Master's Thesis in Computing Science

Selecting the roots of polynomial equations using combinatorial optimization

E.P. Braad
Supervised by Dr. H. Bekker

Institute for Mathematics and Computing Science
University of Groningen
February 2005

Abstract

Given a system of two bivariate polynomials, say $f(x,y) = 0 \ \wedge \ g(x,y) = 0$, we may use a computer program to approximate the roots numerically at runtime. Conventionally, this is done by eliminating y from $f(x,y) = 0$, which results in a univariate polynomial $h(x) = 0$. Solving it for x yields the roots $x_1 \cdots x_n$. These results are substituted back into the system by $f(x_i, y) = 0$ and $g(x_i, y) = 0$ to obtain possible solutions for y: $y_1^f \cdots y_n^f$ and $y_1^g \cdots y_m^g$. Among these roots we can select one value of y that is matched with x_i to form a solution (x_i, y_j) to the system, and this can be repeated for each x_i.

However, this approach is sometimes problematic. For example, in the back substitution step the equation $f(x_i, y) = 0$ may become (near) degenerate. Moreover, we must take adequate measures to prevent a y-root to be selected more than once.

In this thesis we propose and test a different approach, which does not suffer from these problems. From the system $f(x,y) = 0 \ \wedge \ g(x,y) = 0$ we generate two univariate polynomials by eliminating x and, subsequently, y from $f(x,y) = 0 \ \wedge \ g(x,y) = 0$, thereby obtaining $h_1(x) = 0$ and $h_2(y) = 0$. Both can be solved, which generates two sequences of roots $x_1 \cdots x_n$ and $y_1 \cdots y_m$. The problem is now to select the best n root pairs (x_i, y_j) among the n^2 possibilities, such that each root x_i and each root y_j is used exactly once. Moreover, the error in the final solution should be minimised. We solve this combinatorial problem by representing it in the form of a bipartite graph, and computing its minimum-weight matching. In the last chapter we discuss a technique for extending this approach to more than two variables, thereby making it applicable to multi-variate systems.

Contents

Chapter 1

Introduction

The problem of finding the roots of polynomial equations was known to mankind in ancient history; the Sumerian people had already found a closed formula for describing the roots of arbitrary quadratic polynomial equations by 3,000 B.C. Over time, the study of polynomials in general and polynomial equations in specific has provided mathematics with a platform for devising new techniques. In particular, abstract thinking and formal notation have benefited from the extensive research in this area.

Nowadays, we are no longer in search of finding closed formulas – it has been proven that above a certain degree, polynomial roots can not be caught by a single closed description. However, we make use of polynomial equations in many areas, for example computational geometry and computer algebra heavily rely on polynomial equations. These areas find practical applications in industry and other fields.

Although algebraic packages exist which may approximate numerically precise roots of the most complicated types of polynomial equations, in most applications we need to make use of functionality available to a programmer at runtime. If speed or memory limitations are of concern, less sophisticated methods must be used.

In this thesis, we present a robust method for selecting the roots of bivariate polynomial equations. This method can guarantee that the solution with the smallest error is found and that error depends only on which internal solver is used. By making use of well-known existing algorithms, the error may be reduced as much as possible. Moreover, the correctness of the solution is guaranteed and no roots are missed by this procedure. Our solution considers real roots as well as complex roots. In the final chapter, we discuss a technique for extending our results to polynomials that involve multiple variables.

In general, the method works as follows. Given a system of bivariate polynomials $f(x, y) = 0 \wedge g(x, y) = 0$, two univariate polynomials $h_1(x)$ and $h_2(y)$ are computed by elimination of x and y respectively. The roots for x are insrted into one bipartition X of a bipartite graph G. The roots for y are inserted into the other bipartition Y of G. An edge $e_{x,y}$ with $x \in X$ and $y \in Y$ can now be used to represent a possible solution to the original system: (x_i, y_j). The edge weight for $e_{x,y}$ is set to the error in such a solution pair. Computing a minimum-weight bipartite graph matching now corresponds to computing a set of n solutions of the system, such that the total error is minimal. This method is presented in detail in Chapter 5.

1.1 Outline

After the introduction, the basic principles and structures are discussed in Chapter 2. Here, we present what a system of polynomial equations is exactly and introduce the basic concepts of bipartite graphs and computing matchings. In Chapter 3 a brief introduction of the field of solving polynomial equations is given. We limited the discussion to the concepts and methods that apply to the method proposed in this thesis.

Attention is shifted to a pratical case-study in Chapter 4. By analyzing the details of similarity measures based on Minkowski-addition, we find an interesting problem in which a system of polynomials needs to be solved. Due to the geometric nature of the TVT-problem, many degenerate instances of the polynomials arise, which provide a good test case for our method.

The main idea of our approach is presented in Chapter 5. We formulate the problem explicitly and then describe how the correct roots can be selected using bipartite graph matching. This approach is motivated by a detailed example. Moreover, the test results are presented here.

Because the bipartite-graph approach is limited to systems involving two variables only, we felt the need to provide directions for solving multi-variate systems using a combinatorial approach. In Chapter 6, we generalise the problem to the linear assignment problem and discuss an optimised algorithm for solving such problems.

A wide range of topics is covered in this thesis. Remember, however, that the main idea is simple in nature: selecting the roots of a system of equations using a combinatorial approach.

Chapter 2

Preliminaries

To fully appreciate the method proposed in this thesis, it is required that the reader is familiar with a number of constructs and concepts. In this chapter these concepts are briefly introduced and illustrated.

2.1 Bivariate polynomial equations

In this thesis we will present an alternative way of finding roots of a special class of equations. In this section this class of *polynomial* equations is introduced.

2.1.1 Polynomial functions

An important subset of the mathematical functions is formed by the class of polynomials – functions that are defined as a sum of weighted powers. Such functions occur in many applications; predominantly in geometric computations. A polynomial in one variable is, in general, of the form

$$\begin{aligned} f(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ &= \sum_{i=0}^{n} a_i x^i \end{aligned} \tag{2.1}$$

Here, the symbolic constants $a_0 \cdots a_n$ are called the *coefficients* and each term $a_i x^i$ is called a *monomial*. The *degree* of a univariate polynomial equals the highest power occurring in the formulation of Equation 2.1; in this case n.

Definition 2.1. The highest power occurring in the definition is called its (polynomial) *degree*.

When we consider a specific polynomial, say $f(x) = 2x^2 - 4x + 1$, we can regard it in various ways. Most naturally, we would say it is of degree 2. But all the same, we could argue that $f(x)$ is of degree 3, with the monomial $a_3 = 0$. The problem is even worse: $f(x)$ is of any degree $d \geq 2$. To remedy this, we define the concept of *monicity*. When a polynomial is *monic*, we are guaranteed that the term with the highest power – that of the degree of that polynomial – occurs with a non-zero coefficient. Thus the discussed $f(x)$ is monic in degree two (since $a_2 \neq 0$) but not monic in degree three and up (since $a_3 = 0$ and in general $i > 2 \rightarrow a_i = 0$).

Definition 2.2. A polynomial f of degree d is called *monic* if the coefficient of the term in which d occurs is non-zero.

Polynomials may occur with any non-negative number of parameters. We refer to these polynomials as *univariate* if one variable is involved, *bivariate* if two variables are involved and *multivariate* if more than two variables are involved. In this thesis we consider only bivariate polynomials, which are generally of the form

$$\begin{aligned} f(x,y) &= a_{0,0} + a_{1,0}x + a_{2,0}x^2 + \cdots + a_{n,0}x^n + \\ &\quad a_{1,1}xy + a_{1,2}xy^2 + \cdots + a_{1,m}xy^m + \cdots + a_{n,m}x^n y^m \\ &= \sum_{i=0}^{n}\sum_{j=0}^{m} a_{i,j}x^i y^j \end{aligned} \tag{2.2}$$

This bivariate definition can be further extended to multivariate polynomials. As we will only be concerned with bivariate polynomials, this definition is omitted. A strict definition of polynomial functions is given below.

Definition 2.3. A *polynomial function* is a function defined as a sum of weighted powers of its variables. When the function is defined in one variable, i.e. $f : R \to R$, then f is called *univariate*. If the function is defined in two variables, i.e. $f : R^2 \to R$, then f is called *bivariate*. A polynomial in more variables is called *multivariate*.

For multivariate polynomials, the polynomial degree is defined differently. For bivariate polynomials in particular, the degree is defined as the sum of the degree in the individual variables.

Definition 2.4. The *degree* of a bivariate polynomial $f(x,y) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_{i,j}x^i y^j$ is $m+n$.

2.1.2 Polynomial equations

As with any function, equations can be formed from polynomials as well. When a polynomial occurs on one side of the equation, we speak of a *polynomial equation*. There may or may not be real solutions to the equation. Usually, we are especially interested in equations that have a polynomial on one side and zero on the other side – this is no limitation, since any polynomial equation $f(x) = c$ can be transformed to this form by $f(x) - c = 0$. The solutions to such equations are called the *roots* of the polynomial.

Definition 2.5. A *polynomial equation* is an equation in which a polynomial is equated with zero. The solutions to such an are called the *roots* of that polynomial.

The roots of an equation may be either *complex* or *real*; in general, they are complex. If the monomials are all real, the complex solutions appear with equal real parts but conjugated imaginary parts. This will be elaborated upon in Chapter 3.

2.2 Bipartite graphs

In this section we will gently introduce the concept of bipartite graphs. We will start out by recalling the more common graph theory, and then extend the basics to bipartite graphs. The material presented in this section is freely after [13].

2.2.1 Graphs

Consider the problem of planning a delivery route for delivering packages to receivers in various cities. We may think of a fixed number of cities to be visited and these cities are interconnected by roads of known length. Many routes for the delivery may exist, but some are shorter than others. The "best" route may be considered a route that is of the shortest possible length, and the problem of a logistics planner is to find such a route.

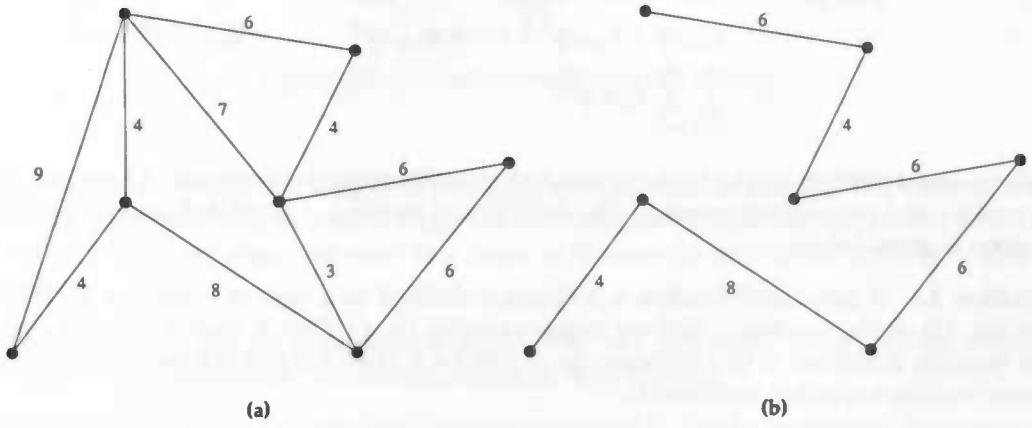


Figure 2.1: *The package delivery problem represented as a graph. In (a) we see the cities and the interconnecting roads with their respective lengths. In (b) a possible route is shown.*

The above problem is very well suited to be represented as a graph. We can represent each city as a node. Then, for each road connecting two cities, we add a line connecting the corresponding nodes. Such a line is called an *edge*. Finally, we associate with each edge the length of the corresponding road. We have then constructed a representation as shown in Figure 2.1(a) and such a representation is called a *graph*. A possible shortest route then consists of a sequence of nodes, connected by edges.

Many other problems occurring in all day's life are suited for graph representation; examples are relationships (who knows who?), hierarchies (who works for whom?) and computer networks (which computer can connect to which computers?). Formally, a graph is defined as follows.

Definition 2.6. A *graph* G is a structure consisting of two sets $G = (V_G, E_G)$. The elements of V_G are called the *vertices* and the elements of E_G are called the *edges* of G. Each edge $e \in E_G$ has a set of one or two vertices $u, v \in V_G$ associated with it, which are called its *endpoints*.

The edges in a graph connect vertices; two at a time. In the delivery problem each edge represents a road and, as the existence of a road from A to B implies a road from B to A exists as well, each edge connects two vertices in both directions. Such an edge is called *undirected*, since it runs both from A to B and B to A. But in many problems *directed* edges are required. For example, in the relationship-setting, everyone may know some celebrity who does not know everyone else in turn.

A graph can be either directed or undirected, indicating a significant ordering on the endpoints of its edges. In this thesis we will only consider undirected graphs but we will present the definition here for the sake of completeness.

