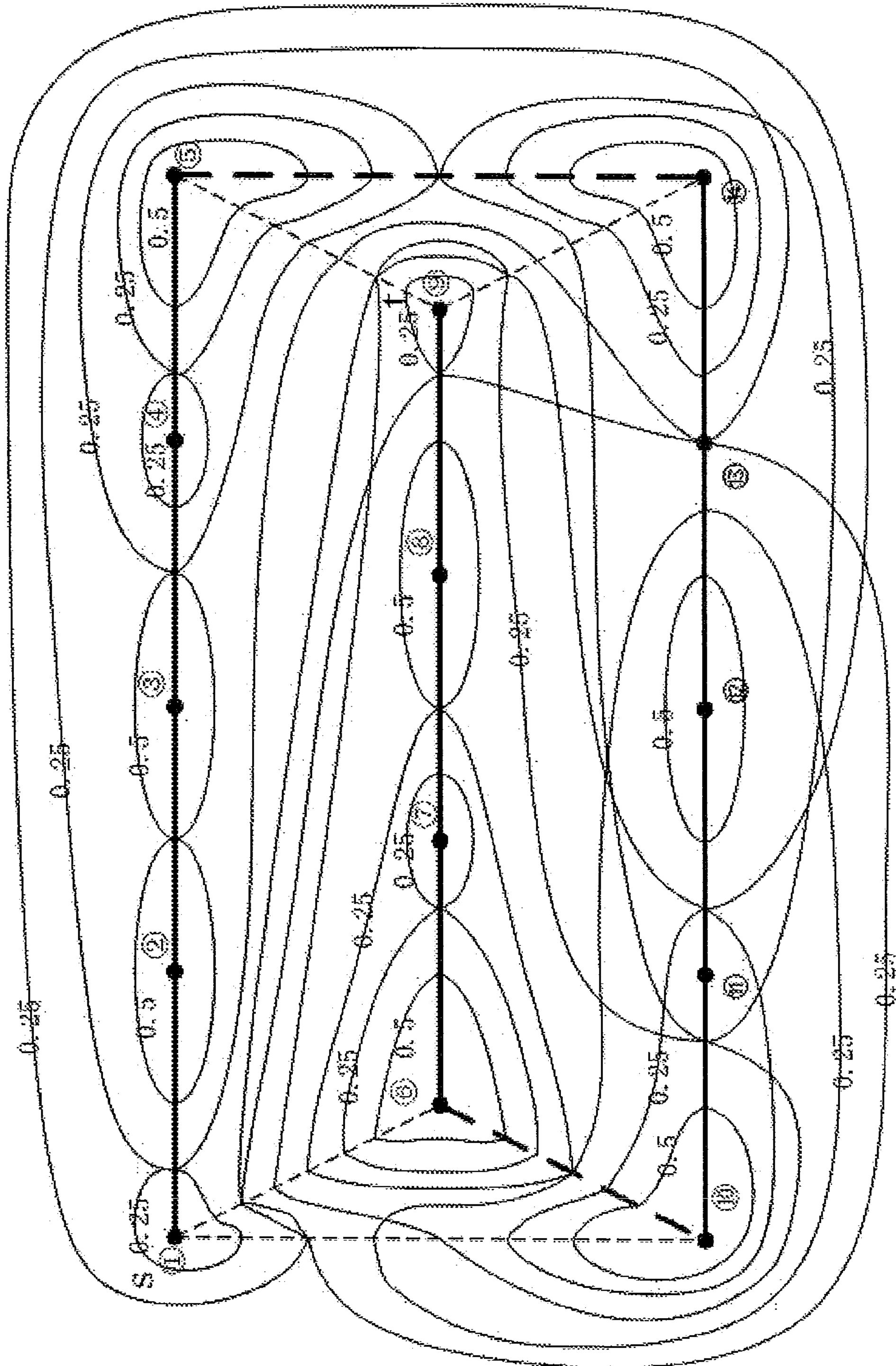


Figure 2.4: Test case 1

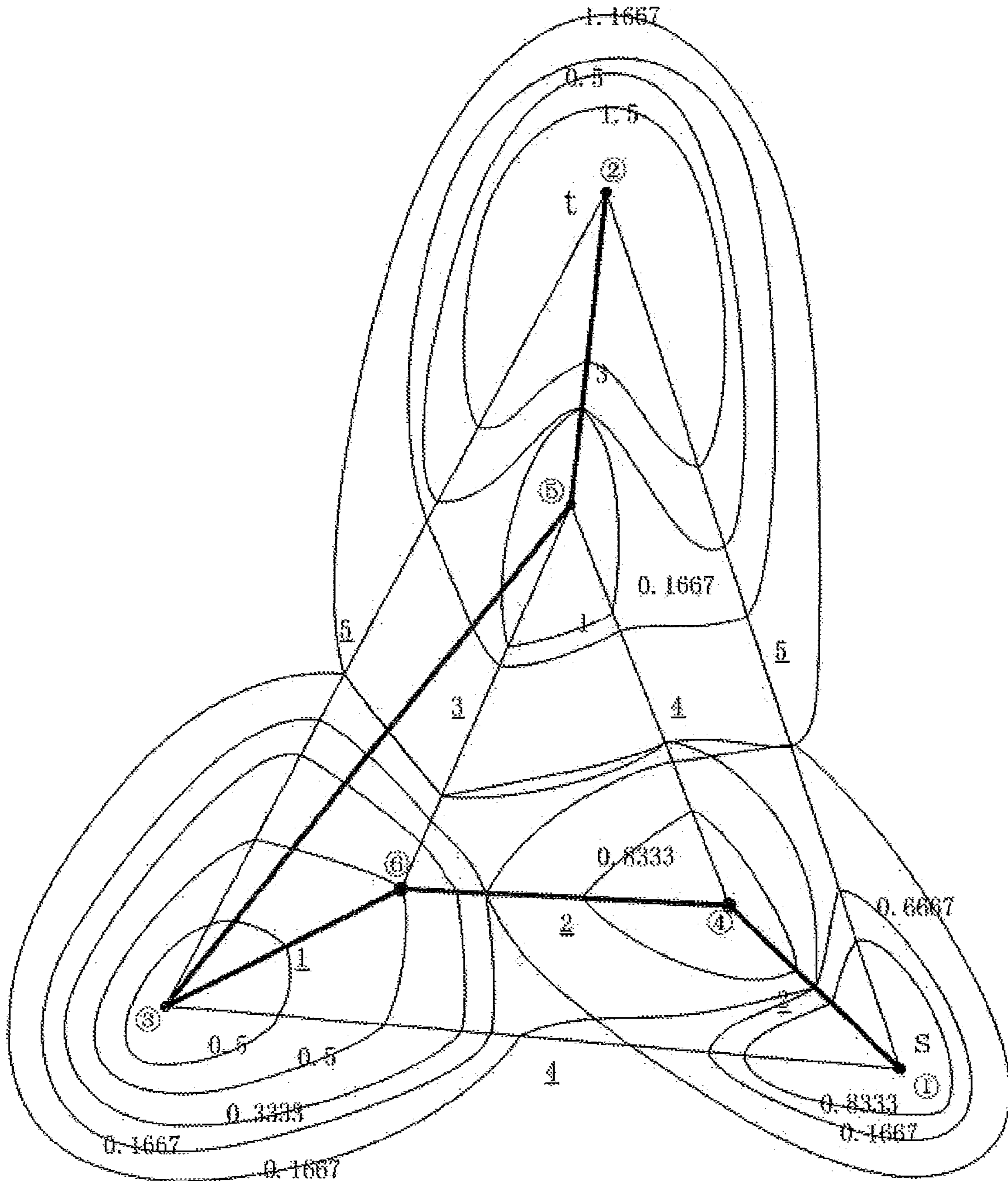


Figure 2.5: Test case 2

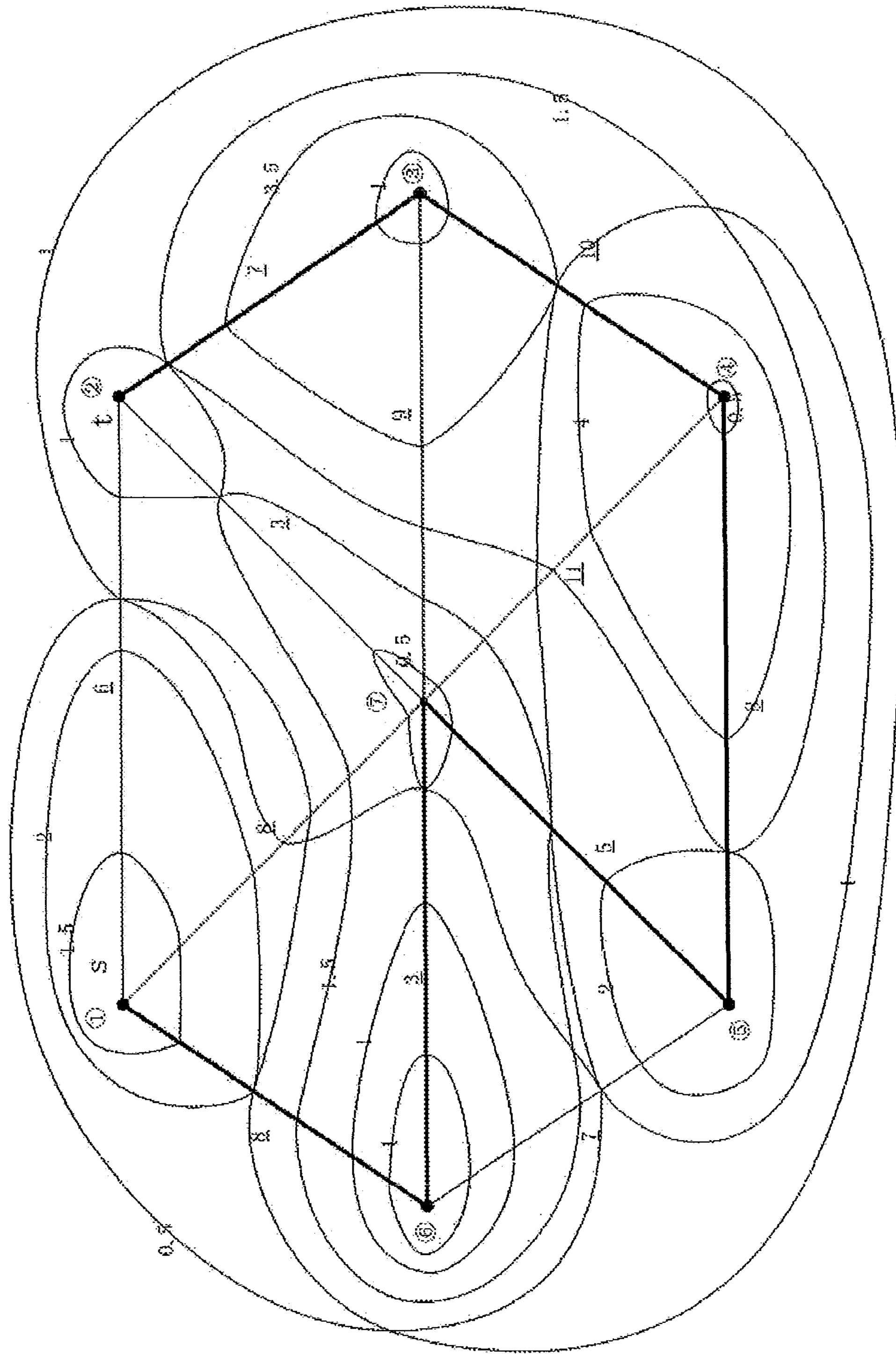


Figure 2.6: Test case 3

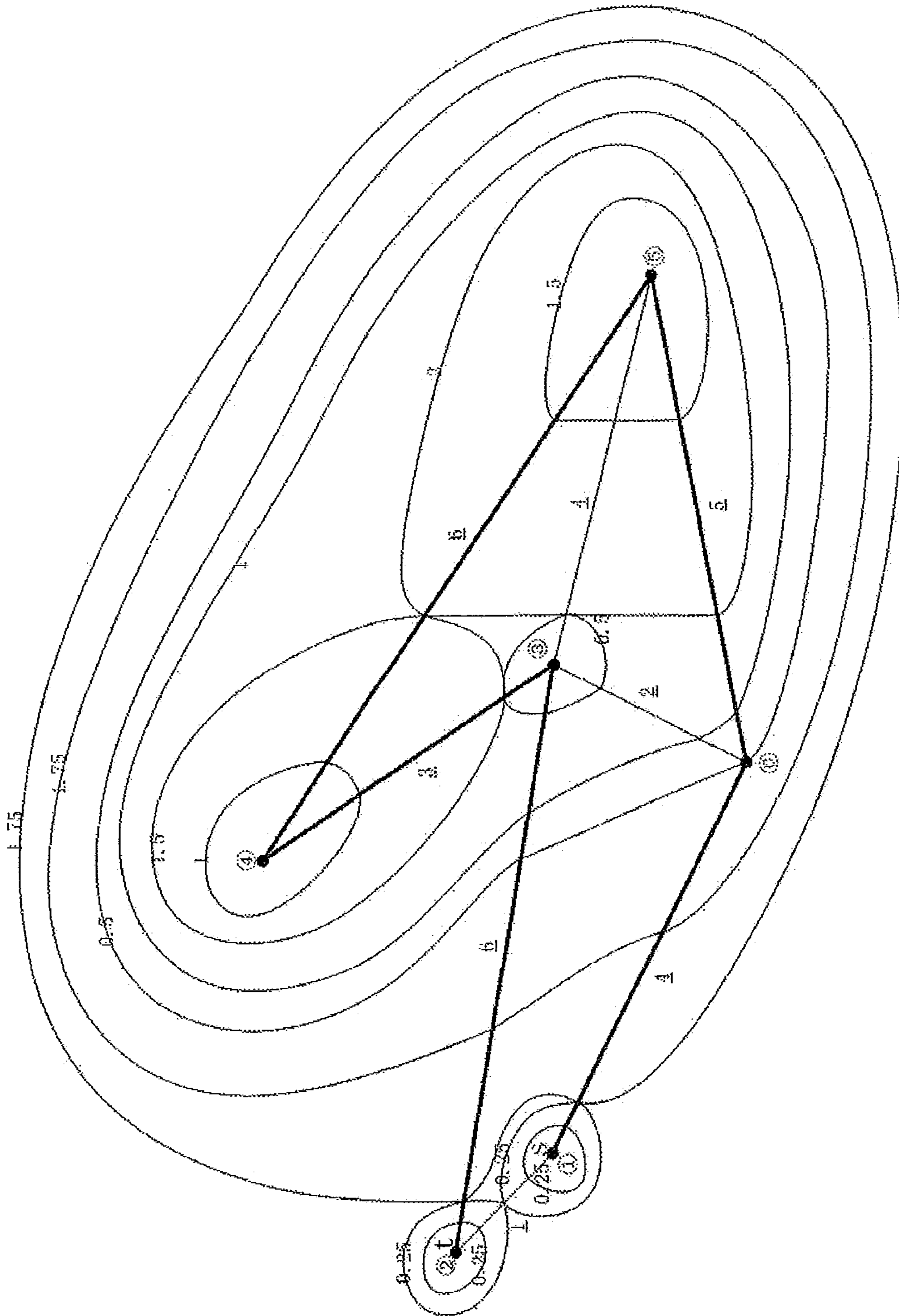


Figure 2.7: Test Case 4

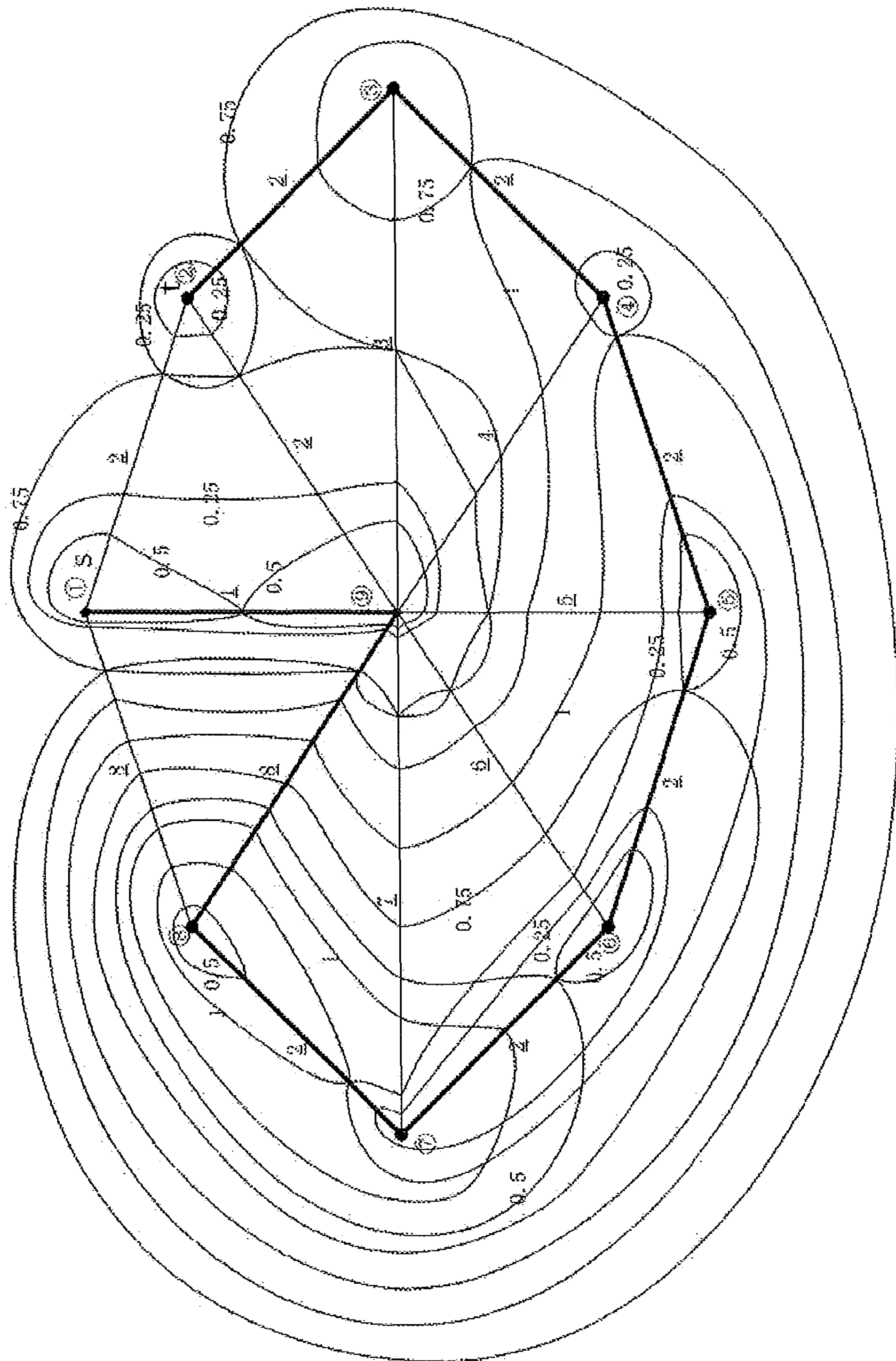


Figure 2.8: Test case 5

Chapter 3

TSP Heuristics

In Chapter 2 we present our linear program formulation for TSP. And in this chapter we develop a heuristic based on our relaxation, trying to restrict the approximation factor to 4/3. First, we recall the LP from last chapter and its dual:

$$
\begin{aligned}
& Minimize \quad \sum_{e \in E} c_e x_e \\
\text{s.t. :} \quad & x(\delta(S)) + y(S) \geq \begin{cases} 2, & \text{if } s \notin S,\ \forall S \subseteq V,\ S \neq \emptyset \\ 0, & \text{if } s \in S,\ \forall S \subseteq V,\ S \neq \emptyset \end{cases} \\
& x(\delta(S)) - y(S) \geq \begin{cases} 2, & \text{if } t \notin S,\ \forall S \subseteq V,\ S \neq \emptyset \\ 0, & \text{if } t \in S,\ \forall S \subseteq V,\ S \neq \emptyset \end{cases} \\
& y(V) = 0 \\
& x_e \geq 0, \quad \forall e \in E \\
& y(v) \text{ free}, \quad \forall v \in V
\end{aligned}
\qquad (\text{ LP})
$$

$$
\begin{aligned}
& Maximize \quad \sum_{s \notin C} 2w_C + \sum_{t \notin D} 2u_D \\
\text{s.t. :} \quad & \sum_{e \in \delta(C), C \subseteq V} w_C + \sum_{e \in \delta(D), D \subseteq V} u_D \leq c_e, \qquad \forall e \in E \qquad \text{(DLP)} \\
& \sum_{v \in C, C \subseteq V} w_C - \sum_{v \in D, D \subseteq V} u_D = \alpha, \qquad \forall v \in V \\
& w_C \geq 0, \qquad \forall C \subseteq V \\
& u_D \geq 0, \qquad \forall D \subseteq V \\
& \alpha \quad free
\end{aligned}
$$

In the description of the heuristic we adopt the terminology defined in Chapter 2, i.e., moats, tight edges, etc. In Section 3.1 we show how the matching idea and algorithms could help with building a TSP path. During this construction unwanted situations may occur. For example, some nodes may have degree greater than two, or tight edges may close a cycle which doesn't cover all nodes. These make the solution infeasible, so we deal with them in Sections 3.2 and 3.3. Then in Section 3.5 we analyze this problem from a different aspect, using matroids.

3.1 Finding augmenting paths

Before we actually move on to the specific algorithm and heuristic, it may be helpful to review the precise description of the Traveling Salesman problems first: Given a graph $G = (V, E)$ and a weight vector $\vec{c}$, the goal is to find a tour that visits every node exactly once and has the minimal possible total weight. When we try to solve a problem, a very natural idea is to relate it to some similar well-studied ones, and see if we can borrow some ideas and methods from them. So in the first section of this chapter we show how we use matching algorithms to help with our own problem.

