Algorithms for Maximum Independent Set Applied to Map Labelling

Tycho Strijk, Bram Verweij, and Karen Aardal

September 27, 2000

Abstract

We consider the following map labelling problem: given distinct points $\boldsymbol{p}^1, \boldsymbol{p}^2, \ldots, \boldsymbol{p}^n$ in the plane, and given σ, find a maximum cardinality set of pairwise disjoint axis-parallel $\sigma \times \sigma$ squares $Q_1, Q_2, \ldots, Q_r$. This problem reduces to that of finding a maximum cardinality independent set in an associated graph called the conflict graph. We describe several heuristics for the maximum cardinality independent set problem, some of which use an LP solution as input. Also, we describe a branch-and-cut algorithm to solve it to optimality.

The standard independent set formulation has an inequality for each edge in the conflict graph which ensures that only one of its endpoints can belong to an independent set. To obtain good starting points for our LP-based heuristics and good upper bounds on the optimal value for our branch-and-cut algorithm we replace this set of inequalities by the set of inequalities describing all maximal cliques in the conflict graph. For this strengthened formulation we also generate lifted odd hole inequalities and mod-k inequalities.

We present a comprehensive computational study of solving map labelling instances for sizes up to $n = 950$ to optimality. Previously, optimal solutions to instances of size $n \leq 300$ have been reported on in the literature. By comparing against these optimal solutions we show that our heuristics are capable of producing near-optimal solutions for large-scale instances.

1 Introduction

The automated map labelling problem is a well-known problem in cartographic and graphical information systems research. Manual label placement is a time-consuming task and it is natural to try to automate it. Automating map labelling requires an abstract formulation of the problem. Early formulations, which were based on rules, were given by Imhof [21], and Yoeli [43]. In this paper we consider a highly abstracted formulation that can be found in which it is only enforced that labels are placed close to the point they refer to and do not overlap. Such formulations are commonly used in the literature (see, e.g., Forman and Wagner [16]).

The map labelling community has mainly focused on heuristics. For modern applications, such as embedding automated map labelling in real-time systems for navigation, being able to produce a good map fast is essential. However, when one wants to produce a map that is going to be printed it can be well worth the effort to try and find an optimal labelling.

Christensen, Marks, and Shieber [7, p. 219] reported that optimising is impractical for map labelling instances with more than 50 cities. In the same year, Wagner and Wolf [37] mentioned that they were able to solve instances with up to 300 cities to optimality. Verweij and Aardal [36] were the first to show that it is computationally feasible to compute provably optimal labellings for maps with up to 800 cities. In this paper we address both heuristic algorithms and optimization algorithms for map labelling. Using the techniques proposed in this paper we can solve to optimality instances with up to 950 cities within reasonable time.

1.1 Problem Definition

The basic map labelling problem can be described as follows: given a set $P = \{\boldsymbol{p}^1, \boldsymbol{p}^2, \ldots, \boldsymbol{p}^n\}$ of n distinct points in $\mathbb{R}^2$, determine the supremum σ^* of all reals σ for which there exist n pairwise disjoint, axis-parallel $\sigma \times \sigma$ squares $Q_1, Q_2, \ldots, Q_n \subset \mathbb{R}^2$, where $\boldsymbol{p}^i$ is a corner of Q_i for all $i = 1, \ldots, n$. By "pairwise disjoint squares" we mean that no overlap between any two squares is allowed. Once the squares are known, they define the boundaries of the areas where the labels can be placed. We will refer to this problem as the *basic* map labelling problem. The *decision variant* of the map labelling problem is to decide, for any given σ, whether there exists a set of squares $Q_1, \ldots, Q_n$ as described above. Formann and Wagner [16] showed that this latter problem is $\mathcal{NP}$-complete.

Kučera, Mehlhorn, Preis, and Schwarzenecker [26] observed that there are only $O(n^2)$ possible values that σ^* can take. Optimising over those can be done by solving the decision variant only $O(\log n)$ times, using binary search with different values of σ. So, the basic map labelling problem reduces to the decision variant. Kučera *et al.* also present an exponential-time algorithm that solves the decision variant of the map labelling problem to optimality. They do not, however, report on computational experiments. Wagner and Wolf [37] mention that they were able to use the algorithm proposed by Kučera *et al.* to solve problems with up to 300 cities to optimality.

Another variant of the map labelling problem, which we will refer to as the *optimisation variant*, has the label size σ as input, and asks for as many pairwise disjoint squares of the desired characteristic as possible. If the number of squares in an optimal solution to the optimisation variant of the map labelling problem equals n, then this solution is a feasible solution to the decision variant, and vice versa. Hence, the decision variant of the map labelling problem reduces to the optimisation variant via a polynomial reduction. It is the optimisation variant of the map labelling problem that is the subject of this paper. Since the decision variant is $\mathcal{NP}$-complete, the optimisation variant is $\mathcal{NP}$-hard.

1.2 Related Literature

A recent survey on map labelling is given by Neyer [31]. An exhaustive bibliography concerning map labelling is maintained by Wolff and Strijk [39]. Formann and Wagner [16] developed a $\frac{1}{2}$-approximation algorithm for the size optimisation variant of the map labelling problem. They also proved that no algorithm exists with a guaranteed approximation factor better than $\frac{1}{2}$ provided that $\mathcal{P} \neq \mathcal{NP}$. Wagner and Wolff [37] proposed a hybrid heuristic using the above $\frac{1}{2}$ approximation algorithm.

Agarwal, Kreveld and Suri [1] presented a polynomial time approximation scheme for the optimisation problem. They also showed that when the labels are not restricted to have equal height, the problem can be approximated within $O(\log n)$ of the optimum. For the optimisation problem different heuristic algorithms (including simulated annealing) are discussed by Christensen, Marks, and Shieber [7]. Van Dijk, Thierens, and de Berg [13] considered genetic algorithms, Cromly [8] proposed a semi-automatic LP-based approach for finding feasible solutions to the optimisation variant. Zoraster [44, 45] used Lagrangian relaxation to make a heuristic algorithm for the optimisation variant.

All the results mentioned so far concern the problem in which each point is labelled with one (square) region from a finite set of candidate (square) regions. Kakoulis and Tollis [22] exploit this to unify several slightly more general map labelling problems. A different approach reported on by van Kreveld, Strijk, and Wolff [25] and Klau and Mutzel [24] is to allow a label to take any position, as long as its corresponding point is on its boundary. This leads to so-called *sliding* map labelling models. The advantage of a sliding model is that more labels can be placed without overlap. In more recent studies also other shapes of the label regions are considered (see e.g. Qin, Wolff, Xu, and Zu [34]).

In this paper we study the optimisation variant of the basic map labelling problem. In the remainder of this paper we will refer to it as the map labelling problem. This study is an extension of the work by Verweij and Aardal [36, 35].

1.3 Outline

In Section 2 we show that the map labelling problem reduces to a maximum independent set problem on an associated graph. Section 3 reviews several formulations of the maximum independent set problem that can be used for algorithmic design. We approach map labelling instances using algorithms for finding large or optimal independent sets. These are the subject of Sections 4 and 5, respectively. In Section 6 we present computational results of both our heuristics and our optimisation algorithms from Sections 4 and 5. Our computational results are obtained on graphs derived from an interesting class of randomly generated map labelling problem instances that is taken from the literature. We show that our heuristic algorithms yield close to optimal solutions, and that our optimisation algorithms are capable of solving map labelling problem instances with up to 950 cities to optimality within reasonable time.

1.4 Notation

If E and S are sets, where E is finite, then S^E is the set of vectors with $|E|$ components, where each component of a vector $\boldsymbol{x} \in S^E$ is indexed by an element of E, i.e., $\boldsymbol{x} = (x_e)_{e \in E}$. For $F \subseteq E$ the vector $\chi^F \in S^E$, defined by $\chi^F_e = 1$ if $e \in F$ and $\chi^F_e = 0$ if $e \in E \setminus F$, is called the *incidence vector* of F. For $F \subseteq E$ and $\boldsymbol{x} \in S^E$, the vector $\boldsymbol{x}_F \in S^F$ is the vector with $|F|$ components defined by $\boldsymbol{x}_F = (x_e)_{e \in F}$. We use $\boldsymbol{x}(F)$ to denote $\sum_{e \in F} x_e$. For $\boldsymbol{x} \in S^E$, the set $\operatorname{supp}(\boldsymbol{x}) = \{e \in E \mid x_e \neq 0\}$ is called the *support* of $\boldsymbol{x}$, and the set $\{e \in E \mid x_e \notin \mathbb{Z}\}$ is called the *fractional support*. All vectors are column vectors, unless stated otherwise.

An *(undirected) graph* $G = (V, E)$ consists of a finite nonempty set V of nodes and a finite set E of edges. For each $S \subseteq V$, let $\delta(S) = \{\{u, v\} \mid u \in S, v \in V \setminus S\}$ be the set of edges that have exactly one endpoint in S. For $v \in V$, we write $\delta(v)$ instead of $\delta(\{v\})$. Given a subset $S \subseteq V$ of nodes, we use $E(S) = \{\{u, v\} \in E \mid u, v \in S\}$ to denote the set of edges with both endpoints in S. The graph with node set S and edge set $E(S)$ is called the *induced graph* of S and is denoted by $G[S] = (S, E(S))$. Given a subset $F \subseteq E$ of edges, we use $V(F) = \bigcup_{e \in F} e$ to denote the set of nodes that occur as endpoints of one or more edges in F.

A *walk* from v_0 to v_k in G is a finite sequence of nodes and edges $W = v_0, e_1, v_1, \ldots, e_k, v_k$ $(k \geq 0)$ such that for $i = 1, 2, \ldots, k$, $e_i = \{v_{i-1}, v_i\} \in E$. Node v_0 is the called the *start* of W and node v_k is called the *end* of W. The nodes on W are denoted by $V(W) = \{v_0, v_1, \ldots, v_k\}$, and the edges on W are denoted by $E(W) = \{e_1, e_2, \ldots, e_k\}$. The nodes $\{v_1, v_2, \ldots, v_{k-1}\}$ are called the *internal nodes* of W. A *path* in G is a walk in G in which all nodes are distinct. We will denote a path from node u to node v by $u \rightsquigarrow v$. A *cycle* (*directed cycle*) in G is a walk in G with $v_0 = v_k$ in which all internal nodes are distinct and different from v_0. A *chord* in a cycle C is an edge $\{u, v\} \in E$ with $u, v \in V(C)$, but $\{u, v\} \notin E(C)$. A *hole* in G is a cycle in G without chords.

For $U \subseteq V$, let $N(U)$ denote the set of *neighbours* of U in G, i.e.,

$$N(U) = \{v \in V \setminus U \mid \{u, v\} \in \delta(U)\}.$$

For singleton sets $\{u\}$ we will abbreviate $N(\{u\})$ to $N(u)$. For any natural number k, the *k-neighbourhood* of a set of nodes $S \subseteq V$ in a graph G, denoted by $N_k(S)$, consists of all nodes in G that can be reached from a node in S by traversing at most k edges, i.e.,

$$N_k(S) = \begin{cases} N(T) \cup T, \quad \text{where } T = N_{k-1}(S), & \text{if } k > 0, \text{ and} \\ S, & \text{if } k = 0. \end{cases}$$

For singleton sets $\{v\}$ we will abbreviate $N_k(\{v\})$ to $N_k(v)$.

The *length* of a path P in G is the number of edges $|E(P)|$ in P. The *diameter* of G, denoted by $\operatorname{diam}(G)$, is the maximum length of a shortest path connecting two nodes in G, i.e.,

$$\operatorname{diam}(G) = \max_{u,v \in V} \min\{|E(P)| \mid P \text{ is a path from } u \text{ to } v \text{ in } G\}.$$

A *connected component* in G is a connected induced graph $G[S]$ that is maximal with respect to inclusion of nodes, where $S \subseteq V$.

2 Reduction from Map Labelling to Independent Set

An instance of the map labelling problem in which the labels are rectangles of uniform size consists of a finite set of points $P \subset \mathbb{R}^2$ and a label size $\sigma_1 \times \sigma_2$. In the following, we assume that the label size (σ_1, σ_2) is fixed. Under this assumption, an instance of the map labelling problem is completely specified by the point set P.

For any point $\boldsymbol{p} \in P$, there are four possible placements of a rectangular label Q such that $\boldsymbol{p}$ is a corner of Q, each occupying a different rectangular region in $\mathbb{R}^2$. Denote these rectangular regions by $Q_{\boldsymbol{p}i}$ with $i \in \{1, 2, 3, 4\}$. Let $\mathcal{Q}(P)$ be the set of all rectangular regions that correspond to possible label placements in a solution to the map labelling problem instance P:

$$\mathcal{Q}(P) = \{Q_{\boldsymbol{p}i} \mid \boldsymbol{p} \in P,\, i \in \{1, 2, 3, 4\}\}.$$

A *solution* to the map labelling problem instance P is a set of rectangular regions $\mathcal{S} \subseteq \mathcal{Q}(P)$. A *feasible solution* to the map labelling problem instance P is a solution $\mathcal{S}$ such that for any pair of rectangular regions $Q, R \in \mathcal{S}$ we have that if $Q \cap R \neq \varnothing$ then $Q = R$. Note that this definition correctly prevents a point $\boldsymbol{p} \in P$ from having more than one label, as the intersection of those labels would contain $\boldsymbol{p}$ and would therefore be non-empty. We will sometimes refer to solutions as *labellings*. The map labelling problem is the problem of finding a feasible solution of maximum cardinality. An optimal solution to a map labelling problem on 250 cities is depicted in Figure 1, Figure 5 shows one on 950 cities. All labelled cities are drawn as black discs, all unlabelled cities are drawn as black circles. All possible rectangular

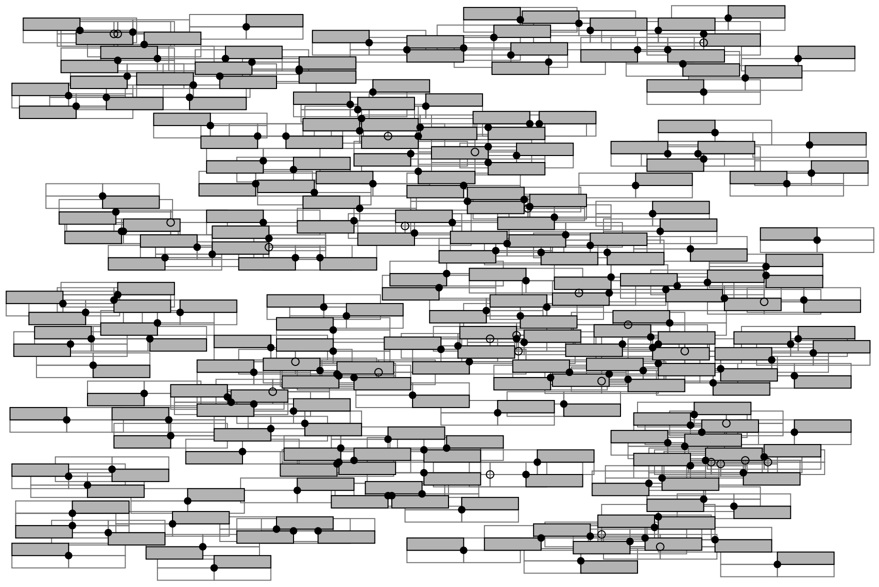


Figure 1: A Solution to a Map Labelling Problem on 250 Cities

regions representing a label position are shown, those that are coloured grey are part of the solution.

