The University of Edinburgh School of Mathematics

A branch-and-cut approach to solve the Hamiltonian Cycle Problem

by

Marco Colombo

Dissertation Presented for the Degree of
MSc in Operational Research

2003

Abstract

This project studies a nonconvex quadratic formulation for the Hamiltonian Cycle Problem, solved within a branch-and-cut framework. The formulation adopted combines a relaxation of the integrality constraint with a penalty term in the objective. Despite the nonconvexity of the resulting quadratic problem, an interior point algorithm is applied. Two different types of cutting planes have been designed and implemented. A computational experience has compared two implementations of one type of cutting plane, within three different formulations of the problem: a complete QP formulation, a linear formulation and a hybrid QP formulation. The results suggest that the branch-and-cut framework with hybrid QP objective outperforms the two other formulations for large scale problems.

Acknowledgements

I would like to thank Dr. Jacek Gondzio for proposing such an interesting and challenging project for me. This allowed me to focus simultaneously on three of my favourite subjects, Graph Theory, Optimization and Software Development. Moreover, his dedicated support has been crucial to my successful completion of this project.

Many thanks also to Mr. George Maistros and Mr. Ignasi Cos Aguilera for their invaluable hints and suggestions in the creation of the Cut structure and the related functions.

I am particularly indebted to Andrea, who has proofread several drafts of this document. All remaining errors are most likely due to my last-minute changes.

Table of Contents

Introduction

Integer programming problems require the solution to be integer. One of the techniques used to solve these problems is branch-and-cut. A branch-and-cut approach adds constraints that remove noninteger points and split the feasible set into separate regions. The subproblems thus generated are simpler, as the areas to be investigated get smaller and smaller.

In this project, the branch-and-cut methodology is applied to solve the Hamiltonian Cycle Problem. This is the problem of finding a cycle in a graph such that all vertices are visited exactly once. The Hamiltonian Cycle Problem is one of the most difficult problems in mathematics, as it belongs to the class of NP-complete problems.

Typical approaches to solving this problem are based on a linear programming formulation. They use the simplex method to find basic solutions to the relaxed problem whilst imposing constraints on the creation of subcycles.

In contrast, this project is inspired by an entirely different perspective. Indeed, the problem undergoes a quadratic reformulation that combines a relaxation of the integrality constraint with a penalty term in the objective.

Although the resulting formulation is nonconvex, an interior point algorithm is employed. This choice stems from the fact that interior point methods stay in the interior and do not converge too quickly to an integer solution, which is likely to contain a series of subcycles.

Within this project, much effort has been placed on integrating a branch-and-cut approach to encourage the creation of integer solutions. The approach relies on cutting planes, generated according to the two heuristics that have been designed and implemented.

Three different formulations of the problem (a complete QP formulation, a linear formulation and a hybrid QP formulation) have been compared in a series of test problems.

Chapter 1

Introductory concepts

1.1 Graphs

A graph G is defined by two finite sets, the set $V = (v_1, v_2, \ldots, v_m)$ of *vertices* (or nodes) and the set $E = (e_1, e_2, \ldots, e_n)$ of *edges* (or arcs), where each e_k has the form $e_k = (v_i, v_j)$, for $v_i, v_j \in V$. Two nodes connected by an edge are said to be *adjacent*. An edge is *incident* with a vertex if it has that vertex as one of its endpoints.

A graph is *directed* when each edge is associated with an *ordered* pair of vertices. In this case, $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_1)$ are distinct edges. Such a graph is often referred to as *digraph*. In this study, we will consider directed graphs as these are the most general way of representing a graph. In fact, an undirected graph is a digraph whose edges go in both directions.

Often the term "arc" is reserved to directed graphs and "edge" to undirected graphs. Here we will not make this distinction and we will use both of these terms when referring to digraphs.

A graph is *simple* when there are no loops (edges of the form (v_i, v_i), for $v_i \in V$) and allows only a single edge between a pair of distinct vertices (no parallel arcs). As loops and parallel arcs do not provide any additional information to the graph, for our purposes the analysis will be restricted, without loss of generality, to simple graphs.

A graph is *complete* if there is an edge between every pair of distinct vertices.

When G is a digraph, there are two numbers associated with each vertex $v \in V$. The *indegree* of v is the number of edges directed toward v; the *outdegree* is the number of edges directed away from v. Indegree and outdegree are also called *semi-degrees*, and the sum of the semi-degrees of a vertex defines the *degree* of that vertex.

A *path* in a graph from a node u to a node v is a sequence of distinct vertices $(v_1, v_2, \ldots, v_k)$ such that $v_1 = u$ and $v_k = v$ and such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$. Vertices v_1 and v_k are the endpoints of the path. The length of the path is simply the number of arcs in the path.

A graph is *connected* when it contains at least one undirected path between every pair of nodes. A digraph is *strongly connected* if, for every pair of distinct vertices x and y, there exists an (x, y)–path and a (y, x)–path.

If a path ends at the vertex where it started from, then the path is closed and forms a *cycle*.

The literature focuses mainly on two particular types of paths:

- An *Eulerian path* is one which traverses every edge in E once and only once.
- A *Hamiltonian path* is one which visits all vertices in V once and only once.

Where these paths are also cycles, they are called *Eulerian cycle* and *Hamiltonian cycle*, respectively.

1.2 Ways of representing graphs

Graphs can be represented in several different ways, according to the specific needs that have to be addressed. Each way has its strengths and weaknesses, and often can be used in conjunction with one another.

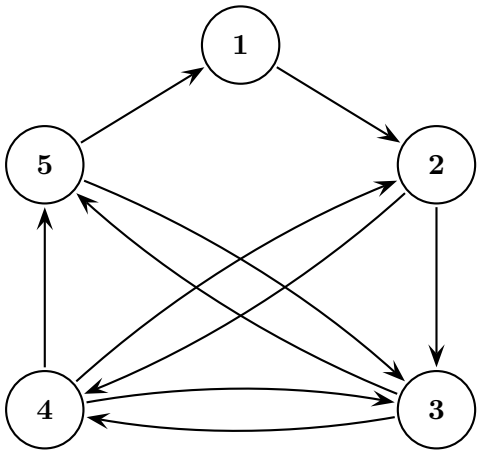
Let us consider the following simple graph on 5 nodes and 10 arcs defined by:

$$\begin{cases} V = \{1, 2, 3, 4, 5\}, \\ E = \{(1, 2), (2, 3), (2, 4), (3, 4), (3, 5), (4, 2), (4, 3), (4, 5), (5, 1), (5, 3)\}. \end{cases}$$

This representation in the form of a set of vertices and a list of edges, directly originates from the definition of graph given above.

Another way is the pictorial representation. In this case, the vertices of a graph may be represented by circles. Since the vertices are indexed, the corresponding index appears inside the circle. Each arc (u, v) is represented by a line segment connecting u and v. In a directed graph, an arrowhead is added at the end of the line representing the arc.

The above graph can be pictorially represented as:



Pictorial representation is especially useful for examining the structure of a graph overall. It is less valuable, however, for larger or complicated graphs. Furthermore, this representation is unsuitable for computer use.

A graph can also be represented by storing the information about vertices and edges in a matrix. The most important of these are the *adjacency matrix* and the *node–arc incidence matrix*.

Given a graph G, its adjacency matrix is the $m \times m$ matrix whose generic element is defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if there is an arc connecting node } i \text{ to node } j; \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix for our example graph is the following:

$$\text{nodes} \underbrace{\begin{Bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} \end{Bmatrix}}_{\text{nodes}}$$

Moving to the node–arc incidence matrix, each row of the matrix corresponds to a vertex of G and each column corresponds to a distinct edge. Thus, the matrix has dimension $m \times n$. The generic element a_{ij} of the incidence matrix can assume one of three different values:

$$a_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves node } i; \\ +1 & \text{if edge } j \text{ enters node } i; \\ 0 & \text{otherwise.} \end{cases}$$

To continue the example, our graph has the following incidence matrix:

$$\text{nodes} \underbrace{\begin{Bmatrix} \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & -1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & -1 \end{bmatrix} \end{Bmatrix}}_{\text{edges}}$$
$$\begin{matrix} (1,2) & (2,3) & (2,4) & (3,4) & (3,5) & (4,2) & (4,3) & (4,5) & (5,1) & (5,3) \end{matrix}$$

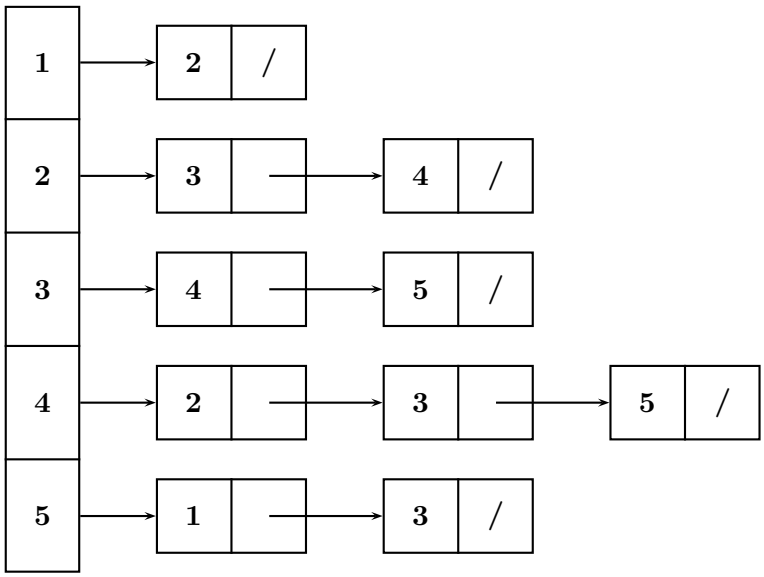
The representation through the node–arc incidence matrix is particularly useful when the graph needs to be embedded in a linear programming problem.

The incidence matrix of a graph is a special case of the class of all matrices having

0, 1 or -1 for elements. As we will see in Section 2.2, such matrices have a particularly important property.

Another way of representing a graph for computational purposes is by its *adjacency lists*. This way consists of a pair of arrays, each of which contains m linked lists. For each node v, the linked list in one array contains all vertices leaving v, while the other array contains all vertices entering v. Using these adjacency lists, it is possible to quickly obtain all out-neighbours and in-neighbours of a given vertex.

According to our example, the adjacency list for the out-neighbours looks like:



The representation discussed above can be used as a starting point for the computer implementation of a graph structure to solve specific problems encountered.

1.3 The Hamiltonian Cycle Problem

Given a graph $G = (V, E)$, a Hamiltonian cycle is a cycle that includes every vertex of G. That is, a Hamiltonian cycle for G is a sequence of adjacent vertices and distinct edges in which every vertex of G appears exactly once.

The *Hamiltonian Cycle Problem* asks to find a Hamiltonian cycle in a graph, or to state that such a cycle does not exist.

Different problems exist that are closely related to the Hamiltonian Cycle Problem. In the this section we will describe two of them.

1.3.1 The Travelling Salesman Problem

In the *Travelling Salesman Problem*, a series of locations must be visited such that each city is visited exactly once, starting and ending in the same city, and the total distance is minimized.

This problem can be represented in terms of a graph, where each node corresponds to a city, and an edge (i, j) exists if it is possible to go directly from city i to city j. A distance, or cost, is assigned to each edge.

In these terms, the Travelling Salesman Problem involves finding a Hamiltonian cycle that minimizes the total distance travelled for an arbitrary directed graph. It is clear that the Hamiltonian Cycle Problem is a special case of Travelling Salesman Problem in which all arcs have the same cost.

A way in which the problem can be solved, is to enumerate all Hamiltonian cycles which start and end at a particular vertex, and, for each cycle, calculate the total distance. The solution is the one with minimum total distance.