To make things simpler and more intuitive, we need some restrictions and preparations for the graphs we work on. In this thesis we always work on symmetric graphs satisfying the triangle inequality. And instead of $G = (V, E)$ and its weight vector $\vec{c}$, we work on its metric completion $G' = (V, E')$ - we keep all the nodes unchanged, and there is an edge e'_{vu} between every pair of nodes v and u in G'. For the new weight vector $\vec{c'}$, if $e_{vu} \in E$, then $c'_{vu} = c_{vu}$, otherwise $c'_{vu} = p_{vu}$ where p_{vu} is the length of the shortest path between v and u in G.

Instead of building a tour directly, we try to build a path that covers all the nodes. And in the end add the edge between the two end points to the path, to make the solution a tour. We can do this because we have proved in Section 2.1 that TSP paths and TSP tours are equivalent.

In Edmonds' blossom algorithm, the essential structure, which is kept all the time, is an augmenting tree that has alternative matched and unmatched edges along every branch. If we can restrict this augmenting tree to an augmenting path, then it will be what we need when a perfect matching is found - a path that covers all the nodes.

In the other direction, if we find two completely different perfect matchings who share no common edge at all, then combining these two matchings together gives a collection of cycles that cover all the nodes: Consider two different matchings M and M', and any node v. Assume v is matched with node u in M, then in M' it must be matched with another node w, $u \neq w$. Thus any node v has two different incident edges (v, u) and (v, w), one as an incoming edge and the other as an outgoing edge. If we can further restrict the combined solution to have only one connected component, it will be a tour that visits all the nodes and return to the starting point.

This lead us to the idea of two alternating matchings. In our LP formulation there are two given nodes, s meaning the starting point of the path and t meaning the ending point. We want to build two maximal matchings, one called s-matching that covers node s but doesn't cover node t, and the other called t-matching that covers t but not s. For graphs having even number of nodes we need to add a dummy node t' and an edge $e_{tt'}$ with cost $c_{tt'} = 0$, since otherwise any maximal matching in such graphs cover both s and t at the same time. If we alternatively grow these two matchings - s-matching and t-matching - using Edmonds' blossom idea, and when growing one matching, treat the matched edges of the other one as augmenting edges, then hopefully we will get the path we want. We say that a node gets *discovered* when it's incident to some tight edge for the first time.