Definition 2.7. A *directed* edge is an edge one of whose endpoints is designated as the *tail* and one as the *head*. The edge is said to be *directed* from its tail to its head. Any edge that is not explicitly

directed is undirected by definition. A graph containing directed edges only is a directed graph and a graph containing only undirected edges is an undirected graph.

Sometimes, we want to denote the vertex that is on the other end of some edge. If a directed edge $e_{u,v}$ connects a vertex u to a vertex v, we have $\text{head}(e_{u,v}) = u$ and $\text{tail}(e_{u,v}) = v$. We may also describe the relation between u and v by saying v is a vertex connected to u, denoted $v(u)$.

In the delivery problem we associated a number with each edge, representing the length of a road. We can associate such *weights* with edges at any time, and such a graph is a *weighted graph*.

Definition 2.8. With each edge e a weight $\text{weight}(e)$ can be associated. A graph containing such *weighted edges* is a *weighted graph*.

Some edges connect a vertex with itself, thus creating a loop. In the relationship-setting, for example, one will always know oneself. Such a *self-loop* is defined formally below. See also Figure 2.2.

Definition 2.9. A *self-loop* is an edge that has only one vertex associated with it. An edge that is associated with two distinct vertices is called a *proper* edge. A *multi-edge* is a set of two or more edges having identical endpoints. A graph that contains no self-loops and no multi-edges is called *simple*.

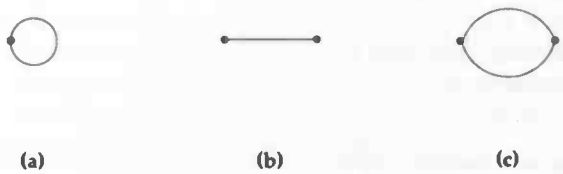


Figure 2.2: *Various types of edges: (a) a self-loop, (b) a proper edge and (c) a multi-edge.*

In some graphs, all nodes are connected to each other by one edge – in such a graph one cannot add an edge between two vertices, because all possible edges exist. Naturally, this can only occur in simple graphs, thus allowing only single connections between two distinct vertices. Such graphs are called *complete*; an example is shown in Figure 2.3.

Definition 2.10. A graph is called *complete* if it is simple and every pair of vertices $u, v \in V_G$ is joined by an edge $e \in E_G$.

2.2.2 Bipartite graphs

In order to motivate the concept of bipartite graphs, let us consider another problem. Suppose that at some company a number of tasks needs to be performed and that a number of employees is available to do these tasks. But not every employee is suited for every task equally well. We assume that we know how good each employee is at each task. Our problem is to compute a task assignment such that the overall quality of the performed tasks is maximal and, naturally, every task is done.

This problem also maps well onto a graph structure: for every employee and every task, introduce a corresponding vertex. An assignment of a task to an employee now corresponds to an

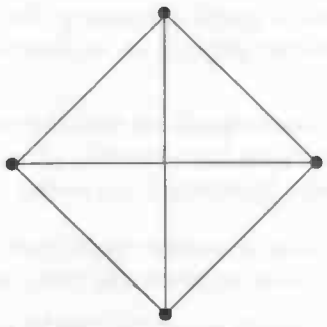


Figure 2.3: *A complete graph with four vertices.*

edge between them, and the weight of that edge indicates the suitedness. But there is one difference: we are restricted in how we can connect vertices. Edges may only exist between an employee and a task; edges between two tasks or between to employees are meaningless and redundant.

In bipartite graphs, the set of vertices is partitioned into two parts. Only vertices in different partitions may be connected. The formal definition is as follows:

Definition 2.11. A graph G is *bipartite* when the vertex-set V_G can be partitioned into two subsets U and W, such that each edge has one endpoint in U and one endpoint in W. Naturally, it is required that $U \cap W = \emptyset$ and $U \cup W = V_G$. The pair U, W is called the *vertex bipartition* of G, and U and W are called the *bipartition subsets*. The bipartition to which a single vertex $v \in V_G$ belongs, is denoted $\mathit{bipartition}(v)$.

Note that, by definition, a bipartite graph can contain no self-loops. Furthermore, by our current definition, a bipartite graph can never be complete since edges between two vertices of the same bipartition are not allowed. Therefore we redefine the concept of completeness.

Definition 2.12. A bipartite graph G is called *complete* if it is simple and every pair of vertices $u, v \in V_G$ with $\mathit{bipartition}(u) \neq \mathit{bipartition}(v)$ is joined by an edge $e \in E_G$.

2.2.3 Bipartite matching

With the definitions of bipartite graphs, we can continue to formulate a solution to the task-assignment problem. Let the vertex-set of some graph G consist of all tasks T and all employees E together, i.e. $V_G = E \cup T$. For every employee $e \in E$ and every task $t \in T$ we add an edge from e to t if the employee is capable of performing the task. Moreover, we associate with that edge a weight that indicates the suitability of that task assignment. This would result in a bipartite graph describing the situation. The solution could then be described by a bipartite graph G' in which an edge from e tot t would stand for an assignment of the task to that employee. Of course, this solution form would have to satisfy that only possible assignments are performed and that each task is assigned only once. Formally this is:

i) for every edge $e' \in G'$ a corresponding edge $e \in G$ exists, but not necessarily *vice versa*.

ii) for every vertex $u \in G'$ only one edge e may have $\mathrm{tail}(e) = u$ or $\mathrm{head}(e) = u$.

In Figure 2.4 the task assignment problem is shown in a bipartite graph representation.

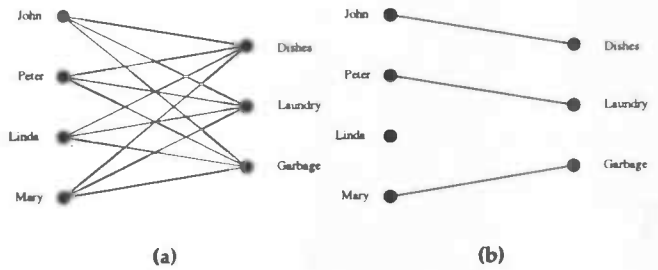


Figure 2.4: *A task assignment problem represented as a bipartite graph. In (a) the problem situation is shown; each edge represents a possible assignment. In (b) a possible solution is shown; each edge represents a task assignment. Lucky for her, Linda is not scheduled to do any task.*

Alternatively, such a solution can be seen as a set of edges M that is a subset of E_G and in which each vertex occurs only once in some edge. A subset of edges of a graph chosen such that no vertex is associated with more than one edge, is called a *matching* in that graph.

Definition 2.13. A set M of edges in a graph G is called a *matching* in G if no two edges in the set M have an endpoint in common. A *perfect matching* is a matching in which every vertex is an endpoint of one of the edges (this requires that the bipartitions are equal in size).

We have not yet taken into account the suitability of a task assignment scheme. Or, in general terms, so far we have only discussed unweighted matchings. Usually, we are interested in matchings that minimise some criterion, in this case the overall "unsuitability" of the task assignment. Such a matching is called a *bipartite minimum-weight matching*.

Definition 2.14. A *bipartite minimum-weight matching* in a graph G is a matching M with total edge weight w_M such that no matching M' exists with total edge weight $w_{M'} > w_M$.

Chapter 3

Polynomial root finding

The problem of finding roots of a polynomial equation has been known to mankind for many centuries – in fact the Sumerians found a closed formula for describing roots of quadratic polynomial equations around 3.000 B.C. From then, the study of polynomial equations has pushed mathematics forward altogether, in particular contributing to shaping formal notation and elaborating abstract thinking [14]. In this chapter we will provide a brief overview of what is known about polynomial equation solving – we limit ourselves to those findings that relate to our topic, since the field is wide and dense.

3.1 Univariate polynomial equations

As we saw in Chapter 2, polynomial equations come in various types. Let us first study equations that involve only one variable, say x – the so-called univariate polynomial equations. These are of the form $p(x) = 0$, where $p(x)$ is a polynomial, thus

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{3.1}$$

We are interested in all values for x such that the equation is satisfied – the values for which $p(x)$ becomes zero are called the roots of the equation.

3.1.1 Fundamental theorem of algebra

We want to find all roots, and we thus need to know how many roots exist. If a polynomial is defined as in Equation 3.1, and it is monic according to Definition 2.2, we can compute the number of roots of a polynomial equation in that polynomial. This claim is known as the fundamental theorem of algebra [10]:

Definition 3.1. *Fundamental Theorem of Algebra:* If $p(x)$ is a polynomial of degree $n \geq 1$ with real or complex coefficients, then $p(x) = 0$ has exactly n roots.

As we saw earlier, roots of polynomial equations may be either complex or real. In general, the roots of $p(x)$ lie in the complex plane $\mathbb{C}$. In Definition 3.1, the number of roots indicates real roots as well as complex roots. We do not know how many of the n roots are complex and how many

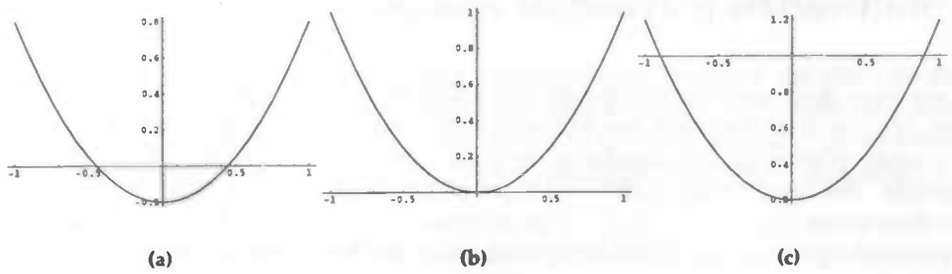


Figure 3.1: *Multiple roots may join together to form one double root or even zero (real) roots. In (a) clearly two separate roots can be distinguished, while in (b) they come together to form one double root. In (c) no real-valued root exists at all.*

are real, however. Also, some roots may occur more than once. This follows from degeneracies in the function; we can easily see this in the graphs of Figure 3.1. In (a), the curve intersects the x-axis twice, indicating two distinct roots. But in (b) the function has been slightly adjusted. Now, the x-axis is touched but not intersected – the two roots of (a) coincide as a multiple root in (b). Similar degeneracies may occur in the complex plane. In Definition 3.1 such multiple roots are counted as two separate roots (with the same value). Roots may also be threefold and up. When the polynomial $p(x)$ has real coefficients $a_0 \cdots a_n$, then its complex roots appear in conjugated pairs (with respect to their imaginary parts).

3.1.2 Abel's impossibility theorem

One may recall from high school that some univariate polynomial equations are easily solved. Of course, the degenerate case where $p(x) = a_0 = 0$ can be solved only when a_0 is zero and in that case any x will do as a solution. Polynomials of degree one are well known; these are so-called linear equations $p(x) = a_0 + a_1 x = 0$. These can be solved by simple rearrangement of terms, arriving at $x = -\frac{a_0}{a_1}$. Note that by the assumption of monicity, a_1 is non-zero and thus x is defined. For polynomials of higher degrees, similar solving formulae exist. The well-known *quadratic formula* yields the (possibly complex, possibly multiple) roots of a second-degree polynomial. For reminiscence, it is defined as

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.2}$$

Even for fourth-degree polynomial equations such a formula may be given [2]. It algebraically describes all roots of a fourth-degree polynomial. However, for fifth-degree polynomials and up, such closed formulae do not exist. For a long time, mathematicians have sought for such formulae, particularly in the 19th-century, until Galois and Abel proved they do not exist. This claim is since known as Abel's impossibility theorem, found by the tragic Niels Abel in 1826. [1]

Definition 3.2. *Abel's Impossibility Theorem:* In general, for polynomial equations of degree five and up, no algebraic solutions exist.

This is meant in terms of a finite number of additions, subtraction, multiplications, divisions and root extractions. The even more tragic Everiste Galois proved the same by deriving that an algebraic equation is solvable if and only if its group is solvable, and subsequently deriving that the group of polynomials of degree five and up is not solvable.

3.2 Multivariate polynomial equations

So far, we have discussed the solving of univariate polynomial equations. From a polynomial equation $p(x) = 0$ of degree n, we know how to find the sequence of roots $x_0, \cdots, x_n \in \mathbb{C}$. In many cases in practice, the equations are more complex than can be described by univariate polynomials. We need to use multivariate polynomials to describe such situations; instead of $p(x)$ we thus write $p(x_0, x_1, \cdots, x_n) = 0$ or, alternatively, $p(\mathbf{x}) = 0$. The roots of such multivariate polynomial equations are, in the latter case, described by the sequence $\mathbf{x}_0, \cdots, \mathbf{x}_n \in \mathbb{C}^n$ where $p(\mathbf{x}_i) = 0$.