Yet this method is impractical, even for a medium-sized number of cities. This is because any algorithm which employs this method has an exponential order of complexity. For example, a complete graph with 20 vertices would have 19! different Hamiltonian cycles to check.

No algorithm has been found that is more efficient for solving the Travelling Salesman Problem to optimality. However, some algorithms exist which can find solutions that, whilst not having the minimum possible total distance, have total distances which are smaller than most other Hamiltonian cycles.

1.3.2 The Knight's Tour Problem

The *Knight's Tour Problem* asks to determine a sequence of moves of a knight over a chess board of size $n \times n$ in such a way that the knight visits every square exactly once.

The connection between the Knight's Tour Problem and the Hamiltonian Cycle Problem is immediate when the former is stated in terms of a graph. Each square of the board represents a vertex of the graph; each edge connects two squares that can be reached by a move of the knight.

Having translated the problem in graph terms, finding a solution to the knight's tour on the chess board is equivalent to finding a Hamiltonian cycle on the associated graph.

There are no knight's tours on $n \times n$ boards when n is odd. Moreover, only for $n \geq 6$ a cycle can be found.

1.4 Theorems on Hamiltonian cycles

In studying the Hamiltonicity of a graph, parallel edges and loops can be excluded from consideration since they do not affect the construction of any Hamiltonian cycle. A loop may not occur in a Hamiltonian cycle since it would isolate the incident vertex (as a vertex must be visited only once). If parallel arcs are present, only one can be used, otherwise the pair of endpoint vertices would create a cycle of length 2.

It appears to be clear that G must be connected if it is to have a Hamiltonian cycle. In the case of directed graphs, the requirement is that the graph must be strongly connected. That is, any digraph with at least one vertex of zero semi-degree is not Hamiltonian.

Theorems give simple criteria for determining whether or not a given graph has an Eulerian cycle. Unfortunately, there are no analogous theorems for determining whether or not a given graph has a Hamiltonian cycle

There is only a special case, covered by the following theorem, which concerns complete graphs.

Theorem 1.1 *Every complete graph of order $m \geq 3$ contains a Hamiltonian cycle.*
Proof: Choose an arbitrary vertex V and traverse a path which never returns to the same vertex twice. It is not possible to get stuck at a vertex u if there is a vertex w not already on the path because, by the assumption of completeness, there is an edge (u, w) in the graph and so we can go from u to w. The process stops only when all vertices are on the path. At this point, by adding the edge (v_m, v_1), the path becomes a cycle.

Some sufficient conditions for general graphs can be stated as well. They rely on the following consideration: by adding edges to a Hamiltonian graph, the Hamiltonicity is not lost, as the resulting graph will still contain the original Hamiltonian cycle. Therefore, one basic approach is to attempt to force the graph to have a relatively large number of edges, since in this case it is more likely to be Hamiltonian.

Two conditions, given in the following theorems, are easily checked. For proofs, see [5, p. 171] and [11, pp. 146–148], respectively.

Theorem 1.2 (Dirac, 1952) *Let G be a graph of order $m \geq 3$. If for any vertex v of G, $deg(v) \geq \frac{1}{2}m$, then G is Hamiltonian.*

Theorem 1.3 (Ore, 1960) *Let G be a graph of order $m \geq 3$. If for every pair of distinct nonadjacent vertices u and v, $deg(u) + deg(v) \geq m$, then G is Hamiltonian.*

A stronger result is proved in [9, p. 352]:

Theorem 1.4 (Bondy – Chvátal, 1976) *Let G be a graph of order $m \geq 3$ and suppose that u and v are distinct nonadjacent vertices of G such that $deg(u)+deg(v) \geq m$. Then G is Hamiltonian if and only if $G + (u, v)$ is Hamiltonian.*

While the previous theorems were dealing with undirected graphs, the following theorems concern directed graphs. An extensive survey of sufficient conditions for Hamiltonicity in directed graphs is provided in [1, pp. 240–246].

Theorem 1.5 (Ghouila-Houri, 1960) *If the degree of any vertex in a digraph of order m is at least m, then the graph is Hamiltonian.*

Theorem 1.6 (Meyniel, 1973) *Let G be a strongly connected digraph of order $m \geq 2$. If $deg(u) + deg(v) \geq 2m - 1$ for all pairs of non-adjacent vertices u and v, then G is Hamiltonian.*

The next theorem generalizes Dirac's result to directed graphs.

Theorem 1.7 *Let G be a digraph of order m. If the minimum semi-degree of G is at least $\frac{1}{2}m$, then G is Hamiltonian.*

All these criteria, while theoretically interesting, are too loose to be of value for arbitrary graphs. Moreover, none of these theorems are constructive, in that they do not provide an algorithmic way to generate a Hamiltonian cycle.

1.5 Complexity issues

The Eulerian cycle problem, which finds a path that touches every edge exactly once, can be solved in linear time by using Fleury's algorithm (see [11, p. 126]). In contrast, there are no known algorithms that are guaranteed to solve the Hamiltonian cycle problem efficiently.

The formal notion of *efficiency* is that a problem has an algorithm running in time proportional to a polynomial function of its input size. That is, we consider an algorithm efficient if it runs in time $O(n^k)$ on any input of size n, for some constant $k > 0$.

When a problem is stated in such a way that the answer can be either 'yes' or 'no', the problem is said to be a *decision problem*. Also the Hamiltonian cycle problem can be stated as a decision problem: in this case, a 'yes' instance would be any simple circuit in the graph that includes all vertices.

The class P contains those decision problems that can be solved in polynomial time: problems in this class require polynomial time to state either 'yes' or 'no'.

A *non-deterministic* machine is an abstract concept. It refers to machines in which when the algorithm reaches a point where there is a choice from a finite number k of alternatives, these are explored simultaneously. This is often described as the machine creates k copies of itself at the point at which the choice arises. If any copy finds it has made an incorrect choice, it stops executing. If any copy finds a solution then all copies stop executing. With a non-deterministic machine, backtracking is avoided.

In contrast, a *deterministic* machine can only proceed to explore a single alternative at a time, and must return later to explore the others.

The class NP contains those decision problems solvable by a non-deterministic machine in polynomial time. Problems in the class NP do not require that every instance can be answered in polynomial time. It is only required that a 'yes' answer can be checked in polynomial time for validity.

Among all problems known to be in NP, there is a subset, known as the class of NP-complete problems, which contains the hardest, under a certain ordering of problems by difficulty.

The following theorem has an immediate impact on the problem we are studying. A proof is given in [5, pp. 230–234].

Theorem 1.8 *The problem to check whether a given digraph has a Hamiltonian cycle in NP-complete.*

The class of NP-complete problems has the following properties:

- No NP-complete problem can be solved by any known polynomial algorithm;
- Any problem in the class can be reduced to all other problems in the class by a polynomial algorithm.

Therefore, if a polynomial time algorithm is found for any NP-complete problem, then there are algorithms for all NP-complete problems. On the other hand, if it is proven that no polynomial-time algorithm exists for some problem in the class, then no polynomial-time algorithms exist for any of them.

We are interested in solving problems in time bounded by a polynomial in the problem size. As noted by Schrijver (see [10, p. 14]):

> "To indicate the significance of polynomial-time solvability, if we have an algorithm with running time 2^n (where n is the problem size), then a quadrupling of the computer speed will *add* 2 to the size of the largest problem that can be solved in one hour, whereas if we have an algorithm with running time n^2, this size will be *multiplied* by 2."

NP-complete problems are said to be "inherently intractable" from a computational point of view. Any algorithm that correctly solves an NP-complete problem will require, in the worst case, an exponential amount of time, and hence will be impractical for all but small instances.

Chapter 2

Integer programming

2.1 Integer linear programs

The class of integer linear programming problems (IP) is of particular interest in our analysis. Indeed, the problem of finding a generic cycle in a graph corresponds to choosing some of the edges to be in the cycle, and all other edges to be left out. Therefore we are dealing with a combinatorial problem, or, more specifically, a binary integer programming problem.

The computational difficulty of integer programming problems depends on two main factors:

1. The number of integer variables involved;

2. The existence of any special structure in the problem.

In the case of binary integer problems with n variables, exhaustive enumeration requires 2^n solutions to be considered, and an increase in size by 1 doubles the number of solutions. Each solution examined must be checked for feasibility, and, if feasible, the value of the objective function is determined. Therefore, a full-scale enumeration cannot be performed by any computers, when problems have more than a few constraints and variables.

It is generally not true that the optimal solution to the linear programming (LP) relaxation of an integer programming problem is integer as well. Nevertheless, several types of IP problems (shortest-path problem, assignment problem, maximum flow problem) exist for which the corresponding LP relaxation finds an integer solution. This result holds for these types of problems because they are characterized by a special property that every basic feasible solution is integer. As we will see in the next section, this property is total unimodularity.

2.2 Unimodular matrices

In this section we outline a characterization of a linear programming problem with integer data such that its optimal solution is integer.

A square, integer matrix B is called *unimodular* if its determinant is ± 1. An integer $m \times n$ matrix A is called *totally unimodular* if every square, nonsingular submatrix of A is unimodular.

The following characterization theorem concerns integer matrices with generic element $a_{ij} = 0, \pm 1$, such that every column contains exactly two nonzero elements. More characterizations of total unimodularity are presented in [10, pp. 269–272].

Theorem 2.1 (Heller – Tompkins, 1956) *A matrix is unimodular if and only if its rows can be partitioned into two disjoint sets I_1 and I_2 such that, if the two nonzero elements are of opposite sign, either both are in I_1 or both are in I_2, and if they have the same sign, one is in I_1 and the other is in I_2.*

There is an immediate corollary that is useful for our analysis.

Corollary 2.2 *The incidence matrix of a digraph is totally unimodular.*
Proof: By definition of incidence matrix, every column contains exactly one $+1$ entry and a -1 entry. Thus we can put every row into the set I_1 and let $I_2 = \emptyset$. This decomposition satisfies Theorem 2.1 and the unimodular result follows.

It should be noted that if a submatrix has two nonzero entries in every column, then the condition of the theorem implies that

$$\sum_{i \in I_1} a_{ij} = \sum_{i \in I_2} a_{ij}, \quad 1 \leq j \leq n.$$

In the case of the incidence matrix of a graph, this means that the sum of the entries of each column is zero.

Totally unimodular matrices have an interesting feature: when A is totally unimodular, a basic solution of the system $Ax = b$, for any integer right-hand side b, is always integer.

By considering the partition of $A = [B, N]$ of basic and nonbasic variables, then the solution of the system is given by

$$x_B = B^{-1}b \quad \text{and} \quad x_N = 0. \tag{2.1}$$

Recalling that

$$B^{-1} = \frac{1}{det(B)} \cdot adj(B),$$

then B^{-1} is an integer matrix, as product of an integer matrix and an integer scalar. Therefore, also the product $B^{-1}b$ is between integer quantities, hence the result x_B is again an integer vector.

The following theorem (see [2, p. 147]) provides an important result that can be used in linear programming.

Theorem 2.3 (Hoffman – Kruskal, 1956) *Let A be a matrix with integral elements, and let b, b', l and u be vectors of integers; then if the matrix A is totally unimodular, each face of the convex polyhedron*

$$\{x \mid b' \leq Ax \leq b,\ l \leq x \leq u\}$$

contains a point all of which the coordinates are integers.

As a particular case covered by the theorem, the vertices of the polyhedron, which are faces of dimension zero, are points with integer coordinates. Therefore, integer linear programming problems with totally unimodular constraint matrices are no more difficult to solve than the corresponding linear program relaxations.

It is important to note that not all totally unimodular matrices are network matrices or their transposes.