To reduce the map labelling problem to a maximum independent set problem, we will make use of the following notion from graph theory. In the following definition we will take a more general perspective of sets of regions. Also, v_Q always denotes some node uniquely associated with the region Q.

Definition 2.1. Given a set of regions $\mathcal{Q}$, the *intersection graph* $G_{\mathcal{Q}} = (V_{\mathcal{Q}}, E_{\mathcal{Q}})$ is given by

$$V_{\mathcal{Q}} = \{v_Q \mid Q \in \mathcal{Q}\}, \text{ and} \tag{1}$$

$$E_{\mathcal{Q}} = \{\{v_Q, v_R\} \subseteq V_{\mathcal{Q}} \mid Q \cap R \neq \varnothing\}. \tag{2}$$

Here, (1) ensures that for each region Q in $\mathcal{Q}$ there is a node v_Q in $G_{\mathcal{Q}}$, and (2) ensures that for each pair (Q, R) of intersecting regions in $\mathcal{Q}$ there is an edge $\{v_Q, v_R\}$ in $G_{\mathcal{Q}}$. Given an instance P of the map labelling problem, we define the *conflict graph* of P as the intersection graph of $\mathcal{Q}(P)$.

Theorem 2.1. *Let $P \subset \mathbb{R}^2$ be an instance of the map labelling problem. There is a bijection between the feasible solutions $\mathcal{S}$ of P and the independent sets in the conflict graph of P.*

Proof. Suppose that $\mathcal{S}$ is a solution of P. Let $S \subseteq V_{\mathcal{Q}(P)}$ be the unique set of nodes of the conflict graph of P that correspond to the labels in $\mathcal{S}$, i.e. $S = \{v_Q \mid Q \in \mathcal{S}\}$. Because $\mathcal{S}$ is a solution to P, for any two distinct labels $Q, R \in \mathcal{S}$ we have that $Q \cap R = \varnothing$. Hence $\{v_Q, v_R\} \notin E_{\mathcal{Q}}$. It follows that S is an independent set in $G_{\mathcal{Q}(P)}$.

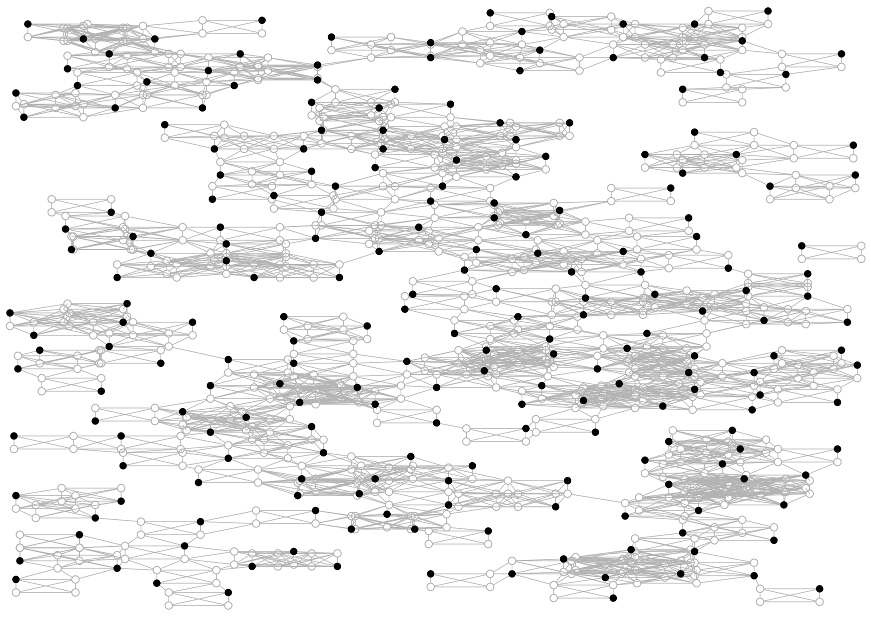


Figure 2: The Conflict Graph of a Map Labelling Problem on 250 Cities

Now, suppose that S is an independent set in $G_{\mathcal{Q}(P)}$. Let $\mathcal{S}$ be the unique set of rectangular regions associated with the nodes in S, i.e. $\mathcal{S} = \{Q \mid v_Q \in S\}$. Because S is an independent set in $G_{\mathcal{Q}(P)}$ and $\{v_Q, v_R\} \in E_{\mathcal{Q}(P)}$ for all distinct $Q, R \in \mathcal{Q}$ such that $Q \cap R \neq \varnothing$ we have that $Q \cap R = \varnothing$ for all distinct $Q, R \in \mathcal{S}$. Hence $\mathcal{S}$ is a solution to P. $\square$

The conflict graph of the map labelling instance shown in Figure 1 is depicted in Figure 2. The conflict graph has 1000 nodes, and the corresponding maximum independent set is indicated by the solid nodes.

3 Formulations of the Maximum Independent Set Problem

To obtain an integer programming formulation for the independent set problem, we use the decision variables $\boldsymbol{x} \in \{0,1\}^V$, where for node $v \in V$ we have $x_v = 1$ if v is in the independent set, and $x_v = 0$ if v is not. Given an undirected graph $G = (V, E)$, let

$$P_E = \{\boldsymbol{x} \in \mathbb{R}^V \mid \boldsymbol{x}(\{u,v\}) \leq 1 \text{ for all } \{u,v\} \in E, \boldsymbol{x} \geq \mathbf{0}\}.$$

The convex hull of incidence vectors of all independent sets in G is denoted by

$$P_{\text{IS}} = \operatorname{conv}(P_E \cap \{0,1\}^V).$$

The maximum independent set problem can be formulated as the following integer programming problem:

$$\max\{\boldsymbol{x}(V) \mid \boldsymbol{x} \in P_E \cap \{0,1\}^V\}. \tag{3}$$

We call this formulation the *edge formulation* of the maximum independent set problem.

3.1 LP Relaxations

By relaxing the integrality constraints in the edge formulation, we obtain the basic LP relaxation

$$\max\{\boldsymbol{x}(V) \mid \boldsymbol{x} \in P_E\}$$

of the independent set problem.

Let $C \subseteq V$ be a clique in G. It follows directly from the definition of independent sets that any independent set can have at most one node from C. Hence, the *clique inequality*

$$\boldsymbol{x}(C) \leq 1 \tag{4}$$

is a valid inequality for P_{IS} for every clique C. If C is maximal, we call (4) a *maximal* clique inequality. Padberg [32] showed that the clique inequality on C is facet-defining for P_{IS} if and only if C is a maximal clique. Let $\mathcal{C}$ be a collection of not necessarily maximal cliques in G such that for each edge $\{u, v\} \in E$, there exists a clique $C \in \mathcal{C}$ with $\{u, v\} \subseteq C$, and let

$$P_{\mathcal{C}} = \{\boldsymbol{x} \in \mathbb{R}^V \mid \boldsymbol{x}(C) \leq 1 \text{ for all } C \in \mathcal{C}, \boldsymbol{x} \geq \boldsymbol{0}\}.$$

By the condition on $\mathcal{C}$ we have that $P_{\mathcal{C}} \subseteq P_E$. From the validity of the clique inequalities it follows that

$$\max\{\boldsymbol{x}(V) \mid \boldsymbol{x} \in P_{\mathcal{C}}\}$$

is a relaxation of the maximum independent set problem. We call the associated IP formulation the *clique formulation* of the independent set problem.

The sets P_{IS}, P_E, and $P_{\mathcal{C}}$, are related as follows:

Proposition 3.1. *Let $\mathcal{C}$ be a collection of cliques such that there exists $C \in \mathcal{C}$ with $\{u, v\} \subseteq C$ for each $\{u, v\} \in E$. Then, $P_{\text{IS}} \subseteq P_{\mathcal{C}} \subseteq P_E$.*

3.2 The Maximal Clique Formulation

Suppose we are given a set $\mathcal{Q}$ of axis-parallel rectangular regions in $\mathbb{R}^2$. It was observed by Imai and Asano [20] that there exists a correspondence between the cliques in $G_{\mathcal{Q}}$ and maximal non-empty regions that are contained in the intersection of subsets of regions from $\mathcal{Q}$.

Lemma 3.2. *Let $\mathcal{Q}$ be a set of axis-parallel rectangular regions. Then, there is a bijection between the cliques in the intersection graph $G_{\mathcal{Q}}$ and the non-empty intersections of subsets of rectangular regions from $\mathcal{Q}$.*

Proof. Let $\mathcal{S} \subseteq \mathcal{Q}$ be a collection of rectangular regions with non-empty intersection. Clearly, the rectangular regions in $\mathcal{S}$ intersect pairwise. Hence, for each $Q, R \in \mathcal{S}$, $\{v_Q, v_R\} \in E_{\mathcal{Q}}$. So, the set $C = \{v_Q \mid Q \in \mathcal{S}\}$ is a clique in $G_{\mathcal{Q}}$.

Conversely, let C be a clique in $G_{\mathcal{Q}}$, and let $\mathcal{S} = \{Q \mid v_Q \in C\}$ be the rectangular regions in $\mathcal{Q}$ that correspond to the nodes in C. Because C is a clique, all rectangular regions in $\mathcal{S}$ intersect pairwise. This implies that the region $R = \bigcap_{Q \in \mathcal{S}} Q$ is non-empty by the following argument (see e.g. Danzer and Grünbaum [11]). The intersection of a set of axis-parallel rectangular regions is non-empty if and only if the intersection of the projections of the rectangular regions on the axis are non-empty. The projections of the rectangular regions on the axes are sets of line-segments in $\mathbb{R}$. By Helly's theorem, a set of line-segments has a non-empty intersection if and only if all pairs of line-segments in the set intersect. $\square$

Let $\mathcal{C}$ be the collection of all maximal cliques in $G_{\mathcal{Q}}$, and let

$$\mathcal{R} = \{\textstyle\bigcap_{v_Q \in C} Q \mid C \in \mathcal{C}\}.$$

It follows from the proof of Lemma 3.2 that $\mathcal{R}$ contains all maximal regions that are contained in the intersection of a maximal subset of rectangular regions from $\mathcal{Q}$ that has a non-empty intersection. The following lemma shows that the regions in $\mathcal{R}$ are mutually disjoint.

Lemma 3.3. *Let $R_1, R_2 \in \mathcal{R}$. If $R_1 \cap R_2 \neq \varnothing$, then $R_1 = R_2$.*

Proof. Let $\mathcal{S}_i \subseteq \mathcal{Q}$ be the maximal set of rectangular regions such that $R_i = \bigcap_{Q \in \mathcal{S}_i} Q$ ($i \in \{1, 2\}$), and assume that $R_1 \cap R_2 \neq \varnothing$. Let $C = \{v_Q \mid Q \in \mathcal{S}_2\}$. Note that C is a maximal clique in $G_{\mathcal{Q}}$ because $R_2 \in \mathcal{R}$. Choose $Q_1 \in S_1$ arbitrarily. Because $R_1 \cap R_2 \neq \varnothing$, also $Q_1 \cap R_2 \neq \varnothing$, which in turn implies that $Q_1 \cap Q \neq \varnothing$ for all $Q \in \mathcal{S}_2$. This implies that $\{v_{Q_1}, v_Q\} \in E_{\mathcal{Q}}$ for all $Q \in \mathcal{S}_2 \setminus \{Q_1\}$. Because C is a maximal clique, this means that $v_{Q_1} \in C$, so $Q_1 \in \mathcal{S}_2$. Since Q_1 was chosen arbitrarily, it follows that $\mathcal{S}_1 \subseteq \mathcal{S}_2$. Similarly, we find that $\mathcal{S}_2 \subseteq \mathcal{S}_1$. This implies that $\mathcal{S}_1 = \mathcal{S}_2$ and that $R_1 = R_2$. $\square$

If the clique formulation is derived from the collection of all maximal cliques in G we call it the *complete clique formulation*. We are interested in finding all maximal cliques in the intersection graph of $\mathcal{Q}$ in order to obtain the complete clique formulation of the independent set problem on $G_{\mathcal{Q}}$. The following lemma states that the number of maximal cliques in the intersection graph of $\mathcal{Q}$ is not too large.

Lemma 3.4. *The number of maximal cliques in $G_{\mathcal{Q}} = (V_{\mathcal{Q}}, E_{\mathcal{Q}})$ is at most $|V_{\mathcal{Q}}| + |E_{\mathcal{Q}}|$.*

Proof. We will show that there exists an injection $f : \mathcal{R} \to V_{\mathcal{Q}} \cup E_{\mathcal{Q}}$. Since $|\mathcal{R}| = |\mathcal{C}|$, this proves the lemma.

Consider any region $R \in \mathcal{R}$. Since R is the intersection of a subset of rectangular regions from $\mathcal{Q}$ we can write $R = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$, where l_1 and l_2 are determined by the boundary of some rectangular regions $Q_1, Q_2 \in \mathcal{Q}$, respectively. Note that Q_1 and Q_2 do not have to be unique. Consider any pair Q_1, Q_2 as above. If $Q_1 = Q_2$, we define $f(R)$ to be v_{Q_1}. Otherwise, $\boldsymbol{l} \in Q_1 \cap Q_2$, so $\{v_{Q_1}, v_{Q_2}\} \in E_{\mathcal{Q}}$, and we define $f(R)$ to be $\{v_{Q_1}, v_{Q_2}\}$.