We use Figure 3.1 as an example to describe the essential process of building such a path. Given any graph satisfying the triangle inequality, get its metric completion, namely $G = (V, E)$, and weight vector $\vec{c}$ on it. Here we have $V = \{s, v, u\}$, $E = \{c_{sv}, c_{su}, c_{uv}\}$, $c_{sv} = 5$, $c_{vu} = 3$ and $c_{su} = 8$, as illustrated in Figure 3.1(a). The heuristic chooses arbitrarily a node as its initial root, and in this example we choose s. We adopt the word moat from last chapter, representing the dual variables w_C or u_D in (DLP). In order to keep the constraints $\sum_{v \in C, C \subseteq V} w_C - \sum_{v \in D, D \subseteq V} u_D = \alpha, \forall v \in V$ true all the time, when we say that we grow a moat W_v around some node v, we grow half of it as w_C and the other half as u_D, i.e., $W_v = w_{\{v\}} + u_{\{v\}}$ and $w_{\{v\}} = u_{\{v\}}$. Later in this section we will grow moats containing a set of nodes S, and when we grow W_S we make sure that $W_S = w_S + u_S, w_S = u_S$. In our heuristic we want the moats to cut the path once or twice: once if they contain exactly one of the two nodes s

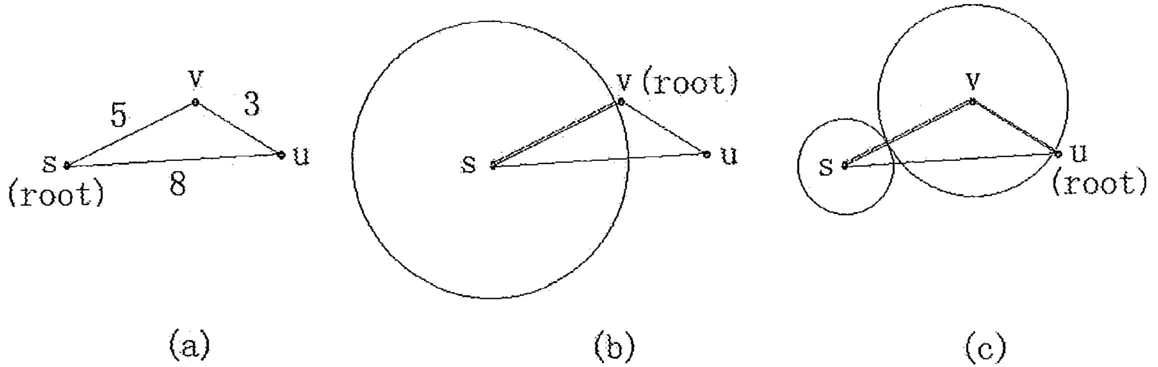


Figure 3.1: grow path

and t, twice if they contain none. It is because of the objective function of our DLP: *Maximize* $\sum_{s \notin C} 2w_C + \sum_{t \notin D} 2u_D$. As stated, we don't distinguish w_C and u_C, so in this case the objective function of DLP is equivalent to *Maximize* $\sum_{s \notin C} W_C + \sum_{t \notin C} W_C$, which means moats will be counted once if they contain only one of s and t, and twice if they contain none. We start by growing a moat W_s around the root s. We continue growing this moat until some edge becomes tight. In this example it is e_{sv}. We match this tight edge as an s-matching, and make node v the new root. Now we are about to grow the t-matching from the current root v.

As with Edmonds' blossom algorithm, first the tight edges (which are also the already-matched ones in our heuristic) are to be augmented to form a path, as the double-lined edge e_{sv} in Figure 2.1(b). Then grow the moats around the root and all nodes that are at an even hop distance from the root. Recall that in Edmonds' algorithm we use the word *distance* to indicate the number of edges along the path from a node on the augmenting tree to the root, so if there are even(or odd) number

of edges on this path we say this node has even(or odd, respectively) distance from the root. And we keep two sets of nodes: node set B that contains tree nodes which have even distance from the root, and set A contains odd distance tree nodes. So here what we grow are the B-nodes in Edmonds' blossom algorithm. Similarly, in order not to make any edge over-tight, we also shrink all the moats around odd-distance nodes(which are A-nodes in Edmonds') at the same time and by the same amount. In our example it means growing W_v and shrinking W_s, as in Figure 2.1(c). This process goes on until the edge e_{vu} becomes tight. Since we are now building the t-matching, the tight edge e_{vu} will be t-matched. Then we make node u the new root, and we are ready to build the s-matching again from root u. In this example a TSP path has already been found. If we have larger graphs containing more nodes, we repeat this process of building s-matching and then t-matching, until the path hopefully covers all nodes.

If at some time a moat shrinks to zero but no new edge becomes tight, as in Figure 3.2(a), we group the whole augmenting path and treat it as the new root. Then we continue to grow its moat(which is the current root) until some new edge becomes tight, as in Figure 3.2(b). (In Figure 3.2(b) we draw the big moat as an ellipse rather than a circle only to save some paper space. Note that in Figure 3.2 and in later sections we only draw the "original" edges while omitting the metric completing ones.)

We succeed if we can get a path of two maximal matchings by repeating this process. But usually we can't, since this process runs in polynomial time but TSP is NP-Complete. In Figure 3.3, we have already grown the path p_{yz} and the current root is node z. Moat W_v has shrunk to zero, so we grow a moat containing the whole

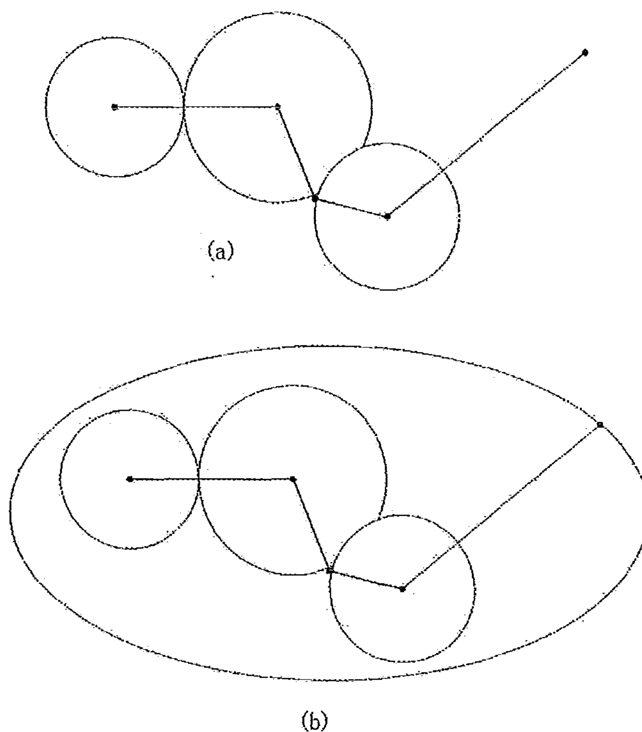


Figure 3.2: make big group when necessary

path p_{yz}, to expand the path from root z. But as the growing continues, edge e_{wx} becomes tight. We cannot simply match this edge e_{wx} because node w is already two-matched (both s-matched and t-matched). We call w a higher-degree problem node. In the next section we see how these higher-degree problem nodes are treated.

3.2 Solving higher-degree problem nodes

As shown in the last section, we are trying to construct a TSP path, consisting of alternating s-matchings and t-matchings. So hopefully every node (except the starting point s and the current root) on this path has two matched edges incident to

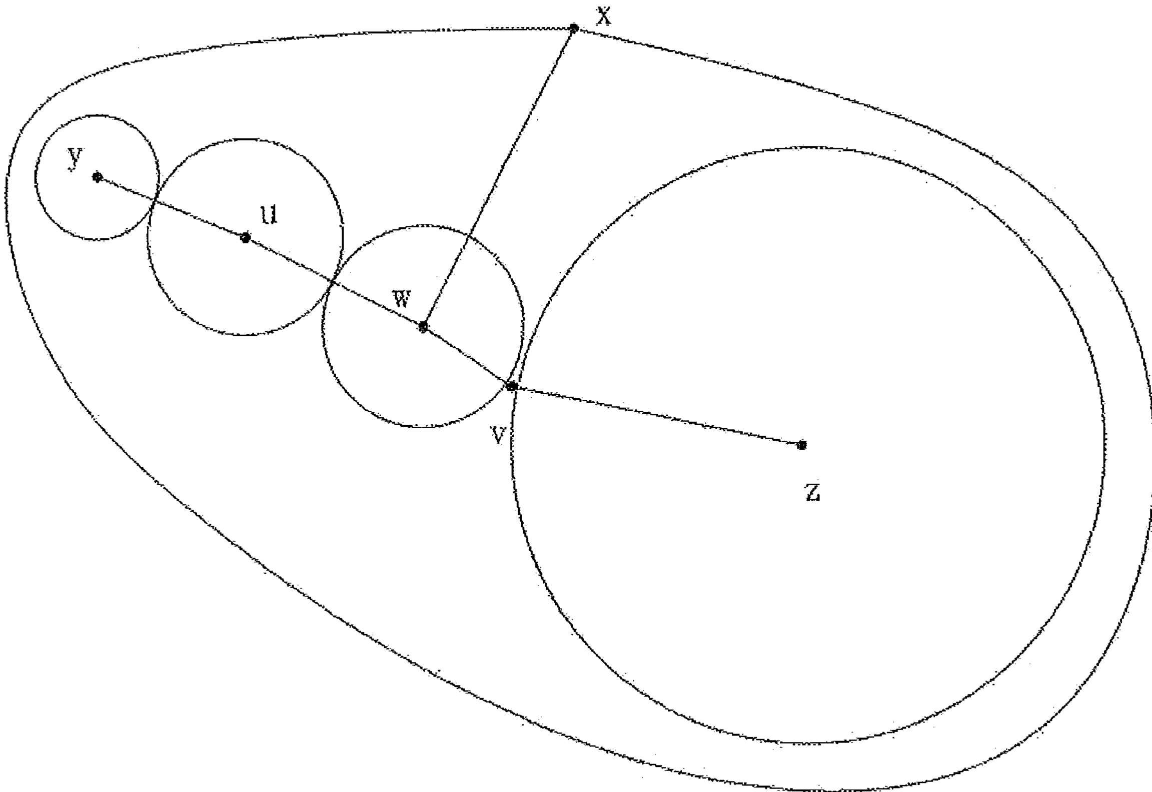


Figure 3.3: higher-degree problem node

it: one edge is in s-matching and the other is in t-matching. We use the word *degree* to represent the number of matched edges incident to a node. So we want that all nodes on this path have degree 2, except s and the current root that are currently of degree 1. But as the growing-and-shrinking process goes on, some 2-matched node may be connected to an unmatched node by a tight edge before the current root gets 2-matched, as node x is incident to w in Figure 3.3, and we cannot simply match this edge e_{wx} since then w will have degree 3. We call this a higher-degree problem and w a higher-degree problem node.

While solving this problem, we want to include the node x into the path we've

already built, so we not only solve a problem but also further expand our path. Consider node w, it has two matched edges e_{uw} and e_{vw} incident to it already. If we want to match the edge e_{wx}, we have to unmatch e_{uw} or e_{vw} first, to keep the degree of node w. We also want to keep the path we've already built - or at least the most part of it. So we try to make the edge e_{ux} (or e_{vx}) tight, unmatch e_{uw} (or e_{vw} respectively) and then match e_{ux} (or e_{vx} respectively) and e_{xw}. To make edge e_{ux} or e_{vx} tight, we would like to grow the moats around node x, u, v and shrink the moat around node w. So at first the moat W_x grows and the moat $W_{p_{yz}}$ shrinks. When $W_{p_{yz}}$ shrinks to zero, we get Figure 3.4, and we continue to grow the moats W_x, W_u, W_v and shrink W_w, as indicated by the arrows. (Arrows pointing to center mean shrinking and pointing out mean growing).

If edge e_{ux} becomes tight first, then we can match is as the same type (s-matching or t-matching) as edge e_{uw}, match e_{xw} as the other type, unmatch edge e_{uw} and switch types for all the edges along the path from node w to the current root z. By switching type for an edge, we mean that if it was an s-matched edge then it becomes t-matched, and if it was a t-matched edge it becomes s-matched. Similarly, if edge e_{vx} becomes tight first, we match e_{wx} as the same type as e_{wv}, match e_{xv} as the other type, unmatch e_{wv} and switch types for all the edges along the path from node v to the current root z.