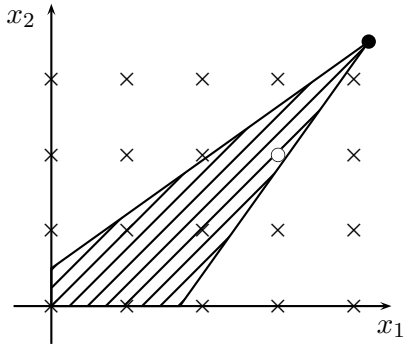
2.3 Solving integer programs

Consider the problem

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b, \\ & x \geq 0, \text{ integer.} \end{aligned}$$

Consider now the polytope defined by this set of inequalities when the integrality constraint is relaxed. As already mentioned, all the extreme points of this polytope have integer coordinates for every choice of integer-valued b if and only if A is totally unimodular.

If total unimodularity cannot be established, one can solve the relaxed problem by the simplex method, or by any other method, and round-off noninteger solutions. Unfortunately, the true integer optimum point may not be among the rounded off results. Moreover, this procedure may produce an infeasible result, as illustrated in the following diagram.



In the figure, the lined area represents the feasible region and the points with integer coordinates are marked by '×'. The solution of the relaxed linear program (represented by '•') is indeed far distant from the integer solution (the '○' in the middle of the feasible region). Additionally, the rounded result, the origin of the Cartesian plane, does not belong to the feasible area.

Therefore, a completely different approach is needed for solving integer programming problems. There are two basic concepts used to develop such methods:

1. Implicitly enumerate "all" solutions until the optimal result is found.

2. Introduce additional constraints that remove noninteger solutions but do not remove integer solutions until the integer optimum is found.

The first concept is at the basis of the branch-and-bound algorithm; the second one defines the cutting plane method.

2.3.1 Branch-and-bound

Branch-and-bound is an optimization technique that tries to intelligently enumerate solutions. It involves calculating upper and lower bounds on the objective function, in order to discard many possible solutions and thus shorten the enumeration.

The branch-and-bound approach also relies on the concept of "divide to conquer". Since the original problem is too difficult to be solved directly, it is broken up into smaller and smaller subproblems until these subproblems can be solved.

The branching is done by partitioning the entire set of feasible solutions into two or more smaller subsets. This can happen in different ways according to the specification of the problem. The branching into subproblems can be portrayed by a tree, referred to as the solution tree (or branching tree).

The bounding uses the objective function value of the relaxed linear programming solution as a lower bound on the cost of the integer solution. The upper bounds do not change unless an integer solution is obtained from the linear program, in which case the best solution to date changes. A subproblem is discarded if its bound indicates that it cannot possibly contain a better solution than the one currently found.

A generic algorithm for branch-and-bound follows this procedure:

1. Given a list of subproblems that have not yet been explored, select one to examine next. If there are none, terminate the search and conclude that the optimal solution is the best one found so far.

2. Examine the solution obtained, and decide whether it is possible to update the bounds on the integer solution. Choose the branching variable and generate new subproblems.

Typically, there are multiple criteria for the choice of the node of the branching to be examined, and these change depending on whether a feasible solution has been reached.

Branching rules can range from breadth-first to depth-first. In the first case, all possible branches from a node are explored before going deeper into the tree. In the second case, a node is explored as deeply as possible, backtracking then to other nodes in the same level.

Hybrid strategies can also be devised, in which both criteria are combined according to the contingent situation of the solution tree. An example is the best-first search, which chooses the node that seems most promising, such as the one with the smallest lower bound.

The most common approach for the choice of the branching variable is to select the variable with the most fractional value. The branching variable must either not exceed the next lower integer or it must equal or exceed the next higher integer. The current problem is split into subproblems, and each of them contains one of these inequalities.

Branch-and-bound terminates when the solution tree has been completely explored. That is, when all nodes have been analyzed or discarded, and no further branching is possible.

2.3.2 Cutting plane method

The *cutting plane method* was introduced by Gomory in the late 1950s to enable the use of the simplex method to solve integer linear programs. It is named after the effect of adding a new constraint to the existing ones, by which unwanted regions of the feasible set are cut away.

In a cutting plane method for integer problems, the linear programming relaxation of the integer program is solved. If the optimal solution is integer, it also solves the integer program; otherwise, a hyperplane which separates the solution of the relaxed problem from the set of feasible integer solutions is generated. This is appended to the set of existing constraints: the new linear program is solved, and the process is repeated.

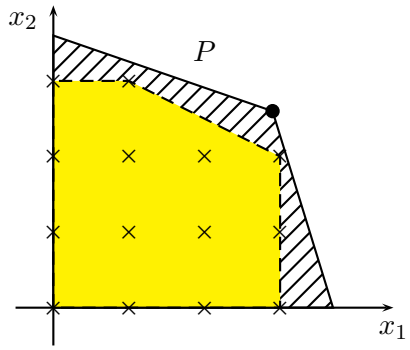
Each hyperplane, simply called cut, has two properties:

- Any feasible integer point will satisfy the cut;
- The optimal solution to the current linear programming relaxation will violate the cut.

This way, the feasible region for the LP relaxation is reduced without eliminating any feasible solutions for the IP problem. Eventually, the solution of the relaxed linear program under the augmented set of constraints also solves the integer problem.

The following formalization is inspired by [10, pp. 339–342].

Let us denote with P the original set of constraints in a problem, and call P_I the convex hull of feasible integer points in P. In the following figure, the convex hull P_I corresponds to the shaded area inside the feasible region P. The solution of the linear programming relaxation is marked by '•'.



Generating P_I is extremely difficult and it is not actually required: it is sufficient to make the optimal integer solution an extreme point and remove all noninteger extreme points with a better objective function value.

The idea behind the cutting plane method is to devise successive hyperplanes h_i such that

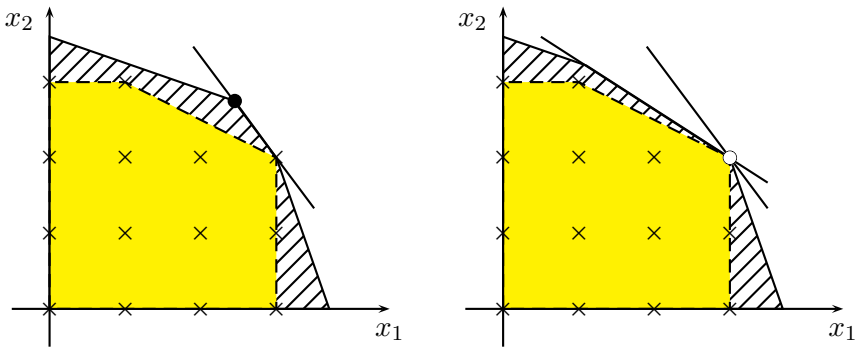
$$P \equiv P_0 \supseteq P_1 \supseteq \ldots \supseteq P_I,$$

where

$$P_i = \bigcap_i h_i \cap P_{i-1}, \quad i = 1, 2, \ldots$$

The cutting plane method solves the LP relaxation over P, P_1, P_2, ..., until an optimal solution with integer components is found.

In the figures below, the introduction of cutting planes moves the solution of the relaxed program to the integer solution.



Hillier and Lieberman (see [7, p. 629]) suggest the following procedure for generating cuts in a binary integer problem:

"Consider any functional constraint in $\leq$ form with only non-negative coefficients.
Find a group of variables such that:

(a) The constraint is violated if every variable in the group equals 1 and all other variables equal 0.

(b) The constraint becomes satisfied if the value of *any one* of these variables is changed from 1 to 0.

By letting N denote the number of variables in the group, the resulting cutting plane has the form:

$$\text{Sum of variables in the group} \leq N - 1."$$

2.4 Branch-and-cut

Branch-and-bound and cutting plane method can be successfully integrated into the so called *branch-and-cut* approach.

This technique aims to combine the best aspects from both those methods. On one hand, it relies on creating branches: this way, the dimension of the problem reduces and the subproblems obtained become easier to solve. Furthermore, bounding strategies can be adopted as well. On the other hand, the branching does not happen on only one variable at a time, as in branch-and-bound, but it may involve several variables simultaneously. For this reason, this method can be thought of as "constraint branching".

The cuts generated in this method are substantially different from those obtained by a conventional cutting plane method, in that they partition the feasible set P into two disjoint subsets P_1 and P_2, such that $P \supset P_1 \cup P_2$, where:

$$\begin{aligned} P_1 &= P \cap h_1, \qquad h_1 = a^T x \leq b - 1, \\ P_2 &= P \cap h_2, \qquad h_2 = a^T x \geq b. \end{aligned}$$

In these expressions, the term $a^T x$ represents a linear combination of the variables in the problem, and b is a convenient integer. As we can see, each subproblem contains the same constraint but with opposite signs and different right-hand sides.

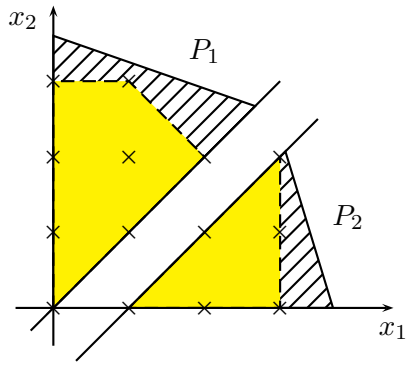
Since two subproblems are created at each iteration, the branching tree is always binary.

Branch-and-cut methods for integer programming problems solve a sequence of relaxed linear programs. Each problem in the sequence allows the generation of two inequalities. At this point the branching happens: each subproblem will contain the same set of constraints as its predecessor; another constraint is the cut just generated, which is added to the problem.

The next figure shows how, with the introduction of the cuts

$$x_1 - x_2 \leq 0 \quad \text{and} \quad x_1 - x_2 \geq 1,$$

the feasible area is partitioned into two disjoint subsets.



These observations can be made:

- The cuts are legitimate, as they cut off the current solution of the relaxed program and do not remove any of the feasible integer points.
- As in the case of Gomory cuts, the feasible region for the linear relaxation has been trimmed.
- In this case, also the convex hull of integer points has been reduced: its area was 8 and now is 5.5.

One of the aims of the cut is to determine a region such that the corresponding subproblem cannot contain an optimal solution: in such a case that subproblem can be discarded and no further analysis is required. In this respect, the difference with classic cutting plane methods become more evident.

The branch-and-cut approach uses the structure of the problem to generate very good cuts. This needs a problem-by-problem analysis, but provides very efficient solution techniques.

Chapter 3

A nonconvex QP formulation of the Hamiltonian Cycle Problem

3.1 Outline of the approach

Here we present the formulation of the Hamiltonian Cycle Problem as a nonconvex quadratic problem.

The approach chosen in solving the problem is based on modelling the graph as a set of linear constraints. This is done by building a node–arc incidence matrix. Other rows and columns are appended to the incidence matrix to obtain a complete model of the problem, as detailed in the next section. This will let us interpret the Hamiltonian Cycle Problem as an optimization problem.

In this problem, we associate a binary variable x_{ij} to every edge (i, j). Such a variable may assume two values:

$$x_{ij} = \begin{cases} 1 & \text{if edge } (i,j) \text{ is used in the Hamiltonian cycle;} \\ 0 & \text{otherwise.} \end{cases}$$

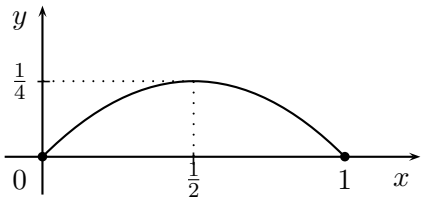
To be able to use a generic solver (an interior point method solver, in our case), we need to relax the integrality constraint. However, the formulation proposed is different from an ordinary relaxation, in that we are introducing a penalty term, which happens to be quadratic.