It remains to show that f is indeed an injection. Let $R_1, R_2 \in \mathcal{R}$ and suppose that $f(R_1) = f(R_2)$. Let $\boldsymbol{l}^i, \boldsymbol{u}^i \in \mathbb{R}^2$ be the points such that $R_i = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{l}^i \leq \boldsymbol{x} \leq \boldsymbol{u}^i\}$ (for $i \in \{1, 2\}$). Because $f(R_1) = f(R_2)$, by construction of f we have that $\boldsymbol{l}^1 = \boldsymbol{l}^2$, so $R_1 \cap R_2 \neq \varnothing$. By Lemma 3.3 we find that $R_1 = R_2$. Hence f is an injection. $\square$

All single-node components of $G_{\mathcal{Q}}$ define their own maximal clique. As these maximal cliques can be reported separately, suppose that $G_{\mathcal{Q}}$ does not have isolated nodes. The following algorithm is a brute-force algorithm that exploits Lemma 3.3 to find all maximal cliques. For each edge $\{v_Q, v_R\} \in E_{\mathcal{Q}}$, we compute a maximal clique C that contains a fixed corner $\boldsymbol{l}$ of $Q \cap R$, and then test whether C is a maximal clique in $G_{\mathcal{Q}}$. If this is the case, we report C, otherwise we proceed with the next edge. Computing C can be done by testing for each node $v_S \in N(\{v_Q, v_R\})$ whether $\boldsymbol{l} \in S$ and if so, adding v_S to C. C is a maximal clique in $G_{\mathcal{Q}}$ if for each node $v_S \in N(\{v_Q, v_R\}) \setminus C$ we have that $S \cap \bigcap_{v_Q \in C} Q = \varnothing$.

Theorem 3.5. *The maximal cliques in $G_{\mathcal{Q}}$ can be found in $O(|V_{\mathcal{Q}}||E_{\mathcal{Q}}|)$ time.*

Proof. The correctness of the brute-force algorithm described above follows directly from Lemma 3.2 and Lemma 3.3. It remains to analyse its time complexity. For each edge $\{u, v\} \in E_{\mathcal{Q}}$, the brute-force algorithm described above uses $O(|\delta(u)| + |\delta(v)|)$ time. Hence, the total time complexity needed for reporting all maximal cliques in $G_{\mathcal{Q}}$ is

$$\begin{aligned}\sum_{\{u,v\}\in E_{\mathcal{Q}}} O(|\delta(u)| + |\delta(v)|) &= O(\sum_{\{u,v\}\in E_{\mathcal{Q}}} |\delta(u)| + |\delta(v)|) = O(\sum_{u\in V_{\mathcal{Q}}} \sum_{e\in\delta(u)} |\delta(u)|) \\ &= O(\sum_{u\in V_{\mathcal{Q}}} |\delta(u)|^2) \leq O(|V_{\mathcal{Q}}||E_{\mathcal{Q}}|).\end{aligned}$$

□

The direct consequence of Theorem 3.5 is that, given an instance of the map labelling problem P, we can compute the complete clique formulation of the independent set problem in $G_{\mathcal{Q}(P)}$ in $O(|V_{\mathcal{Q}(P)}||E_{\mathcal{Q}(P)}|)$ time.

3.3 Other Valid Inequalities

Let $\boldsymbol{x} \in P_E$ be a fractional solution. An *odd hole* in G is a hole in G that contains an odd number of nodes. If H is an odd hole in G, then the *odd hole inequality* defined by H is

$$\boldsymbol{x}(V(H)) \leq \lfloor |V(H)|/2 \rfloor.$$

It was shown by Padberg [32] that the odd hole inequality is valid for P_{IS}, and facet-defining for $P_{\text{IS}} \cap \{\boldsymbol{x} \mid \boldsymbol{x}_{V\setminus V(H)} = \mathbf{0}\}$. Given an odd hole H, a *lifted* odd hole inequality is of the form

$$\boldsymbol{x}(V(H)) + \sum_{v\in V\setminus V(H)} \alpha_v x_v \leq \lfloor |V(H)|/2 \rfloor \tag{5}$$

for some suitable vector $\boldsymbol{\alpha} \in \mathbb{R}^V_+$. Lifted odd-hole inequalities are not added a-priori to our models, but dynamically. This is explained in Section 5.7.

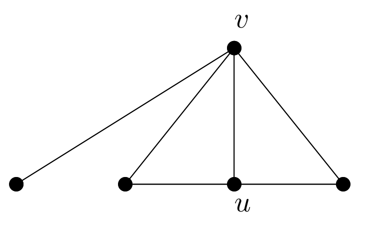
A more general class of valid inequalities that is known in the literature consists of the so-called *mod-k inequalities.* These are developed by Caprara, Fischetti, and Letchford [4, 5].

3.4 Pre-processing and Dominance Criteria

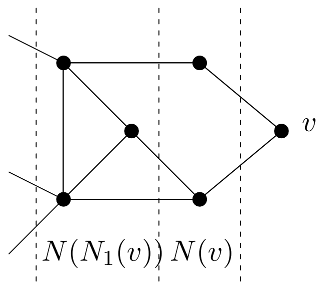
In this section we consider ways to recognise nodes that belong to a maximum independent set, or that can be removed from the graph without reducing the size of the maximum independent set in it. Variables corresponding to nodes that belong to a maximum independent set can be set to one, and and variables corresponding to nodes that can be removed from the graph without reducing the size of the maximum independent set can be set to zero before starting the branch-and-cut algorithm.

The following result allows us to identify nodes in a graph that belong to a maximum independent set. A weighted version was already mentioned by Nemhauser and Trotter [30, Theorem 1].

Proposition 3.6. *If $I \subseteq V$ is a maximum independent set in $G[N_1(I)]$, then there exists a maximum independent set in G that contains I.*



v is node-dominated by u



v is set-dominated by $N(v)$

Figure 3: Node- and Set-Dominance Criteria

Proof. Suppose that $I \subseteq V$ is a maximum independent set in $G[N_1(I)]$. Let I^* be a maximum independent set in G, $I_1 = I^* \setminus N_1(I)$, and $I_2 = I_1 \cup I$. By construction, I_2 is an independent set. Observe that

$$|I_1| = |I^* \setminus N_1(I)| = |I^*| - |I^* \cap N_1(I)| \geq |I^*| - |I|$$

because $I^* \cap N_1(I)$ is an independent set in $G[N_1(I)]$. Hence

$$|I_2| = |I_1 \cup I| = |I_1| + |I| \geq |I^*|.$$

So I_2 is a maximum independent set that contains I. $\square$

A *simplicial node* is a node whose neighbours form a clique. As a corollary to Proposition 3.6 we can set x_v to one if v is a simplicial node. Checking whether a given node v is simplicial can be done by marking its neighbours, and then for each neighbour of v, counting whether it is adjacent to all other neighbours of v. This check takes at most $O(|\delta(v)| + \sum_{u \in N(v)} |\delta(u)|)$ for node v, summing up to a total of $\sum_{v \in V} O(|\delta(v)|^2) \leq O(|V||E|)$ time for all nodes.

The following two propositions, illustrated in Figure 3, are special cases of the dominance criteria reported by Zwaneveld, Kroon and Van Hoesel [46].

Proposition 3.7. (Node-Dominance) *Let $u, v \in V$ be nodes in G such that $N_1(u) \subseteq N_1(v)$. Then there exists a maximum independent set in G that does not contain v.*

Proof. Let I^* be a maximum independent set in G. If $v \notin I^*$, we are done. Otherwise, let $I_1 = I^* \setminus \{v\}$, and $I_2 = I_1 \cup \{u\}$. Because $v \in I^*$ and $N_1(u) \subseteq N_1(v)$, $I_1 \cap N_1(u) = \varnothing$. It follows that I_2 is an independent set with $|I_2| = |I^*|$, hence it is maximum and $v \notin I_2$. $\square$

If the condition of Proposition 3.7 is satisfied, we say that v is *node-dominated* by u. Searching for a node u that node-dominates v can be done in a similar way as testing whether v is simplicial. As a corollary to Proposition 3.7, we can set x_v to zero if v is node-dominated.

Proposition 3.8. (Set-Dominance) *Let $v \in V$ be a node in G. If for each independent set I in $G[N(N_1(v))]$ there exists a node $u \in N(v)$ with $N(u) \cap I = \varnothing$, then there exists a maximum independent set in G that does not contain v.*

Proof. Let I^* be a maximum independent set. If $v \notin I^*$, we are done. Otherwise, let $I_1 = I^* \setminus \{v\}$, $u \in N(v)$ a node with $N(u) \cap I_1 = \varnothing$, and $I_2 = I_1 \cup \{u\}$. It follows directly from the choice of u that I_2 is an independent set with $|I_2| = |I^*|$ that satisfies the requirement. $\square$

If the condition of Proposition 3.8 is satisfied, we say that v is *set-dominated* by $N(v)$. Zwaneveld, Kroon and Van Hoesel presented weighted versions of the node-dominance and set-dominance criteria.

We now focus our attention on how to determine whether a node is set-dominated.

Proposition 3.9. *Let $v \in V$. There exists a node $u \in N(v)$ with $N(u) \cap I = \varnothing$ for each independent set I in $G[N(N_1(v))]$ if and only if there does not exist an independent set I in $G[N(N_1(v))]$ with $N(v) \subseteq N(I)$.*

There does not seem to be a simple way to determine whether a node is set-dominated or not. Instead, we use a recursive algorithm, by Zwaneveld *et al.*, that searches for an independent set I in $G[N(N_1(v))]$ with $N(v) \subseteq N(I)$. Although the algorithm has an exponential worst-case time bound, it turns out to work efficiently on conflict graohs of map labelling instances.

The algorithm has as input two sets of nodes, denoted by U and W, and returns true if and only if there exists an independent set in $G[W]$ that has as neighbours all nodes in U. If U is empty the algorithm returns true. If U is not empty but W is, the algorithm returns false. Otherwise it selects a node $v \in U$. For each $w \in W \cap N_1(v)$, the algorithm recursively searches for an independent set in $G[W \setminus N_1(w)]$ that has as neighbours all nodes in $U \setminus N_1(w)$. If any of the recursive calls returns true, the algorithm returns true; otherwise it returns false. If the algorithm returns false when applied to $U = N(v)$, and $W = N(N_1(v))$, then v is set-dominated.

The correctness of the algorithm can be shown by induction, where the induction step follows from the observation that if there exists an independent set in $G[W \setminus N_1(w)]$ that has as neighbours all nodes in $U \setminus N_1(w)$, then we can extend it with node w to become an independent set in W that has as neighbours all nodes in U. As a corollary to Proposition 3.8 we can set x_v to zero if v is set-dominated.

4 Heuristic Algorithms

Finding labellings of large, but not necessarily maximum cardinality can be accomplished using heuristic algorithms. We consider two heuristics that treat the problem directly as a map labelling problem. These are the subjects of Sections 4.1 and 4.2. Furthermore, we study heuristics that reformulate the map labelling problem as an independent set problem, and then search for a large independent set. These heuristics can be classified as local search algorithms, the subject of Sections 4.3 and 4.4, and LP-based rounding algorithms, the subject of Section 4.5.

4.1 Simulated Annealing

A simple optimisation strategy is *neighbourhood search*. The solution space consists of all possible labellings of the points. Every point can be labelled in one of the four positions or receive no label at all. We define the *neighbourhood* of a labelling L as all labellings that can be formed by changing the label of a single point including deleting a label or adding a label. Such a change from one labelling to another labelling is called a *move*. The objective function z is the number of labels that is not overlapped by other labels. Our objective is to maximise z such that there are no overlapping labels. The neighbourhood search works as follows:

1. Compute an initial labelling $L = L_0$
2. Repeat the following until no improvement is made:
    (a) find a labelling L' in the neighbourhood of L that improves the objective value z.

The computation of L_0 is done as follows. Each point feature gets assigned randomly one of the four potential positions, or with probability 0.1 it receives no initial label.

A disadvantage of this method is that it easily gets stuck in local maxima. To escape from local optima in the search space, *simulated annealing* can be used, see Kirkpatrick, Gelatt, and Vecchi [23]. Moves in the search space that make the solution worse are allowed with a certain probability. Simulated annealing uses the following approach:

1. Compute an initial labelling $L = L_0$
2. Initialise the temperature T to T_0.
3. Repeat until the rate of improvement falls below a given threshold:
    (a) Decrease the temperature, T, according to the annealing schedule
    (b) Choose a move, that is pick a point p at random, its label is randomly placed in any of the potential positions or with probability 0.1 point p receives no label.
    (c) Compute ΔE, the change in the objective function z caused by the move.
    (d) If the new labelling is worse, $\Delta E < 0$, undo the label repositioning with probability $1.0 - e^{\Delta E/T}$.

We used the same annealing schedule as Christensen *et al.* [7]. The initial temperature is $T_0 = 1/\log(1.5)$. The temperature is decreased by 10 percent after $50n$ iterations, where n is the number of point features. If more than $5n$ moves are accepted at a temperature stage, the temperature is decreased immediately.

The calculation of ΔE is done as described by Christensen *et al.* [6] by using the conflict graph and a overlap counter for each label position that counts the number of other labels in the current labelling that overlap the label position. So the time for one iteration is $O(\delta)$ with δ the maximum degree of the conflict graph. And the total time $O(\delta\, n_{\text{iter}})$, with n_{iter} the number of iterations performed by the algorithm.

The simulated annealing method as described by Christensen *et al.* [7] has been used in several comparisons of label algorithms. Examples of this are Wagner, Wolff, Kapoor, and Strijk [38] using an algorithm that repeatedly applies three simple rules, and van Dijk, Thierens, and de Berg [12] that describes genetic algorithms. The results were that these algorithms had roughly same performance as simulated annealing although the running times differed. Since simulated annealing is present in most other comparisons of label algorithms we choose this algorithm to include in our tests.

4.2 Diversified Neighbourhood Search

Briefly, the method explores part of the solution space by moving at each iteration to the best neighbour of the current solution, even if this leads to a deterioration of the objective function z. To avoid cycling and to escape from local maxima we use a penalised objective function z_{divers}. This function is a long term memory of the algorithm, since points are given a penalty proportional to the number of times a point was chosen as the best move by the algorithm.

Diversified Neighbourhood Search takes the following approach:

1. Compute an initial labelling $L = L_0$
2. Repeat the following steps, until we reach an optimal solution or until a certain number of iterations n_{iter}.
    (a) find the move m_{best} in the neighbourhood of L with maximum z
    (b) if m_{best} gives a solution with $z > z_{\text{best}}$ then carry out m_{best} and update z_{best}.
    (c) else carry out the move m_{divers} in the neighbourhood of L with maximum z_{divers}.
    (d) update all data structures used for selecting the best move.

4.2.1 Initial Solution

We place repeatedly a label at the label position that has the least number of possible overlaps with other label positions. The label positions overlapped by this chosen label cannot be chosen anymore. In this way we get an initial labelling with no overlapping labels. This method is implemented in the following way. First, all label positions are inserted in a priority queue with as priority the number of overlaps of that label position. At each iteration the minimum of the priority queue gives us the label position that becomes a label. The priority queue is updated by removing all label positions that overlap the chosen label. The priority of the neighbours of those removed label positions is then decreased by one to reflect the fact that the number of overlaps has decreased for those label positions. This method for computing the initial solution has running time $O(n(\delta + \log n))$ with δ the maximum degree in the conflict graph.