3.2.1 Systems of polynomial equations

So far, we discussed finding roots of single equations. In practice, virtually all problems require that we solve a number of polynomial equations simultaneously. That is, given a number of individual equations, we need to find roots that satisfy each of the equations individually at the same time. In this section we discuss how such *systems* of polynomial equations may be solved. In general, if we want to solve equations in which n variables are involved, we need a system of n such equations as well.

In a system of polynomial equations, we are interested in the common solutions of a set of equations. Consider a system of polynomial equations $p(x, y) = 0$ and $q(x, y) = 0$. We are interested in solutions (x, y) such that $p(x, y) = 0 \wedge q(x, y) = 0$. Only $p(x, y) = 0$ is not enough, and neither is $q(x, y) = 0$ individually. A system of equations and its solution is formally defined as follows.

Definition 3.3. A system of equations is a set S of equations of the form $f_i(\mathbf{x}) = 0$ with corresponding functions $f_0(\mathbf{x}), \cdots, f_n(\mathbf{x})$. The solution of S is a set containing all solutions $\mathbf{x}$ such that $\forall i f_i(\mathbf{x}) = 0$.

3.2.2 Ordinary approach

In the case of multivariate polynomials, the methods and techniques described previously do not apply directly. We can, however, reduce the complexity of multivariate polynomials in various ways, such that we arrive at (a number of) univariate polynomials. In general, two major techniques exist for eliminating variables: Groebner bases and resultants. We will first discuss how a multivariate polynomial equation may be solved using reduction; afterwards, we will motivate both reduction methods.

Consider a problem in which we need to solve some polynomial system of equations $f(x, y) = 0 \wedge g(x, y) = 0$. Using a reduction method, we may compute a univariate polynomial $p(x)$. Solving $p(x) = 0$ is straightforward and yields a sequence of roots $x_1 \cdots x_n$. In this sequence, each x_i corresponds to a root of the system. In order to determine the corresponding root y_j for each x_i, each x_i is back substituted into $f(x, y) = 0$ and $g(x, y) = 0$. This yields two equations $f(x_i, y) = 0$ and $g(x_i, y) = 0$. From each of these two equations, a sequence of y-roots may be computed; say $y_1^f \cdots y_n^f$ and $y_1^g \cdots y_m^g$. The value y_j occurring in both sequences is the y-root that belongs to x_i – we have thus found a pair (x_i, y_j) such that $f(x_i, y_j) = 0 \wedge g(x_i, y_j) = 0$. The process is repeated for all x-roots, which in the end yields all possible pairs.

3.2.3 Resultants

As we saw, we require the ability to reduce a multivariate polynomial equation to univariate polynomials. One way to achieve that is by using resultant-methods. In its most simple form, the resultant is defined directly in terms of the monomials of two polynomials. Given some f with roots $\alpha_0 \cdots \alpha_n$ and some g with roots $\beta_0 \cdots \beta_n$, the resultant is described by a product

$$\rho(f,g) = \prod_{i=0}^{n} \prod_{j=0}^{m} (\beta_j - \alpha_i) \tag{3.3}$$

The result is a polynomial of a lower dimension than f and g, of which the roots correspond to those of f and g. Thus, if we take f and g as in our previous example to be $f(x,y)$ and $g(x,y)$, then $p(x) = \rho(f,g)$ is a univariate polynomial, with roots corresponding to x-roots of $f(x,y) = 0$ and $g(x,y) = 0$. The resultant method is asymmetric, that is, $\rho(f,g) \neq \rho(g,f)$. As may be expected, $\rho(g,f)$ yields the y-roots that correspond to $f(x,y) = 0$ and $g(x,y) = 0$. [3]

3.2.4 Groebner bases

The second method for eliminating a variable from a multivariate polynomial equation is by means of Groebner bases. A Groebner basis, founder by Bruno Buchberger in 1956 [8], is a mathematical equivalence system of polynomials. The key feature is that the set of polynomials in a Groebner basis has the same collection of roots as the original polynomial. Among these, we can select a number of polynomials such that a new system can be constructed that has the same roots as the original system. By defining an order – or preference – on the variabls, we can make that choice explicit.

We will omit the details of computing Groebner bases and using Groebner bases to reduce the degree of polynomial system. What is important is that, in practice, one can use them similarly to resultants. Thus, given two bivariate polynomial equations we can compute a univariate polynomial $p(x)$ or $q(y)$ with roots corresponding to the initial polynomials, either by resultants or Groebner bases.

3.3 Numerical approximation

According to Abel's impossibility theorem we cannot symbolically find roots of high-degree polynomial equations. Research, however, has continued and many methods have been devised for numerically approximating polynomial equations. The introduction of the computer increased demand for such techniques and at the same time provided applications in computational geometry and other fields.

The various polynomial solving algorithms, and their rich variations, have various characteristics. Some require you to give a number of estimations of roots – for example Müller's method takes three approximations x_0, x_1 and x_2 to yield the roots of $p(x)$ within a specified numeric tolerance. Other methods require no initial approximations. Sometimes, the roots yielded by the algorithms are rather imprecise – in this case special *root polishing* algorithms may be used. In general, the algorithms differ in the precision and robustness with which roots are found.

In our experiments we have used Laguerre's method for finding roots. This method requires no initial guesses – it converges to a complex root from any starting position – and is therefore easy to use. Moreover, it is able to find all roots of a univariate polynomial when called repeatedly.

Finally, an implementation of it is included in a well-known library of computer algorithms. A drawback of Laguerre's method is that, compared to some other techniques, the precision of the roots found is sub-optimal – especially when dealing with multiple roots.

3.3.1 Problems in numerical solving

In numerical solving we are, of course, limited to machine precision. Some packages exist that provide arbitrary precision computations, but such packages are slow when compared to ordinary double or float arithmic. The machine-precision limitations introduce some hard problems which do not exist when solving algebraically.

First, when back substituting a root x_i into $f(x, y) = 0$ we hope to arrive at a new equation involving only y. The coefficients $a_0, \cdots, a_n$ describing $f(x, y)$ may cause this substitution to reduce the equation to $0 = 0$, which is useless. If it reduces to exactly $0 = 0$ we speak of an *exact degeneracy* – the zeroes are exact, which may be detected at run time. Sometimes, the equation reduces to nearly $0 = 0$, but numerically one of the sides of the equations is very small but non-zero. In this case we speak of a *near degeneracy*. Near degeneracies are hard to detect and may go unnoticed – the solving procedure may fail altogether. A similar reasoning holds for back substituting $y = 0$ into $g(x, y)$. When the number of solutions is finite, it cannot occur that both equations become degenerate at the same time.

Second, there may be problems with detecting multiple roots. While in symbolical solving the values of a double root, say x_i and x_j are identical, in numerical solving the values may differ – we cannot simply check wether $x_i = x_j$. Epsilon math may be used to decide that the roots are equal when their difference is smaller than some small number ε, but this cannot guarantee all multiple roots are detected as such.

Third, we need to properly administrate which roots x and y belong in which root pair (x, y). Each root may only be used once in some pair, thus we need to administrate that if x_i was used with y_j to form some pair, we cannot re-use x_i to match with y_k $(k \neq j)$. This may be especially complicated in the context of the multiple root problem.

Note that none of these problems is insurmountable – using appropriate programming techniques and using domain-specific knowledge, each of the three may be avoided or reduced. However, this may prove to be a hard case, in particular when dealing with complicated or near-degenerate problems. The technique proposed in this thesis does not require such extra programming work and cleanly solves the system regardless of these problems.

3.3.2 Algebraic packages and libraries

Various algebraic computation packages exist for computer systems; examples are Mathematica and Maple. Such packages may, when supplied with the right input, compute the solutions of complex polynomial equations symbolically. Internally, they use various strategies and techniques to produce a symbolic result. However, when in practice a problem is defined symbolically but needs to be solved efficiently and many times for various instances of the constants, one cannot use such a system. In that case, the problem needs to be coded in a programming language (for example C++) and the resulting program run with changing parameters.

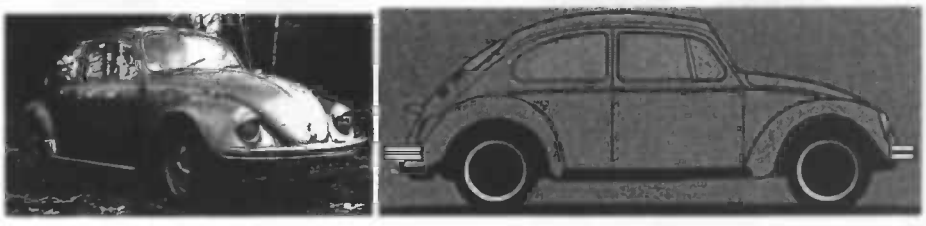
Luckily, many additional libraries exist that provide useful functionality with regard to solving in a computer program. In particular, the package *Numerical Recipes* [15] provides many implementations of well-known solving algorithms and *Synaps* [11] deals with polynomial equations

in specific. Compared to algebraic computation packages such as Mathematica and Maple, the results yielded by these runtime libraries are usually of a smaller precision than an algebraic system would yield. Moreover, we may still be faced with some or all of the problems described in the previous paragraph.

Chapter 4

The TVT-problem

In the previous chapter we discussed rather theoretically what methods are in use for finding roots of polynomial equations. However, finding roots is in itself a rather boring activity. Only when put in the context of a problem – and preferrably a practical and real problem – can we seriously examine the various methods. In this chapter we will introduce such a problem in detail: the TVT-problem. This will serve as a context for discussing our proposed method in Chapter 5.



(a) (b)

Figure 4.1: *Retrieving objects from a database using object comparison: (a) what kind of car is this? (b) it's a turqouise 1972 Volkswagen Beetle 1300.*

4.1 3D Object comparison

Consider a computer system that is used to identify three-dimensional objects. The input comes from a camera or some other device which, after some image processing, yields the geometry of the object in 3D space. The presented object is now compared to objects in some database, in order to find a nearest match – see Figure 4.1. At the heart of this system lies some routine for comparing two objects, and computing a similarity value for those specific objects. The TVT-problem is a pre-processing step in computing such a similarity measure, as we will explain shortly.

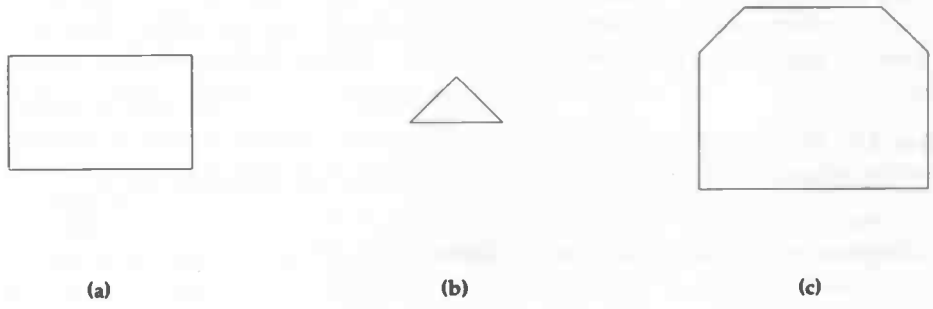


Figure 4.2: *Minkowski addition for two-dimensional objects. The second object shown in (b) is dragged around the border of the first object shown in (a) to produce their Minkowski sum shown in (c). Note that the computed object bears similarities to both input objects.*

4.1.1 Comparing 3D objects using Minkowski addition

In [7] the computation of similarity measures is discussed in much detail – here, special interest is taken in using Minkowski addition to compute similarity measures for two objects. We will not discuss in detail how the Minkowski sum is used to that end, but it is important to notice some properties of the Minkowski sum. The Minkowski sum of two objects can intuitively be seen as the shape described by dragging the second object shape around the border of the first shape. In this way, the shapes of both objects can be combined into some sort of "mean" shape. For a two-dimensional case an example is shown in Figure 4.2. Formally, the Minkowski sum (denoted $\oplus$) is defined as follows.

Definition 4.1. *Minkowski sum:* The Minkowski sum of two objects A and B is defined as the set $A \oplus B = \{a + b : a \in A, b \in B\}$.

Here, the $+$-sign indicates the vectorial sum or a displacement of a over b. A different way of thinking about the Minkowski sum is the following, analogous to the definition. For each point $a \in A$, compute a new set of points by displacing a over al vectors $b \in B$. The Minkowski sum of A and B is then the union of all those newly computed point sets, as is reflected by Definition 4.2. Notice that for both a and b infinitely many choices exist, since the objects span a continuous subspace which cannot be described by a finite list of points. For now, it suffices that based on the object described by the Minkowski sum of two input objects – or more specifically, based on its volume – a similarity value for A and B may be computed.

4.1.2 Convex polyhedra

In the foregoing we discussed similarity measures for three-dimensional objects, but we have not defined what objects are suitable for comparison and how they should be represented. In fact, similarity measures based on the Minkowski sum are currently limited to *convex objects* – objects that contain no holes and no dents. More precisely, any two points within a convex object can be connected by an imaginary line segment such that the line is also fully contained in the object.