Let us sketch here how this transformation is done.

To model a binary variable x without using an integrality constraint, we can express it as a continuous variable $0 \leq x \leq 1$ such that

$$x(1 - x) = 0.$$

To visualize why we choose such a nonconvex approach, we provide the graphical representation of the function $y = x(1 - x)$.



This function is a parabola with vertex in $(\frac{1}{2}, \frac{1}{4})$. As we can see, in the interval $[0, 1]$ the function is non-negative, and attains its minimum at the extreme points, which have integer coordinates.

Therefore, the minimization of such a nonlinear nonconvex function leads to the same result as the imposition of an integrality constraint.

3.2 The objective function

The presentation given here follows what is done in [3].

Consider a node i, and let $\mathcal{A}(i) = \{j \mid (i, j) \in E\}$ be the set of nodes adjacent to node i. Since we are considering directed graphs, the set $\mathcal{A}(i)$ contains the nodes reachable from node i, the out-neighbours.

In our quest to find a cycle in the graph, we want to choose only one node among those in $\mathcal{A}(i)$; this is equivalent to choosing only one arc among those leaving node i.

Using the network flow terminology, we demand a flow of 1 to leave node i. Thus, for all i, we require that

$$\sum_{j \in \mathcal{A}(i)} x_{ij} = 1. \tag{3.1}$$

If x_{ij} is binary, condition (3.1) is easily satisfied. Unfortunately, if the integrality constraint is relaxed, this condition is not enough to guarantee that only one arc will be chosen. Indeed, there are infinitely many combinations of the continuous variables x_{ij} that satisfy it.

Now, let us apply a technique similar to the one outlined above to model the flow x_{ij} that traverses arc (i, j).

The requirement that we choose only one arc among those leaving from node i can be expressed by the minimization of the following quantity:

$$\left(\sum_{j \in \mathcal{A}(i)} x_{ij} \right)^2 - \sum_{j \in \mathcal{A}(i)} x_{ij}^2 = \sum_{k \neq l} x_{ik} x_{il}.$$

The last term corresponds to the sum of all cross products. Given that $0 \leq x_{ij} \leq 1$, this quantity is non-negative. If we now ask for

$$\sum_{\substack{k, l \in \mathcal{A}(i) \\ k \neq l}} x_{ik} x_{il} = 0, \tag{3.2}$$

then no two variables can be different from zero at the same time. This is equivalent to saying that we will choose at most one of them to be strictly positive.

Now, by using $e = (1, 1, \ldots, 1)^T$, we can write the same expression in matrix notation:

$$\begin{aligned}(e^T x_i)^2 - x_i^T x_i &= x_i^T e e^T x_i - x_i^T x_i \\ &= x_i^T (e e^T - I) x_i \\ &= x_i^T Q_i x_i,\end{aligned}$$

where

$$Q_i = \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 0 & & 1 \\ \vdots & & \ddots & \\ 1 & 1 & & 0 \end{bmatrix} \quad \text{and} \quad x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{in_i} \end{bmatrix}.$$

Matrix Q_i is an $n_i \times n_i$ matrix, with $n_i = |\mathcal{A}(i)|$.

The quantity $x_i^T Q_i x_i$ is zero when at most one node in $\mathcal{A}(i)$ is chosen. In all other cases, this quantity is strictly positive. We can think of it as a penalty term that penalizes us when we assign fractional values to the variables. The minimization of $x_i^T Q_i x_i$, leads us to assign integer values to the elements of x_i.

Clearly, the same applies to all nodes in the graph. Thus we will consider

$$\sum_{i \in V} x_i^T Q_i x_i = x^T Q x,$$

where

$$Q = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_m \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}.$$

This is a block diagonal $n \times n$ matrix, where each of the m blocks has dimension $n_i \times n_i$.

The matrix, as just defined, has many nonzero elements. However, it is possible to improve its sparsity pattern. If we consider the introduction of an auxiliary variable y_i, we can write

$$x_i^T Q_i x_i = (x_i^T e)(e^T x_i) - x_i^T x_i = y_i^2 - x_i^T x_i, \tag{3.3}$$

where

$$y_i = x_i^T e = \sum_{j \in \mathcal{A}(i)} x_{ij}. \tag{3.4}$$

This sparser formulation corresponds to the system ${\tilde{x}_i}^T \tilde{Q}_i \tilde{x}_i$, where

$$\tilde{Q}_i = \begin{bmatrix} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & -1 & \\ & & & & 1 \end{bmatrix} \quad \text{and} \quad \tilde{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{in_i} \\ y_i \end{bmatrix}, \tag{3.5}$$

and $\tilde{Q}_i$ now has dimension $(n_i + 1) \times (n_i + 1)$.

The advantage in this case is that the matrix is diagonal; on the other hand, its dimension has increased by one and we have the additional constraint (3.4). Overall, the dimension of the problem has m more variables and m more constraints.

Having transformed the matrix Q into a diagonal matrix, we have obtained a separable problem. This problem is computationally much easier than the previous one, despite its increase in dimension.

3.3 The constraints

While in the previous section we were mostly referring to the objective function, here we discuss the modifications that have to be made to the constraints. To visualize the changes in the formulation, we will refer to the example graph already seen in Chapter 1.

For each node, we want the total amount of flow entering it to be equal to the total amount of flow leaving it. That is, no vertex is a source or a sink of the network. This is the so called *flow conservation law*, and can be expressed as

$$\sum_{j \in \mathcal{A}(i)} x_{ij} \; = \sum_{k:\, i \in \mathcal{A}(k)} x_{ki}, \quad i = 1, \ldots, m.$$

The constraint $Ax = 0$, where A is the node–arc incidence matrix of the graph, guarantees that the flow conservation law is satisfied.

According to our example, we have

$$\left[\begin{array}{c|cc|cc|ccc|cc} -1 & & & & & & & & 1 & \\ 1 & -1 & -1 & & & 1 & & & & \\ & 1 & & -1 & -1 & & 1 & & & 1 \\ & & 1 & 1 & & -1 & -1 & -1 & & \\ & & & & 1 & & & 1 & -1 & -1 \end{array}\right] \cdot \begin{bmatrix} x_{12} \\ x_{23} \\ \vdots \\ x_{51} \\ x_{53} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Now we consider the addition of an auxiliary variable y_i, one for each node of the

graph, such that (3.4) is satisfied, or equivalently:

$$\sum_{j \in \mathcal{A}(i)} x_{ij} - y_i = 0.$$

Adding this set of constraints is equivalent to considering the following matrix:

$$\tilde{A} = \left[\begin{array}{cc|ccc|ccc|cccc|ccc} -1 & & & & & & & & & & & & 1 & & \\ 1 & & -1 & -1 & & & & & 1 & & & & & & \\ & & 1 & & & -1 & -1 & & & 1 & & & & 1 & \\ & & & 1 & & 1 & & & -1 & -1 & -1 & & & & \\ & & & & & & 1 & & & & 1 & & -1 & -1 & \\ \hline 1 & -1 & & & & & & & & & & & & & \\ & & 1 & 1 & -1 & & & & & & & & & & \\ & & & & & 1 & 1 & -1 & & & & & & & \\ & & & & & & & & 1 & 1 & 1 & -1 & & & \\ & & & & & & & & & & & & 1 & 1 & -1 \end{array}\right],$$

where

$$\tilde{x} = [\ x_{12} \quad y_1 \mid x_{23} \quad x_{24} \quad y_2 \mid x_{34} \quad x_{35} \quad y_3 \mid x_{42} \quad x_{43} \quad x_{45} \quad y_4 \mid x_{51} \quad x_{53} \quad y_5\ ]^T.$$

To satisfy constraints (3.1) we should append the following row to matrix $\tilde{A}$, with a corresponding right-hand side of 1:

$$\left[\begin{array}{cc|ccc|ccc|cccc|ccc} 1 & 0 & & & & & & & & & & & & & \\ & & 1 & 1 & 0 & & & & & & & & & & \\ & & & & & 1 & 1 & 0 & & & & & & & \\ & & & & & & & & 1 & 1 & 1 & 0 & & & \\ & & & & & & & & & & & & 1 & 1 & 0 \end{array}\right]$$

However, by simultaneously considering constraints

$$\sum_{j \in \mathcal{A}(i)} x_{ij} - y_i = 0 \qquad \text{and} \qquad \sum_{j \in \mathcal{A}(i)} x_{ij} = 1,$$

it is immediately clear that $y_i = 1$ for all i. Therefore, we can remove the constraints (3.1) by fixing $y_i = 1$.

The final formulation of the problem is the following:

$$\begin{array}{rll} \min & \tilde{x}^T \tilde{Q} \tilde{x} & \\ \text{s.t.} & \tilde{A}\tilde{x} = 0, & \\ & y_i = 1, & i = 1, \ldots, m, \\ & x \geq 0. & \end{array} \tag{3.6}$$

There is no need to explicitly introduce upper bounds $x \leq 1$, as these are embedded in the existing constraints.

3.4 Interior point methods

In this section we provide an overview of interior point methods for quadratic problems. We refer to [6] and [12, pp. 4–14] for the following discussion.

Consider a generic quadratic problem:

$$\begin{aligned} \min \quad & c^T x + \tfrac{1}{2} x^T Q x \\ \text{s.t.} \quad & Ax = b, \\ & x \geq 0. \end{aligned}$$

In a convex problem, Q is a symmetric $n \times n$ positive semidefinite matrix.

Any interior point method replaces the non-negativity constraint on the variables with a logarithmic barrier term. The logarithmic term penalizes the objective function when a variable approaches the value 0. This forces the solution point to be in the interior of the feasible region, hence the name of these methods.

The so called barrier problem is:

$$\begin{aligned} \min \quad & c^T x + \tfrac{1}{2} x^T Q x - \mu \sum_{j=1}^{n} \ln x_j \\ \text{s.t.} \quad & Ax = b. \end{aligned}$$

The parameter μ, used in the objective function, controls the weight assigned to the logarithmic term. As the iterations proceed, its value is decreased enabling us to get closer and closer to the boundaries of the feasible region, and thus to the solution vertex.

From the Lagrangian of the barrier problem

$$L(x, \lambda, \mu) = c^T x + \frac{1}{2} x^T Q x - \mu \sum_{j=1}^{n} \ln x_j - \lambda^T (Ax - b),$$

we obtain the first order conditions for a minimizer:

$$\begin{aligned} \nabla_x L &= c + Qx - \mu X^{-1} e - A^T \lambda = 0, \\ \nabla_\lambda L &= Ax - b = 0, \end{aligned}$$

where we used $X = \operatorname{diag}(x_1, x_2, \ldots, x_n)$.

Using the notation $s = \mu X^{-1} e$ and $S = \operatorname{diag}(s_1, s_2, \ldots, s_n)$, the first order condi-

tions can be rewritten as

$$F(x,\lambda,s) = \begin{bmatrix} A^T\lambda + s - Qx - c \\ Ax - b \\ XSe - \mu e \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.7}$$

This system of nonlinear equations is solved by Newton's method. Newton's method constructs a linear approximation of the nonlinear system (3.7) and obtains a search direction d by solving the Newton equation $\nabla F d = -F$:

$$\begin{bmatrix} -Q & A^T & I \\ A & 0 & 0 \\ S & 0 & X \end{bmatrix} \cdot \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta s \end{bmatrix} = \begin{bmatrix} c + Qx - A^T\lambda - s \\ b - Ax \\ \mu e - XSe \end{bmatrix}.$$

Newton's method generates iterates (x^k, λ^k, s^k) in which x^k and s^k are strictly positive.