4.2.2 Objective Function

For a labelling L let set $P_{\text{obstructed}}(L)$ be the points that have a label that is overlapped by another label, $P_{\text{deleted}}(L)$ the set of points without a label and the set $P_{\text{free}}(L)$ all points with a non-overlapped label. Clearly these three sets are disjoint and $P = P_{\text{obstructed}}(L) \cup P_{\text{deleted}}(L) \cup P_{\text{free}}(L)$. The objective function we use is:

$$z(L) = |P_{\text{free}}(L)| = n - |P_{\text{obstructed}}(L)| - |P_{\text{deleted}}(L)|$$

This is the same objective as used in simulated annealing, see Section 4.1. z_{best} is the maximum z encountered so far during the algorithm.

4.2.3 Neighbourhood Structure

We use the same neighbourhood as in the simulated annealing algorithm. So the neighbourhood of a labelling L are all labellings that can be formed by changing the label of a single point including deleting a label or adding a label. Such a change from one labelling to another labelling is called a move.

4.2.4 Diversification

To avoid that the same points are chosen very often without giving a better solution, we will penalise those points. This enables the algorithm to escape from local maxima and prevents too much cycling. We count how often a point has changed its label position and normalise these values with the maximum occurring frequency. These values are substracted from z:

$$z_{\text{divers}}(L) = z(L) - c_{\text{freq}} \sum_{p \in P} \text{freq}(p)/\text{freq}_{\max}$$

with $\text{freq}_{\max} = \max_{p \in P} \text{freq}(p)$. The value of c_{freq} was set to 1.75. After the first $n_{\text{iter}}/4$ iterations the frequencies are reset to zero. At that stage the solution has already improved considerably, so resetting the frequency gives it fresh start with respect to the frequencies.

4.2.5 Running Time

We let the algorithm run for $n_{\text{iter}} = 50n$ iterations. We maintain for each point the change in the objective functions z and z_{divers} for the best move of that point. These values are kept in two queues $\mathcal{Q}_z$ and $\mathcal{Q}_{zd}$. In that way we can, at each iteration, select the best move with respect to z and z_{divers} by looking at the maximum of the queue. To keep $\mathcal{Q}_z$ and $\mathcal{Q}_{zd}$ up to date we recompute the change in the objective functions z and z_{divers} for the points that are neighbours in the conflict graph of the changed point, and of the neighbours of the neighbours of the changed point. It is not necessary to recompute other points. So, at each iteration at most $O(\delta^2)$ points need a re-computation where δ is the maximum degree in the conflict graph. Recomputing the change of z and z_{divers} costs $O(\delta)$ time, and inserting the new value in the queue costs $O(\log n)$ time. Thus one iteration costs $O(\delta^2(\delta + \log n))$ time. So the total update time of the algorithm is $O(n_{\text{iter}} \cdot \delta^2(\delta + \log n))$.

Every time $\text{freq}_{\max}$ increases, all priorities in $\mathcal{Q}_{zd}$ must be changed. This costs $O(n \log n)$ per increase of the maximum frequency. If all points are chosen equally often, then $\text{freq}_{\max}$ would be n_{iter}/n. This is generally not the case but in all experiments we have that $\text{freq}_{\max} < 2 \cdot n_{\text{iter}}/n$. The time spent during the whole algorithm on updating the frequencies is $O((2n_{\text{iter}}/n)(n \log n)) = O(2n_{\text{iter}} \log n)$.

Since the update time dominates the time for changing the frequencies, the total running time of the algorithm is $O(n_{\text{iter}} \cdot \delta^2(\delta + \log n))$.

4.3 The k-Opt Heuristic

Here we consider local search neighbourhoods and iterative improvement algorithms for the maximum independent set problem.

Definition 4.1. Let S be an independent set in G, and $k \geq 1$ be an integer. The *k-opt* neighbourhood of S, denoted by $\mathcal{N}_k(S)$, is defined as the collection of independent sets S' of cardinality $|S'| = |S| + 1$ that can be obtained from S by removing $k - 1$ nodes and adding k nodes.

So, if $S' \in \mathcal{N}_k(S)$, then $S' = (S \setminus U) \cup W$, for some U and W with $U \subseteq S$, $|U| = k - 1$, $W \subset (V \setminus S) \cup U$, and $|W| = k$. The k-opt neighbourhood is undefined if $|S| < k - 1$. Because we do not require that $U \cap W = \varnothing$ we have that, if $\mathcal{N}_k(S)$ is defined, then $\mathcal{N}_j(S) \subseteq \mathcal{N}_k(S)$ for all $j \in \{1, \ldots, k\}$.

Definition 4.2. An independent set S is *k-optimal* if the k-opt neighbourhood of S is empty.

There is no guarantee that the rounding algorithms presented in Section 4.5 produce k-optimal independent sets for any $k \geq 1$. This motivates our interest in the k-opt neighbourhoods of independent sets.

Proposition 4.1. *If an independent set S is k-optimal, then S is l-optimal for all $l \in \{1, \ldots, k\}$.*

Proof. The proposition holds because $\mathcal{N}_l(S) \subseteq \mathcal{N}_k(S)$ for all $l \in \{1, \ldots, k\}$. $\square$

The k-opt algorithm starts from an independent set S, and replaces S by an independent set $S' \in \mathcal{N}_k(S)$ until S is k-optimal. Optimising over the k-opt neighbourhood can be done by trying all possibilities of U and W. There are $\binom{|S|}{k-1}$ possible ways to choose U, and at most $\binom{|V|}{k}$ possible ways to choose W. Checking feasibility takes $O(|E|)$ time. It follows that searching the k-opt neighbourhood can be done in $O(|V|^{2k-1}|E|)$ time.

Note, that in order to compute a 1-optimal solution it is sufficient to look at each node only once, and only checking feasibility on arcs adjacent to this node. Therefore, a 1-optimal solution can be computed in $O(|V| + |E|)$ time.

The following proposition tells that we can take advantage of the sparsity of a graph when looking for neighbours in the k-opt neighbourhood of a $(k-1)$-optimal independent set S.

Proposition 4.2. *Let S be a $(k-1)$-optimal independent set for some $k > 1$, and $S' \in \mathcal{N}_k(S)$. Then $S' = (S \setminus U) \cup W$ for some sets $U \subseteq S$ and $W \subseteq N(U)$ such that $G[N_1(U)]$ is connected.*

Proof. Suppose by way of contradiction that $W \not\subseteq N(U)$. Then, there exists $v \in W \setminus N_1(U)$, and because $S' \in \mathcal{N}_k(S)$ we have that $S \cup \{v\}$ is an independent set, so S is not 1-optimal, contradicting our choice of S. It follows that $W \subseteq N(U)$.

From the $(k-1)$-optimality of S it follows that $|U| = k - 1$. Now, let $X \subseteq N_1(U)$ be the node set of a connected component in $G[N_1(U)]$ with $|(U \cap X)| = |(W \cap X)| - 1$. Note that such a node set exists because $|S'| = |S| + 1$. Then, the set $I = (S \setminus (U \cap X)) \cup (W \cap X)$ is an independent set with $|I| = |S| + 1$. It follows from the $(k-1)$-optimality of S that $|U \cap X| = k - 1$. But then, $U \cap X = U$, so $U \subseteq X$, which implies that $G[N_1(U)]$ is connected. $\square$

So, when searching the k-opt neighbourhood of S we can limit our choice of U and W using the above observations. This results in a more efficient search of the k-opt neighbourhood on sparse graphs with a large diameter.

4.4 Tabu Search using the Independent Set Formulation

Here we present a local search algorithm that uses tabu search. Instead of applying tabu search directly to the independent set problem, we use it to find augmentations. This has as advantage that the neighbourhood during the tabu search is small. Hence, it is easy to optimise over efficiently.

The tabu search is embedded in a procedure that decides on where to try to augment the current solution. This procedure is the main loop of our augmentation algorithm and is described in Section 4.4.1. The tabu search itself is the subject of Section 4.4.2.

4.4.1 The Main Loop

The main loop of our augmentation algorithm maintains an independent set S, a list of *active nodes* L, and works in iterations. Initially $S = \varnothing$ and $L = V$. As long as $L \neq \varnothing$, the main loop proceeds by extracting the first node, say v, from L. We try to find an independent set S' of size $|S| + 1$ using tabu search starting from the set $S \cup \{v\}$. If we do not succeed in doing so v becomes inactive and we proceed with the next iteration.

If we do succeed, then $S' \in \mathcal{N}_k(S)$ for some value of k and we can write $S' = (S \setminus U) \cup W$. Here U, W are the sets of nodes that are removed from S, and added to S, respectively, to obtain S'. We replace S by S' and make the set of inactive nodes adjacent to W active by appending all $u \in N(W)$ with $u \notin L$ to the end of L. Having done this, we proceed with the next iteration. The main loop of our augmentation algorithm terminates when L becomes empty.

Since each successful augmentation increases the size of the independent set, there are at most $|V|$ successful augmentations. Also, the number of consecutive failed augmentations is at most $|V|$. Hence the number of tabu searches performed by the main loop is bounded by $O(|V|^2)$.

4.4.2 Finding an Augmentation using Tabu Search

We employ the same neighbourhood as proposed by Friden, Herz, and de Werra [17]. Given a set S_0 with $|S_0| = \alpha_0$, we try to find a set S that satisfies $|S| = \alpha_0$ and minimises the objective function

$$f(S) = |E(S)|.$$

For this purpose we maintain a partition $(S, V \setminus S)$ of the nodes and a set of *tabu nodes.* Initially, $S := S_0$ and none of the nodes are tabu. The tabu search operates in iterations, each iteration we take a node $u \in S$ and a node $w \in V \setminus S$, and set $S' := (S \setminus \{u\}) \cup \{w\}$, where $u \in V(E(S))$. Here u and w are chosen such that the resulting set $|E(S)|$ is minimal subject to the condition that neither u nor w is tabu. We *move* from S to S' by setting $S := S'$.

After each move in which the set $E(S)$ does not strictly decrease in size, the nodes involved in it are made tabu. Nodes that are made tabu stay so for a fixed number τ of iterations. Whenever $|E(S)|$ becomes zero, S is an independent set and we terminate the search.

Let S^* denote the set S that minimises $|E(S)|$ over the current search. If, after c_1 iterations we did not see any change in S^*, we conclude that the search has reached some local optimum. Our next goal is to escape from this local optimum. To achieve this we perform a constant number, say $c_2 - 1$, of attempts at escaping from it. Each attempt consists of at most c_1 iterations, of which the first c_3 iterations are performed with a perturbed objective function. This can be interpreted as applying *intensification* and *diversification* techniques (see Hertz, Taillard and de Werra [19]). The perturbed objective function is given by

$$\tilde{f}(S) = |E(S)| + \frac{1}{2} \sum_{v \in V} \boldsymbol{p}^T \chi^S,$$

where $\boldsymbol{p} \in \mathbb{N}^V$ is a vector that counts for each node $v \in V$ how often node v has been involved in a conflict, i.e., p_v is the number of iterations in which $v \in V(E(S))$. Also, we let the number of iterations τ that tabu nodes stay tabu depend on the number of the attempt at escaping from the current local optimum. In attempt i, we take $\tau = \tau(i)$, where $\tau(i)$ is defined as

$$\tau(i) = \tau_{\min} + i(\tau_{\max} - \tau_{\min})/c_2$$

for some parameters $\tau_{\min}$ and $\tau_{\max}$. As soon as we find a set S with $|E(S)| < |E(S^*)|$ we proceed as normal, using f as objective function, and with $\tau = \tau_{\min}$. The search terminates if none of the attempts succeeds.

To complete the description of our tabu search we give our choices of parameters. After doing some experiments we concluded that taking $\tau_{\min} = 2$, $\tau_{\max} = 7$, $c_1 = 100$, $c_2 = 10$, and $c_3 = 3$ are good choices of parameters. These are the settings that we use in our experiments reported on in Section 6.1.

If one keeps track of the number of neighbours of each node that are in S, each iteration of the tabu search can be implemented to work in $O(|\delta(V(E(S)))|)$. Since $0 \leq f(S) \leq |E|$ and each improvement decreases $|E(S^*)|$ by at least one, we can have at most $c_1 c_2 |E|$ iterations. The worst-case running time of the search can then be bounded by $O(|E| k \max_{v \in V} |\delta(v)|)$, where $|V(E(S))| \leq k \leq |V|$. This could be as large as $O(|V|^2|E|)$. As a result the worst-case bound on the time complexity of the main loop becomes $O(|V|^4|E|)$.

On conflict graphs of map labelling problems we have observed that $|V(E(S))|$ and $\max_{v \in V} |\delta(v)|$ are bounded from above by small constants. Assuming that these observations hold, we obtain a bound of $O(|E|)$ on the running time of the tabu search and a bound of $O(|V|^2|E|)$ on the running time of the main loop. Still, these bounds are crude since they do not take into account that augmentations only consider a small part of the graph.

4.5 LP Rounding Algorithms

Here we consider heuristics for finding hopefully large independent sets, starting from an LP solution in P_E.

4.5.1 Simple LP Rounding

Suppose we are given a point $\boldsymbol{x} \in P_E$. Let S be the set of nodes that correspond to $\boldsymbol{x}$-variables with a value strictly greater than one half, i.e., $S = \{v \in V \mid x_v > \frac{1}{2}\}$. Because $\boldsymbol{x} \in P_E$, we know that for each edge $\{u, v\} \in E$ at most one of u and v can have a value strictly greater than one half and thus be in S. It follows that S is an independent set in G. The simple LP rounding algorithm rounds $\boldsymbol{x}_S$ up to $\mathbf{1}$, and $\boldsymbol{x}_{V \setminus S}$ down to $\mathbf{0}$ to obtain the vector $\boldsymbol{x}' = \chi^S \in P_E \cap \{0,1\}^V$, which it returns as a solution. This can be done in $O(|V|)$ time.

The quality of the solution of the LP rounding algorithm can be arbitrarily bad. For instance, the vector $\boldsymbol{x} \in P_E$ with $x_v = \frac{1}{2}$ for all $v \in V$ is rounded to $\mathbf{0}$, corresponding to the empty independent set. On the other hand, the algorithm is capable of producing any optimal solution. For example, if $\boldsymbol{x} = (1-\epsilon)\chi^{S^*} + \epsilon\chi^{V \setminus S^*}$ for some maximum independent set S^* and any ϵ such that $0 \leq \epsilon < \frac{1}{2}$, then the algorithm produces χ^{S^*} as an answer, an optimal solution.

Nemhauser and Trotter [29] have shown that if $\boldsymbol{x}$ is obtained by maximising any objective function over P_E, then $\boldsymbol{x}$ is a vector with components 0, $\frac{1}{2}$, and 1 only. In this case the simple LP rounding algorithm degenerates to selecting the components of $\boldsymbol{x}$ with value 1.