Definition 4.2. *Convexity:* An object A is convex if for any two points p and q with $p, q \in A$ it holds that the line segment $pq \in A$.

We will also use a discrete object representation, that is, the objects are represented by a finite list of bounded planar surfaces. These surfaces – which may be very small to approximate curved surfaces – are called *faces*. The lines where adjacent faces meet are called *edges*, and the points where three or more faces meet are called *vertices*. Objects defined in those terms are called *polyhedra*.

Definition 4.3. *Polyhedron:* An object defined in terms of a limited number of vertices, which are connected by edges, which in turn are bordered by faces is called a polyhedron.

In the TVT-problem we deal with *convex polyhedra* only.

4.1.3 Critical orientations

Imagine yourself comparing two real life three-dimensional objects; two coffee cups, for example. One of the first things most people will do before comparing the objects is re-orient one of the cups to approximately match the orientation of the other. We find it easier to compare if the top holes of the cups are both up, and if the handles point in the same direction – see Figure 4.3. This pre-comparison step is what the TVT-problem is: re-orienting one of the objects such that it matches the orientation of the other object best.

(a) (b)

Figure 4.3: *Aligning cups before comparison. In (a) the cups are oriented different from one another, resulting in a different Minkowski-based similarity measure. In (b) the cups are oriented likewise and the correct measure can be computed.*

In the computer system described at the start of this chapter, this pre-processing step is required because the Minkowski sum is not invariant to rotation. If the orientations of two identical objects differ, the computed similarity measure will not be 1, but less. Note also that, if the objects are oriented similarly, the similarity measure will be maximal. The TVT-problem is concerned with finding rotations for the second object, such that it matches best with the first object – in other words, such that the maximum similarity may occur.

Given two objects, multiple *critical orientations* may exist for which the maximum similarity may occur. The one for which it actually does occur is called the *optimal orientation*. We need to produce *all* critical orientations in order to be able to compute the optimal orientation and thus the maximum similarity measure for this pair. For two identical objects, again, the similarity measure is of course maximal when the orientations are exactly the same. If we miss some of the orientations, we may miss the optimal orientation and produce the wrong measure.

Note that multiple orientations may exist that produce an optimal orientation – for example when

comparing two cups with two handles each. In that case, the cups may be rotated 180° around their longitudal axes without changing the similarity.

4.2 Computing critical orientations

In the last paragraph, we described the TVT-problem as the problem of finding all critical rotations for an object to become oriented similar to a given other object. In this section we will present the mathematic derivation that describes this problem formally. Then, solving the mathematical problem corresponds to solving the critical orientations finding problem.

4.2.1 Orienting vector triples

We will need to know something about critical orientations. When may they occur? In [16] it is shown that maximum similarity only occurs if three faces in one object align with three edges in the other object. By aligning a face to an edge, we mean that of a face f the normal N_f is perpendicular to the edge e – or simply $N_f \cdot e = 0$. Thus, we need to orient a vector triple such that it aligns with another vector triple. This is where the TVT-problem acronym stems from: TVT stands for *two vector triples*.

Computing the critical orientations for two objects A and B means finding a rotation such that faces in A align as much as possible with edges in B. More precisely, given three normal vectors N_{f_1}, N_{f_2} and N_{f_3} of faces in A and three edges e_1, e_2 and e_3 in B, we want to compute a rotation $\mathbf{R}$. This rotation $\mathbf{R}$ needs to satisfy

$$N_{f_1} \cdot \mathbf{R}e_1 = 0 \wedge N_{f_2} \cdot \mathbf{R}e_2 = 0 \wedge N_{f_3} \cdot \mathbf{R}e_2 = 0 \tag{4.1}$$

which is equivalent to aligning two vector triples $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and $(\mathbf{k}, \mathbf{l}, \mathbf{m})$:

$$\mathbf{k} \cdot \mathbf{Ra} = 0 \wedge \mathbf{l} \cdot \mathbf{Rb} = 0 \wedge \mathbf{m} \cdot \mathbf{Rc} = 0 \tag{4.2}$$

Some selections of $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and $\mathbf{k}, \mathbf{l}, \mathbf{m}$ cannot be aligned in this sense. These cases are discarded before we compute the orientation, and thus do not need to be considered.

4.2.2 Pre-conditioning

The problem described by Equation 4.2 clearly requires six vectors and $\mathbf{R}$ should be computed accordingly. However, various degenerate cases may complicate the solving strategy. Using preconditioning techniques we try to simplify the algorithm.

First of all, the problem is invariant to permutations of $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ when paired with a corresponding permutations of $\mathbf{k}$, $\mathbf{l}$ and $\mathbf{m}$ [4]. This property is used to create a situation where $\mathbf{a}$ and $\mathbf{b}$ are linearly independent and $\mathbf{k}$ and $\mathbf{l}$ are linearly independent.

Second, we can reduce the number of equation by a simple math trick. Suppose we have a second problem defined as

$$\mathbf{k}' \cdot \mathbf{Ra}' = 0 \wedge \mathbf{l}' \cdot \mathbf{Rb}' = 0 \wedge \mathbf{m}' \cdot \mathbf{Rc}' = 0 \tag{4.3}$$

and assume that the new vectors are related to their corresponding vector in Equation 4.2 by two orthogonal transformations **S** and **T**. Thus

$$\mathbf{a} = \mathbf{S}\mathbf{a}' \quad \mathbf{b} = \mathbf{S}\mathbf{b}' \quad \mathbf{c} = \mathbf{S}\mathbf{c}'$$
$$\mathbf{k} = \mathbf{T}\mathbf{k}' \quad \mathbf{l} = \mathbf{T}\mathbf{l}' \quad \mathbf{m} = \mathbf{T}\mathbf{m}'$$

Now, let us assume that we have found some rotation matrix **R** that satisfies Equation 4.2. Combining this with the previous two derivations, we get

$$\mathbf{T}^{-1}\mathbf{R}\mathbf{S}\mathbf{a}' \cdot \mathbf{k}' = 0 \wedge \mathbf{T}^{-1}\mathbf{R}\mathbf{S}\mathbf{b}' \cdot \mathbf{l}' = 0 \wedge \mathbf{T}^{-1}\mathbf{R}\mathbf{S}\mathbf{c}' \cdot \mathbf{m}' = 0 \tag{4.4}$$

Thus, if we find a solution **R** to the problem in Equation 4.2 we also find a solution $\mathbf{R}'$ to the problem in Equation 4.3. The mapping is given by $\mathbf{R}' = \mathbf{T}^{-1}\mathbf{R}\mathbf{S}$.

Since the orthogonal transformations **S** and **T** are left completely free, they may be chosen such that the triples **a**, **b**, **c** and **k**, **l**, **m** are positioned such that the problem in Equation 4.2 is simplified. Thus, we choose **S** such that **a** is oriented along the positive x-axis and we choose **T** such that **k** is oriented along the positive z-axis. This is allowed and ensures that **a** and **k** are known and are known to be mutually orthogonal. More precisely:

$$\mathbf{a} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}, \mathbf{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \mathbf{l} = \begin{pmatrix} l_0 \\ l_1 \\ l_2 \end{pmatrix}, \mathbf{m} = \begin{pmatrix} m_0 \\ m_1 \\ m_2 \end{pmatrix} \tag{4.5}$$

4.2.3 Rodrigues rotation matrix

An orthogonal rotation transform can be described by a rotation over a certain angle around an axis through the origin. This can be described by one vector

$$\mathbf{r} = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$$

, where (u, v, w) describes the rotation axis and the length $|\mathbf{r}|$ describes the amount of rotation. Such a rotation can, alternatively, be described by a Rodrigues rotation matrix **R** as follows:

$$\mathbf{R} = \frac{1}{1+u^2+v^2+w^2} \begin{pmatrix} 1+u^2+-v^2-w^2 & 2(uv-w) & 2(uw+v) \\ 2(uv+w) & 1-u^2+v^2-w^2 & 2(vw-u) \\ 2(uw-v) & 2(vw+u) & 1-u^2-v2+w^2 \end{pmatrix} \tag{4.6}$$

In fact, in Equation 4.6, the matrix is built up from individual components for each Euclidean axis. Thus, we can describe individual rotations around the x-axis and z-axis by the following two matrices:

$$\mathbf{R}_x = \frac{1}{1+u^2} \begin{pmatrix} 1+u^2 & 0 & 0 \\ 0 & 1-u^2 & -2u \\ 0 & 2u & 1-u^2 \end{pmatrix}$$
$$\mathbf{R}_z = \frac{1}{1+w^2} \begin{pmatrix} 1-w^2 & -2w & 0 \\ 2w & 1-w^2 & 0 \\ 0 & 0 & 1+w^2 \end{pmatrix}$$

These two matrices can be combined to a sequence of rotations $\mathbf{R}_{zx}$, that is, a rotation around the x-axis as described by $\mathbf{R}_x$ followed by a rotation around the z-axis as described by $\mathbf{R}_z$. This

combined rotation is described by

$$\mathbf{R}_{zx} = \frac{1}{(1+u^2)(1+w^2)} \begin{pmatrix} (1-w^2)(1+u^2) & -2w(1-u^2) & 4uw \\ 2w(1+u^2) & (1-w^2)(1-u^2) & -2u(1-w^2) \\ 0 & 2u(1+w^2) & (1+w^2)(1-u^2) \end{pmatrix} \tag{4.7}$$

Recall that we preconditioned the system such that $\mathbf{a}$ is orthogonal to $\mathbf{k}$. The rotation described by $\mathbf{R}_{zx}$ satisfies $\mathbf{k} \cdot \mathbf{R}_{zx}\mathbf{a} = 0$, thus this property is maintained. In other words, the first equation of Equation 4.2 is satisfied by $\mathbf{R}_{zx}$. Moreover, $\mathbf{R}_{zx}$ is described in terms of u and w only; hence we transformed a problem involving three non-linear equations in three variables to a simpler problem involving two non-linear equations in two variables.

4.3 Solving the system

We have now reduced the problem to a simpler form. We can rewrite this as a system of polynomial equations. In this section we discuss this last derivation and describe a number of degenerate cases and how we deal with them.

4.3.1 Bivariate polynomial system

We assume that no rotations of over π will occur, so the factor $\frac{1}{(1+u^2)(1+w^2)}$ in Equation 4.6 will be non-zero and thus will not influence the solutions other than by some scalar factor. For simplicity we therefore ignore this factor in the following. Using Equation 4.7 and substituting it in Equation 4.2, we get the two equations in u and w explicitly:

$$\begin{aligned} f(u,w) &= (-l_0b_0 + l_1b_1 - l_2b_2)u^2w^2 + (2l_1b_0 + 2l_0b_1)u^2w + (2l_1b_2 + 2l_2b_1)uw^2 \\ &\quad +(-l_1b_1 - l_2b_2 + l_0b_0)u^2 + 4l_0b_2uw + (l_2b_2 - l_0b_0 - l_1b_1)w^2 \\ &\quad +(-2l_1b_2 + 2l_2b_1)u + (2l_1b_0 - 2l_0b_1)w + (l_0b_0 + l_1b_1 + l_2b_2) \\ &= 0 \end{aligned} \tag{4.8}$$

$$\begin{aligned} g(u,w) &= (-m_0c_0 + m_1c_1 - m_2c_2)u^2w^2 + (2m_1c_0 + 2m_0c_1)u^2w + (2m_1c_2 + 2m_2c_1)uw^2 \\ &\quad +(-m_1c_1 - m_2c_2 + m_0c_0)u^2 + 4m_0c_2uw + (m_2c_2 - m_0c_0 - m_1c_1)w^2 \\ &\quad +(-2m_1c_2 + 2m_2c_1)u + (2m_1c_0 - 2m_0c_1)w + (m_0c_0 + m_1c_1 + m_2c_2) \\ &= 0 \end{aligned} \tag{4.9}$$

As can be seen, both (4.8) and (4.9) are polynomial equations in two variables. Thus, we have described our original orientation problem in terms of solving a system of two bivariate equations. This TVT-problem is a context for the solution method we will describe in Chapter 5.

Chapter 5

Root-selection using bipartite graphs

We have discussed the common methods for solving polynomial equations or systems of polynomial equations and we have extensively discussed a practical problem in which a system of multivariate polynomials needs to be solved. In this chapter we present our main result, an alternative approach to solving such systems. We will first discuss the general procedure and then present some variations. We will also present the results from experimenting with these methods.

5.1 Selecting roots

Given a system of two bivariate polynomial equations, we may use one of the discussed elimination methods (Gröbner bases or resultant methods) to compute a univariate polynomial of which the roots correspond to the roots for one of the two variables under discussion. Whereas in the normal solving strategy one such polynomial is computed, and its results back substituted in both polynomials, we propose a different tactic. We will compute two univariate polynomials, one for each variable, and use its roots to define a set of all possible roots. Of this set we will compute the optimal solution.