3.4.1 Application to our problem

An important assumption that is made in the development of interior point methods is that the problem is convex. If the problem does not have this property, the algorithm may fail.

In computer implementations, some regularizations have to be made to convexify the problem. This works when the nonconvexity is not too accentuated. In case of strong nonconvexity, however, a large regularization term may yield undesired perturbations.

As noticed, the formulation we are using is nonconvex. Therefore an adjustment has to be made when the problem is actually solved. The idea used in [3] is to decrease the nonconvexity of the problem, thus making it easier.

This is done by introducing a parameter α in the expression (3.3), that is:

$$x_i^T Q_i x_i = y_i^2 - \alpha x_i^T x_i.$$

This modification does not make the problem convex, however it allows us to obtain a smaller negative curvature as α is decreased.

3.5 Comparison with the simplex method

This section attempts to justify the choice of an interior point algorithm as a solution method for our problem. The analysis made here is not dependent on the formulation adopted, but abstracts from it and considers a generic linear program. Therefore, it is possible to make a comparison with the simplex method.

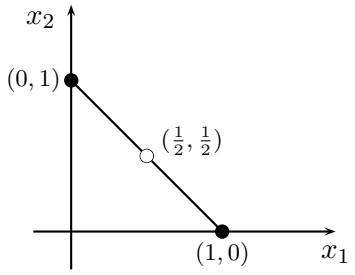
The simplex method represents the solution in terms of basic and nonbasic variables. A basic solution contains at most m nonzero variables, while the others are set to zero, as shown in (2.1). A basic feasible point corresponds to a vertex of the polytope defined by the set of constraints.

Interior point methods, on the contrary, do not distinguish between basic and nonbasic variables, as they do not move on the boundaries of the feasible region, but in the interior. Therefore, there is no upper limit on the number of nonzero variables.

All iterates generated by the simplex method are vertices, so it follows that the termination will happen at a vertex solution, even if an entire edge of face of the feasible polytope is optimal.

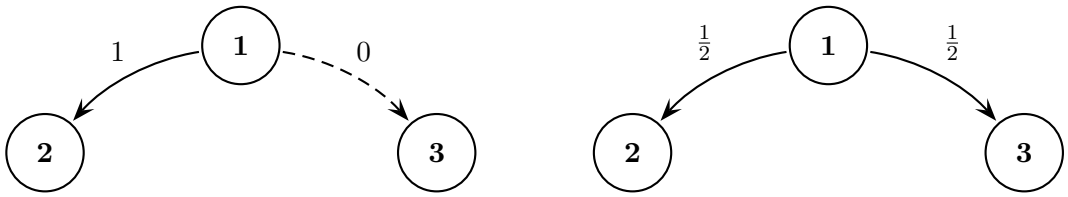
A feature of any interior point algorithm is that in the case of several optimal points lying on an edge or face of the feasible region, it will terminate in their relative interior, rather than at a vertex.

Consider this simple example.



The simplex method would choose either one of the points on the axes, represented by '•'. In contrast, the interior point solution would correspond to the '○'. In some sense, the interior point method finds both solutions simultaneously.

To show how this affects our problem, consider the next example, concerning a node with two out-neighbours. The picture on the left shows one of the two possible solutions obtained by the simplex method. On the right, the solution given by the interior point method is shown.



While the simplex chooses a local minimum (corresponding to a basic optimal solution), the interior point solution corresponds to a combination of basic optimal points. In the case of the example above, it is easy to determine what these points are by an inspection of the flow.

It would be interesting to extend this approach to the Hamiltonian Cycle Problem. Unfortunately, even for small order graphs, it is too difficult to attempt such an

analysis. In fact, a Hamiltonian cycle is just one of many possible solutions, most of which are subcycles, that is cycles containing a subset of vertices. Therefore, extracting a solution from a feasible flow is virtually impossible.

Chapter 4

Generating cutting planes

4.1 Heuristic approaches

A heuristic method for an optimization problem is an algorithm which, given an instance of the problem, finds some solution, the quality of which is generally not guaranteed when compared to an optimal solution.

Heuristics often have an intuitive justification but they lack a theoretical foundation. This does not allow to establish mathematical results on their efficiency or reliability.

In Chapter 2 we have discussed three of ways of solving hard integer programs to optimality. Depending on the specific application, these algorithms may be fast to find an optimal solution or, on the contrary, prohibitively time consuming for a computer. For example, in some cases branch-and-bound can take an enormous amount of time and memory, due to an inordinate growth of the solution tree. Gomory cuts, on the other hand, can be very slow as they may obtain only modest reductions of the feasible region.

When it is impractical to compute an optimal solution, one has to settle for a good (but not necessarily optimal) solution. This is when heuristic approaches come into play.

In the peculiar case of the Hamiltonian Cycle Problem, the notion of "optimal solution" is unconventional. If the solution of system (3.6) can be interpreted as a Hamiltonian cycle, then we have solved the Hamiltonian Cycle Problem for the given graph, and the solution is optimal; in all other cases, we have not solved the problem. In some respects, the concept of "optimality" in this problem can be represented by a binary variable and not by a continuous variable. Thus, the way we will refer to heuristics is somewhat different.

We incorporated two different heuristics in our algorithm:

1. Heuristics to generate the cutting planes;

2. Heuristics to reduce the size of the graph.

Particular attention has been paid to the first type of heuristics, as this is an essential element of the branch-and-cut approach we are proposing. Heuristics of the second type play a valuable role, but they will receive less attention as they are not typically characteristic of branch-and-cut.

4.2 The generation of cuts

This section describes how cuts are generated such that we can solve the Hamiltonian Cycle Problem.

When generating cutting planes, the flow information obtained by solving system (3.6), is important and necessary. Nevertheless, it should be combined with the information stored in the graph. In fact, by knowing the graph, more intelligent decisions can be made.

The problem we are solving is a binary integer problem. A classic branch-and-bound approach would choose a variable and fix its value to 0 or 1: we can call this approach "variable branching". A different approach, implemented by Jacek Gondzio in `hcp` (see Chapter 5 for more details), considers a vertex of the graph to be the entity upon which the branching happens. As many subproblems as there are out-neighbours of the chosen vertex are generated. In each subproblem, only one arc is kept, while all others are discarded. This approach derives from the obvious observation that in a Hamiltonian cycle only one of the arcs leaving a node is chosen. Let us call this strategy "vertex branching".

Both approaches, variable branching and vertex branching, share the same fundamental scheme: the value of one or more variables is fixed to be integer.

The approach proposed here uses quite a different perspective. The underlying idea is to force some of the fractional variables to move towards integrality. This is done by adding a constraint concerning those variables, by which we ask the value (flow) assumed by those variables to be changed in the next iteration. We may name this scheme "constraint branching".

In a sense, this may have a drawback in that the next solution will contain as many fractional values as the current one, although different. On the other hand, it is perceived that fixing one variable at a time, as it is done in the branching approaches outlined above, may be slow or lead to extremely large branching trees. For this reason, constraint branching works on more than one variable at a time, and allows the optimization problem to determine the optimal values for those variables, on the assumption that the decisions made by the solver are somewhat more "informed".

By leaving the freedom of making its own decisions to the optimization program, we are hoping to see that less important arcs (those that do not belong to any Hamiltonian cycle, or, in particular, to the one that is emerging) will receive a very small flow, and will be removed by the reduction heuristic described in Section 4.3.

By reconsidering the flow formulation, the branch-and-cut approach discussed here tries to attain two different objectives at the same time:

1. Rule out the current solution;

2. Create "some movement" in the network.

The first point can be considered as such: that we are trying to "teach" the model that the current solution is unacceptable, as it does not lead to a Hamiltonian cycle. With the second point, we are trying to create some areas of concentred information, in the hope that this will propagate to the rest of the graph.

The right-hand side of a cut

Let us assume that we are able to generate a cut, that is we know how to choose some of the existing arcs to appear in the cut.

The choice of the right-hand side associated to the cut must respect two conditions:

- As we want the flow carried by the arcs in the cut to be integer, the right-hand side must also be an integer number;

- As two cuts are generated with a different inequality sign and we do not allow to remove any integer solution, the difference in right-hand side for the two cuts must be 1.

The idea proposed here is to evaluate the total flow T shipped through the arcs in the cut, and it is determined from the latest solution of system (3.6). This is most likely to be a fractional number, as each arc in the cut receives a fractional flow.

Since we want the value associated to each arc to be integer, the total flow should be integer as well. Thus, we discard the current fractional solution by asking:

$$\begin{aligned} \text{arcs in cut} &\geq \lceil T \rceil, \\ \text{arcs in cut} &\leq \lfloor T \rfloor, \end{aligned}$$

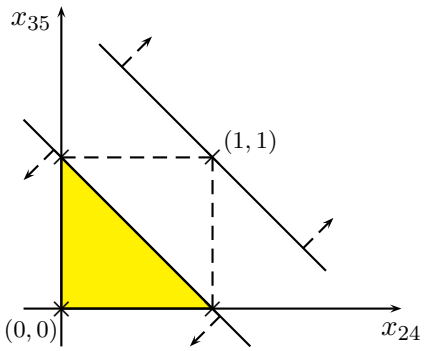
where $\lceil T \rceil$ and $\lfloor T \rfloor$ denote the next integer immediately above and below T, respectively.

An example

To visualize what happens, suppose that among all arcs of the graph, we choose arc $(2, 4)$, with current flow 0.7, and arc $(3, 5)$, with flow 0.4, to generate the cuts:

$$x_{24} + x_{35} \leq 1 \quad \text{and} \quad x_{24} + x_{35} \geq 2.$$

These cuts are depicted on the Cartesian plane by arrows pointing towards the region where the inequalities are satisfied:



In the figure, the feasible region is the shaded area plus the point $(1, 1)$; the feasible integer points are denoted by '$\times$'.

The cuts remove noninteger points as, according to our analysis, they do not lead to the creation of a Hamiltonian cycle, but they leave those points that could be needed untouched.

4.2.1 A naive cut

This type of cuts chooses two fractional arcs. It is required that the total flow carried by these arcs does not equal 1, as in such a case a constructive cut would not be generated. Indeed, in this case we would have an inequality with right-hand side 1: the current solution already satisfies it, so in the branch of that inequality there will be no improvement towards the solution.

These cuts are called "naive" because they do not exploit the graph information, but base their decisions on only the latest solution of system (3.6).

4.2.2 An intelligent cut

A more insightful approach has been studied. This approach chooses arcs at random among those leaving from vertices that are not directly connected.

Therefore, in this case, the flow information obtained by solving system (3.6) is combined with the graph information, to generate cuts that are as effective as possible.

The generation of these cuts follows this algorithm:

1. Initialize all nodes to be "unvisited".

2. Choose an unvisited node and mark it as "visited". If no unvisited vertices exist, stop.

3. Among the arcs leaving from the chosen node, select some to be placed in the cut according to the following criteria:

(a) The destination node of the arc must be an unvisited vertex;

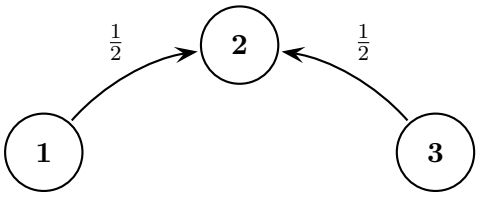
(b) Add as many arcs as necessary to reach a certain threshold on the amount of flow shipped through those arcs;

(c) If the addition of the last arc has added all of the arcs leaving from the node, remove it.

4. Mark as visited the destination nodes of the arcs put in the cut.

5. Repeat from step 2.

Consider the following example:



If we consider node 1 and add arc $(1, 2)$ in the cut, then node 2 becomes visited. Therefore, no other arcs leaving it or going into it can be chosen.