Another possibility is that during the growing-and-shrinking process edge e_{uv} becomes tight before e_{ux} or e_{vx}. This tight edge prevents the current moats from growing or shrinking any more, so neither e_{ux} nor e_{vx} can be tight. Since this edge e_{uv} can be very short compared to e_{ux} or e_{vx}, we cannot afford to include it in our solution because then the approximation factor can approach 2 in the worst case.

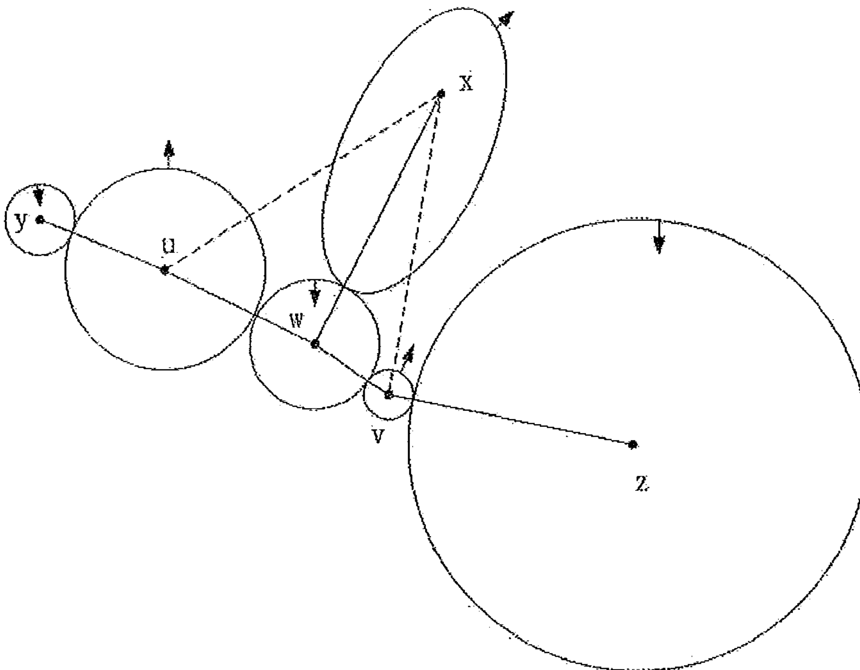


Figure 3.4: Shrinking higher degree moats

In section 3.1 we used the node set containing the whole augmenting path when the process of growing-and-shrinking got stuck by some moat shrinking to zero. Here the rule is a little bit different: we group together most nodes, matched and undiscovered, except a few special ones on which we are working. As in Figure 3.5, we group all the nodes together except nodes w and x. This big group will be cut so that the growing-and-shrinking process can resume. We keep growing $W_{\{yuvz\}}$ and the moat W_x, shrinking the moat W_w, until e_{ux} or e_{vx} becomes tight and the problem is fixed.

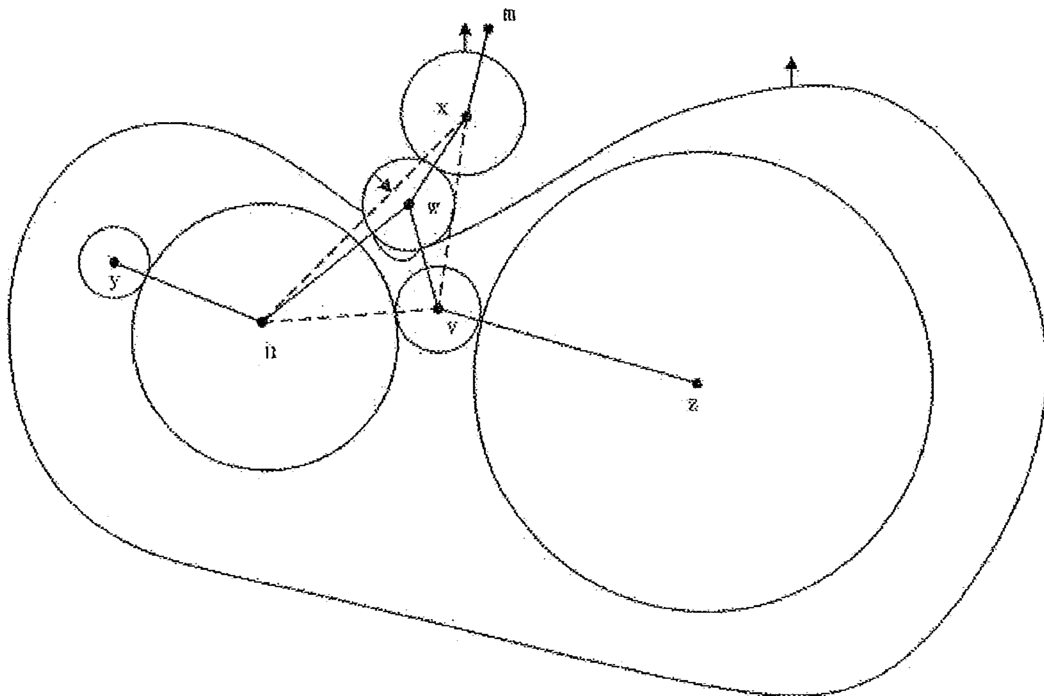


Figure 3.5: Making big group when tight triangle occurs

Another unwanted possibility is that edges between node x and other undiscovered nodes become tight before e_{ux}, e_{vx} or e_{uv}, (e.g., edge e_{mx} in Figure 3.5). This tight edge will interrupt the growing-and-shrinking process.

Node x is actually a potential higher-degree problem node, just like node w in the previous stage. So following the same strategy, we would like to shrink the moat around x, and start to grow the moat of node m. In order to deal with the augmenting path, the grouping idea will be used, since we don't want the moat W_w

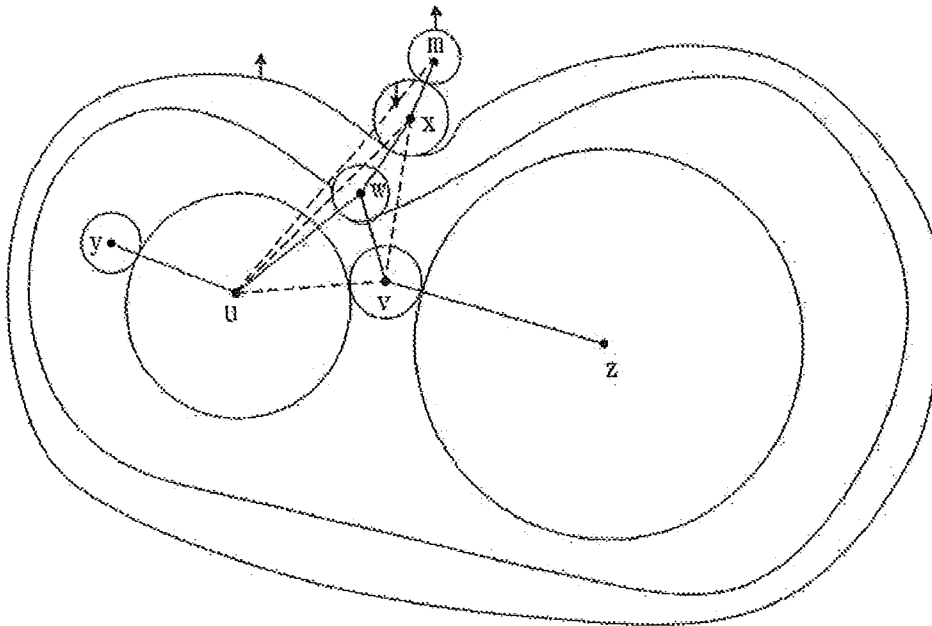


Figure 3.6: Making big group when new node is discovered

to grow. As illustrated in Figure 3.6, this time the group will contain all nodes except x and the newly discovered m.

The essential idea behind this process is to connect u or v by a tight edge to the "farthest" node on this "problem-node path", which in our example means path p_{wm}, and the "farthest" indicate the node-wise distance from the augmenting path we've already built. So in Figure 3.6 the farthest is node m, and we are trying to make edge e_{um} or e_{vm} tight. If we achieve this goal, say we make edge e_{um} tight, then

we can augment this "problem-node path" by matching e_{um} and all the edges along path p_{mw}, and then unmatching edge e_{uw} gives us an augmenting path $p_{yumxuvz}$ as we want.

We define an operation called *flipping*. When we say that we flip a moat W_S, we mean that first we set $W_{\bar{S}} := W_S$ for $\bar{S} = V - S$, and then $W_S := 0$.

We establish two criteria to decide if a step of the heuristic improves the solution: (1) after this step either of the s-matching or the t-matching augments; (2) or, the total dual value increases after this step. These criteria help deciding when the heuristic should terminate. We don't terminate until all nodes are discovered and augmented, so there exists no further improvement from the first criterion. But it is much more difficult to tell if the current set of duals could grow into a better dual set with greater total value. So we don't have a good criterion to tell when the heuristic should terminate. We leave it as a open problem.

The heuristic is stated as follow:

1.	Given a complete graph $G = (V, E)$ and the weight vector $\vec{c}$, choose arbitrarily an initial root s. Set both the s-matching and the t-matching empty.
2.	Begin with s. Grow W_s until some incident edges become tight. Pick one arbitrarily and include it in the s-matching. Make the newly discovered and 1-matched node the new root r.
3.	Augment the alternating s-matching and t-matching as the current augmenting path. Decide which matching is going to be grown: if the last matched edge was s-matched then we are going to grow the t-matching, otherwise if we just grew the t-matching then we are going to grow the s-matching.

4. Grow the root moat W_r. In order not to make any edges over-tight or any matching edges untight, shrink the moats which have odd distance from the root and grow the even-distance ones, all by the same amount, until one of the following happens:

   case 1: An edge between the root and an unmatched node becomes tight: Match this edge as the type decided in step 3 — s-matching or t-matching. Make the newly discovered and 1-matched node the new root, and go to step 3.

   case 2: A single node moat shrinks to zero: Group the whole augmenting path together and treat it as the new root. Go to step 4.

   case 3: A group moat shrinks to zero: Now all nodes inside this group moat are exposed and can be operated on. Go to step 4.

   case 4: An edge between a 2-matched node w and an unmatched node x becomes tight: (see Figure 3.3) Shrink the moats around the 3-degree node w and grow its 3 neighbors x, u and v, until some edge becomes tight: (see Figure 3.4)

   case (1): e_{uv} becomes tight: Group all the nodes except w and x. Continue to grow this big group and W_x, and shrink W_w, until e_{ux} or e_{vx} becomes tight. Go to the following case (2).

   case (2): e_{ux} or e_{vx} becomes tight: If e_{ux} becomes tight, unmatch e_{uw} and match e_{ux} as the same type as e_{uw}, s-matching or t-matching. Match e_{wx} as the other type, and switch types all along the path from w to the current root. If e_{vx} becomes tight, unmatch e_{vw}, match e_{wx} as the same type as e_{vw}, match e_{vx} as the other type, and switch types along the path from v to root. Go to step 4.

case (3): An edge between x and an undiscovered node m becomes tight:

Grow the farthest (farthest from the original higher-degree problem node) moat (W_m in Figure 3.6), shrink the one next to it (W_x in Figure 3.6) and grow a moat around all the other nodes, which are grouped together.

(i) If some edge between m and an undiscovered node n becomes tight, go to case (3), treating node n as m before and node m as x before.

(ii) If an edge between m and a 2-matched node on the augmenting path becomes tight, then a tight-edge cycle occurs. We discuss this situation in Section 3.3.

(iii) If W_x shrinks to zero, flip the group moat, so the new flipped group moat will contain node x and m.

Then if the original higher-degree problem node(w in Figure 3.6) still has group moat around it, go to case (3);

Otherwise, if the original higher-degree problem node w has only one single node moat around it, as in Figure 3.7(a), we shrink this moat W_w, grow the group moats $W_{\{xm\}}$ and $W_{\{yuvz\}}$, until W_w shrinks to zero. Then flip the moat $W_{\{yuvz\}}$ to let it contain nodes w, x and m, as in Figure 3.7(b). Match u with the farthest node (m in Figure 3.7(b)) as the same type as e_{uw}, and switch types along the path from m to the current root. Go to step 4.

case (4): Any edge between two 2-matched nodes on the augmenting path becomes tight:
We discuss this situation in Section 3.3.

case 5: Tight edges close any cycle:
We discuss this situation in Section 3.3.

case 6: All nodes have been discovered:
As an open problem, we need further criteria to decide when we should terminate.