5.1.1 Root selection problem

Consider a system of polynomial equations:

$$f(x, y) = 0 \wedge g(x, y) = 0 \tag{5.1}$$

We can now compute two univariate polynomials, one for each variable. We thus have $h_1(x) = 0$ and $h_2(y) = 0$ such that the roots $x_1 \cdots x_n$ and $y_1 \cdots y_n$ correspond to the original system. These roots may be computed using well known methods for univariate solving. By exhaustively forming pairs, we thus have n^2 possible roots of the original system; each pair (x_i, y_j) is a valid candidate. However, from the degree of f and g we know that only n actual roots exist – the problem is now to select n pairs among the n^2 candidate pairs. In other words, for each x_i from the roots of $h_1(x) = 0$ we need to find some y_j among the roots of $h_2(y) = 0$ and construct a root (x_i, y_j) of Equation 5.1. There are, however, two conditions to be respected. First, not all candidate pairs represent roots of the original systems; some x_i's simply do not belong to a specific y_j. Second, for the total set of n such pairs, each root x_i and each root y_j must be used exactly once.

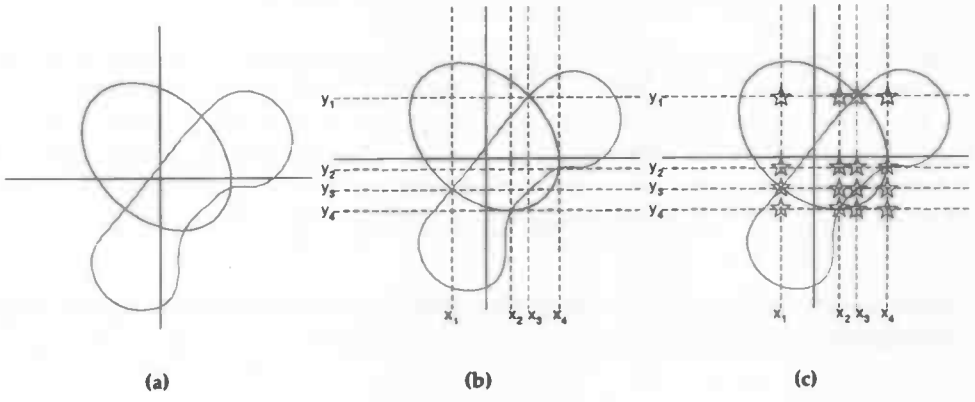


Figure 5.1: *Solving a system of polynomial equations using matching: in (a) two arbitrary polynomials are shown in a zero-plane implicit plot. In (b) it is shown how the individual x-roots and y-roots limit the possibile solution. In (c) it is shown that, nonetheless, n^2 possible solutions remain.*

The previous situation is shown in Figure 5.1. In the first image, the zero-plane plot of two arbitrary polynomials are shown. Of these we computed the two univariate resultant polynomials by elimination of either variable. In the second images, we overlayed the effect of the roots of $h_1(x)$ and $h_2(y)$: in the zero-plane they represent straight lines on which a root lies somewhere. More specifically, roots may lie only at intersections of the lines. As shown in the last image, there are generally n^2 such intersections of which only n form actual roots of Equation 5.1.

5.1.2 Minimising the total error

We propose the following method to solve this problem. We can guarantee that the minimum error solution is found in which each root x_i and each root y_j is used exactly once.

First, we create a bipartite graph G with the roots computed from the univariate polynomials at its vertices. More precisely, the roots $x_1 \cdots x_n$ of $h_1(x)$ are inserted into the X-bipartition of G. Analogously, the roots $y_1 \cdots y_n$ of $h_2(y)$ are inserted into the y bipartition. In such a graph, an edge between $x_i \in X$ and $y_j \in Y$ can be used to represent an assignment of x_i to y_j or, in other words, a potential root (x_i, y_j) of the original system of Equation 5.1.

As a preparation step, we represent the n^2 possible root pairs by connecting each x_i to each y_j – in essence, we create a complete bipartite graph G with two bipartitions of each n vertices and a total number of n^2 edges, as shown in Figure 5.2. As discussed in Chapter 2, we can now use a bipartite matching algorithm to compute an assignment of exactly n edges, such that each vertex is used exactly once. This would represent a possible solution to the original system, but we only fulfilled the second of the two conditions.

We require some measure of how suitable some pair is as a root of the original system. Straightforwardly, we can use back substitution of some pair in the functions f and g. To obtain an adequate error measure, we use the root mean square error:

$$E(x_i, y_j) = \sqrt{|f^2(x_i, y_j)| + |g^2(x_i, y_j)|} \tag{5.2}$$

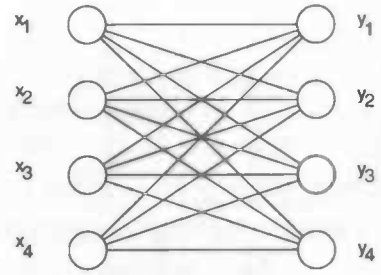


Figure 5.2: *An example of a bipartite graph representing all n^2 possible solutions. The edge weights are omitted for clarity.*

This can be incorporated into the bipartite matching as follows. Since an edge between $x_i \in X$ and $y_j \in Y$ represents a root (x_i, y_j), we can use the edge weight to specify the suitability of such a pair as a solution to Equation 5.1. We thus assign as the weight of an edge $e_{i,j} \in G$ the error $E(x_i, y_j)$. Finally, if we now compute a minimum-weight bipartite matching in G, we automatically select edges such that the total weight of the n solutions is minimised. This weight is $\sum_{i=1}^{n} E(x_i, y_j(x_i))$. By the very definition of matching, we automatically select each root only once. This situation is shown in Figure 5.3

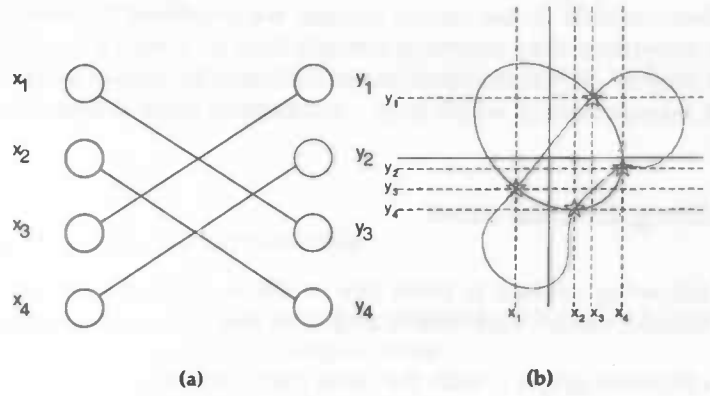


Figure 5.3: *The result of computing a matching in a bipartite graph. In (a) the effect in the graph is shown; here n edges are selected among the n^2 possibilities, such that their total weight is minimal. In (b) the effect in the zero-plane is shown; here n of the n^2 possible roots are selected, such that they together form the best solution in the sense of a minimised total error.*

We have now, finally, reached the essence of our research – this approach is an easy to implement and robust way of finding roots of bivariate polynomial equation systems. It guarantees that the best solution in terms of the minimum total error is found and that each individual root is used only once. This method has been published in [5]. In the next sections we will discuss some alternative methods based on the same ideas.

5.1.3 Minimising the maximum error

Perhaps in some context we are not so much interested in the minimum total error, but instead we would like the maximum error occurring in the final solution to be minimal. By slightly changing

the scheme we discussed so far, this can be easily achieved as follows.

Again, we start with the n^2 candidate solution pairs and, as before, we construct a bipartite graph G with two partitions X and Y. In X we insert the roots of $h_1(x)$ and in Y we insert the roots of $h_2(y)$. We also compute the error of each pair (x_i, y_j) as defined by Equation 5.2, but instead of assigning them directly as edge weights we store them in a list L. This list is sorted based on the error, such that the solution pair with the smallest error $L[1]$ comes first and the pair with the largest error $L[n^2]$ comes last. Next, we will assign edge weights based on the index in the list as follows.

The edge corresponding to the pair at $L[1]$ gets assigned edge weight 1. Each subsequent list item's edge gets assigned a weight based on the weights preceeding it; the weight will be the sum of previous edge plus one:

$$w(e_{ij}) = 1 + \sum_{k=0}^{m-1} e_{ij} \text{ where } e_{ij} \text{ corresponds to } L[m]$$

Thus, with increasing error, the weights become $1, 2, 4, 8, 16, \cdots$ until all edges have a weight assigned.

The matching procedure is left unchanged – that is, based on the edge weights the minimum-weight bipartite graph matching is computed. However, because each subsequent root is assigned an edge weight larger than the sum of all previous edge weights, solution pairs that have a smaller error are now always chosen in favour of pairs with a larger error. In [6] a proof of the optimality and correctness of this approach is given.

5.2 Example problem

In this section we work out an example problem step by step. We generate a problem at random and derive the two univariate polynomials $h_1(x)$ and $h_2(y)$. We then continue the solving process, as described in the previous section. We use roots computed by Mathematica for reference.

5.2.1 Problem

An instance of the TVT-problem generated at random by our test program is given by the following vectors (they are already preconditioned in the sense of Section 4.2.2):

$$\begin{array}{lllll} a & = \{ & 1.000, & 0.000, & 0.000 \} \\ b & = \{ & 0.795, & 0.607, & 0.000 \} \\ c & = \{ & 0.974, & 0.134, & 0.183 \} \\ k & = \{ & 0.000, & 0.000, & 1.000 \} \\ l & = \{ & 0.000, & 0.809, & 0.587 \} \\ m & = \{ & 0.732, & 0.630, & 0.259 \} \end{array}$$

The corresponding resultants $h_1(u)$ and $h_2(w)$ can be symbollically derived from Equation 4.8 and Equation 4.9. The values of the symbolical constants are substituted with these explicit values. The results are as follows:

$$
\begin{aligned}
h_1(u) &= -6.598 \times 10^{-02} - 1.006 \times 10^{-01}u - 1.212 \times 10^{-01}u^2 - 4.149 \times 10^{-01}u^3 \\
&\quad -2.129 \times 10^{-01}u^4 - 5.066 \times 10^{-01}u^5 - 7.761 \times 10^{-01}u^6 \\
&\quad -3.101 \times 10^{-02}u^7 - 5.671 \times 10^{-01}u^8 \\
h_2(w) &= 1.835 \times 10^{-01}5.206 \times 10^{-01}w2.345 \times 10^{-01}w^2 - 3.925 \times 10^{-01}w^3 \\
&\quad 2.273 \times 10^{-01}w^4 1.068 \times 10^{+00}w^5 - 3.539 \times 10^{-02}w^6 \\
&\quad -6.131 \times 10^{-01}w^7 1.797 \times 10^{-01}w^8
\end{aligned}
$$

This situation is shown in Figure 5.4.

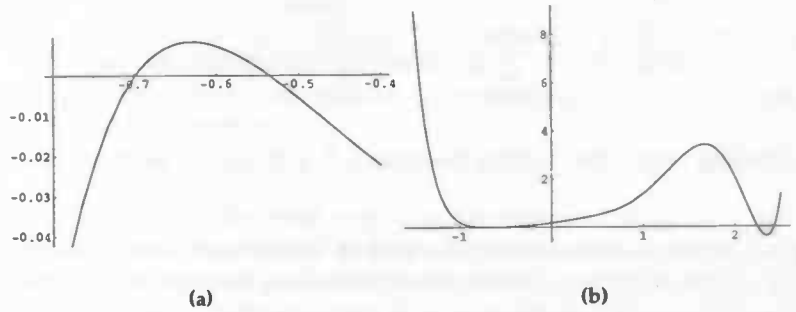


Figure 5.4: *The univariate polynomials derived for this particular problem instance. In (a) $h_1(u)$ is shown and in (b) $h_2(w)$ is shown.*

5.2.2 Solving univariate polynomials

We can let some algebraic computation package solve these univariate polynomials for us. Mathematica gives us the following real and complex roots:

u-roots		w-roots	
-0.699767		-0.69961	$-0.024372i$
-0.536136		-0.69961	$+0.024372i$
-0.033480	-0.69154	-0.54688	$-0.408868i$
-0.033480	$+0.69154i$	-0.54688	$+0.408868i$
$+0.188737$	$-1.07016i$	$+0.61927$	$-0.662472i$
$+0.188737$	$+1.07016i$	$+0.61927$	$+0.662472i$
$+0.435350$	$-0.59864i$	$+2.24400$	
$+0.435350$	$+0.59864i$	$+2.42194$	

At runtime, we would use an external library to find these individual roots. We used Laguerre's method from the *Numerical Recipes* [15] library. The computed individual roots are similar, only a bit less precise. Using these results we can compute the exhaustive list of combinations of x_i and y_j; this yields our $8 \times 8 = 64$ possible solutions to the original problem, of which 8 must be selected.