4.5.2 Minimum Regret Rounding

Denote by $\mathcal{I}$ the collection of all independent sets in G. Suppose we are given a vector $\boldsymbol{x} \in P_E$ with some fractional components. We are going to round $\boldsymbol{x}$ by repeatedly choosing an independent set $I \in \mathcal{I}$ with $\mathbf{0} < \boldsymbol{x}_I < \mathbf{1}$, rounding $\boldsymbol{x}_I$ up to $\mathbf{1}$, and $\boldsymbol{x}_{N(I)}$ down to $\mathbf{0}$, until

$\boldsymbol{x}$ is integral. Let $I \in \mathcal{I}$. The rounding operation above defines a function f that maps $\boldsymbol{x}$ on a vector $\boldsymbol{x}' \in P_E$ using I:

$$f : P_E \times \mathcal{I} \to P_E : (\boldsymbol{x}, I) \mapsto \boldsymbol{x}', \qquad \text{where } x'_u = \begin{cases} 1 & \text{if } u \in I, \\ 0 & \text{if } u \in N(I), \\ x_u & \text{otherwise.} \end{cases}$$

We say that f *rounds up* $\boldsymbol{x}_I$.

Lemma 4.3. *Let $\boldsymbol{x} \in P_E$ and $I \in \mathcal{I}$. Then $f(\boldsymbol{x}, I) \in P_E$.*

Proof. Let $\boldsymbol{x}' = f(\boldsymbol{x}, I)$. Since I is an independent set and by construction of f, $\boldsymbol{x}'_{N_1(v)\setminus\{v\}} = \mathbf{0}$ for all $v \in I$, we have that $\boldsymbol{x}'_{N_1(I)} \in P_E$. Since $\mathbf{0} \le \boldsymbol{x}'_{V\setminus I} \le \boldsymbol{x}_{V\setminus I}$, we have $\boldsymbol{x}'_{V\setminus I} \in P_E$. The proof follows because by definition of $N_1(I)$ there do not exist edges $\{u, v\}$ with $u \in V \setminus N_1(I)$ and $v \in I$. ☐

We first study the effect of f on the objective function $\boldsymbol{x}(V)$. Again, let $I \in \mathcal{I}$ and $\boldsymbol{x}' = f(\boldsymbol{x}, I)$. Define the function $r : P_E \times \mathcal{I} \to \mathbb{R}$ as the difference in objective function value between $\boldsymbol{x}$ and $\boldsymbol{x}'$:

$$\begin{aligned} r(\boldsymbol{x}, I) &= \sum_{v\in V} x_v - \sum_{v\in V} (f(\boldsymbol{x}, I))_v = \sum_{v\in V} (x_v - x'_v) = \sum_{v\in N_1(I)} x_v - |I| \\ &= \boldsymbol{x}(N_1(I)) - |I|. \end{aligned}$$

Because we will later apply f to a vector $\boldsymbol{x} \in P_E$ that is optimal with respect to $\boldsymbol{x}(V)$, we have that $\boldsymbol{x}(V) \ge \boldsymbol{x}'(V)$, so $r(\boldsymbol{x}, I) \ge 0$ is the decrease in the objective function value in that case. Since we do not like the objective function to decrease, we call $r(\boldsymbol{x}, I)$ the *regret* we have of rounding $\boldsymbol{x}_I$ to $\mathbf{1}$.

Now take $\boldsymbol{x} \in P_E, I \in \mathcal{I}$ with $|I| > 1$, and choose non-empty sets $I_1, I_2 \subset I$ where $I_1 = I \setminus I_2$. Then,

$$\begin{aligned} r(\boldsymbol{x}, I) &= \boldsymbol{x}(N_1(I)) - |I| \\ &= \boldsymbol{x}(N_1(I_1)) + \boldsymbol{x}(N_1(I_2)) - \boldsymbol{x}(N_1(I_1) \cap N_1(I_2)) - |I_1| - |I_2| \\ &= r(\boldsymbol{x}, I_1) + r(\boldsymbol{x}, I_2) - \boldsymbol{x}(N_1(I_1) \cap N_1(I_2)). \end{aligned}$$

This shows that if

$$\boldsymbol{x}(N_1(I_1) \cap N_1(I_2)) = 0, \tag{6}$$

then the regret of rounding $\boldsymbol{x}_I$ to $\mathbf{1}$ is the same as the combined regret of rounding $\boldsymbol{x}_{I_1}$ and $\boldsymbol{x}_{I_2}$ to $\mathbf{1}$. It follows that we can restrict our choice of I to independent sets that cannot be partitioned into subsets I_1, I_2 satisfying condition (6). This is the case if and only if the graph induced by the support of $\boldsymbol{x}_{N_1(I)}$ is connected.

If we choose I in such a way that $\mathbf{0} < \boldsymbol{x}_I < \mathbf{1}$, then $f(\boldsymbol{x}, I)$ has at least $|I|$ fewer fractional components than $\boldsymbol{x}$. We will use this to define a greedy rounding algorithm as follows. The algorithm has as input a vector $\boldsymbol{x} \in P_E$ and an integer $t > 0$ and repeatedly replaces $\boldsymbol{x}$ by $f(\boldsymbol{x}, I)$ for some set I, rounding $\boldsymbol{x}_I$ to $\mathbf{1}$. This is done in t phases, numbered $t, t-1, \ldots, 1$. In phase k, we only work with sets $I \in \mathcal{I}$ satisfying

$$|I| = k, \tag{7}$$

$$\mathbf{0} < \boldsymbol{x}_I < \mathbf{1}, \text{ and} \tag{8}$$

$$G[\operatorname{supp}(\boldsymbol{x}_{N_1(I)})] \text{ is connected.} \tag{9}$$

During phase k, the next set $I \in \mathcal{I}$ is chosen so as to minimise the regret $r(\boldsymbol{x}, I)$ within these restrictions. Phase k terminates when there are no more sets $I \in \mathcal{I}$ satisfying these conditions.

We name this algorithm the *minimum regret rounding algorithm* due to the choice of I. Note that at any time during the algorithm, $\boldsymbol{x}_F$ is an optimal LP solution to the maximum independent set problem in $G[F]$ if the original vector $\boldsymbol{x}$ was one in G, where $F = \{v \in V \mid 0 < x_v < 1\}$ is the fractional support of $\boldsymbol{x}$. It follows that the value $\boldsymbol{x}(V)$ never increases over any execution of the algorithm.

Phase k is implemented in iterations as follows. We maintain a priority queue Q that initially contains all sets $I \in \mathcal{I}$ satisfying conditions (7)–(9), where the priority of set I is the value of $r(\boldsymbol{x}, I)$. In each iteration, we extract a set I from Q with minimum regret. If $\boldsymbol{x}_I$ has integral components or if the graph induced by the support of $\boldsymbol{x}_{N_1(I)}$ is not connected, then we proceed with the next iteration. Otherwise we update Q. This is done by decreasing the priorities of all $I' \in Q$ with $N_1(I') \cap N_1(I) \neq \varnothing$ by $\boldsymbol{x}(N_1(I') \cap N_1(I))$. We replace our current vector $\boldsymbol{x}$ by $f(\boldsymbol{x}, I)$ and proceed with the next iteration. Phase k terminates when Q is empty.

Lemma 4.4. *Let $F \subseteq V$ be the fractional support of $\boldsymbol{x}$ upon termination of phase k of the minimum regret rounding algorithm. For any $F' \subseteq F$ such that $G[F']$ is a connected component of $G[F]$,*

$$\operatorname{diam}(G[F']) < 2(k-1).$$

Proof. Let $G[F']$ be a connected component of $G[F]$ and suppose, by way of contradiction, that the graph $G[F']$ has a diameter of at least $2(k-1)$. Then, there exists nodes $u, v \in F'$ for which the shortest path P in $G[F']$ has length exactly $2(k-1)$. Let

$$P = (u = v_0, e_1, v_1, \ldots, e_{2(k-1)}, v_{2(k-1)} = v).$$

Consider the set $I = \{v_0, v_2, \ldots, v_{2(k-1)}\}$. Observe that $I \subset F'$. We argue that I is an independent set. Suppose for some $v_i, v_j \in I$ we had $\{v_i, v_j\} \in E$, thus $\{v_i, v_j\} \in E(F')$. Since we can assume without loss of generality that $i < j$, P can be shortened by replacing the sequence $v_i, e_{i+1}, v_{i+1}, \ldots, e_j, v_j$ by the sequence $v_i, \{v_i, v_j\}, v_j$, and still be a path in $G[F']$. As this contradicts our choice of P it follows that no such $v_i, v_j \in I$ exist. Thus I is an independent set in $G[F']$. Because $|I| = k$, $F' \subseteq F$, and $G[F']$ is connected, I satisfies conditions (7)–(9). This contradicts the termination of phase k. □

Theorem 4.5. *Let $\boldsymbol{x} = \chi^S$ be the vector returned by the minimum regret rounding algorithm with input $\boldsymbol{x}'$ and some $t > 0$, where $\boldsymbol{x}' \in P_E$. Then S is an independent set.*

Proof. Since $\boldsymbol{x}$ is obtained from $\boldsymbol{x}'$ by iteratively applying f, we have $\boldsymbol{x} \in P_E$ by Lemma 4.3. From Lemma 4.4, we have that upon termination of phase $k = 1$, the diameter of each connected components of the fractional support of $\boldsymbol{x}$ is strictly less than 0. This implies that the graph induced by the fractional support is empty, hence $\boldsymbol{x}$ is an integer vector. □

We have implemented the minimum regret rounding heuristic for $t = 1$ and $t = 2$. Let us analyse the time complexity of the minimum regret rounding algorithm for those values of t. We start by analysing the time complexity of phase $k = 1$, the only phase of the algorithm if $t = 1$. In this phase, condition (9) is automatically fulfilled, and any fractional component x_v defines its own singleton independent set in Q. Since the regret of the set $\{v\}$ can be computed in $O(|\delta(v)|+1)$ time for each $v \in V$, Q can be initialised in $O(|V|\log|V|+|E|)$ time. Extracting a node with minimum regret from Q takes at most $O(\log|V|)$ time. Moreover, for each node $v \in V$, x_v is set to 0 at most once, and when this happens at most $|\delta(v)|$ priorities have to be decreased. Since decreasing a priority takes $O(\log|V|)$ time, the total time spent in decreasing priorities is at most

$$\sum_{v\in V}(|\delta(v)| \cdot O(\log|V|)) = O(\log|V|) \cdot \sum_{v\in V}|\delta(v)| = O(|E|\log|V|).$$

Summing everything together, phase $k = 1$ of the rounding algorithm can be implemented to work in $O((|V| + |E|)\log|V|)$ time.

Next we analyse the time complexity of phase $k = 2$, which precedes phase $k = 1$ in the case that $t = 2$. Each node $v \in V$ occurs in at most $|V| - 1$ independent sets of size two, and $N_1(v)$ intersects with $N_1(I)$ for at most $|\delta(v)| \cdot (|V| - 1)$ possible choices of $I \in \mathcal{I}$ with $|I| = 2$. So, the number of independent sets of size two is at most $O(|V|^2)$, and their regret values can be initialised in time

$$O(\sum_{v\in V}|\delta(v)| \cdot (|V| - 1)) = O(|V||E|).$$

It follows that Q can be initialised in $O(|V|^2\log|V| + |V||E|)$ time. For $v \in V$, when x_v is set to 0, at most $|\delta(v)|\cdot(|V|-1)$ priorities have to be decreased, each decrease of a priority taking $O(\log|V|)$ time, summing up to a total of $O(|V||E|\log|V|)$ time. Finally, we have to check condition (9). This can be done by keeping track of the sizes of the sets $\mathrm{supp}(\boldsymbol{x}_{N_1(v)\cap N_1(w)})$ for each $\{v, w\} \in \mathcal{I}$ $(v \neq w)$. These sizes can be stored in a $|V| \times |V|$ matrix. This matrix has to be updated each time we set x_v to 0 for some $v \in V$. This takes at most $O(|V||E|)$ time in total. It follows that phase $k = 2$ of the rounding algorithm can be implemented in $O((|V| + |E|)|V|\log|V|)$ time. This term dominates the time complexity of the algorithm.

We complete this section with observing that for any natural number $k > 2$, the number of different independent sets of size k is at most $|V|^k$, which is polynomial in $|V|$ for fixed k. As a consequence, implementing the minimum regret rounding algorithm in a brute-force fashion will yield a polynomial algorithm for any fixed value of t.

5 Optimisation Algorithms for Independent Set

Here we consider solving maximum independent set problems to optimality using LP-based branch-and-bound algorithms. Section 5.1 reviews the basic branch-and-cut and cutting plane/branch-and-bound algorithm. The initial LP-formulation is given in Section 5.2. To speed up our branch-and-bound algorithms we use various techniques for setting variables. These are the subjects of Sections 5.3–5.6. We end this section considering how to apply the separation algorithms on graphs obtained from map labelling instances in Section 5.9.

5.1 Branch and Cut

A branch and cut algorithm is a branch and bound algorithm where we may call a *cutting plane algorithm* in each node of the search tree. Here we give a short description of a basic version of branch and bound and of a cutting plane algorithm. For further details we refer to the book by Wolsey [42].

Consider the problem of determining $z_{\mathrm{OPT}} = \max\{z(\boldsymbol{x}) : \boldsymbol{x} \in P, \boldsymbol{x} \text{ integer}\}$, where z is a linear function in $\boldsymbol{x}$, and where P is a polyhedron. We refer to this problem as problem Π. Branch and bound makes use of the *linear programming relaxation* (LP-relaxation) $\bar{\Pi}$: $\bar{z} = \max\{z(\boldsymbol{x}) : \boldsymbol{x} \in P\}$. It is easy to see that $\bar{z} \geq z_{\mathrm{OPT}}$. At the top level, or *root node*, of the branch and bound tree we have problem $\bar{\Pi}$. At level k of the tree we have a collection of problems, say $\bar{\Pi}_1, \ldots, \bar{\Pi}_l$ such that the corresponding polyhedra $P_1, \ldots, P_l$ are pairwise disjoint, and such that all integral vectors in P are contained in $P_1 \cup \cdots \cup P_l$.