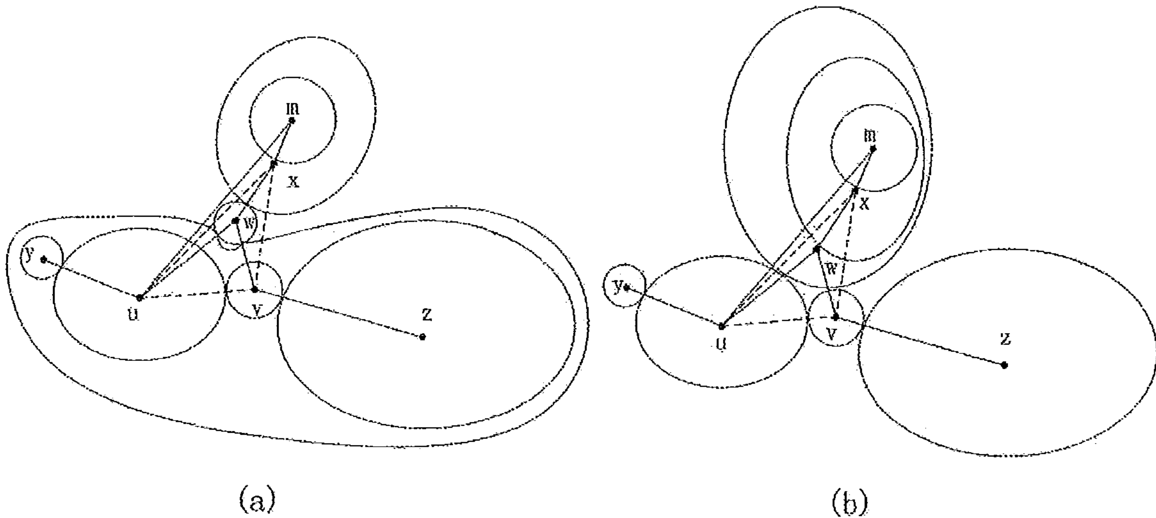


Figure 3.7: Solved Higher Degree Problem Nodes

3.3 When tight-edge cycles occur

In Section 3.2 we mentioned the situations in which tight-edge cycles occur as case 5 of the heuristic. In this section we discuss some of the possibilities. We don't have proof that all the possibilities are covered in this section. We only present what we

have.

From Figure 3.5, 3.6 and 3.7 we can see that sometimes higher-degree problem nodes may extend to a really long path. If we repeat the process in case 4.(3) of the heuristic we can finally solve all of them, as long as the path doesn't get back to any already two-matched node on our augmenting path and closes a cycle. But since we grow the farthest moat and the group moat together (e.g. W_x and $W_{\{yuvz\}}$ in Figure 3.5, or W_m and $W_{\{yuwvz\}}$ in Figure 3.6), it is possible that edges cut by these two moats become tight. As illustrated in Figure 3.8, $W_{\{yuwvz\}}$ and W_m grow together and make the edge e_{mz} tight, which prevents W_x from shrinking.

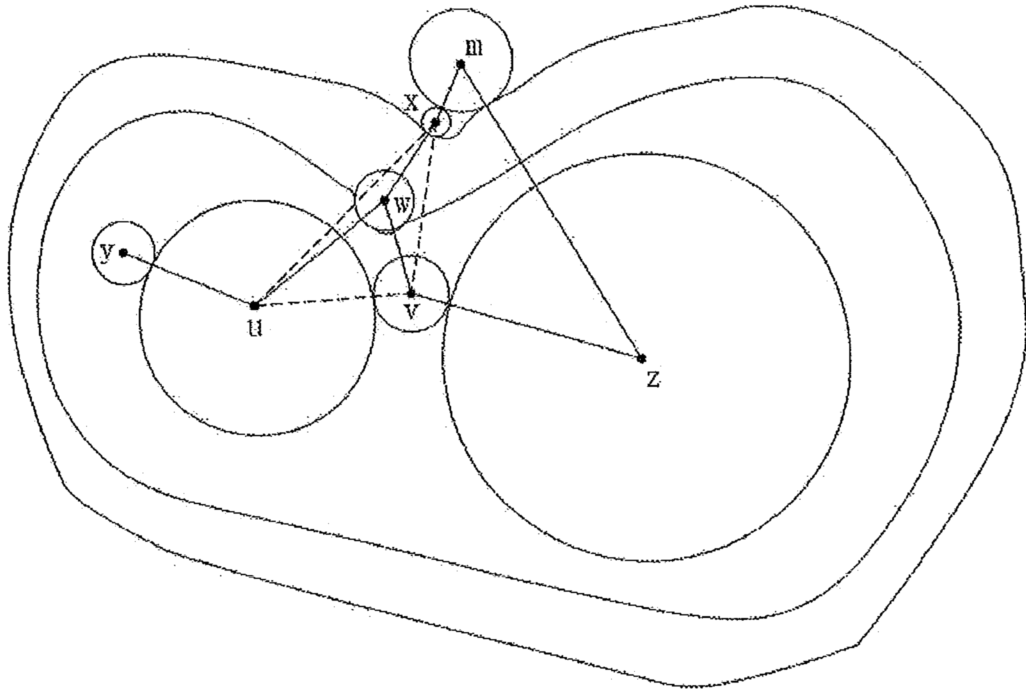


Figure 3.8: Tight edges close a cycle

The node z now is actually a higher-degree problem node, since it was already two-matched and now incident to node m. To follow the same strategy on solving higher-degree problem nodes, we would like to firstly flip $W_{\{yuwvz\}}$ and then $W_{\{yuvz\}}$, so this problem node can be exposed. Then the next step would preferably shrink the moat to which this problem node belongs and grow all its neighboring moats. But if we check carefully, we may find it impossible to do so in some situations. Since we want to shrink this problem node z, the flipped $W_{\{wxm\}} = W_{\overline{\{yuvz\}}}$ needs to grow and moats W_u and W_v to shrink. This will make both edges e_{uv} and e_{vz} untight and the augmenting path broken into disconnected parts.

Other situations may also create tight-edge cycles of different kinds. A comparably simple case is presented in Figure 3.9:

In Figure 3.9 we are rooted at node q, and the solid-edge path is the augmenting path that we've already built. Assume that there are still some nodes left undiscovered. As we can see, since we are growing the moat around the root q, the group moat $W_{\{suvwxy\}}$ has to be growing at the same time. Then at some point moats W_q and $W_{\{suvwxy\}}$ make edge e_{wq} tight, close a tight-edge cycle and stop the root's moat growing.

In section 3.1, when, while building a path, some moat shrinks to zero and stops the root from growing, we make a new moat to contain the whole path. So adopting the same idea here we would like to start growing the moat which contains the whole path from s to q (which means it contains the cycle C_{wxyzq}). Then growing $W_{\{suvwxyzq\}}$ grows the moat around the root, and hopefully this will lead us to a new matched edge from root q.

There is another less simple situation in which tight edge cycle may occur.

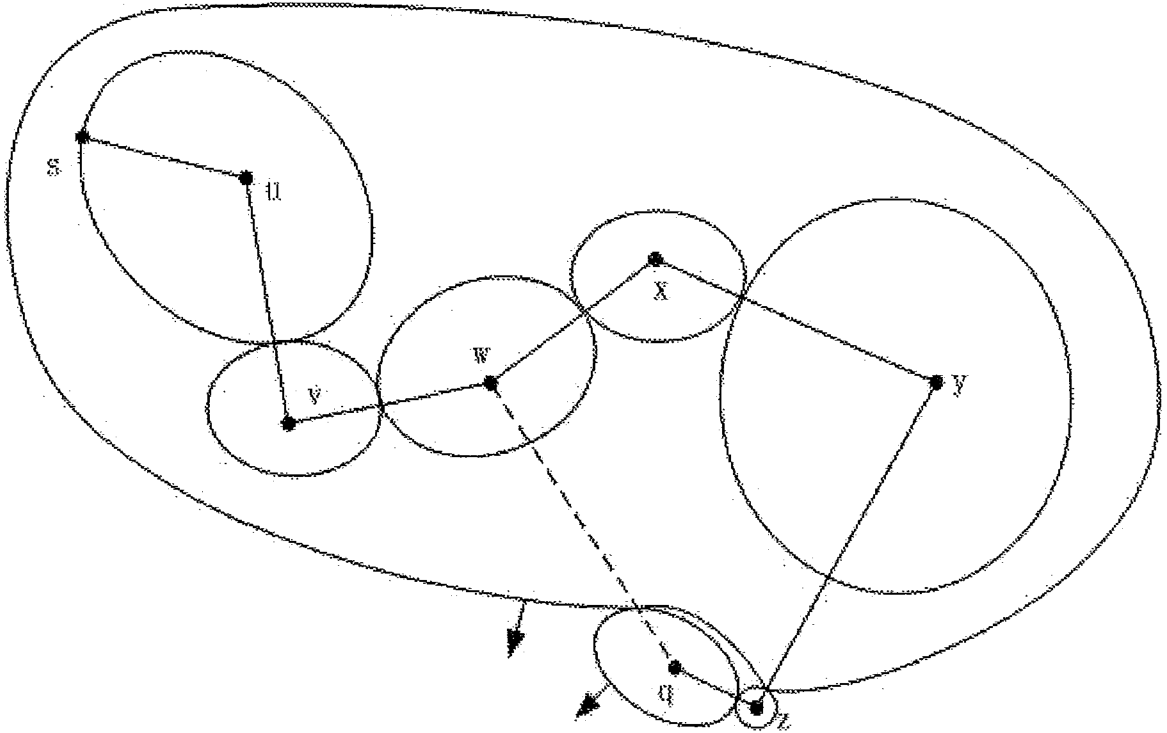


Figure 3.9: Another case of tight edge cycle

Consider the example in Figure 3.10, (we omit most nodes and present only the part of path where the problem occurs). In the example, w is the original problem node, edge e_{vx} becomes tight before e_{vz} and e_{xz} during the shrinking of moat W_w, so we make a moat containing every one except node w and z, and flip this group moat after W_w shrinks to zero. Now we can match the tight edge e_{vz} and the higher degree problem is solved. Then we should get back to the process of growing the moat around the root. But from this situation we can't tell if a node, say node y, should grow or shrink responding to the growing of the root's moat. If the growing of the current root's moat leads W_y to grow, then W_y, W_u and $W_{\{w,z\}}$ will grow, and both W_v and W_x will shrink. This is not a problem since e_{vx} has been unmatched and we

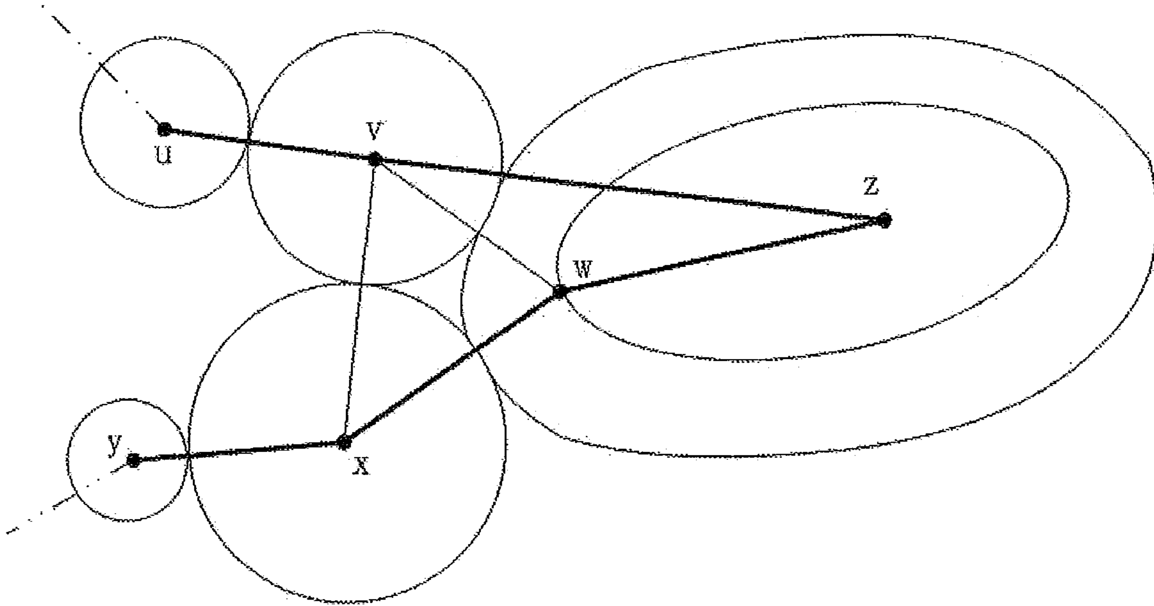


Figure 3.10: Tight edge cycle after solving higher degree problem node

don't care if it becomes untight. But if node y has even distance from the current root, then W_y, W_u and $W_{\{w,z\}}$ will shrink, and both W_v and W_x will grow, and we are in trouble since edge e_{vx} can not be over-tight. In this case we grow the moat containing the whole path, so that we can preserve the path structure we've already built and at the same time enable the augmenting at the current root.