5.2.3 Total solution

For reference, we first use Mathematica to compute the solutions of the original problem directly. This yields us the following eight roots of $f(u,w) = 0 \wedge g(u,w) = 0$:

u		w	
-0.43535	$-0.59864i$	-0.69961	$+0.024372i$
-0.43535	$+0.59864i$	-0.69961	$-0.024372i$
-0.18874	$-1.07016i$	$+0.61927$	$-0.662472i$
-0.18874	$+1.07016i$	$+0.61927$	$+0.662472i$
$+0.03348$	$-0.69154i$	-0.54688	$+0.408868i$
$+0.03348$	$+0.69154i$	-0.54688	$-0.408868i$
$+0.53614$		$+2.24401$	
$+0.69977$		$+2.42194$	

We apply the methods proposed in the previous section to calculate solutions at runtime. Note that we compute how many roots actualy apply to the current TVT-problem instance and discard the other roots. The skipped roots are computed however, and in other applications they may be used. Moreover, we truncate all results to real-valued roots. In this particular case, the results for the two methods coincide. The program yields the following result.

u	w	ε
-0.6997667	2.421942	3.324101×10^{-12}
-0.5361356	2.244010	5.160439×10^{-12}
Total error:	8.484540×10^{-12}	

This is the solution that has the minimum total error. In this case, it is an ordinary error of $\approx 10^{-12}$; by using more sophisticated univariate solving strategies, one may optimise this value. Therefore, in the following we will not study the precision of the roots – this precision is not a characteristic of our approach but of the underlying solving technique used.

5.3 Results

We performed various experiments to test our methods in differing situations. We looked at how well the algorithms perform in terms of finding all roots in the complex plane and how long it takes the program to calculate these results. We used the TVT-problem from Chapter 4 to test our approach, and compared our method to the runtime library Synaps [11].

5.3.1 Peformance

We experimented with various problem types within the context of the TVT-problem to obtain a reasonable measure of the performance of the root selection procedure we proposed. By performance we mean a measure of how good a method performs on a problem, that is, if it correctly computes the roots. This is distinct from accuracy – as our method simply selects roots computed by some other, univariate, solving method the accuracy depends on the chosen method and not on the matching procedure.

For various initial orientations of the two objects to be compared in the TVT-problem, degenerate configurations occur which are harder to solve. We generated random problems of varying degeneracies and measured the performance. Here, we find that in the more degenerate cases, ordinary solving techniques such as implemented in Synaps fail to compute the roots whereas our method always computes the roots. In general over 10% of the roots was missed by the Synaps package. In Table 5.1 the errors of the various methods are shown.

total error min.		maximum error min.		Synaps	
average	max	average	max	average	max
8.2388E-011	1.4557E-009	8.2388E-011	1.4557E-009	4.4288E-001	6.9353E+000
7.4460E-017	5.4873E-016	7.4460E-017	5.4873E-016	3.2078E+261	2.8549E+263
3.8511E-011	7.0850E-010	3.8511E-011	7.0850E-010	6.4918E-001	5.6151E+000
1.4672E-003	2.5676E-002	1.4672E-003	2.5676E-002	4.2465E+000	1.2245E+001
2.0264E-010	4.2725E-009	2.0264E-010	4.2725E-009	3.5758E+000	8.3778E+000
1.3765E+001	9.8000E+001	6.5319E-004	2.3914E-002	1.0720E+000	1.9311E+001

Table 5.1: *The accuracy of the various methods for increasingly degenerate problems. Each error is the average of multiple experiments.*

As can be seen from the table, the total error minimisation and maximum error minimisation approaches work well. However, Synaps' averaged performance turns out relatively bleak, since sometimes no roots are computed at all. In other, less degenerate, cases, Synaps may compute the roots correctly.

5.3.2 Running times

In order to make a quantative analysis of the running times, we let each method under investigation run for a large number of cases. We subjected our three variations of the previous section to this test, as well as the runtime library Synaps. For the tests we included the preparation steps for each algorithm in the measurements, such as building the graph or sorting a list of possible solutions. However, we left out the computations related to any specific problem such as the TVT-problem. The results are shown in Table 5.2.

Method	10^4 cases	(per case)	10^5 cases	(per case)
Total error minimisation	1.56s	0.156ms	13.65s	0.137ms
Maximum error minimisation	3.84s	0.384ms	35.34s	0.353ms
Synaps	51.3s	5.13ms	475.40s	4.75ms

Table 5.2: *Comparison of runtimes of the various methods under investigation.*

5.3.3 Using complex numbers

We can choose at which point to truncate our values from complex to real representation, if desired at all. We can truncate all roots found by univariate solving to real values, evaluate the error and make a selection subsequently. Alternatively, we can compute the optimal selection using complex numbers all along, and of the final selection discard the complex parts.

From our experiments, we can deduce that the last approach works best. Because the errors in the graph are numerically in the same range, it ensures that no roots get discarded because a

small complex part is included. Moreover, this approach allows the end user to choose which roots to keep and whether or not to truncate complex roots to real-valued roots.

Chapter 6

Extension to multiple variables

In the previous, we discussed how a bipartite graph and the minimum-weight bipartite graph matching algorithm can be used to compute the best root selection among the individual roots of bivariate polynomials. However, no multi-dimensional generalization of bipartite graphs exists. To make it possible to apply the proposed combinatorial approach to multivariate polynomials, we need to regard the problem from a different perspective. In this chapter, we will discuss linear assignment problems and present a new algorithm for solving such problems. We will end the chapter by showing how this approach can be applied to the root selection problem.

6.1 Linear assignment problems

In the bipartite graph matching problem, we computed the best matching of n items from some category X to n items of some other category Y. The cost of an assignment (x_i, y_j), with $x_i \in X$ and $y_j \in Y$ was known in advance. For example, when assigning employees to tasks, we computed a minimum-weight assignment in the sense that the best suited employees were assigned to the most appropriate tasks. The total cost of the assignment equaled the sum of the cost of all involved individual assignments. Such problems, where the category sizes are equal and the total cost equals the sum of all individual costs, are called linear assignment problems (LAP).

6.1.1 The linear assignment problem

We can make the definition of the LAP more precise as follows. The cost of each assignment can be represented in one matrix C, known generally as the cost matrix. In such an $n \times n$ matrix, each element $C[i, j]$ represents the cost of an assignment (x_i, y_j), with $x_i \in X$ and $y_j \in Y$. An assignment in C – which is a collection of n such individual assignments – can then be represented by a matrix A. Such an assignment matrix contains only 0's and 1's: a 1 at A[i,j] indicates the selection of the corresponding element $C[i, j]$ in C and a 0 indicates no selection.

Since we know the cost of any individual assignment from C, we can compute the cost of an assignment A with respect to C. This is given by the cost function $\text{cost}(C, A)$.

$$\text{cost}(C, A) = \sum_{i=1}^{n} \sum_{j=1}^{n} A[i, j]C[i, j]$$

Recall that we required that in each column and each row exactly one individual assignment must be made, respectively. In terms of an assignment matrix A, we require that $\sum_{i=1}^{n} A[i,j] = 1$ for all j and that $\sum_{j=1}^{n} A[i,j] = 1$ for all i. Hence $\sum_{i=1}^{n}\sum_{j=1}^{n} A[i,j] = n$. This condition is called the feasibility of an assignment A.

Definition 6.1. *Feasiblity of LAP*: an assignment A is feasible in the sense of the LAP if in each column and each row, respectively, exactly one 1 occurs.

An instance of the LAP can be given by a cost matrix C, as we saw. We can imagine the set of all possible feasible solutions to some LAP instance C; this set is denoted $\mathbb{P}$. Assuming that a solution exists, thus $\mathbb{P} \neq \emptyset$, we can explicitly formulate the LAP.

Definition 6.2. *Linear assignment problem (LAP)*: the LAP is the problem of finding the optimal assignment A^*:

$$A^* \in \arg\min \text{cost}(C, A) \quad \text{with } A \in \mathbb{P}$$

For a more detailed discussion of the LAP and further reading, see [9].

6.1.2 Solving the LAP

The linear assignment problem is well known in the field of combinatorial optimization, and many efficient algorithms for solving it exist. However, the solution cannot be computed straightforwardly or in a greedy fashion. When the problem size n increases, performance is of concern in solution algorithms. All current algorithms are based on the shortest augmenting path method, the implementation of which is based on the Köning-Egervary Theorem [9]. These algorithms all have a time complexity of $\mathcal{O}(n^3)$.

In this chapter, we want to propose a new method which is not based on the shortest augmenting path method; this method works directly on the cost matrix. To motivate our method, we need to introduce two concepts first: a relaxed version of the problem and the notion of a reduced cost matrix. These concepts are discussed in the next few paragraphs. We will subsequently describe a class of algorithms, branch-and-bound algorithms, on which our method is based. After that thorough introduction, we will present our method and describe how it can be used to compute the best selection of roots.

6.1.3 The relaxed LAP

We now consider a relaxed version of the linear assignment problem, known as the relaxed linear assignment problem (RLAP). Suppose that in the task assignment problem, we would allow multiple employees to perform a specific huge task. For example, three workers set out to work through a stack of paperwork. The total amount of work does not change and each worker can only do a part of the work. Hence the cost of assigning these workers to the same task is the sum of assigning each worker individually. The cost function as defined for the LAP can remain unchanged.

However, the feasibility of an assignment in the RLAP is changed. We relieve the restriction on the number of individual assignments in one column. In terms of an assignment matrix A, we still require that $\sum_{i=1}^{n} A[i,j] = 1$ for all j but not that $\sum_{j=1}^{n} A[i,j] = 1$ for all i. This more relaxed condition is called the feasibility of an assignment A in sense of the RLAP.

Definition 6.3. *Feasiblity of RLAP*: an assignment A is feasible in the sense of the RLAP if in each row exactly one 1 occurs.

As with the LAP, we can think of the set of all feasible solutions to some instance C of RLAP. This set is denoted $\mathbb{Q}$ and again we assume that $\mathbb{Q} \neq \emptyset$. Then the RLAP can be defined as follows.

Definition 6.4. *Relaxed linear assignment problem (RLAP)*: the RLAP is the problem of finding the optimal assignment A^*:

$$A^* \in \arg\min \text{cost}(C, A) \quad A \in \mathbb{Q}$$

Because the RLAP is a more relaxed version of the LAP, it clearly holds that $\mathbb{P} \subseteq \mathbb{Q}$. Thus, any solution to the LAP is also a solution to the RLAP, but not necessarily vice versa. However, some solutions of the RLAP may also be solutions to the LAP. Because the RLAP is more relaxed than the LAP, the cost of a solution $A \in \mathbb{P}$ and thus $A \in \mathbb{Q}$ forms a lower bound to the optimal solution A^* of the LAP. Thus, with $A \in \mathbb{Q}$ and $A^* \in \mathbb{P}$ for some LAP instance C, it always holds that $\text{cost}(C, A) \leq \text{cost}(C, A^*)$. This property proves to be useful for using algorithms based on the branch-and-bound paradigm, discussed in the next section.

From hereon in, we will assume that the problem instance C is known from the context. Therefore we will write shortly $w(A)$ instead of $\text{cost}(C, A)$ to indicate the cost, or more generally, the weight of an assignment A.

6.1.4 Reduced cost matrix

Until now, we used an assignment matrix to indicate which individual assignments are made. This was done by setting the corresponding elements $A[i, j]$ to 1 in A. We can, however, employ a more efficient and more sophisticated scheme, where the assignments are stored *within* the cost matrix.

According to [12], subtracting a constant value c from all elements in a row or a column of a cost matrix C does not change the optimal assignment. Thus we may subtract constant values, as long as it is done per row or per column. The minimum element, computed per column and per row respectively, is a constant value and we thus may apply this theorem. By subtracting the minimum element row-wise or column-wise, we achieve $C[i, j] = 0$ whenever $C[i, j]$ equals the minimum element.

We opt for subtracting row-wise. If in each row the minimum element is computed, and subsequently subtracted from all elements in that row, we know that in each row at least one zero exists. The resulting matrix is called the reduced matrix C_r of C. We can use $C_r[i, j] = 0$ instead of $A[i, j] = 1$ to indicate an assignment (x_i, y_j).

Note that this is not of conceptual use only; when large problems are being solved, it may reduce the memory requirements of the program by large factors. Using this scheme, the algorithm may operate directly on the cost matrix instead of storing a sparse assignment matrix of the same dimensions as the cost matrix.

Alternatively this can be seen in graph form, like we did in the previous chapters. The cost matrix C can be seen as a bipartite graph $G(V_1 \bigcup V_2, E)$. The vertices in V_1 represent the rows in C and the vertices in V_2 represent the columns in C. An edge $e_{i,j} \in E$ then corresponds to the element $C[i, j]$ of the cost matrix. A matching M in G is then analogous to the reduced cost matrix, with $e_{i,j} \in M$ whenever $C_r[i, j] = 0$.