The algorithm works as follows. We maintain a set of *open problems*, the best known value of an integer solution $z^* = z(\boldsymbol{x}^*)$, and the corresponding integer solution $\boldsymbol{x}^*$. At first, problem $\bar{\Pi}$ is the only open problem. In iteration i, we select an open problem $\bar{\Pi}^i$ and solve it. If problem $\bar{\Pi}^i$ is infeasible or if the value $z(\bar{\boldsymbol{x}}^i)$ of its optimal solution, $\bar{\boldsymbol{x}}^i$, is less than z^* we remove $\bar{\Pi}^i$ from the list of open problems and continue with the next iteration. If the optimal solution to $\bar{\Pi}^i$ is integral, i.e., $\bar{\boldsymbol{x}}^i$ is a solution to problem Π, we set $\boldsymbol{x}^* := \bar{\boldsymbol{x}}^i$ if $z(\bar{\boldsymbol{x}}^i) > z^*$, we remove $\bar{\Pi}^i$ from the list of open problems and we proceed to the next iteration. Otherwise, we identify a component j of the vector $\bar{\boldsymbol{x}}^i$ such that $\bar{x}^i_j$ is not integral, and "branch" on x_j, i.e., we formulate two new open problems, say $\bar{\Pi}^i_1$ and $\bar{\Pi}^i_2$ by adding constraints $x_j \leq \lfloor \bar{x}^i_j \rfloor$ and $x_j \geq \lceil \bar{x}^i_j \rceil$ to P^i. If x_j is a 0-1 variable we add constraints $x_j = 0$ and $x_j = 1$. Note that $\bar{\boldsymbol{x}}^i$ neither belongs to P^i_1 nor to P^i_2. The value of $z(\bar{\boldsymbol{x}}^i)$ is an upper bound on the value of any solution to the problems $\bar{\Pi}^i_1$ and $\bar{\Pi}^i_2$. We replace $\bar{\Pi}^i$ by $\bar{\Pi}^i_1$ and $\bar{\Pi}^i_2$ in the set of open problems and proceed to the next iteration. The algorithm stops after the set of open problems becomes empty.

When using branch and bound to solve integer programming problems it is crucial that we obtain good lower and upper bounds on the optimal value as the bounds are used to prune the search tree. In order to obtain good upper bounds we strengthen the LP-relaxation by adding *valid inequalities*.

In a *cutting plane algorithm* we maintain an LP-relaxation of a problem Π. We start with the formulation P. Solve $\bar{\Pi}$ given P. If the optimal solution $\bar{\boldsymbol{x}}$ is integral, then we stop. Otherwise, we call the *separation algorithms* based on all families of valid inequalities that we consider. If any violated inequalities are identified we add them to P. If no violated inequalities are found we stop.

In a *cutting-plane/branch-and-bound* algorithm we run a cutting plane algorithm to obtain a strong linear relaxation, and then use this relaxation in an LP-based branch-and-bound algorithm.

5.2 GUB Formulation

We can rewrite the clique formulation by introducing the slack variables $\boldsymbol{s} \in \{0,1\}^{\mathcal{C}}$. The resulting equality constraints can then be interpreted as so-called *generalised upper bound (GUB) constraints.* A GUB constraint models the situation in which we have to choose one option from a set of mutually exclusive options. GUB constraints were introduced by Beale and Tomlin [2].

The advantage of having GUB constraints is that they can be used to define a branching scheme known as branching on GUBs. Let

$$P^{\mathrm{GUB}}_{\mathcal{C}} = \{(\boldsymbol{x}, \boldsymbol{s}) \in \mathbb{R}^V \times \mathbb{R}^{\mathcal{C}} \mid \boldsymbol{x}(C) + s_C = 1 \text{ for all } C \in \mathcal{C}, \boldsymbol{x} \geq \boldsymbol{0}, \boldsymbol{s} \geq \boldsymbol{0}\}.$$

This leads to the following formulation of the maximum independent set problem, which we call the *GUB* formulation:

$$\max\{\boldsymbol{x}(V) \mid (\boldsymbol{x}, \boldsymbol{s}) \in P_{\mathcal{C}}^{\mathrm{GUB}} \cap \{0,1\}^{V \cup \mathcal{C}}\}. \tag{10}$$

In our branch-and-cut algorithm we use the GUB formulation.

5.3 Reduced Cost Variable Setting

The branch-and-bound algorithm is correct as long as we do not discard any solution that is better than our current best solution from the remaining search-space. Consider iteration i of the branch-and-bound algorithm in which the LP relaxation was feasible. Let the LP relaxation in iteration i be denoted by

$$\begin{aligned} \max \quad & \boldsymbol{x}(V) && \text{(11a)}\\ \text{subject to} \quad & A\boldsymbol{x} + I\boldsymbol{s} = \boldsymbol{b} && \text{(11b)}\\ & \boldsymbol{l}^i \leq \boldsymbol{x} \leq \boldsymbol{u}^i && \text{(11c)} \end{aligned}$$

for an appropriate choice of slack variables $\boldsymbol{s}$ and right-hand side $\boldsymbol{b}$. We can exploit the information obtained from the optimal solution of the LP relaxation to tighten the lower and upper bounds $\boldsymbol{l}^i$ and $\boldsymbol{u}^i$, respectively, on the variables. These improved bounds are based on the value of the best known integer solution, the value of the LP relaxation, and the reduced cost of non-basic variables in an optimal LP solution.

Let $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi})$ be an optimal primal-dual pair to (11) where $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ for some basis B. Further, let $z^{\mathrm{LP}} = z(\boldsymbol{x}^{LP})$, and let L, U be the sets of variable indices with $\boldsymbol{c}_L^{\boldsymbol{\pi}} < \boldsymbol{0}$ and $\boldsymbol{c}_U^{\boldsymbol{\pi}} > \boldsymbol{0}$. The reduced cost $c_j^{\boldsymbol{\pi}}$ can be interpreted as the change of the objective function per unit change of variable x_j. From the reduced cost optimality conditions (see e.g. Dantzig and Thapa [10]) it follows that $x_j = u_j^i$ if $c_j^{\boldsymbol{\pi}} > 0$ and $x_j = l_j^i$ if $c_j^{\boldsymbol{\pi}} < 0$. Using these observations and the difference in objective function between the optimal LP solution and $\boldsymbol{x}^*$ we can compute a new lower bound for x_j if $c_j^{\boldsymbol{\pi}} > 0$, and a new upper bound for x_j if $c_j^{\boldsymbol{\pi}} < 0$. For zero-one integer programming problems, this means that we can set $l_j^i = 1$ if $x_j^{\mathrm{LP}} = 1$ and $c_j^{\boldsymbol{\pi}} > z^{\mathrm{LP}} - z^*$, and we can set $u_j^i = 0$ if $x_j^{\mathrm{LP}} = 0$ and $c_j^{\boldsymbol{\pi}} < z^* - z^{\mathrm{LP}}$, where $j \in V \cup \mathcal{C}$ is any variable index.

Denote the node in the branch-and-bound tree that is associated with iteration i by v_i, and the sub-tree of the branch-and-bound tree that is rooted at node v_i by T_{v_i}. The improved bounds can be used in all iterations of the branch-and-bound algorithm that are associated with any node in the branch-and-bound tree in the sub-tree rooted at node v_i.

When a variable index j satisfies $l_j^i = u_j^i$ we say that x_j is *set* to $l_j^i = u_j^i$ in iteration i (node v_i). When a variable is set in the root node of the branch-and-bound tree, it is called *fixed*. If $l_j^i < u_j^i$, we say that x_j is *free* in iteration i (node v_i). Variable setting based on reduced cost belongs to the folklore and is used by many authors to improve the formulation of zero-one integer programming problems (for example by Crowder, Johnson, and Padberg [9] and Padberg and Rinaldi [33]).

Note that the new bounds are a function of $\boldsymbol{\pi}, z^{\mathrm{LP}}$, and z^*. As a consequence, each time that we find a new best integer feasible solution in the branch-and-bound algorithm we can re-compute the bounds. Suppose we find an improved primal solution in iteration k. An original feature of our implementation of the branch-and-bound algorithm is that we re-compute the bounds in all nodes v_i of the branch-and-bound tree with $i \in \{1, \ldots, k\}$ that are on a path from the root node to a node $v_{k'}$ with $k' > k$. In order to be able to do this, we store a tree T' that mirrors the branch-and-bound tree. Each node w_i in T' corresponds to some iteration i of the branch-and-bound algorithm, and with w_i we store its parent $p(w_i)$ in T', and the values of $\boldsymbol{\pi}$, z^{LP}, and z^* for which we last computed the bounds in w_i, and the bounds that we can actually improve in node w_i. The values of $\boldsymbol{\pi}$ are stored implicitly by storing only the differences of the optimal LP basis between node w_i and node $p(w_i)$.

The actual re-computation of bounds is done in a lazy fashion as follows. In iteration k of the branch-and-bound algorithm, we compute the path P from w_1 to w_k in T' using the parent pointers. Next, we traverse P from w_1 to w_k, and keep track of the final basis in each node using the differences, and of the best available bounds on each variable using the improved bounds that are stored in the nodes on P. Consider some node w_i in this traversal. If the value of z^* that is stored in w_i is less than the actual value, we re-compute the bounds in w_i. If any of the bounds stored in node w_i contradicts with bounds stored in a node w_j that preceded w_i in the traversal, we have found a proof that there do not exists integer solutions that satisfy the bounds of iteration k and we fathom node w_k. If any of the bounds stored in node w_i is implied by a bound stored in a node w_j that preceded w_i in the traversal, we remove it from node w_i.

Consider an execution of the branch-and-bound algorithm and let η denote the number of times we improve on the primal bound. For $w_i \in T'$ let J' denote the non-basic variables in the final basis of node w_i. Assuming that $n \gg m$, the time spent in the re-computation of bounds of node w_i is dominated by the re-computation of the reduced cost from the final basis of node w_i, which is of the order

$$O(\eta|\mathrm{supp}(A_{J'})|). \tag{12}$$

In a typical execution of the branch-and-bound algorithm, we improve on the value of z^* only a few times. Moreover, we use a branch-and-cut algorithm that calls sophisticated and time-consuming subroutines in each iteration of the branch-and-bound algorithm. These observations imply that the bound (12) is dominated by the running time of the other computations performed in iteration i of the branch-and-bound algorithm in our applications. We believe that the benefit of having strengthened formulations is worth the extra terms (12) in the running time of the branch-and-bound algorithm, as the improved formulations help in reducing the size of the branch-and-bound tree.

5.4 GUB Constraints and Variable Setting

Whenever we have a formulation in which some of the constraints are GUB constraints, we can exploit this by strengthening the bounds on those variables that are in a GUB constraint using a slightly stronger argument than the one presented in Section 5.3. Consider again iteration i of the branch-and-bound algorithm in which the LP relaxation was feasible. Denote this LP relaxation by

$$\begin{aligned} \max \quad & z(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} && \text{(13a)}\\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b} && \text{(13b)}\\ & \boldsymbol{l}^i \le \boldsymbol{x} \le \boldsymbol{u}^i && \text{(13c)} \end{aligned}$$

Let I' be the set of row indices corresponding to maximal clique constraints with explicit slack variables.

For each clique $C \in \mathcal{C}$, the slack variable s_C (as introduced in Section 5.2) can be interpreted as a variable corresponding to an extra node in G, say v_C, that has an objective function coefficient of 0 and is connected to all nodes in C. Denote this extended graph by $G' = (V', E')$.

For $j \in V'$, whenever x_j has value one, the GUB constraints imply that $\boldsymbol{x}_{N(j)} = \boldsymbol{0}$, and whenever x_j has value zero the GUB constraints imply that for at least one $k \in N(j)$ x_k has value one. The strengthened argument for modifying the upper bound on x_j takes into account the reduced cost of the variables $\boldsymbol{x}_{N(j)}$, and the strengthened argument for modifying the lower bound on x_j takes into account the reduced cost of the variables x_k and $\boldsymbol{x}_{N(k)}$ for some properly chosen $k \in N(j)$.

Let z^* again denote the value of the best known integer solution. Consider iteration i in which the LP relaxation is feasible and let $((\boldsymbol{x}_V^{\mathrm{LP}}, \boldsymbol{x}_{\mathcal{C}}^{\mathrm{LP}}), \boldsymbol{\pi})$ be an optimal primal-dual pair to it, where $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ for some basis $B \subseteq V'$ and A denotes the constraint matrix. Let $z^{\mathrm{LP}} = z(\boldsymbol{x}^{LP})$, and let $L, U \subseteq V'$ be the sets of variable indices with $\boldsymbol{c}_L^{\boldsymbol{\pi}} < \boldsymbol{0}$ and $\boldsymbol{c}_U^{\boldsymbol{\pi}} > \boldsymbol{0}$. The strengthened bounds $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \{0,1\}^{V'}$ are defined as

$$\tilde{l}_j^i = \begin{cases} \max(0, 1 + \lceil (z^* - z^{\mathrm{LP}}) / \min\{-\tilde{c}_k^{\boldsymbol{\pi}} \mid k \in N(j)\} \rceil), & \text{if } j \in U, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$\tilde{u}_j^i = \begin{cases} \min(1, \lfloor (z^* - z^{\mathrm{LP}}) / \tilde{c}_j^{\boldsymbol{\pi}} \rfloor), & \text{if } j \in L, \\ 1, & \text{otherwise,} \end{cases}$$

where for each $j \in L$

$$\tilde{c}_j^{\boldsymbol{\pi}} = c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(N(j) \cap U).$$

Proposition 5.1. *All integer $\boldsymbol{x}^{\mathrm{IP}}$ that are feasible to (13) with $z(\boldsymbol{x}^{\mathrm{IP}}) \geq z^*$ satisfy $\tilde{\boldsymbol{l}}^i \leq \boldsymbol{x}^{\mathrm{IP}} \leq \tilde{\boldsymbol{u}}^i$.*

Proof. Let $\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi}, z^{\mathrm{LP}}, L, U$ be as in the construction of $z^*, i, \tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. Assume that there exists an integer vector $\boldsymbol{x}^{\mathrm{IP}}$ that is feasible to (13) with $z(\boldsymbol{x}^{\mathrm{IP}}) \geq z^*$. A fundamental result from the theory of linear programming is that we can rewrite the objective function in terms of the reduced cost. This yields

$$\begin{aligned} z(\boldsymbol{x}^{\mathrm{IP}}) &= \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{x}_L^{\mathrm{IP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{x}_U^{\mathrm{IP}}, \text{ and} \\ z(\boldsymbol{x}^{\mathrm{LP}}) &= \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{l}_L^i + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{u}_U^i. \end{aligned}$$

Observe that the feasibility of $\boldsymbol{x}^{\mathrm{IP}}$ implies $\boldsymbol{x}_L^{\mathrm{IP}} \geq \boldsymbol{0}$ and $\boldsymbol{x}_U^{\mathrm{IP}} \leq \boldsymbol{1}$, so $\boldsymbol{x}_U^{\mathrm{IP}} - \boldsymbol{1} \leq \boldsymbol{0}$. Now, choose $j \in L$ arbitrarily. Either $x_j^{\mathrm{IP}} = 0$ or $x_j^{\mathrm{IP}} = 1$. In the case that $x_j^{\mathrm{IP}} = 0$ we directly have $x_j^{\mathrm{IP}} \leq \tilde{u}_j^i$ since $\tilde{u}_j^i \geq 0$. So assume that $x_j^{\mathrm{IP}} = 1$. It follows that $\boldsymbol{x}_{N(j)}^{\mathrm{IP}} = \boldsymbol{0}$. But then,

$$\begin{aligned} z^* - z^{\mathrm{LP}} &\leq z(\boldsymbol{x}^{\mathrm{IP}}) - z(\boldsymbol{x}^{\mathrm{LP}}) \\ &= (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T (\boldsymbol{x}_L^{\mathrm{IP}} - \boldsymbol{l}_L^i) + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T (\boldsymbol{x}_U^{\mathrm{IP}} - \boldsymbol{u}_U^i) \\ &= (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{x}_L^{\mathrm{IP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T (\boldsymbol{x}_U^{\mathrm{IP}} - \boldsymbol{1}) \\ &\leq c_j^{\boldsymbol{\pi}} x_j^{\mathrm{IP}} + (\boldsymbol{c}_{U \cap N(j)}^{\boldsymbol{\pi}})^T (\boldsymbol{x}_{U \cap N(j)}^{\mathrm{IP}} - \boldsymbol{1}) \\ &= c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j)). \end{aligned}$$

Since $c^{\boldsymbol{\pi}}_j - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j)) < 0$, we find

$$(z^* - z^{\mathrm{LP}})/(c^{\boldsymbol{\pi}}_j - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j))) \geq 1.$$

Hence $x^{\mathrm{IP}}_j = 1 \leq \tilde{u}^i_j$. So both $x^{\mathrm{IP}}_j = 0$ and $x^{\mathrm{IP}}_j = 1$ imply $x^{\mathrm{IP}}_j \leq \tilde{u}^i_j$. Because j was chosen arbitrarily we have $\boldsymbol{x}^{\mathrm{IP}} \leq \tilde{\boldsymbol{u}}^i$.