Suppose that we do not satisfy the condition stated in the algorithm and we generated the cuts:

$$x_{12} + x_{32} \leq 0 \quad \text{and} \quad x_{12} + x_{32} \geq 1.$$

The current flow already satisfies the '$\geq$' cut: the solution of the corresponding subproblem will not be different from the current one, as this cut is not constructive.

What happens in the other subproblem, corresponding to the '$\leq$' inequality, is even worse. To satisfy the cut, we have to drop both edges. But, as these are the only edges that enter in node 2, this subproblem will be removed as with this cut the problem has become infeasible.

4.2.3 Cuts in the case of subcycles

A solution vector x yields a subcycle if and only if for some nonempty subset $V_1 \subset V$ and $V_2 = V - V_1$ we have

$$\sum_{i \in V_1} \sum_{j \in V_2} x_{ij} = 0,$$

that is, none of the arcs that go from V_1 to V_2 are used.

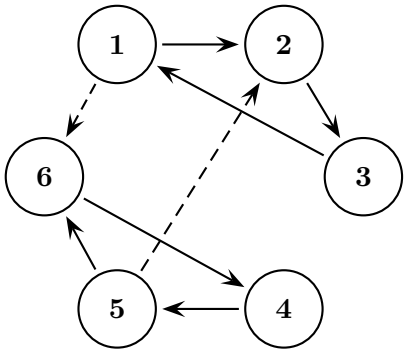
Complete cycles are the only solutions that satisfy

$$\sum_{i \in V_1} \sum_{j \in V_2} x_{ij} \geq 1,$$

for all $V_1 \subset V$, $V_1 \neq \emptyset$, $V_2 = V - V_1$.

If we find two or more subcycles that span all the vertices, we could be rather close to the solution. This is not guaranteed, but it seems worthwhile to attempt to "fix" the current situation rather than eliminate the subproblem from the branching tree.

Consider the following case, where the arcs in the current solution are represented by a solid line, while unused arcs have a dashed line.



Given the existing arcs, it is possible to recover the Hamiltonian cycle. To obtain it, we simply have to drop arcs $(1, 2)$ and $(5, 6)$, and use the dashed arcs instead.

It is important to note that in a similar situation, we do not want to create branches before having removed the subcycle. Therefore, here we will only generate one inequality.

The idea is to force a flow of 1 to leave any of the nodes in one subcycle and to enter any of the nodes in the other subcycle. By denoting $V_1 = \{1, 2, 3\}$ and $V_2 = \{4, 5, 6\}$, we can add the cut:

$$\sum_{\substack{i \in V_1 \\ j \in V_2}} x_{ij} + \sum_{\substack{i \in V_1 \\ j \in V_2}} x_{ji} \geq 2.$$

A slightly different idea is to break one of the subcycles. In other words, we want to forbid one of the cycles to appear. This is easily done with a constraint like

$$x_{12} + x_{23} + x_{31} \leq 2,$$

that avoids the case where all the variables in the subcycle have coefficient 1. This is most effective when we know of the existence of one cycle but we don't know what happens in the rest of the graph.

In the general case in which there are k subcycles that cover all vertices, the following set of cuts is added:

$$\sum_{i \in I_j} x_i \leq n_j - 1 \quad j = 1, \ldots, k,$$

where I_j is the set of indices corresponding to the nodes in subcycle j and $n_j = |I_j|$.

4.3 Reduction heuristics

The term "reduction heuristics" refers to ways of reducing the size of the graph by removing certain arcs. They have been considered for two main reasons:

- The more edges there are, the more combinations of them there are, and so the problem has many more local minima, most of them leading to subcycles in the solution.
- Every arc corresponds to a variable in the problem and in a column in the constraint matrix $\tilde{A}$ of model (3.6). This increases the sparse storage and the number of iterations required to obtain a solution.

As we have seen, after having solved the model, we get a solution interpretable in terms of flow. We can consider the amount of the flow as an indicator of the relative importance of an arc. On this basis, when an arc has a sufficiently small flow, it may mean that it is rarely used in the creation of cycles, and of Hamiltonian cycles in particular. Therefore, we may think of removing it from the graph, hoping that we are not losing the Hamiltonicity in the reduced graph thus obtained.

If the graph contains a node with a large number of outgoing arcs, it may be that the flow shipped by each of these arcs is very small in absolute terms. Clearly this should not mean that all of the arcs have to be removed. Therefore, the threshold measure must be adjusted by considering the number of out-neighbours of the node.

This type of reduction heuristic is substantially the same as the one used in [3]. In that context, however, there was a solid probabilistic justification for it. Indeed, the Markov decision processes framework adopted there allowed the translation of a flow solution into a frequency space: arcs used less frequently have a smaller probability to be needed in a Hamiltonian cycle.

In the Markov decision process approach, and here even more clearly, there is no guarantee that the arc being removed is not essential to preserve the Hamiltonicity of the graph. We are again dealing with a heuristic rule that, while effective on many occasions, may be in error on others.

To comment upon the importance of having some reduction heuristics, we notice that even just a single arc removal may lead to a series of subsequent simplifications in the graph, obtainable by logical analysis of the reduced graph. These will be discussed in Section 5.2.3.

Chapter 5

The computer implementation

5.1 Overview

A large proportion of time during this project has been devoted to the implementation of a branch-and-cut algorithm. In particular, this required the creation of a new structure to store the information relating to cutting planes, and of the functions to generate and update them.

This implementation has been based on code originally written by Jacek Gondzio in the C programming language. This code was designed to solve, using branch-and-bound, the Hamiltonian Cycle Problem expressed in terms of the nonconvex quadratic formulation presented in Chapter 3 and embedded in a Markov decision process framework (consult [3] for more details). We will refer to this program as `hcp`.

The source code provided contained a large set of routines written to manage graphs, enumeration trees, and the interfacing with the `HOPDM` (Higher-Order Primal-Dual Method) interior point solver. Despite all these pre-existing functions, the changes that had to be made were substantial.

Using somebody else's code and getting acquainted with it is a major task. It takes some time to get used to the conventions in the code, and to understand why certain design principles have been adopted.

The programming part of this project involved the following steps:

- The first step was to remove the Markov decision process formulation. This was relatively straightforward, as the changes were confined to specific functions.
- The removal of the branch-and-bound implementation required suppressing the functions that were choosing the branching variable, and the occurrences of that variable. These changes were more difficult, as they were spread in several places in the code.
- Functions in the `hcp` code were few and large. To get a clearer picture of what they were used for, they have been split into smaller routines.

This needed a deep understanding of what the functions were used for, what parameters they expected, and what changes they were or were not allowed to make. A careful design of calling points and return codes was required as well. This overhaul made the code more modular and manageable. The time spent on this step proved to be time saved when facing debugging problems.

- At this point, almost everything was set up for the creation of the Cut structure. This was a challenging job, as the structure had to be designed from scratch. Most of this work was independent from the rest of the code, as the Cut structure was designed to be a self-contained component.
- The next step was to integrate the Cut structure into the main program. This required writing other code to glue together these distinct portions of code. Moreover, since it was desirable that the set of cuts matched the graph they were associated with, the storage of the Cut structure was moved inside the Graph structure. Luckily, this required only minor changes to the Graph structure and the related functions.
- The second major difficulty concerned the routines actually generating cuts. Details about those functions are given in Section 5.3.

5.2 Implementation details

5.2.1 Algorithm

The program developed follows this algorithm:

- Read the graph file.
- Initialize the root node of the solution tree.
- Solve the root problem and generate a cutting plane.
- While the iteration limit has not been reached:
    - Look for the most attractive leaf node of the tree.
    - Analyze the chosen node:
        - Create two children nodes, append the latest cut generated with a different inequality sign in each subproblem.
        - For each subproblem, solve the subproblem and generate a cutting plane.

The analysis of each subproblem executes the following steps:

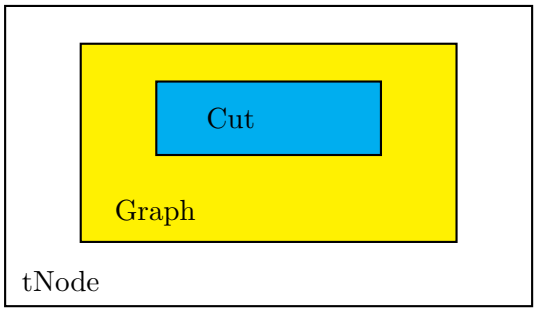
- Apply simplification logics.

- Formulate and solve the QP version of the problem by calling `HOPDM`.
- Apply the reduction heuristic and eliminate those edges for which the flow is particularly small.
- Generate a cut and append it to the problem.
- Create two branches.
- Solve the problem in each branch.

5.2.2 Structures

There are three main structures used in the program, the tNode structure, the Graph structure and the Cut structure. While the first two have been inherited from the `hcp` code, the Cut structure has been written from scratch.

The relationship between these structures may be visualized in the following figure.



Details of their content are given next. Not every member variable of the structures will be listed, as some of them have only book-keeping purposes, and their explanation would only interfere without providing any additional insight.

The tNode structure

The tNode structure is the outermost one. It keeps track of the information concerning a node of the branching tree.

```
struct  {
  /* Tree-related information */
  ptNode   father;   // father of this treenode
  ptNode*  sons;     // array of sons of this treenode
  int      NoSons;   // number of sons of the node
  /* Graph-related information */
  Graph*   G;        // Graph structure
} tNode;
```

The Graph structure

The Graph structure contains the list of origins and destinations of all arcs and also the linked lists.

```
struct {
  /* Graph-related information */
  int     m;         // number of nodes
  int     n;         // number of arcs
  int*    orig;      // array of origins of arcs
  int*    dest;      // array of destinations of arcs
  /* Adjacency lists */
  int*    OrigFirst; // array of first out-neighbour of a node
  int*    OrigNext;  // array of next out-neighbour of a node
  int*    DestFirst; // array of first in-neighbour of a node
  int*    DestNext;  // array of next in-neighbour of a node
  /* Cut-related information */
  Cut*    SetOfCuts; // Cut structure
} Graph;
```

The Cut structure

The Cut structure stores all the cutting planes that have been generated, together with the associated inequality signs and right-hand signs.

A single cut is stored in a dense array whose dimension corresponds to the number of arcs in the graph. The index corresponding to an arc in the cut contains the value 1, otherwise 0.

Its definition is the following:

```
struct {
  int n;      // number of cuts
  int dim;    // length of the cut
  int* rhs;   // array of right-hand sides
  int* sign;  // array of inequality signs: 0 for <=, 1 for >=
  int** cuts; // set of arrays, one for each cut
} Cut;
```

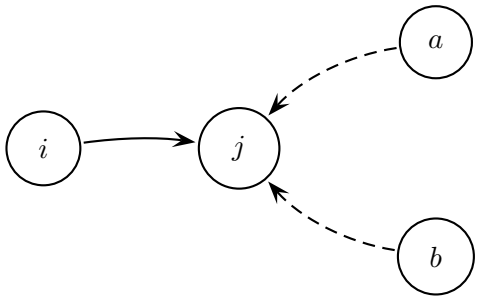
5.2.3 Simplification logics

Simplification logics aim to reduce the size of the graph by a logical analysis of the graph. In fact, in some cases it is certain that some of the edges will never be used. Therefore, their removal will not compromise the Hamiltonicity of the graph.

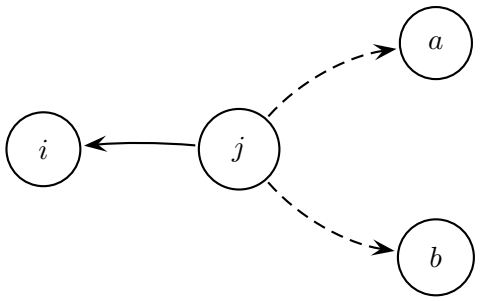
It is the case that an arc removal done by the reduction heuristic may lead to a series of consequent arc removals done by logical rules.