But there are situations in which we can not simply group the whole path to solve the tight edge cycle problems. Consider the situation in Figure 3.11:

When we are trying to solve the problem node y by shrinking W_y and growing W_x and W_z, moats W_v, W_q and other moats on the path have to grow together. This could make some edge like e_{vq} tight and create a tight cycle, which stops the growing and shrinking process. The previous grouping of the whole path doesn't work here, since if we group the whole path into one moat, it would include the problem node

y, and its growth would make the problem even worse.

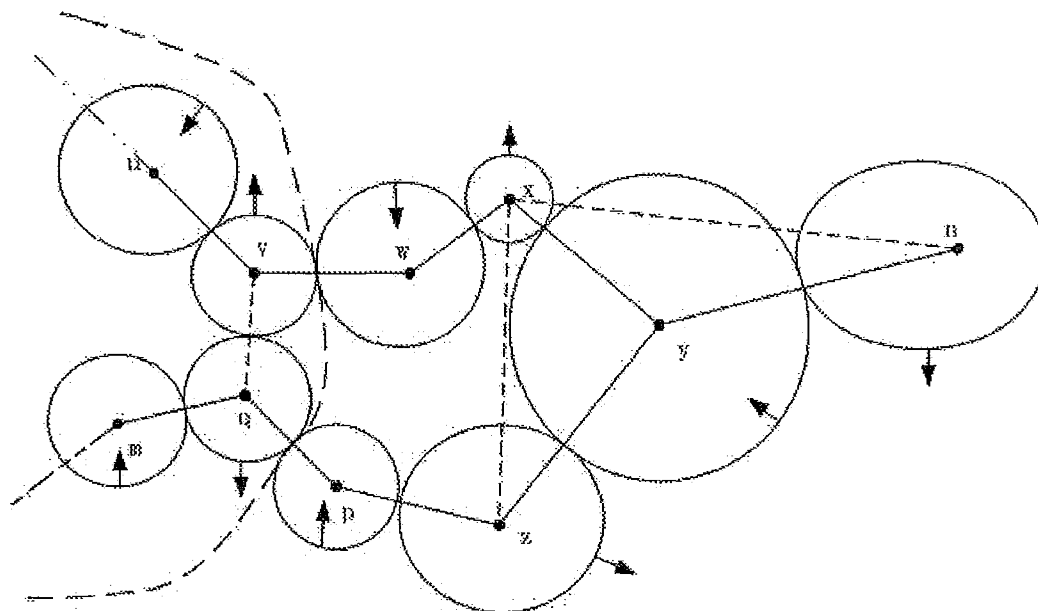


Figure 3.11: General case of tight edge cycle

So we need a new kind of grouping. We want the problem node and its neighbors to be exposed, and others may be grouped. Here we let the group cover node v, u and all those along the path from u to s, as well as q, m and all those along the path from m to the current root, together with all the undiscovered nodes, as indicated by the dashed moat in Figure 3.11. Now it's possible to continue shrinking the problem node y, and instead of v, u growing together, the group will grow. After solving the higher-degree problem on node y and augmenting the path to node n, we simply flip the dashed moat and let it be around node w, x, n, y, z and p. It is easy to see that this group is fine because it cuts the current path exactly twice.

Till now we've seen several kinds of grouping: when some moat on the path

shrinks to zero; or new node is discovered during the process of solving higher degree problem nodes; or the root meets a two-matched node through a tight edge; or two nodes on different side of a problem node get connected by a tight edge. In all these situations there is one question we need to answer: how do we treat the undiscovered nodes? Should we include them in the group, or leave them outside?

To answer this question, we examine what is meant by containing a node inside a group. When we cover some nodes with a group moat, we should be prepared that these nodes may not be exposed again. So what we include in this kind of group should be a nice structure and potentially part of the final solution. But we do have an operation (flipping) which basically release the node set of a group moat and cover its complementary set. So, when we do a grouping knowing we will flip it later, e.g., in the process of solving higher degree problem nodes, as in Figure 3.5, we should include all the undiscovered nodes inside the group $W_{\{yuvz\}}$ since after flipping they will be exposed again. And if we do a grouping knowing that we will leave it as it is, e.g., when some moat shrinks to zero, we should pack only the augmenting path inside the group moat, because at least for now it is a nice structure where every node has degree of 2, and we hope it will be part of the final TSP path.

Before closing this section, we discuss the approximation factor of our heuristic. As we've seen, if a tight cycle like the one in Figure 3.8 or Figure 3.11 occurs, we don't have any good method to break it and we are forced to keep it until some later stage. At the termination of the heuristic, when we have already discovered and augmented all the nodes, it will possibly look like Figure 3.12: solid straight lines indicate the matched edges and wavy lines indicate matched paths.

The final solution has to be a tour, so we need to give up some accuracy and

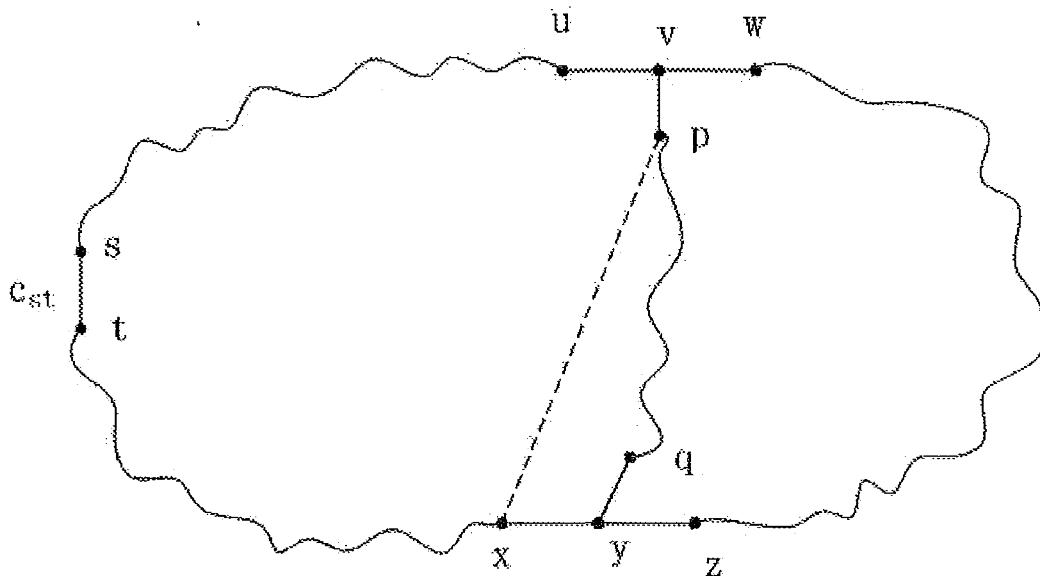


Figure 3.12: four-thirds approximation factor

add an extra edge which is not fully-paid by the duals. We can add the edge e_{px} (we may choose other extra edges like e_{zp} or e_{uq} etc. to complete the tour, but the idea and the approximation factor are the same). Since we always work on the metric completion of the graph, by the triangle inequality, we know that the length of e_{px} is no greater than the length of path $p_{\{pusx\}}$, path $p_{\{pwyx\}}$ or path $p_{\{pqyx\}}$. That is,

$$c_{(p,x)} \leq min(p_{\{pusx\}}, p_{\{pwyx\}}, p_{\{pqyx\}})$$

All edge costs are non-negative, so we know that $p_{\{vus\}} + p_{\{tx\}} \leq p_{\{vusx\}} \leq p_{\{pusx\}}$, $p_{\{vwzy\}} \leq p_{\{pwyx\}}$ and $p_{\{pqy\}} \leq p_{\{pqyx\}}$. By substituting $p_{\{vusx\}}$, $p_{\{vwzy\}}$ and $p_{\{pqy\}}$ for $p_{\{pusx\}}$, $p_{\{pqyx\}}$ and $p_{\{pqyx\}}$ respectively, we have

$$c_{(p,x)} \leq min(p_{\{vusx\}}, p_{\{vwzy\}}, p_{\{pqy\}})$$

The three paths together build up the path from x to p, and the duals pay for

this path plus part of edge (x,p), so we know that

$$c_{(p,x)} \leq \frac{1}{3}(p_{\{vusx\}} + p_{\{vwzy\}} + p_{\{pqy\}}) \leq \frac{1}{3} * \text{Total dual optimum} \tag{3.3.1}$$

$$c_{st} + c_{tx} + c_{suvwzyqp} \leq c_{st} + \text{Total dual optimum} \tag{3.3.2}$$

$$(3.3.1), (3.3.2) \Longrightarrow \textit{solution} \leq c_{st} + \frac{4}{3} * \text{Total dual optimum} \;\; \leq \frac{4}{3} OPT_{tour}$$

The approximation factor of $\frac{4}{3}$ applies when the paths of the proof have been formed. As we stated earlier in this section, we don't have proof that all possibilities are covered, so there may be other situations in which approximation factors of greater values apply.

3.4 Test Cases on This Heuristic

In this section we show the running of our heuristic on a variation of the well-known Held-Karp 4/3 example(as seen in [3]). We describe in Section 2.4 and illustrate in Figure 2.2 the Held-Karp 4/3 example, all the displayed horizontal edges have $c_e = 1$, the vertical and oblique edges have $c_e = 2$, and all edges which are not shown have $c_{(u,v)} = \sum_{e \in \hat{p}_{uv}} c_e$, where $\hat{p}_{uv}$ is the shortest path between node u and v. We add two nodes s and t, with $c_{(s,a)} = 1$, $c_{(t,a)} = 1$, $c_{(s,t)} = 1$ and $c_{(u,v)} = \sum_{e \in \hat{p}_{uv}} c_e$ for all $u \in \{s,t\}, v \in V - \{s,t,a\}$.

As stated in Section 2.1, we run our algorithm (which is only a heuristic for now) on every pair of nodes s and t, add the edge (s,t) with cost $c_{(s,t)}$ to each path, and select the minimal cost tour. But in the heuristic we presented in previous sections, we don't specify the ending point t. This problem can be fixed by treating t

as matched all the time before every node $v \in V - \{s, t\}$ is discovered and augmented. Suppose j is the last discovered node in node set $V - \{s, t\}$, then t will be matched to j at termination.