The proof that $\boldsymbol{x}^{\mathrm{IP}} \geq \tilde{\boldsymbol{l}}^i$ is derived similarly starting from an arbitrarily chosen index $j \in U$, assuming that $x^{\mathrm{IP}}_j = 0$, and using the observation that $x^{\mathrm{IP}}_j = 0$ implies $x^{\mathrm{IP}}_k = 1$ for some index $k \in N(j)$, which in turn implies that $\boldsymbol{x}^{\mathrm{IP}}_{N(k)} = \boldsymbol{0}$. □

The strengthened criteria for setting variables based on reduced cost can be taken into account in an implementation that stores the reduced cost of the variables in an array by replacing $c^{\boldsymbol{\pi}}_j$ by $\min\{-\tilde{c}^{\boldsymbol{\pi}}_k \mid k \in N(j)\}$ for all $j \in U$ and by $\tilde{c}^{\boldsymbol{\pi}}_j$ for all $j \in L$ in this array. Having done this the strengthened bounds can be computed as in Section 5.3. The extra time needed for pre-processing the array is $O(|E|)$.

5.5 Logical Implications

In each node of the branch-and-bound tree we solve a linear programming relaxation of the form

$$z^{\mathrm{LP}} = \max\{z(\boldsymbol{x}) = \boldsymbol{x}(V) \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}, \tag{14}$$

for some $\boldsymbol{l}, \boldsymbol{u} \in \{0,1\}^V \times \{0,1\}^{\mathcal{C}}$ where $A, \boldsymbol{b}$ are obtained from the constraint matrix of $P^{\mathrm{GUB}}_{\mathcal{C}}$ and $\boldsymbol{1}$, respectively, by adding the rows and right hand sides of valid inequalities that are produced by our separation algorithms. Recall the notion of setting variables from Section 5.3.

We start with the most elementary of all logical implications:

Proposition 5.2. *Let $v \in V$ be a node in G and let $W = N(v)$ be its set of neighbours. If x_v is set to one, then x_w can be set to zero for all $w \in W$ without changing z^{LP}.*

Proof. Directly from the definition of P_E. □

Proposition 5.3. *Let v, W be as in Proposition 5.2. If x_w is set to zero for all $w \in W$, then x_v can be set to one without changing the value of z^{LP}.*

Proof. Because $z^{\mathrm{LP}} = z(\boldsymbol{x}^{\mathrm{LP}})$ for some optimal solution $\boldsymbol{x}^{\mathrm{LP}}$ to model (14), and $\boldsymbol{x}^{\mathrm{LP}}_W = \boldsymbol{0}$ together with optimality of $\boldsymbol{x}^{\mathrm{LP}}$ implies that $x^{\mathrm{LP}}_v = 1$, the proposition holds. □

If we set x_v to one for some $v \in V$, then we can set x_w to zero for all neighbours w of v by Proposition 5.2. In a formulation of the independent set problem with explicit slack variables, such as model (10), it is possible to interpret the slack variable s_C that is associated with clique $C \in \mathcal{C}$ as a variable that is associated with an extra node that is connected to all nodes in C. Using this interpretation we can also apply Proposition 5.2 to the slack variables s_C for all $C \in \mathcal{C}$. If we set x_v to zero for some $v \in V$, its neighbours may satisfy the conditions of Proposition 5.3.

After applying Proposition 5.2 to all nodes that are set to one, we can restate the linear programming problem 14 in terms of its free variables. The resulting linear programming problem can be interpreted as a (fractional) independent set problem in the graph induced by the nodes that correspond to the free variables (i.e., with a lower bound of $\boldsymbol{0}$ and an upper bound of $\boldsymbol{1}$). Therefore, setting variables to zero can be interpreted as removing nodes from the graph.

Proposition 5.4. *Let v be a node of G, let $U = N(v)$ be the set of neighbours of v, and let $W = N(N_1(v))$ be the set of neighbours of the 1-neighbourhood of v. When removing v from G, only the following cases can occur:*

(i) the nodes in U may become simplicial nodes,

(ii) the nodes in W may become node-dominated by nodes in U, or

(iii) the nodes in W may become set-dominated.

Proof. Directly from Propositions 3.6, 3.7, and 3.8. □

After setting a variable to zero, we check whether any of the three cases of Proposition 5.4 occurs, and if so, we take the appropriate action.

5.6 Variable Setting by Recursion and Substitution

Suppose we are solving an instance of the map labelling problem P with conflict graph $G = (V, E)$ using a branch-and-cut algorithm. Focus on iteration j of the branch-and-cut algorithm just after the termination of the cutting plane algorithm. Recall from Section 5.5 that we have solved an LP of the form (14) for some lower and upper bounds $\boldsymbol{l}, \boldsymbol{u}$. Let $J_1, J_2 \subseteq V$ be the set of nodes whose corresponding variables are set to zero, and one, respectively, and let $F = V \setminus (J_1 \cup J_2)$ be the set of nodes whose corresponding variables are free. Because we exhaustively apply Proposition 5.2, for each $v \in J_2$ we have that $\boldsymbol{l}_{N(v)} = \boldsymbol{0}$, so all neighbours of v are members of J_1. Note that $G[F]$ does not need to be a connected graph, even if G is. Let $\{F_1, \ldots, F_k\}$ be the partition of F such that for each $i \in \{1, \ldots, k\}$, $G[F_i]$ is a connected component of $G[F]$. The following proposition, which is similar to Proposition 3.6, states that we can find the optimal integer solution in the set

$$P_E \cap \{\boldsymbol{x} \in \mathbb{R}^V \mid \boldsymbol{l}_V \leq \boldsymbol{x} \leq \boldsymbol{u}_V\}$$

by combining optimal solutions for the individual F_i, where P_E again denotes the edge-formulation of Section 3.

Proposition 5.5. *Let $i \in \{1, \ldots, k\}$, let J_1, J_2 and F_i be as above, and let $I \subseteq F_i$ be a maximum independent set in $G[F_i]$. Then, there exists an independent set I^* in G with $I \subseteq I^*$ that is maximum under the restrictions that $J_1 \cap I^* = \varnothing$ and $J_2 \subseteq I^*$.*

Proof. Let I_1 be a maximum independent set in G under the restrictions that $J_1 \cap I_1 = \varnothing$ and $J_2 \subseteq I_1$. Let $I_2 = I_1 \setminus F_i$ and $I^* = I_2 \cup I$. Because we exhaustively use Proposition 5.2 and F_i is maximal by the definition of connected components, for all $v \in F_i$ and $\{v, w\} \in \delta(v)$ we have that $w \in F_i \cup J_1$. It follows that I^* is an independent set in G. Because $I_1 \cap F_i$ is an independent set in $G[F_i]$, we find that

$$|I_2| = |I_1 \setminus F_i| = |I_1| - |I_1 \cap F_i| \geq |I_1| - |I|.$$

Because $I \cap I_2 = \varnothing$, this implies that

$$|I^*| = |I_2 \cup I| = |I_2| + |I| \geq |I_1| - |I| + |I| \geq |I_1|.$$

Moreover, $J_1 \cap I^* = \varnothing$ and $J_2 \subseteq I^*$ by choice of I_1 and construction of I^*. So I^* is an independent set in G that is at least as large as I_1, and satisfies $I \subseteq I^*$, $J_1 \cap I^* = \varnothing$ and $J_2 \subseteq I^*$. □

Now, suppose that $|F_1| \geq |F_i|$ for all $i \in \{2, \dots, k\}$. We see F_1 as the main component of the problem we solve in the part of the branch-and-bound tree rooted at v_j, and leave it aside for the remainder of iteration j. (Here v_j again denotes the node of the branch-and-bound tree corresponding to iteration j.) For all $i \in \{2, \dots, k\}$, however, we recursively compute a maximum independent set I_i^* in $G[F_i]$, and then set the variables in I_i^* to one and the variables in $F_i \setminus I_i^*$ to zero. By Proposition 5.5, this does not reduce the value of the optimal solution. After substituting the solutions of the recursively computed independent sets, we obtain a new solution and a new bound for iteration j of the branch-and-cut algorithm. We call this technique *variable setting by recursion and substitution*, or SRS.

5.7 Separation of Lifted Odd Holes

Here we consider the separation algorithm for lifted odd hole inequalities. Let $\boldsymbol{x} \in P_E$ be a fractional solution. Recall from Section 3.3 that a lifted odd hole is of the form

$$\boldsymbol{x}(V(H)) + \sum_{v \in V \setminus V(H)} \alpha_v x_v \leq \lfloor |V(H)|/2 \rfloor$$

for some suitable vector $\boldsymbol{\alpha} \in \mathbb{R}_+^V$. We compute values $\boldsymbol{\alpha} \in \mathbb{N}_+^V$ using sequential lifting (see Padberg [32] and Wolsey [40, 41]), to obtain facet-defining inequalities of P_{IS}.

The separation algorithm for lifted odd hole inequalities consists of two parts. The first part derives an odd hole H from $\boldsymbol{x}^{\mathrm{LP}}$ that defines a violated or nearly violated odd hole inequality. The second part consists of lifting the resulting odd hole inequality so that it becomes facet-defining for P_{IS}. After the lifting, we check whether we have found a violated inequality and if so, we report it.

5.7.1 Identifying a (Nearly) Violated Odd Hole Inequality

We start by describing the separation algorithm for the basic odd hole inequalities, $\boldsymbol{x}(V(H)) \leq \lfloor |V(H)|/2 \rfloor$. Suppose we are given a vector $\boldsymbol{x}^{\mathrm{LP}} \in P_E$. We first find an odd cycle starting from some node $v \in V$ using the construction described by Grötschel, Lovász, and Schrijver [18]. To find a shortest odd cycle containing node v, Grötschel *et al.* construct an auxiliary bipartite graph $\tilde{G} = ((V^1, V^2), \tilde{E})$ and cost vectors $\boldsymbol{c} \in [0,1]^E$ and $\tilde{\boldsymbol{c}} \in [0,1]^{\tilde{E}}$ as follows. Each node $v \in V$ is split into two nodes v^1 and v^2, with v^i included in V^i $(i = 1, 2)$. For each edge $\{u, v\} \in E$, we add the edges $\{u^1, v^2\}$ and $\{u^2, v^1\}$ to $\tilde{E}$, and set $c_{\{u,v\}} = \tilde{c}_{\{u^1,v^2\}} = \tilde{c}_{\{u^2,v^1\}} = 1 - x_u^{\mathrm{LP}} - x_v^{\mathrm{LP}}$. Observe that a path from $u^1 \in V^1$ to $v^2 \in V^2$ in $\tilde{G}$ corresponds to a walk of odd length in G from u to v.

A shortest path from v^1 to v^2 in $\tilde{G}$ corresponds to a shortest odd length closed walk in G containing v. The reason that we are looking for a shortest path is that an odd hole H in G defines a violated odd hole inequality if $\boldsymbol{c}(E(H)) < 1$, and a short closed walk in $\tilde{G}$ is more likely to lead to a violated lifted odd hole inequality than a long one. In our implementation, we restricted ourselves to shortest path with length at most 1.125. Shortest paths in a graph with non-negative edge lengths can be found using Dijkstra's algorithm [14]. Hence, we can find a closed walk

$$C := (v = v_0, e_1, v_1, e_2, v_2, \dots, v_{k-1}, e_k, v_k = v)$$

in G with odd k that is minimal with respect to $\boldsymbol{c}$ and $|C|$ by using Dijkstra's algorithm to find a shortest path (with respect to $\tilde{\boldsymbol{c}}$) of minimal cardinality from v^1 to v^2 in $\tilde{G}$. Some of the v_i may occur more than once, and the walk may have chords.

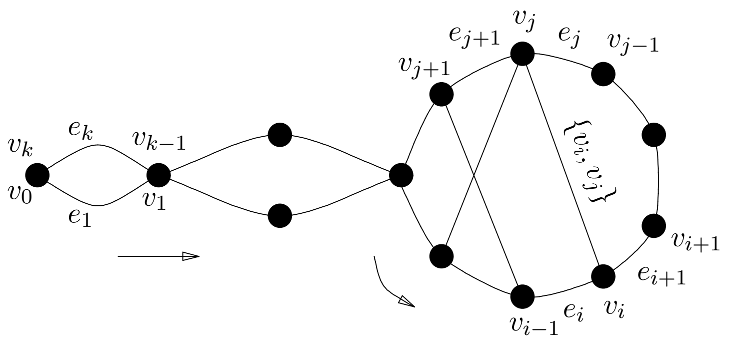


Figure 4: Identifying an odd hole in a closed walk.

Let $j \in 2, \ldots, k-1$ be the smallest index such that there exists an edge $\{v_i, v_j\} \in E$ for some $i \in 0, \ldots, j-2$ (such i, j exist because $\{v_0, v_{k-1}\} \in E$). Let $i \in 0, \ldots, j-2$ be the largest index such that $\{v_i, v_j\} \in E$. Let

$$H := (v_i, e_{i+1}, v_{i+1}, \ldots, e_j, v_j, \{v_j, v_i\}, v_i).$$

The construction of H is illustrated in Figure 4. We proceed by showing that H is indeed an odd hole.