There are two main rules:

- If node i has only one outgoing arc, leading to node j, then arc (i, j) will definitively be used to create any Hamiltonian cycle. Therefore, all other arcs that terminate into node j can be removed, since they are not allowed to ship any positive flow to that node.



- If node i has only one incoming arc, arriving from node j, then arc (i, j) will definitively be used to create any Hamiltonian cycle. Therefore, all other arcs that depart from node j can be removed.



5.3 Implementation of the cuts

The routines that generate cuts have some requirements to satisfy:

1. They need to know the current solution: this is necessary in order to identify the variables that have a fractional value.

2. They must be aware of the current structure of the graph: this is required as we want to exploit as much information as we can, while generating cuts that are not contradictory.

5.3.1 The naive cuts

This naive implementation, introduced in Section 4.2.1 has originally been written so that other parts of the program could be developed.

The following piece of code shows the actual implementation of this type of cuts.

```
// read each element of the solution array
for (i = 0; i < dim; i++) {
  // consider an arc only if its current flow
```

```
  // is not too close to an integer value
  if ((pSol[i] > tol) && (pSol[i] < 1.0 - tol)) {
    aCut[i] = 1;
    arcsInCut++;
    flow += pSol[i];
    // avoid taking two arcs that add up to 1
    // as the cut would not be constructive
    if (fabs(flow - 1.0) < eps) {
      aCut[i] = 0;
      arcsInCut--;
      flow -= pSol[i];
    }
  }
  // stop when the cut contains two arcs
  if (arcsInCut == 2)
    break;
}
```

The flow currently associated to an arc is read from the solution vector `pSol`. If it is sufficiently far from 0 and 1, the arc is placed in the cut. The algorithm tries to avoid the situation when the flow of the arcs in the cut is equal to one (the parameter `eps` is a small number): in that case, the last cut added is removed and the algorithm continues. When the cut contains two arcs, the algorithm stops. The right-hand side associated to this cut is determined by the flow shipped by the arcs in the cut.

Unfortunately, due to time restrictions, it has not been possible to work around the main shortcoming of this routine, that is the cut generated may be exactly the same as the one obtained at the previous iteration.

This happens when the application of a cut changes the solution only slightly. In those cases, the flow is likely to be quite close to the previous flow, and the algorithm will repeat the same choice.

The flow associated to the edges is read sequentially. In the "forward-only" implementation, the solution vector is always read from the beginning. In the so called "mixed-direction" implementation, the reading may start from the beginning, the middle, or the end of the vector `pSol`. The decision on the order by which the vector is read is not completely random, but depends on the current number of edges in the graph, according to this code:

```
int r = dim % 3;
if (r == 0)
  return naiveCutForward(pSol, dim, b);
else if (r == 1)
  return naiveCutMiddle(pSol, dim, b);
else
  return naiveCutBackward(pSol, dim, b);
```

This approach works because the size of the graph may decrease from iteration to iteration, and thus the arcs are read from different positions.

A better solution would probably be to let a random number generator decide which specific cut must be called. This way, the algorithm would not repeat the same cut over and over when the solution changes only slightly.

Another "workaround" would be to compare the generated cut with all other cuts. If the newly created cut is a duplicate of an existing one, it is discarded and a different routine is called.

5.3.2 The intelligent cuts

The implementation of the intelligent cut (refer to Section 4.2.2) is far more complex than the previous one. This is due to the fact that besides taking into account the flow information, we are trying to exploit the Graph structure as well.

To provide a clear picture of this routine, the code will not be presented as a whole, but it will be split into different parts.

First of all, we have to count the number of arcs that leave from every node. These quantities will be stored in the `noOrigs` array. This step is done only once at the beginning of the routine.

```
// count the number of outgoing arcs from every node
for (i = G->n - 1; i >= 0; i--) {
  // add to the numbers only if the arc is being used
  if (G->UsedArcs[i]) {
    noOrigs[G->orig[i]]++;
  }
}
```

The next steps are repeated as many times as there are unvisited nodes. The variable `unvisitedNodes` keeps track of their number.

We start by choosing the node with largest number of out-neighbours. The decision to consider such a node first depends on the intuitive idea that, being highly connected in the graph, this node may provide an effective constraint branching. In the meantime, in the event of the chosen node having only one out-neighbour, we will avoid it, as it would not yield a constructive cut.

```
// choose the node with the largest number of out-neighbours
for (i = 1; i < G->m; i++) {
  if ((noOrigs[i] > noOrigs[node]) && (usedInCut[i] == 0))
    node = i;
}
// avoid a node that has only one out-neighbour
if (noOrigs[node] <= 1) {
  unvisitedNodes--;
  continue; // go to the next iteration
```

```
}
```

Once we have chosen the node, we have to select some of the edges that depart from that node. For every arc, we have to check that the arc does not end in a visited node. Again we check that the flow carried by the arcs that are in the cut does not sum up to 1: this would mean that we have chosen all the arcs leaving from the chosen node, and one of the subproblems would generate be infeasible.

```
// check that the arc does not end in a node already considered
if (visited[G->dest[arc]] == 0) {
  // consider an arc only if its current flow
  // is not too close to an integer value
  if ((pSol[k] > tol) && (pSol[k] < 1.0 - tol)) {
    flow += pSol[k];
    // check that the flow does not add up to 1
    if (1.0 - flow < tol) {
      flow -= pSol[k];
    }
    else {
      aCut[k] = 1;
      visited[G->dest[arc]] = 1;
      unvisitedNodes--;
      if (flow > threshold)
       break; // go to the next node
    }
  }
}
```

When we have finished considering all outgoing arcs of a node, we add the partial flow to the total flow shipped by all arcs present in the cut, and mark visited the node under consideration.

```
if (flow > 0.0) {
  totalFlow += flow;
  visited[node] = 1;
  unvisitedNodes--;
}
```

When there are no more unvisited nodes, the routines need only to determine the right-hand side before leaving:

```
rhs = ceil(totalFlow);
```

While the routine that generates this type of cut has been written and works according to the specifications, several difficulties have been encountered in the actual use of this type of cuts.

The main problem concerns the updating of existing cuts. The problem arises when, after some arc removals, all out-neighbours of a node are left in the cut. In this case, the cut may become infeasible.

Updating existing cuts

When arc removals occur, due to the reduction heuristic or to the simplification logics, the existing cuts, as stored in the Cut structure, have to be updated to reflect the new number of existing arcs.

If an arc being removed is used in the cut, the corresponding entry in the Cut structure has to be changed from 1 to 0. This needs to be reflected onto the right-hand side. Therefore, at this point we have to evaluate the amount of flow carried by the edges left in the cut, and adjust the right-hand side accordingly.

It may occur that after some arc removals, a cut becomes empty. A routine that checks these cases has been implemented.

```
// count the number of arcs in the cut
for (j = 0, nArcsInCut = 0; j < C->dim; j++) {
  if (C->cuts[i][j])
    nArcsInCut++;
}
// mark the cut for removal if it has become empty
if (nArcsInCut == 0) {
  C->cuts[i][0] = -1;
  nRemovedCuts++;
}
```

The routine scans each array that stores the cuts, and counts the number of arcs that appear in the cut. If an empty cut is found, the cut is marked for deletion, by assigning -1 to the first entry of the array. At the next iteration this cut will be discarded from the set of cuts.

5.4 Possible extensions

A sparse representation of cuts

The cuts have been implemented as dense arrays. This decision was taken so the development could proceed swiftly.

A dense representation has a dramatic impact on storage requirements. This is especially true when the number of edges is large, and several cuts have been added to the problem.

Furthermore, this representation is not justified. Each cut contains a subset of arcs (just two arcs in the case of the naive cut), and the indices corresponding to the arcs left out are set to zero. Moreover, the coefficient of the arcs in the cut, in the implementation adopted, is always 1. Therefore, a sparse implementation needs only to store the indices corresponding to the arcs that contribute to the cuts.

Cuts for subcycles

When we encounter a subcycle in a flow solution, we should generate an inequality that attempts to remove the subcycle. To be able to obtain such a cut, we must know which arcs belong to the subcycles. Given the existing implementation, it was too difficult to determine exactly which arcs where involved.

A different implementation of the routine that checks for subcycles is probably needed, so that a list of the nodes in a subcycle can be easily generated.

Checking for duplicate cuts

It may be that the latest generated cut is identical to an existing one. In such a situation, the cut is not constructive, and one subproblem is solved without leading closer to the solution.

Therefore, it seems reasonable to check if the cut created is a duplicate one. This check can be done in two levels of detail:

- Compare the new cut only with the previous one;
- Compare the new cut with all the existing cuts.

The first situation is easily implemented and is not particularly expensive. It may be especially useful when there is no randomness in the generation of cuts, as in those cases, the same cut may be created again and again, without making any progress towards the solution.

The second type of check requires a careful implementation if we want to keep the cost of this check low. Such a routine is certainly too expensive if the cuts are stored in a dense format.

Chapter 6

Computational experience

6.1 Details of the experiments

Test problems

The implementation of the program, as detailed in Chapter 5, has been evaluated in a series of test problems. Each test problem consists of a graph represented in a file with the following form:

```
5 10
  0 1
  1 2
   ⋮
  4 2
```

The first line in the file states the number of nodes and edges in the graph. Each of the following lines describes an arc, in the form origin node – destination node.

The test problems ranged from the example graph of Chapter 1 (5 nodes and 10 arcs), to the graph corresponding to the Knight's Tour Problem on a board of size 20 (400 nodes and 2736 arcs).

The following table displays, for each test problem used, the number of nodes and edges of the corresponding graph.

Problem	Nodes	Arcs
`r5-10`	5	10
`r8-16`	8	16
`star51`	10	30
`r61-182`	61	182
`chess8`	64	336
`chess10`	100	576
`chess12`	144	880
`chess14`	196	1248
`chess20`	400	2736

The last two test problems have been used only on the algorithms that proved to perform better.

Formulations

Alongside the nonconvex QP approach detailed in Chapter 3, two other formulations of the problem have been investigated. Both use the same set of constraints as in model (3.6), but the objective function is built differently.

In one case, a linear objective is used, in the form

$$\min\ e^T x,$$

where $e = (1, 1, \ldots, 1)$.

In another case, a heuristic chooses the nodes to be put in the quadratic term, creating a hybrid formulation in which only some of the nodes contribute to the QP term. At each iteration, the node with the largest number of out-neighbours is designated to appear in the objective, which now has the form

$$\min \sum_{i \in I} \tilde{x}_i^T \tilde{Q}_i \tilde{x}_i,$$

where $\tilde{Q}$ and $\tilde{x}$ are defined in (3.5), and I is the set of indices corresponding to the vertices that contribute to the objective.

Different values of the nonconvexity parameter α, (see Section 3.4.1), ranging from 1.0 to 0.01 have been tested.

Cuts tested

Due to the incomplete implementation of intelligent cuts, the computational experience has considered only the naive cuts.

Two different implementations of the naive cut have been tested:

1. The mixed-direction naive cut;
2. The forward-only naive cut.

Result tables

In the result tables, the problems are ordered by number of nodes. This does not imply that graphs of large orders are more complex than smaller ones. Other considerations come into play, such as the number of edges and the actual connections in the graph. A graph close to being complete is definitely simpler than one in which exactly one Hamiltonian cycle exists.

The problems had a limit of 500 iterations, at which the program would stop. In this case, the table shows (NT), as the program has not terminated.