6.2 Branch-and-bound algorithms

Most of the solution methods to the LAP can be found in the class of branch-and-bound algorithms. These algorithms work by dynamically creating a search tree of possible solutions. This is done by dividing problems into sub problems (branch) and then limiting which sub problems should be worked out and which can be pruned from the tree (bound). The search three is thereby created on the fly, introducing branches and chopping other branches where necessary. By imposing certain conditions on problems and their sub problems, it can be guaranteed that eventually the solution found is the optimal solution to the original problem. In this section we describe a general and typical branch-and-bound approach. In the next section, we use this to motivate the tolerance-based algorithm for LAP, which is based on the branch-and-bound paradigm.

6.2.1 Requirements for branch-and-bound

Branch-and-bound algorithms cannot be used on every problem; there are certain requirements that need to be met in order for branch-and-bound to be applicable. Because in the search tree we need an intermediate representation of a solution, we require the definition of a relaxed version of the problem. Thus, in the search tree, some nodes will represent possible solutions to the original problem – and some of those solutions will be optimal ones – and some other nodes will represent relaxed solutions, which are not applicable to the original problem. However, because of the way we branch problems into sub problems, eventually relaxed problems will generate a non-relaxed solution.

Another requirement is that the cost of a (relaxed) solution found in some sub problem is always at least the cost of its parent. That is, if some node has a relaxed solution S and we branch until a non-relaxed solution S' to the original problem is found, the cost may not be below the cost of that initial solution; or $S' \geq S$. This allows us to prune the tree at some points, knowing that we do not discard any optimal solutions.

There are three key concepts to branch-and-bound algorithms. They are branching and bounding, obviously, and back tracking. They are discussed in the next three sections.

6.2.2 Branching

Branching is the process of dividing some problem P into a number of sub problems $P_1 \cdots P_n$. At first, the original problem or root problem is divided and later on other sub problems may be divided to create sub sub problems and so on. There are two requirements for branching.

First of all, the solutions of the sub problems must form solutions to the parent problem as well. This is shown in Figure 6.1(a). If we use the tilde to describe the 'solved-by' relationship and $P_i \sim s_i$ then it must hold that $P \sim s_i$ for all i. This ensures that, when a sub branch of problems is worked out to form some solution, we know that we have found a solution to the initial problem as well. It is however not certain the optimal solution is found.

The second requirement is that the generated sub problems cover the parent problem. This is shown in Figure 6.1(b). If we solve all sub problems, and compute the weights $w(s_i)$, we must be sure that the minimal solution s_i^* is the minimal solution to the parent problem P. This can be seen as sub problems covering the parent problem, because it requires that there is not some other minimal solution to the parent problem which is not included in the sub problem solutions.

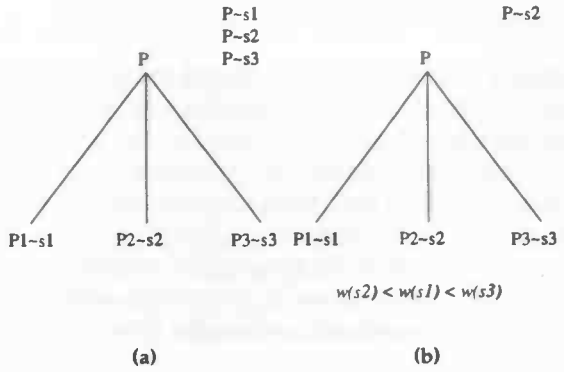


Figure 6.1: *Requirements for branching: (a) each solution to a sub problem is also a solution to its parent problem and (b) the minimum solution of all sub problems is the minimum solution to the parent problem (sub problems cover the parent problem).*

The weight of each sub problem may be used as an indication of which sub problem to traverse into next: the problem with the lowest weight may potentially hold better solutions than the others. However, this is a rule of thumb only – we may need to reconsider this choice and work out the other problems as well.

Branching can be done only on relaxed problems, thereby making it less relaxed. This can continue until a non-relaxed or feasible solution to the original problem is found. Non-relaxed problems cannot be branched further, but there is no need for it either.

6.2.3 Bounding

Using branching, we could unfold the whole search tree and then select the minimal feasible solution. However, usually the problems are complicated and the search tree large. Generally it is infeasible to unfold the whole tree. Using bounding, we can prune the tree at certain points and prevent working out whole branches. Only branches of which we are sure that they do not contain the optimal feasible solution may be pruned, while preserving optimality.

Suppose we have traversed the tree using branching and at some point encountered a feasible solution S to the initial problem. We can compute its weight $w(S)$. We have to search further (using back tracking, explained next) in order to ensure that the optimal solution is found, eventually. During the further search, we process problems differently. We can compute the weight of a relaxed problem P, it is $w(P)$. If it the weight of $w(P) \geq w(S)$, we know from the second branch-and-bound requirement that traversing into P will not generate a feasible solution that is better than S. Therefore we can avoid traversing into P at all. The whole branch starting at P is pruned beforehand, thereby saving the generation and computation of many sub problems.

6.2.4 Backtracking

Suppose we have followed the previous branching approach and we have found some feasible solution S to the original problem. We are not yet certain that S is the optimal solution: at some point in the tree a sub problem P may exist that is not yet worked out, but whose weight $w(P)$

is below the weight $w(S)$ of S. We need to track back to sub problems that, potentially, lead to better feasible solutions.

Many approaches exist for tracking back through the search tree; we may for example choose to use a last-in-first-out (LIFO) order. This means we must first consider all the sibling problems of S and, if a sub problem exists whose weight is smaller than $w(S)$, we must work this problem out by traversing into it. Subsequently, unprocessed nodes further up in the tree may need to be worked out.

If at some point during back tracking we encounter an alternative feasible solution S', with $w(S') < w(S)$, we update the current best solution $S = S'$ and continue back tracking. Back tracking stops when we either have discarded all unsolved sub problems or when we have no sub problems left. The current best solution is, in both cases, guaranteed to be the optimal solution S^* to the initial problem, as can be deduced from the weights.

So, in general a typical branch-and-bound algorithm proceeds as follows for a non-relaxed problem P. First, the problem is subdivided into a number of sub problems. Of these sub problems, we traverse into the problem that has the smallest weight i.e. which looks best from our current viewpoint. This is continued until a feasible solution S is found, which is stored as the current best solution. Then back tracking starts, during which unsolved problems are considered and entered when required. Eventually, the optimal solution S^* will be generated.

6.3 The tolerance-based algorithm for LAP

We now have the required concepts in place to present a different algorithm for solving the LAP. This algorithm is not based on shortest augmenting paths, as all other known algorithms, but rather makes use of tolerances. Hence the name, tolerance-based algorithm (TBA). In general, the algorithm uses the concept of the RLAP to initiate the branch-and-bound search. Then, using tolerances on the weights of the current reduced cost matrix under which the optimality is preserved, we try to converge towards solutions that are feasible to LAP as well. The branch-and-bound paradigm is used to guide the search.

We will first discuss the key concepts of the TBA for LAP, and afterwards present the outline of the algorithm. In the next section, it is discussed how the TBA can be applied to selecting the best roots. In the following we will alternate between the matrix representation and the graph representation, in order to make the transition from bipartite graphs to the LAP as clear as possible.

6.3.1 Column labels

An assignment which is feasible in the sense of the LAP, i.e. some $C_r \in \mathbb{P}$, has per column only one assignment. We could call such columns *served*, indicating that the condition for LAP-feasibility is met. In an assignment feasible in the sense of the RLAP only, columns that are not served may occur. Put loosely, the more the number of assignments in a column deviates from one, the more relaxed the problem is.

Columns that are not served exist in two types: no assignment at all is made in that column or multiple assignments are made in that column. We call the first type *underserved*, since some individual assignment(s) must be inserted into this column in order for this assignment to be feasible in the sense of LAP. The second type is called *overserved*, since some individual assignment(s)

must be removed from this column. By assigning such column labels, we can further decide what to do in order to tighten the relaxedness and converge towards a solution of LAP.

The notion of served, overserved and underserved columns can be made more explicit by using the graph representation. The labels correspond to the values of the in-degree $\mathrm{id}(j)$ of vertices $j \in V_2$. In served columns, the in-degree is exactly one. In an underserved column j the in-degree $\mathrm{id}(j) = 0$ and in an overserved column j' the in-degree $\mathrm{id}(j') > 1$. We want to find a solution to the LAP ultimately, and we thus require that $\mathrm{id}(j) = 1$ for all vertices $j \in V_2$. This corresponds exactly to the feasibility definition of the LAP in Definition 6.1. We thus have found a way of describing the 'relaxedness' of an assignment: the more vertices $j \in V_2$ have in-degree $\mathrm{id}(j) = 1$, the less relaxed the problem is. Finally, when $\sum \mathrm{id}(j) = n$ and $\mathrm{id}(j) = 1$ for all $j \in V_2$, the solution is not relaxed at all – it is a solution feasible to LAP.

6.3.2 Lower bound

We want to change assignments, in order to restore overserved and underserved columns to served columns. However, in the spirit of branch-and-bound, we need that the optimality of the solution is preserved. This can be done by the notion of upper and lower tolerances, which will be explained next. In this paragraph we discuss the lower bound for underserved columns; in the next paragraph the upper bound for overserved columns is introduced.

In an underserved column, no individual assignment was made. The reduced cost matrix thus has no zero in that column. In order to achieve a LAP-feasible assignment, this column must have one element selected. The element that can be selected with the least increase in cost, is of course the smallest element in that column. Put loosely, $i^* = \arg\min C[i, j]$ is called the lower bound of the underserved column j. This is shown in Figure 6.2.

In graph form, this can be seen as follows. We define the maximum increase of the weight of the individual assignments in one column j under which the assignment remains optimal, as the upper tolerance $u(e_{i,j})$. Consider an assignment $C'_{\alpha,a,b}$ which is much like C except for one element $C[a, b]$. It is defined as

$$C'[a, b] = C[i, j] + \alpha$$

and

$$C'[i, j] = C[i, j] \quad \text{with } i \neq a \text{ and } j \neq b$$

The upper tolerance is then the maximum value of α for which the optimality is preserved, thus:

$$u(e_{i,j}) = \max\{\alpha \geq 0 : C_{\alpha,i,j} \in \arg\min \{w(C) | C \in \mathbb{Q}\}\}$$

6.3.3 Lower bound

In an overserved column, too many assignments are made. The reduced cost matrix has multiple zeroes in that column. In order to achieve a LAP-feasible assignment, this column must have only one element selected. The best we can do is, per assignment in this column, see what alternatives there are. More explicitly, for each row i for which $C[i, j] = 0$ in this column j, look at the other elements $C[i, k]$ with $k = 1, \cdots, n$ and $k \neq j$. Of these elements, compute the smallest one – this is the best alternative in the sense that it removes a selection from column j and increases the weight of the assignment least. This is shown in Figure 6.2(b).

The previous scheme, however, removes only one individual assignment violating the LAP-feasibility condition. Since the column is overserved, multiple such assignments may violate the

condition. In such cases, we can do better by elminating them all at once. This is done straightforwardly by looking at all the alternative elements, and summing all but the greatest one. The greatest alternative increases the weight most, and hence is the worst change we can make from this point – it is for now best to leave that selection in this column.

Again, this can be made explicit by using the graph formalism. Analogous to the the upper tolerance, we can define the lower tolerance $l(e_{i,j})$ as follows:

$$l(e_{i,j}) = \max\{\alpha \geq 0 : C_{-\alpha,i,j} \in \arg\min \{w(C)|C \in \mathbb{Q}\}\}$$

The summed up value of lower tolerances is called the lower bound for an overserved column.

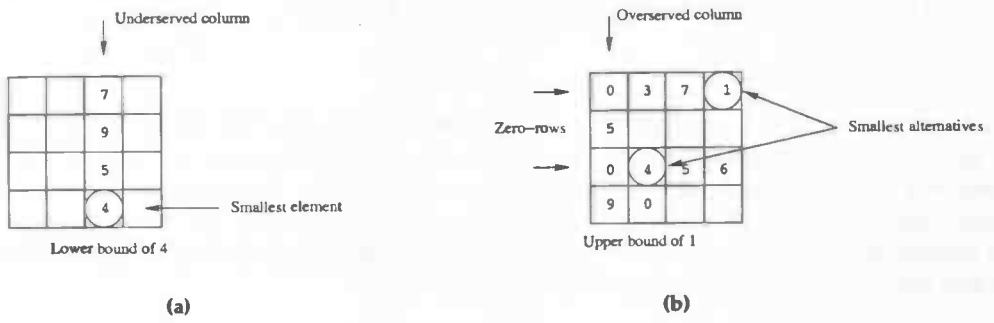


Figure 6.2: *Computation of upper and lower bounds. In (a) an underserved column is higlighted. Of this, the smallest element is taken and chosen as the lower bound. In (b) an overserved column is highlighted. Of this, the smallest alternative element for a row in which the zero lies in the selected column is chosen as the upper bound. For convenience, all irrelevant elements are omitted.*

6.3.4 Outline of the algorithm

The TBA, now, works as follows. Starting with a problem instance C we first compute an initial assignment by computing the reduced cost matrix C_r. Recall from the definition of the reduced cost matrix, that $C_r \in \mathbb{Q}$ i.e. C_r is feasible in the sense of RLAP. The weight of this assignment is $w(C_r)$. By looking at the number of individual assignments per column, we can determine how 'relaxed' this assignment is – this is done by assigning to each column one of the labels *served*, *underserved* or *overserved*. For underserved columns, we compute and store the upper bound and for overserved columns we compute and store the lower bound. Of these bounds, we compute the minimum and the element that generated this minimum is called the *bottleneck* of the current problem.