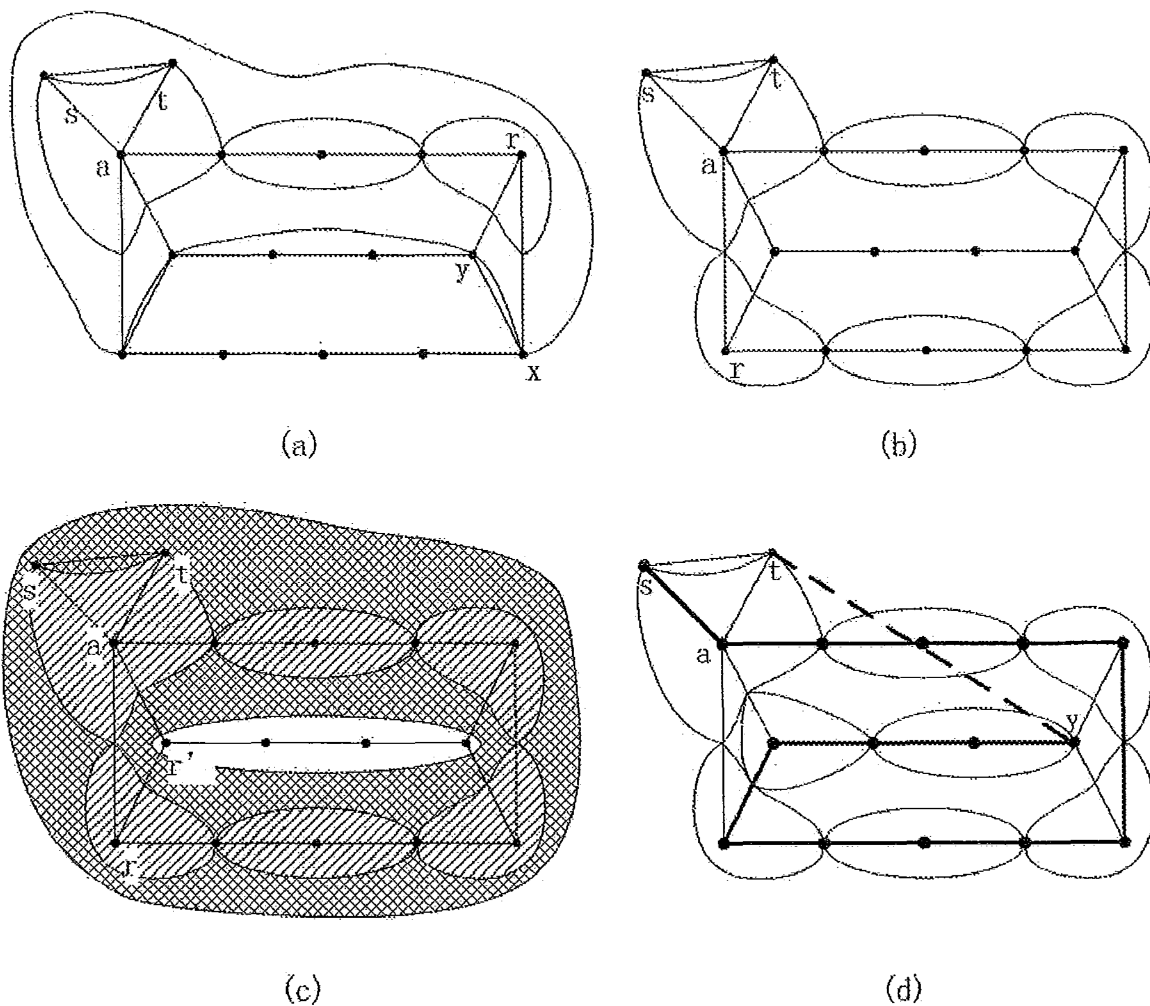
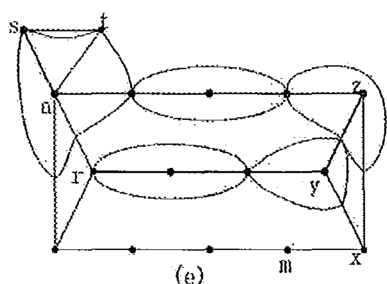


Figure 3.13: Running our heuristic on a variation of Held-Karp 4/3 example

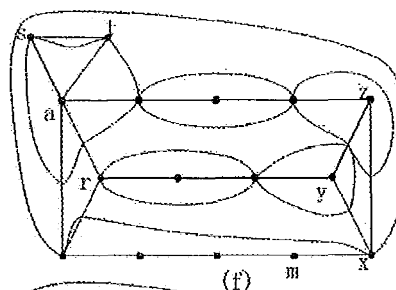
We start by growing a moat around node s. Edges (s, t) and (s, a) become tight at the same time, but since t is the end point, we match (s, a) at this step. Then, we grow the path until r becomes the root as in Figure 3.13(a). We can't continue

growing the root since some node moats have already shrunk to zero. So we create the group moat shown in Figure 3.13(a). Then, among the four newly discovered nodes, we select node x and root there. We continue to grow the path until the situation in Figure 3.13(b) happens. Node r is the root, but we cannot grow its moat any more since a tight edge cycle has formed. As discussed in the previous section, in this situation we can group the whole path we've already grown, as in Figure 3.13(c), as indicated by the shaded ring. Grow this group moat until it discovers r', then we can get back to the basic path-growing process, until we finally get to the situation in Figure 3.13(d). All the matched edges are indicated by bold lines, the dash line (y, t) is not fully paid and brings in the inaccuracy, and (s, t) is the last edge that completes the tour. The total dual value for TSP path p_{st} is 16, and the cost of this path is 22. If we generate this example to graphs containing $3k + 4$ nodes, which are very similar to the family of graphs defined in Section 2.4 with the extra two nodes s and t, the integrality gap for TSP paths should be $\frac{4k+6}{3k+4}$, and the limit value when k goes to infinite is $\frac{4}{3}$. After adding edge (s, t), the total dual value remains the same, but the cost of the tour increases to 23.

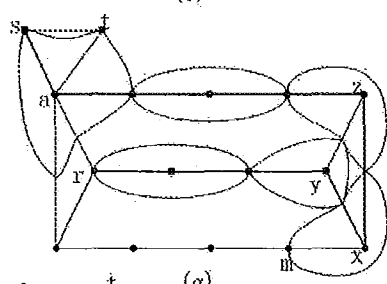
Suppose that when we are in the situation of Figure 3.13(a) we select node y. We extend the path until reaching the situation in Figure 3.13(e). Now we are unable to grow the moat around root r since some moat on the path has already shrunk to zero. So we group the path and grow the group moat as shown in Figure 3.13(f), until the node x is discovered, which causes a higher-degree problem since node y is already two-matched. Directed by the heuristic, we grow the moat around x and shrink the group one, until get to the situation in Figure 3.13(g). The tightness of edge (z, x) solves the higher-degree problem, but now node m is discovered and makes node x



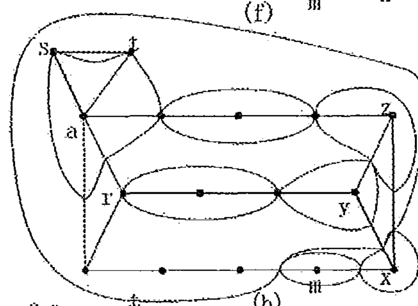
(e)



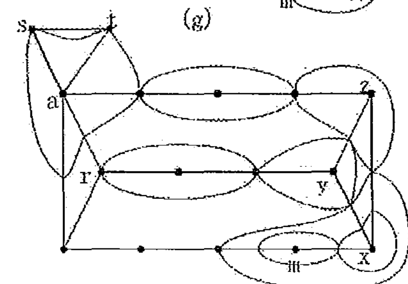
(f)



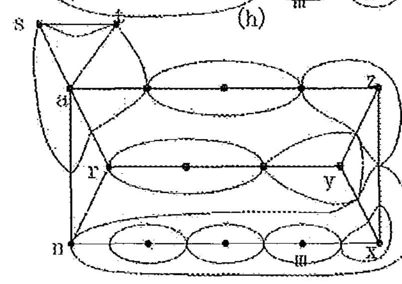
(g)



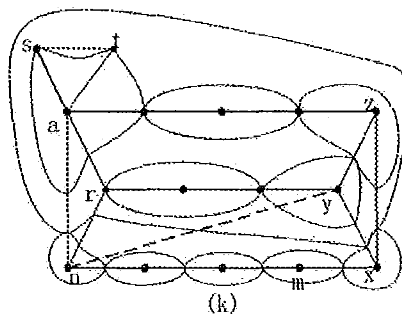
(h)



(i)



(j)



(k)

a new problem. As introduced in Section 3.2, in this situation we grow the moat W_m, shrink W_x and group all the others, as shown in Figure 3.13(h). When some edge becomes tight, we flip the group moat to let it contain m and x, as in Figure 3.13(i), and a new node is discovered again. We repeat this process until reaching the situation shown in Figure 3.13(j). Now the newest discovered node is n, and we want to grow its moat until situation in Figure 3.13(k) occurs. If we see node s as two-matched (since one of its degree has to be given to the ending node of the path), the tightness of edge e_{sn} creates another higher-degree problem and also a tight-edge cycle which prevents us from further progress.

3.5 A Matroid approach

In this section we try to model the TSP problems in the field of Matroid Theory. Matroid Theory was developed from linear algebraic theory at first, and now it is a powerful tool in combinatorial optimization field. We start this section by introducing the definitions and terminologies,then we state in two different theories that matching and 2-factor are both matroids. In the end we discuss the approach to modeling TSP as intersection of two matroids.

Definition 3.5.1. A matroid $M = (E, F)$ is a pair in which E is a finite set of elements, and F is a family of subsets of E, and the following axioms hold:

(1) $\emptyset \in F$, and if a subset S is in F, then all proper subsets of S are also in F;

(2) If two subsets S_p and S_{p+1} are in F and they have cardinalities of p and $p+1$ respectively, then there exist an element $e \in S_{p+1} - S_p$ such that $\{e\} \cup S_p \in F$.

The basic and probably simplest matroid is the matrix matroid. The elements

in it are the columns of a matrix, and F contains all subsets of linearly independent columns. Obviously $\emptyset \in E$, and any subsets of an independent set are independent, so the first axiom holds for sure. And for two independent sets S_p, S_{p+1} of p and $p+1$ columns respectively, S_p together with some column from S_{p+1} create a new independent set of $p+1$ columns. (See [15]).

A subset in F is called an *independent set* of the matroid $M = (E, F)$. And any subset of E which is not independent is called *dependent.* A maximal independent subset of $A \subseteq E$ is called a *base* of A, and for $A \subseteq E$ the *rank* $r(A)$ is the cardinality of any base of A. A base of E is called a *base of the matroid* M, and the rank of it is the *rank of the matroid*, because all bases of M have the same rank.

There are several variations for the second axiom from which we can select the most suitable one when we try to proof that a structure is a matroid. We state three of these equivalent variations in the following lemma.

Lemma 3.5.1. *As the second axiom for the matroid definition, the following three statements are equivalent:*

(2) If two independent sets S_p and S_{p+1} are in F and they have cardinalities of p and $p+1$ respectively, then there exist an element $e \in S_{p+1} - S_p$ such that $\{e\} \cup S_p \in F$;

(2)' If two independent sets S and T are in F and $|S| > |T|$, then there exists an element $e \in S - T$ such that $\{e\} \cup T \in F$;

(2)" For any subset A of E, all bases of A have the same cardinality.

Readers are referred to [15] and [23] for proofs.

One of the well-known matroids is the matching matroid, which is also closely related to our problem. And another one is the graphic matroid.

Theorem 3.5.2. *Let $G = (V, E)$ be a graph. For any subset $N \subseteq V$, if F is the family of all subsets $S \subseteq N$ such that all the nodes in S can be covered by a matching of G, then $M = (N, F)$ is a matroid. We call these structures $M = (N, F)$ matching matroids, and $M = (V, F)$ the matching matroid of graph G.*

Theorem 3.5.3. *Let $G = (V, E)$ be a graph. For any subset $A \subseteq E$, A is independent if the edges in A form an acyclic subgraph. This is called a graphic matroid. And a base of the graphic matroid is a spanning tree.*

Proofs of the above two theorems can be found in [15].

We introduced the b-matching problems and the b-factor problems in Section 1.1 and 2.2. In this section we show that 2-factor problems also have a nice matroid structure. Instead of direct checking the matroid definitions, following [24] we construct a new graph reducing the 2-factor problem to the perfect matching problem. Since we've already seen that perfect matching is a matroid, this construction proves that 2-factor is also a matroid.