In several cases, mainly when the parameter α was set to a large value, many problems were rejected as too many iterations were needed by `HOPDM`. In these situations, the node would not generate any subproblems. If this occurs too often, it is possible that no more tree nodes can be investigated, and the program stops running. These occurrences are denoted in the tables by (n), where n is the number of subproblems solved before the premature termination occurs.

In a few cases, the early termination happens at the root node, and so no subproblem is generated. As the branching procedure is not started, these occurrences are marked by (NS) in the result tables.

For the complete QP and the linear formulations, we report the number of subproblems solved, the number of cuts used in the last subproblem, and the time in minutes required to reach the solution. For the hybrid formulation, the data given also considers the number of QP terms added to the objective. For `hcp`, only the number of subproblems solved and the time used are given.

6.2 Analysis of the results

Here we provide an analysis of the outcomes from the computational experience. In spite of being limited and not conclusive, it shows some remarkable results. Detailed results can be found in the Appendix. Here we will discuss some of the relevant observations that can be drawn from those results.

Let us first concentrate on the complete QP formulation. It appears that this formulation is not robust enough to be competitive. In fact, while it was able to solve all but one problem with $\alpha = 1.0$, it consistently failed with smaller values of α. Why this occurred has yet to be investigated.

The cost of each iteration is much higher than expected. A separable QP problem is generally at least as fast as a linear problem. The fact that the formulation adopted is nonconvex dramatically increases the computing time required to converge to the solution: many iterations are needed, and each of them has a large cost.

The hybrid approach proved to be robust, failing only in one case when $\alpha \geq 0.1$. The number of QP terms needed was always relatively small, being consistently below 40% of the number of nodes of the graph (below 26% when $\alpha = 0.01$ on graphs with more than 10 nodes).

The linear approach is very fast due to its simpler formulation that makes the interior point iterations extremely inexpensive. This compensates for the fact that they have to solve a larger number of subproblems and that they need the generation of more cuts. Its advantage is particularly evident when compared to the other formulations with $\alpha \geq 0.1$. But for $\alpha = 0.01$, as the size of the problems increases, the hybrid approach improves its efficiency and becomes much faster.

To verify this statement, a special series of tests were run, involving problems of

a particularly large size. This involved the hybrid QP approach ($\alpha = 0.01$), the linear approach and `hcp`. Both were solved using the mixed-direction naive cut. The results obtained are presented in the table below.

Problem	Hybrid				Linear			`hcp`	
	Probs	Cuts	QPs	Time	Probs	Cuts	Time	Probs	Time
`chess14`	232	19	55	1.48	264	51	2.18	543	14.58
`chess20`	354	48	119	9.27	670	117	14.42	746	115.31

From this table, it is particularly interesting to compare the number of subproblems solved by the different algorithms. The two branch-and-cut approaches required less than 50% of the number of subproblems solved by `hcp`. The hybrid formulation is only marginally faster on `chess14`, but the performance gap increased on `chess20`. Comparing the two branch-and-cut formulations, the number of cuts needed was double when the objective was linear.

If we now consider the value of α, we notice that it has an important impact on the program. With a small value of α, problems become easier for the solver. Therefore, we gain the chance of investigating the nodes more deeply. Since failures of the `HOPDM` solver are rarer, we minimize the possibility of prematurely discarding tree nodes.

Let us compare the results obtained by the branch-and-cut implementation with those obtained by `hcp`. Recall that `hcp` implements branch-and-bound on a complete QP formulation. The difference in speed is imperceptible on small size problems (up to 10 nodes), but it becomes manifest when problems are bigger.

The case of problem `chess10`, in the hybrid formulation when $\alpha = 1.0$ and the mixed-direction cut is applied, points out the fact that the heuristic may occasionally take incorrect paths, and thus lengthen the exploration of the solution tree. There is no way to predict when this may happen, therefore it must be taken into account in the evaluation of the algorithm.

Despite their simple nature, the naive cuts have managed to solve a fairly large set of test problems. This is considered to be an achievement that shows the effectiveness of the branch-and-cut approach.

It should be noted that the efficiency of the program may have been affected negatively by the current nonsparse implementation of the cuts. This is probably more evident when the number of edges is very large.

The results obtained as the size of the problems increased are not conclusive, as the biggest test problems consisted of large instances of the Knight's Tour Problem. This type of problem has a definite structure, and it is possible that the way the naive cuts are generated allows the exploitation of it. Therefore, the results presented must be considered as preliminary.

Conclusions and further research

The objectives of this dissertation were to study ways of generating effective cutting planes for solving the Hamiltonian Cycle Problem, and to obtain a working implementation of a branch-and-cut algorithm based on these cuts. While different ideas were proposed (naive cuts, intelligent cuts, and cuts in case of subcycles), only two of them have been implemented.

The naive cuts, in both configurations proposed (mixed-direction and forward-only), proved to be stable and fast. They are certainly very quick to generate and update. More intensive testing is needed on a variety of large problems, in order to clarify the strengths and weaknesses of this approach.

Unfortunately, the implementation of the most promising of the proposed techniques, the intelligent cuts, was not so sufficiently advanced as to be tested in the computational experience run.

The nonconvex QP formulation proved to be an interesting theoretical framework. Its practical performance, however, has not displayed those characteristics of robustness and speed that are asked to an effective algorithm. Nonetheless, inside the nonconvex QP framework, it has been possible to devise a hybrid approach which, in the computational experience, has outperformed both the complete QP and the linear formulations.

The hybrid objective function was created by using a very simple heuristic, in order to make the computer implementation as easy as possible. It may well be the case that other heuristics perform better, both in terms of earlier convergence and in terms of computational time. This may depend on the choice of the nodes that contribute to the quadratic objective and on the number of these terms. In our implementation we have chosen the nodes with largest number of outgoing arcs, but other strategies may be explored (for example based on the number of incoming arcs). In order to keep the number of QP terms small, these terms may be used for a certain number of iterations, after which they are removed.

From the results, it appears that by decreasing α, the hybrid method acquires speed. This is true for the values tested, and it seems interesting to verify to which extent this parameter can be reduced without losing robustness or speed.

The code implementation has several shorcomings that have already been pointed out. It appears that there is a certain range of improvements achievable in the efficiency of the program, in particular if a sparse storage of the cuts is implemented.

Appendix

Mixed-direction naive cut

$\alpha = 1.0$

Problem	QP			Hybrid			
	Probs	Cuts	Time	Probs	Cuts	QPs	Time
r5-10	4	1	<0.01	4	1	2	<0.01
r8-16	8	2	<0.01	6	2	2	<0.01
star51	8	3	<0.01	18	5	3	0.01
r61-182	(NS)	-	-	(109)	-	-	-
chess8	56	6	0.37	110	17	27	0.40
chess10	112	13	1.50	450	23	46	5.50
chess12	148	16	3.57	186	31	61	3.18

$\alpha = 0.1$

Problem	QP			Hybrid			
	Probs	Cuts	Time	Probs	Cuts	QPs	Time
r5-10	4	1	<0.01	4	1	2	<0.01
r8-16	8	2	<0.01	2	1	1	<0.01
star51	(7)	-	-	14	2	2	<0.01
r61-182	(77)	-	-	(43)	-	-	-
chess8	(NS)	-	-	148	8	16	0.27
chess10	(NS)	-	-	150	16	30	1.07
chess12	(145)	-	-	246	26	46	2.06

$\alpha = 0.01$

Problem	QP			Hybrid			
	Probs	Cuts	Time	Probs	Cuts	QPs	Time
r5-10	4	1	<0.01	4	1	2	<0.01
r8-16	(7)	-	-	4	2	2	<0.01
star51	8	3	<0.01	12	3	4	<0.01
r61-182	(NS)	-	-	52	7	6	0.04
chess8	(7)	-	-	44	7	13	0.06
chess10	(15)	-	-	54	12	26	0.17
chess12	(9)	-	-	92	41	17	0.41

Forward-only naive cut

$\alpha = 1.0$

Problem	QP			Hybrid			
	Probs	Cuts	Time	Probs	Cuts	QPs	Time
r5-10	4	1	<0.01	4	1	2	<0.01
r8-16	(5)	-	-	6	2	2	<0.01
star51	10	3	<0.01	10	3	3	<0.01
r61-182	(33)	-	-	(155)	-	-	-
chess8	132	10	1.09	90	10	25	0.36
chess10	94	12	1.30	82	11	37	0.50
chess12	202	26	3.38	158	22	66	2.43

$\alpha = 0.1$

Problem	QP			Hybrid			
	Probs	Cuts	Time	Probs	Cuts	QPs	Time
r5-10	4	1	<0.01	4	1	2	<0.01
r8-16	8	2	<0.01	2	1	1	<0.01
star51	10	3	<0.01	8	2	2	<0.01
r61-182	30	6	0.10	8	4	4	0.01
chess8	(NS)	-	-	24	10	12	0.04
chess10	(NS)	-	-	19	9	28	0.29
chess12	(37)	-	-	146	26	39	1.26

$\alpha = 0.01$

Problem	QP			Hybrid			
	Probs	Cuts	Time	Probs	Cuts	QPs	Time
r5-10	4	1	<0.01	4	1	2	<0.01
r8-16	(7)	-	-	4	2	2	<0.01
star51	(15)	-	-	10	3	3	<0.01
r61-182	(NS)	-	-	(85)	-	-	-
chess8	(7)	-	-	24	6	11	0.04
chess10	(5)	-	-	52	11	24	0.16
chess12	(23)	-	-	90	8	36	0.42

Linear programming formulation and `hcp`

Problem	Mixed-direction			Forward-only			`hcp`	
	Probs	Cuts	Time	Probs	Cuts	Time	Probs	Time
r5-10	4	1	<0.01	4	1	<0.01	2	<0.01
r8-16	12	3	<0.01	12	3	<0.01	2	<0.01
star51	12	4	<0.01	10	3	<0.01	8	<0.01
r61-182	190	7	0.14	60	5	0.04	78	0.12
chess8	62	14	0.07	90	20	0.08	43	0.19
chess10	120	27	0.27	98	9	0.20	122	2.11
chess12	190	34	1.12	148	22	0.51	169	4.23

Bibliography

[1] Bang-Jensen, J. and Gutin, G. (2001), *Digraphs: theory, algorithms and applications*, Springer, London.

[2] Berge, C. (1962), *The theory of graphs and its applications*, Methuen, London.

[3] Filar, J., Gondzio, J., and Ejov, V. (2003), *An interior point heuristic for the Hamiltonian Cycle Problem via Markov decision processes*, Technical Report, University of South Australia.

[4] Garfinkel, R. S. and Nemhauser, G. L. (1972), *Integer programming*, John Wiley and Sons, New York.

[5] Gibbons, A., (1985), *Algorithmic graph theory*, Cambridge University Press, Cambridge.

[6] Gondzio, J., (2002), *Practical large-scale optimization*, Lecture Notes, University of Edinburgh.

[7] Hillier, F. S. and Lieberman, G. J. (2001), *Introduction to operations research*, 7th ed., McGraw-Hill, New York.

[8] Papadimitriou, C. H. and Steiglitz, K. (1998), *Combinatorial optimization: algorithms and complexity*, Dover Publications, New York.

[9] Polimeni, A. D., Straight, H. J. (1990), *Foundations of discrete mathematics*, 2nd ed., Brooks/Cole, Pacific Grove.

[10] Schrijver, A. (1986), *Theory of linear and integer programming*, John Wiley and Sons, New York.

[11] Wilson, R. J. and Watkins, J. J. (1990), *Graphs: an introductory approach*, John Wiley and Sons, New York.

[12] Wright, S. J. (1997), *Primal-dual interior-point methods*, SIAM, Philadelphia.