In coherence with the branch-and-bound paradigm, we can now create two sub problems that are less relaxed. The left sub problem C_{left} of a given problem C is computed by eliminating the bottleneck element from C; this can be done by setting its cost to infinity. Effectively, this ensures that in all sub problems of C_{left} the bottleneck element is never selected. The right sub problem C_{right} of C is computed by eliminating the elements sharing a column or row with the bottleneck, but not the bottleneck itself, by setting their costs to infinity. Effictively, this ensures that in all sub problems of C_{right} the bottleneck element is always selected. This is shown in Figure 6.3.

According to branch-and-bound, we must now choose one of the sub problems to process further. It was suggested that the sub problem with the smallest weight should be traversed first. In

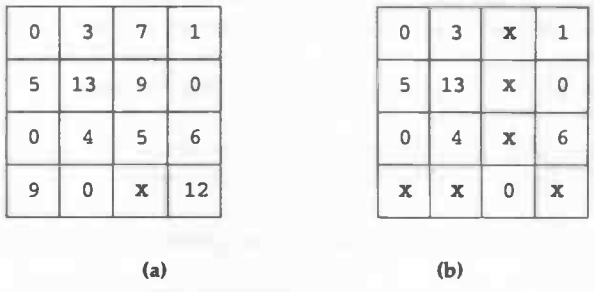


Figure 6.3: *Sub problems are created based on the bottleneck. In (a) the bottleneck element is rejected for further use whereas in (b) it is obligatory for future use.*

the TBA, we choose which problem to work out based on whether the bottleneck was attained on an upper bound or a lower bound. If attained on an upper bound, we branch into the left sub problem in which the bottleneck element was eliminated. If attained on a lower bound, we branch into the right sub problem in which the bottleneck element was made obligatory. The newly generated matrices may not be feasible even in the sense of RLAP. However, this condition can at any point be restored by computing a reduced matrix.

We can continue branching in this way, until a solution that is feasible in the sense of LAP is found. Since at each step we add an assignment to an underserved column or remove assignments from an overserved column, eventually such a solution will always be found.

Suppose the first LAP-feasible solution found is C. Then we must traverse all unsolved sub problems C_i geneated in the tree so far, and check whether $w(C) \leq w(C_i)$. If so, the sub problem branch is pruned, i.e. not worked out. If not, we must traverse into C_i and evaluate until either C_i (or one of its sub problems) achieves a weight W with $W \geq w(C)$ or the sub problem becomes feasible to LAP with a weight $W < w(C)$. In the latter case, we update our best solution $C = C_i$ and continue back tracking. Eventually, since each branch will lead to a LAP-feasible solution or will be pruned beforehand, there will be no more sub problems to be worked out. According to branch-and-bound, we can be certain that the current best solution C will then be the optimal solution C^* to the original problem.

6.3.5 Pseudo-code for TBA

To give a perhaps more intuitive overview of the tolerance-based algorithm for LAP, we present a pseudo-code listing. Only the main branching routine is shown; the other sub routines are assumed to be clear. Also, the back tracking procedure is omitted. It can be defined straightforwardly by using the Branch routine and the list of unsolved sub problems.

Procedure S = **Branch(** C **)**
Input: an $n \times n$ cost matrix C.
Output: the first feasible solution that is derived from C.

```
{Reduce the cost matrix and check for LAP-feasibility.}
C_r = Reduce( C )
if LAP-Feasible( C_r ) then
  Return C_r
end if
```

```
{Compute column labels and the bottleneck}
L = LabelColumns( C_r )
for all columns j in C_r with L(j) == underserved do
  B(j) = ComputeUpperBound( C_r[j] )
end for
for all columns j in C_r with L(j) == overserved do
  B(j) = ComputeLowerBound( C_r[j] )
end for
bottleneck = arg min {B(j) : j = 1, 2, ··· , n}
{Create sub problems and branch}
C_left = CreateLeftSubProblem( C_r, bottleneck )
C_right = CreateRightSubProblem( C_r, bottleneck )
if L(bottleneck) == underserved then
  Return Branch( C_left )
else
  Return Branch( C_right )
end if
```

6.4 Using the TBA for root selection

In the previous sections we presented an alternative approach to selecting roots of polynomial equations, by introducing the linear assignment problem and the tolerance-based algorithm for computing solutions to it. In this section we will briefly discuss how the TBA may be used for multivariate root selection.

6.4.1 Bivariate root selection using TBA

In the discussion of TBA, we already frequently used the graph representation in order to keep the discussion coherent with bipartite graph matching. Therefore, it is almost straightforward how TBA can be applied to selecting the best roots of bivariate polynomial equations. As usual, we compuqte the univariate polynomial equations $h_1(x) = 0$ and $h_2(y) = 0$ and solve them, thereby obtaining the root sequences $x_1 \cdots x_n$ and $y_1 \cdots y_n$. But instead of a bipartite graph, we now construct a $n \times n$ matrix C to represent the problem. Whereas in the bipartite graph the edge weights were used to store the 'cost' of selecting a specific root, now we store these error values in the matrix. Thus, each element $C[i, j]$ is initialized according to

$$C[i, j] = \sqrt{|f^2(x_i, y_j)| + |g^2(x_i, y_j)|}$$

Then, TBA can be applied straightforwardly to find the optimal matching. This yields a matrix representing this optimal matching C^*. The n roots that form the best solution to the polynomial system are those pairs (x_i, y_j) for which $C^*[i, j] = 0$. The error in this selection is given by $\sum C[i, j]$ where $C^*[i, j] = 0$.

6.4.2 Results

We have implemented and tested the TBA algorithm in the context of TVT-problem. With respect to the bipartite graph algorithm proposed initially, the TBA algorithm does not change the performance in terms of the number of detected roots. The TBA algorithm solves the same problem,

only in a different way, and therefore all roots are still found. However, this approach has the potential advantage of allowing multiple variables to be used with this method.

We have tested the TBA algorithm on the same cases as we used for the analysis of the methods in Chapter 5. It turns out that, in these cases, the TBA algorithm is a lot slower than the bipartite matching approach. One case takes much less than a second to solve, whereas ten cases take slightly over a second to solve. However, to solve about a hundred cases the TBA algorithm takes nearly twenty seconds – in the same amount of time, the bipartite matching algorithm solves 10^5 cases easily. The TBA algorithm is still faster than Synaps.

The relatively high running times may be due to the high memory-dependent nature of TBA. Certain optimizations could be made if known in advance that multiple problems need to be solved. In such a situation, the data structures may be left intact and reused for the next problem, thereby saving the allocation and deallocation times. Also, more efficient data structures could be used to reduce memory cost. However, in this research we focused on the basic use of TBA and we have not concerned ourselves much with efficiency.

6.4.3 Multivariate root selection with TBA

With the TBA algorithm, our approach of selecting roots using combinatorial optimization can be extended to systems of multivariate polynomials. As an example, we shortly discuss how an extension to three dimensions can be made. Generalization to higher dimensions is then straightforward.

In three dimensions, the otherwise two-dimensional cost matrix becomes a three-dimensional $n \times n \times n$ cube of costs. The feasibility condition must now hold for each column, each row and also each line in the cube. Thus, in each row i only one selection in $C[i, j, k]$ may be made. Similarly for each column j only one selection in $C[i, j, k]$ and for each line k also only selection in $C[i, j, k]$.

The TBA algorithm can now be applied, by using the upper and lower bounds to move assignments from violated columns, rows and lines. This may be done by applying TBA iteratively in two-dimensional sub problems. However, the exact approach needs to be experimented with in future research.

Conceptually, this approach allows the method we present in this thesis to be applied to systems of polynomial equations with any number of variables. However, the time complexity limits the number of variables for which the approach is feasible. From our experiments we expect that systems with up to eight variables can be solved in a reasonable amount of time, but for a detailed analysis further research is required.

Chapter 7

Conclusion

In the previous chapters we discussed a new method for selecting the roots of systems of polynomial equations. We have shown how the basic approach works, using the bipartite matching algorithm, for two variables. In the last chaper we have discussed how this method may be extended to multi-dimensional cases, by using the TBA-algorithm. We tested both approaches in practice, by applying them to the TVT-problem of Chapter 4, and found that they outperform methods currently available. We may conclude that, when robustness is required in terms of the number of roots returned, especially in degenerate cases, one may be advised to use the approaches presented in this thesis for selecting roots.

Although we have shown how the approach may be modified to support the extension to multiple variables, we have not tested this approach yet for $n > 2$. This is one of the main directions in future research. Moreover, the maximum error minimisation alternative may prove to contain a number of interesting properties. These may be exploited to reduce the number of branching steps in TBA, and thereby provide an upper bound for its running time.

Bibliography

[1] Abel, N.H. (1826): 'Beweis der unmögklichkeit algebraische gleichungen von höheren graden als dem vierten allgemein aufzulösen'. *Journal für die reine und angewandte mathematik*, vol. 65.

[2] Abramowitz, M., Stegun, I.A. *eds.* (1972), *Handbook of mathematical functions with formulas, graphs and mathematical tables*, U.S. Department of Commerce. (ISBN: 1-59124-217-7)

[3] Allgower, E.L., Georg K., Miranda, R. (1992): 'The methods of resultants for computing real solutions of polynomial systems'. *SIAM Journal on Numerical Analysis*, vol. 29, no. 3, pp. 831-844.

[4] Bekker, H. (2005): 'Calculating the rigid rotations that move three vectors orthogonal to three fixed vectors'. In preparation.

[5] Bekker, H., Braad, E.P., Goldgengorin, B. (2005): 'Using bipartite and multidimensional matching to select the roots of a system of polynomial equations'. *International Conference on Computational Science and its Applications* (ICCSA 2005). Forthcoming in the conference proceedings.

[6] Bekker, H., Braad, E.P., Goldengorin, B. (2005): 'Selecting the roots of a small system of polynomial equations by tolerance based matching'. *International Workshop on Efficient and Experimental Algorithms* (WEA 2005). Forthcoming in *Lecture Notes in Computer Science*.

[7] Brink, A.A. (2004): *Sppeeding up the computation of similarity measures based on Minkowski addition*. M.Sc. Thesis, Department of Mathematics and Computing Science, The University of Groningen.

[8] Buchberger, B. (1976): 'Theoretical basis for the reduction of polynomials to canonical forms'. *SIGSAM Bulletin*, vol. 39, pp. 19-24.

[9] Burkard, R.E. (2002): 'Selected topics on assignment problems'. *Discrete Applied Mathematics*, vol. 123, no. 1-3, pp. 257-302.

[10] Courant, R., Robbins, H. (1996): 'The fundamental theorem of algebra'. *What is mathematics? An elementary approach to ideas and methods*, Oxford University Press, pp. 101-103.

[11] Dos, G. *et al.* (2002): 'An environment for symbolic and numeric computation'. *Proceedings of the International Conference on Mathematical Software*, World Scientific, pp. 239-249. http://www-sop.inria.fr/galaad/logiciels/synaps

[12] Goldengorin, B., Sierksma, G. (2003): *Combinatorial optimization tolerances calculated in linear time*, SOM Research Report 03A30, University of Groningen, The Netherlands.

[13] Gross, J., Yellen, J. (1999): *Graph theory and its applications*. CRC Press Inc. (ISBN 0-8493-3982-0)

[14] Pan, V.Y. (1997): 'Solving a polynomial equation: Some history and recent progress'. *SIAM Review*, vol. 39, no. 2, pp. 187-220.

[15] Press, W.H. *et al.* (1988): *Numerical Recipes in C*, Camebridge University Press. (ISBN 0-521-35465-X)
http://www.nr.com

[16] Tuzikov, A.V., Sheynin, S.A. (2002): 'Symmetry measure computation for convex polyhedra'. *Journal of Mathematical Imaging and Vision*, vol. 16, no. 1, pp. 41-56.