Let $G = (V, E)$ be a graph, and for every node $v \in V$, $d_G(v)$ denotes the degree of v in G. For each node $v \in V$, we construct two sets of nodes in the new graph $G' = (V', E')$, set $A(v)$ and $B(v)$:

$$\begin{aligned} A(v) &= \{v_w : w \text{ is adjacent to v by some edge in } E.\} \\ B(v) &= \{v_i : i \in \{1, 2, \ldots, d_G(v) - 2\}\} \end{aligned}$$

$B(v)$ is possibly empty. And V' is the union of all $A(v)$'s and $B(v)$'s:

$$V' = \bigcup \{A(v) \cup B(v) : \forall v \in V\}$$

The new edge set E' consists of two kinds of edges: for each $v \in V$, there is an edge between each pair of v_w and v_i; for each $e_{uv} \in E$, edge $e'_{u_v v_u}$ is in E':

$$E' = \bigcup \{e'_{v_w v_i} : v_w \in A(v), v_i \in B(v), \forall v \in V\} \bigcup \{e'_{u_v v_u} : e_{uv} \in E\}$$

Theorem 3.5.4. *G has a 2-factor if and only if G' has a perfect matching.*

Theorem 3.5.4 was proven by W. T. Tutte in [26]. Another proof and further results on the b-factor problem can be found in [17]. We give an example of this translation of graph in Figure 3.14. Bold edges in Figure 3.14(a) is a 2-factor of the original graph, and bold edges in Figure 3.14(b) is a perfect matching of the translated graph.

Now we return to the TSP. In Chapter 1 we introduced that TSP can be modeled as intersection of three matroids: First we extend the n nodes graph by adding a new node and n edges that connect this new node to all the original ones. The first matroid is a graphic matroid on this new graph, the second matroid is a partition matroid having independent sets containing at most one edge ingoing to any given node, and the third matroid is similar to the second but contains at most one edge outgoing from any given node. But it's known that finding the minimum weight intersection of three matroids is NP-Complete. So we need to reduce the problem of approximating TSP to two matroids. Our initial inspiration comes from the fact that TSP tours are restricted 2-factor solutions, or further, TSP tours are 2-factor solutions that have only one connected component. When considering the connected component, probably the simplest structure is the spanning tree. We have already proved that 2-factors are matroids, and it's also known that spanning trees are matroids. So we are led to the idea of taking minimum weight intersection of

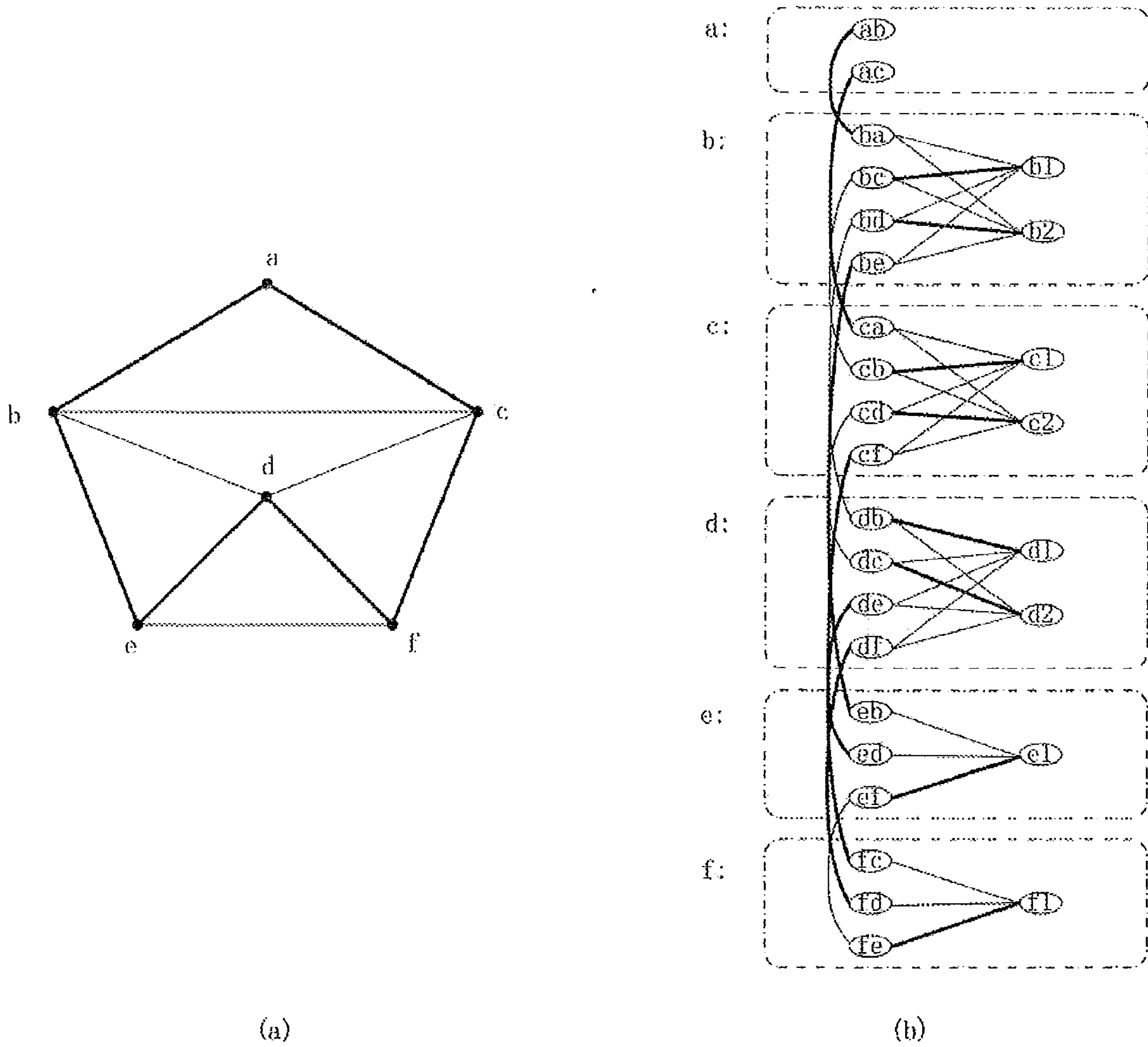


Figure 3.14: Example of the graph translation

these two matroids — 2-factor matroid and spanning tree matroid.

Our difficulty comes from the fact that those two matroids — 2-factor matroid and spanning tree matroid — are over different universes. The universe of a spanning tree matroid is the edge set of the graph, while the universe of the 2-factor matroid is the edge set of the extended graph of Theorem 3.5.4, which is different from the edge set of the original graph. We could not find a way to "merge" these two different universes. This was expected, since if we could do it we would be able to solve TSP in polynomial time using a matroid intersection algorithm [15]. But we could try to "reconcile" the two universes into a single one, paying a small approximation factor that translates into an approximation factor for TSP. We leave this approach as an open problem.

Chapter 4

Conclusions and future work

This thesis provides an LP relaxation for finding the TSP paths, which is inspired by an LP formulation for finding perfect matchings, and proven to provide solutions as good as the path form of Held-Karp relaxation. To show the performance of our LP relaxation, we give five examples, including the well-known Held-Karp 4/3 example([3]). Unlike the Held-Karp relaxation, our relaxation doesn't produce an integrality gap on this example. Then based on our relaxation we build a heuristic finding approximate TSP solutions, with issues as cover of all possibilities and termination conditions left for further research.

We prove that, if an approximation factor α is guaranteed for the TSP paths, then using a certain transforming rule the TSP tours also have the same approximation factor. This ensures our heuristic, which finds TSP paths, also has the same performance on finding TSP tours.

In our heuristic we keep two alternating matchings, s-matching and t-matching. For now they don't provide any contribution to the quality of solutions — we can treat both of them indiscriminately as matched. But we keep this two-alternating-matching structure in our heuristic, since it is where our motivation comes from, and

we believe it has the potential influence on refining our heuristic to be an algorithm.

The biggest deficiency of our heuristic is the lack of termination conditions. To define the improvement of a solution, we establish two criteria: the augmenting of ether s-matching or t-matching, or the increasing of the total dual value. When all nodes are discovered and augmented, we can tell that no further augmenting of matchings are possible. But we haven't found a sufficient condition to say that no further increasing is possible for the current set of duals. Plus, our heuristic may fall into an endless loop: solving one higher-degree problem causes another one on a different node, but solving the second one makes the first appear again. We need some detecting methods to decide when to halt this loop. And since we don't have proof that all possible kinds of tight-edge cycles are covered, there may exist an endless loop in other situations. So the termination conditions rise to be the most important open problem left in this thesis.

On the other hand, as an attempt to model the TSP as a minimal weighted intersection of two matroids, we do not have a method to combine the universe of the 2-factor matroid and the universe of the spanning tree matroid together. How to reconcile these two different universes by paying an approximation factor becomes another open problem left to future research.

Bibliography

[1] C. Berge, *Two theorems in graph theory*, Proceedings of the National Academy of Science of the U.S. **43** (1957), 842–844.

[2] G. Birkhoff, *Tres observaciones sobre el algebra lineal*, Revista Facvltad de ciencias Exactas, Puras y Aplicadas Universidad Nacional de Tucuman, Serie A (1946), 147–151.

[3] R. Carr and S. Vempala, *On the held-karp relaxation for the asymmetric and symmetric traveling salesman problems*, Sandia National Labs, Albuquerque and Department of Mathematics, MIT, 2003.

[4] N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Symposium on New Directions and Recent Results in Algorithms and Complexity (1976), 441.

[5] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, *Combinatorial optimization*, John Wiley and Sons, Chichester/New York/Brisbane/Weinheim/Singapore/Toronto, 1998.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, second edition*, The MIT Press, Massachusetts, 2001.

[7] G. Dantzig, R. Fulkerson, and S. Johnson, *Solution of a large-scale traveling-salesman problem*, Operations Research (1954), 113–122.

[8] J. Edmonds, *Maximum matching and a polyhedron with 0,1-vertices*, Journal of Research of the National Bureau of Standards (B) **69** (1965), 125–130.

[9] M. X. Goemans and D. J. Bertsimas, *Survivable networks, linear programming relaxations and the parsimonious property*, Math. Program. (1993), 145–166.

[10] M. Held and R. M. Karp, *The traveling salesman problem and minimum spanning trees*, Operations Research **18** (1970), 1138–1162.

[11] D. S. Hochbaum, *Approximation algorithms for np-hard problems*, PWS Publishing Company, Boston, 1995.

[12] M. Junger and W. Pulleyblank, *New primal and dual matching heuristics*, Algorithmica **13** (1995), 357–380.

[13] R. M. Karp and M. Held, *The traveling salesman problem and minimum spanning trees, part 2*, Mathematical Programming **1** (1971), 6–25.

[14] B. H. Korte and J. Vygen, *Combinatorial optimization: theory and algorithms – 4th ed.*, Springer Berlin Heidelberg, New York, 2008.

[15] E. Lawler, *Combinatorial optimization : Networks and matroids*, Dover Publications, Mineola-New York, 1976.

[16] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *The traveling salesman problem : a guided tour of combinatorial optimization*, Wiley, Chichester [West Sussex] /New York, 1985.

[17] L. Lovasz and M. D. Plummer, *Matching theory*, North-Holland, Amsterdam, New York, Oxford, Tokyo, 1986.

[18] P. Miliotis, *Integer programming approaches to the travelling salesman problem*, Mathematical Programming **10** (1976), 367–378.

[19] P. M. Pardalos and M. G. C. Resende, *Handbook of applied optimization*, Oxford University Press, New York, 2002.

[20] W. R. Pulleyblank, *Faces of matching polyhedra*, Ph.D. thesis, University of Waterloo, Ontario, 1973.

[21] G. Reinelt, *The traveling salesman : Computational solutions for tsp applications*, Springer-Verlag, Berlin-Heidelberg-New York, 1994.

[22] A. Schrijver, *Theory of linear and integer programming*, John Wiley and Sons, Chichester/New York/Brisbane/Weinheim/Singapore/Toronto, 1986.

[23] A. Schrijver, *Combinatorial optimization: Polyhedra and efficiency*, Springer Berlin Heidelberg, New York, 2003.

[24] T. Szabo, *Tutte-berge theorem and consequences*, Lecture Notes, McGill University, 2009.

[25] C. Tompkins, *Machine attacks on problems whose variables are permutations*, Numerical Analysis **6** (1956), 195–211.

[26] W. T. Tutte, *A short proof of the factor theorem for finite graphs*, Canadian Journal of Mathematics **3** (1954), 347–352.

[27] R. J. Urquhart, *Degree constrained subgraphs of linear graphs*, Ph.D. thesis, University of Michigan, 1967.

[28] R. J. Vanderbei, *Linear programming: foundations and extensions – 2nd ed.*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

12501 43