Proposition 5.6. *Let H be constructed as above. Then, H is an odd hole in G.*

Proof. Because $\{v^1, v^2\} \notin \tilde{E}$, we have that $|H| \geq 3$. Clearly H is a cycle in G. By choice of i and j, H does not contain chords, so H is a hole in G. It remains to prove that $|V(H)|$ $(= j - i + 1)$ is odd. Suppose by way of contradiction that $|V(H)|$ is even. Then,

$$C' := (v = v_0, e_1, \ldots, v_{i-1}, e_i, v_i, \{v_i, v_j\}, v_j, e_{j+1}, v_{j+1}, \ldots, e_k, v_k = v)$$

is an odd length closed walk in G containing v. Moreover,

$$\boldsymbol{c}(\{e_{i+1}, \ldots, e_j\}) = (j-i) - (2 \textstyle\sum_{p=i+1}^{j-1} x^{\mathrm{LP}}_{v_p}) - x^{\mathrm{LP}}_{v_i} - x^{\mathrm{LP}}_{v_j}.$$

It follows from $\boldsymbol{x}^{\mathrm{LP}}(\{v_p, v_{p+1}\}) \leq 1$ that

$$\sum_{p=i+1}^{j-1} x^{\mathrm{LP}}_{v_p} \leq (j-i-1)/2.$$

Therefore,

$$\boldsymbol{c}(\{e_{i+1}, \ldots, e_j\}) \geq (j-i) - (j-i-1) - x^{\mathrm{LP}}_{v_i} - x^{\mathrm{LP}}_{v_j} = c_{\{v_i, v_j\}},$$

so C' is not longer than C with respect to $\boldsymbol{c}$. However, C' is of smaller cardinality, which contradicts our choice of C. Hence H is an odd hole in G. $\square$

If $|V(H)| = 3$, then H is a clique in G, and we ignore it in our computations.

5.7.2 Lifting an Odd Hole Inequality

Let H be an odd hole in G. Assume that we have an ordering of the node set $V \setminus V(H)$ that is given by $\{v_1, v_2, \ldots, v_{|V \setminus V(H)|}\}$. Padberg [32] has shown that a lifted odd hole induces a facet if we choose

$$\alpha_{v_i} = \lfloor |V(H)|/2 \rfloor - \max\{\boldsymbol{x}(V(H)) + \textstyle\sum_{j=1}^{i-1} \alpha_{v_j} x_{v_j} \mid \boldsymbol{x} \in X^i_{\text{IS}}\},$$

where

$$X^i_{\text{IS}} = \{\chi^I \mid I \text{ is an independent set in } G[(V(H) \cup \{v_1, \ldots, v_{i-1}\}) \setminus N_1(v_i)]\}.$$

In order to compute the lifting coefficients, we have to compute several maximum weight independent sets, one for each lifting coefficient.

Nemhauser and Sigismondi [28] observed that $\alpha_v = 0$ for $v \in V \setminus V(H)$ if $|N_1(v) \cap V(H)| \leq 2$. This implies that the independent set problems that have to be solved in order to compute the lifting coefficients $\boldsymbol{\alpha}$ are relatively small in practice. We lift the variables in non-decreasing lexicographic order of the pairs $(|\frac{1}{2} - x_v|, -|N_1(v) \cap V(H)|)$, where ties are broken at random.

To compute the coefficients $\boldsymbol{\alpha}$, we make use of a path decomposition (see Bodlaender [3], and de Fluiter [15]) of the graph induced by the nodes in the hole and the nodes we already lifted. Here we describe how we maintain the path decomposition. How to use path decompositions to compute maximum weight independent sets is described by Verweij [35].

Definition 5.1. A *path decomposition* of a graph $G = (V, E)$ is a sequence $(S_i)_{i=1}^n$ satisfying the following conditions:

$$\textstyle\bigcup_{i=1}^n S_i = V, \tag{15}$$

$$\text{for all } \{u, v\} \in E \text{ there exists } i \in \{1, \ldots, n\} \text{ with } \{u, v\} \subseteq S_i, \text{ and} \tag{16}$$

$$\text{for all } i, j, k \text{ with } 1 \leq i < j < k \leq n \text{ we have } S_i \cap S_k \subseteq S_j. \tag{17}$$

The *width* of a path decomposition $(S_i)_{i=1}^n$ is the value $\max_{i=1}^n |S_i| - 1$.

We may assume without loss of generality that S_i and S_{i+1} differ in only one node, i.e., that $S_{i+1} = S_i \cup \{v\}$ or $S_{i+1} = S_i \setminus \{v\}$ for some $v \in V$. We may also assume without loss of generality that $S_1 = S_n = \varnothing$. A path decomposition satisfying these assumptions is called *normalised*. A normalised path decomposition has $n = 2|V| + 1$. We will present our algorithm to compute the maximum weight of an independent set given a path decomposition in the next sub-section. Here, we proceed by outlining how we obtain and maintain the path decomposition that we use for this purpose.

Given a hole $H = (v_0, e_1, v_1, \ldots, e_n, v_n = v_0)$, an initial path decomposition of the graph $G[V(H)] = (V(H), E(H))$ of width two is given by

$$S_i = \begin{cases} \varnothing, & \text{if } i = 1 \text{ or } i = 2|V(H)| + 1, \\ \{v_0\}, & \text{if } i = 2 \text{ or } i = 2|V(H)|, \\ \{v_0, v_k\}, & \text{if } 2 < i < 2|V(H)| \text{ and } i = 2k + 1, \\ \{v_0, v_k, v_{k+1}\}, & \text{if } 2 < i < 2|V(H)| \text{ and } i = 2(k + 1). \end{cases}$$

Suppose at some stage we want to compute the lifting coefficient for some node $v \in V$. Let V' be the set of nodes that are either in the hole or did already receive a positive lifting coefficient at some earlier stage. Assume that $(S_i)_{i=1}^{2|V'|+1}$ is a path decomposition of the graph induced by V'. A path decomposition of $G[V' \setminus N_1(v)]$ can be obtained from $(S_i)_{i=1}^{2|V'|+1}$ by eliminating the nodes in $N_1(v)$ from all sets S_i $(i = 1, \ldots, 2|V'| + 1)$ and eliminating consecutive doubles (i.e., sets S_i and S_{i+1} that are equal).

For each node that we assign a positive lifting coefficient, we have to update our path decomposition. Suppose at some stage we have found a strictly positive lifting coefficient for some node $v \in V$. Let $V', (S_i)_{i=1}^{2|V'|+1}$ be as before. We have to extend the path decomposition so that it becomes a path decomposition for $G[V' \cup \{v\}]$. We do this in a greedy fashion, by identifying the indices j, k such that $j = \min\{i \mid \{u, v\} \in E, u \in S_i\}$ and $k = \max\{i \mid \{u, v\} \in E, u \in S_i\}$, and adding v to all sets S_i for $i \in \{j, \ldots, k\}$. Having done this, our path decomposition satisfies conditions (15)–(17) for the graph $G[V' \cup \{v\}]$. We normalise the resulting path decomposition to ensure that it satisfies our assumptions on the differences between consecutive sets.

5.8 Separation of Maximally Violated Mod-k Cuts

Suppose $\boldsymbol{x}^{\mathrm{LP}} \in P_E$ is a fractional solution to the maximum independent set problem. We use the algorithm described by Caprara, Fischetti, and Letchford [4, 5] to search maximally violated mod-k cuts. The most time-consuming step of the separation algorithm consists of computing, if it exists, a solution to a system of mod-k congruences that is formed by those constraints of the constraint matrix that are satisfied with equality. For this purpose we compute an LU-factorisation using arithmetic in $GF(k)$ for $k \in \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$.

As input to the mod-k separation algorithm, we use all (globally) valid inequalities for P_{IS} that are present in the formulation of the linear programming relaxation and are satisfied with equality by $\boldsymbol{x}^{\mathrm{LP}}$. The inequalities we use are the following: maximal clique inequalities (4), lifted odd hole inequalities (5), non-negativity constraints on $\boldsymbol{x}_V$, upper bound constraints of 1 on components of $\boldsymbol{x}_V$, and mod-k cuts that were found at an earlier stage of the algorithm.

5.9 Separation for Map Labelling Problems

We can exploit the structure of the independent set problems arising from map labelling by more conservatively calling our separation routines. Because adding cuts or bounds only tends to change the LP solution in a small region surrounding the nodes that are directly effected by them, there is no need to call the separation routines again in parts of the graph where no change of the LP solution has occurred.

We say that a solution $\boldsymbol{x}$ *precedes* $\tilde{\boldsymbol{x}}$ in a branch-and-cut algorithm if $\boldsymbol{x}$ ($\tilde{\boldsymbol{x}}$) is the optimal solution to an LP relaxation P ($\tilde{P}$, respectively), and $\tilde{P}$ is derived in the branch-and-cut algorithm from P by adding cuts or variable bounds. Focus on iteration i of the branch-and-cut algorithm, and let v_j be the parent of v_i in the branch-and-bound tree if $i > 1$. If the current LP relaxation is $\tilde{P}$ and $\tilde{P}$ is not the first iteration of the cutting plane algorithm in iteration i of the branch-and-cut algorithm, then P is the LP relaxation associated with the previous iteration of the cutting plane algorithm. Otherwise, if $i > 1$ and the current iteration of the cutting plane algorithm is its first iteration in iteration i of the branch-and-cut algorithm, then P is the LP relaxation associated with the last iteration of the cutting plane algorithm in iteration j of the branch-and-cut algorithm.

Let $G = (V, E)$ denote the conflict graph of an instance of the map labelling problem. Recall the definition of P_E from Section 3. Focus on iteration i of the branch-and-cut algorithm. Suppose that $\boldsymbol{x} \in P_E$ is an optimal solution to an LP relaxation in iteration i of the branch-and-cut algorithm, and let $\boldsymbol{x}' \in P_E$ be the optimal LP solution that precedes $\boldsymbol{x}$. In each iteration of the branch-and-cut algorithm, we only call the separation algorithm for lifted odd hole inequalities presented in Section 3.3 starting from nodes $v \in V$ for which x_v is fractional and $x_v \neq x'_v$. In this way, we avoid doing the same calculations over and over again in each iteration of the branch-and-cut algorithm. Although there is no guarantee that we find all the lifted odd hole inequalities one could find if one would start the separation from all nodes that correspond to fractional variables, we believe that our cutting plane algorithm still finds most of them. The decrease in processing time needed for each iteration of our branch-and-cut algorithm is considerable.

We now focus our attention on the remaining separation algorithms. For large map labelling instances, the systems of congruences mod-k that we have to solve for each run of the mod-k separation algorithm by Caprara, Fischetti, and Letchford [4, 5] are large as well. As a consequence, the mod-k separation turns out to be very time consuming. Although we suspect that a similar incremental strategy as for separating lifted odd-holes would solve this problem, developing such a strategy is non-trivial, and is still on our to-do list. Therefore, in our implementation we restrict the separation of mod-k inequalities to iteration one of the branch-and-cut algorithm. Since we start out with the complete clique formulation, we do not call our separation routines for finding maximal clique inequalities.

5.10 Enumeration and Branching Schemes

There are two aspects of the branch-and-cut algorithm that need to be specified to complete its description, namely, how we select the next open problem and how we choose to branch on fractional solutions. We use the best-first strategy in selecting the next open problem, i.e., we choose to process that open problem for which the value of the LP relaxation in its parent node is as high as possible.

There are several possible ways to branch on a fractional solution. One way is branching on a fractional variable, another is using a GUB constraint to derive a partitioning of the feasible region. We compute a so-called *pseudo-cost* for each possible branch, and then use these to decide on how to partition the feasible region. Details of our branching scheme can be found in Verweij [35]. Using pseudo-costs dates back to the nineteen seventies and has most recently been reported on by Linderoth and Savelsbergh [27].

6 Computational Results on Map Labelling Instances

We tested our heuristic and optimisation algorithms on the same class of map labelling instances as used by Christensen *et al.* [7] and van Dijk *et al.* [13]. These instances are generated by placing n (integer) points on a standard map of size 792 by 612. The points have to be labelled using labels of size 30×7.[1] For each $n \in \{100, 150, \ldots, 750, 950\}$ we randomly generated 25 maps. We will first turn our attention to the computational behaviour of our heuristic

[1]Since we use closed labels, the label $Q_1 = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{0} \leq \boldsymbol{x} \leq (30,7)^T\}$ intersects the label $Q(\boldsymbol{\alpha}) = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{\alpha} \leq \boldsymbol{x} \leq (\alpha_1 + 30, \alpha_2 + 7)^T\}$ for all $\boldsymbol{\alpha}$ with $|\boldsymbol{\alpha}| \leq (30,7)^T$. From the integrality of the data it follows that either $|\alpha_1| \geq 31$, or $|\alpha_2| \geq 8$, or both if Q_1 and $Q(\boldsymbol{\alpha})$ do not intersect. Some researchers see this as a reason to actually use labels of size $(30-\epsilon, 7-\epsilon)^T$ for some small constant $\epsilon > 0$. The reader should be aware of this when comparing the results presented in this chapter to results found in the literature.

Name	Heuristic	Section
SA	Simulated Annealing	4.1
DS	Diversified Neighbourhood Search	4.2
O1	1-Opt starting from zero	4.3
O2	1-Opt followed by 2-opt starting from zero	4.3
TS	Tabu Search on Independent Set Formulation	4.4
L1	Simple LP rounding followed by 1-opt	4.5.1, 4.3
L2	Simple LP rounding followed by 1-opt and 2-opt	4.5.1, 4.3
R1	Minimum regret rounding with parameter $t = 1$	4.5.2
R2	Minimum regret rounding with parameter $t = 2$	4.5.2

Table 1: Heuristics Applied to Map Labelling Instances

algorithms, then we will discuss our optimisation algorithms, and finally we will discuss the influence of SRS.

6.1 Heuristics

We evaluated the performance of our heuristics for the maximum independent set problem on conflict graphs of map labelling problem instances. The algorithms are summarised in Table 1. All LP-based rounding algorithms are started from a fractional independent set computed by the cutting plane algorithm as described in Section 5.9, using the complete clique formulation.

The average number of labels placed by our LP-based rounding algorithms and the corresponding running times are reported in Table 2. Because the separation of the mod-k cuts is computationally intensive we tried our heuristics with and without them. The column LP shows the average amount of CPU time needed to compute the LP relaxations in seconds. The L1 and L2 columns refer to the algorithms that first apply the simple LP rounding algorithms from Section 4.5.1 to obtain an integer solution. Next, these algorithms invoke an iterative improvement algorithm starting from this integer solution using the 1-opt and 2-opt neighbourhoods, respectively. The R1 and R2 columns refer to algorithms that apply the minimum regret rounding heuristic from Section 4.5.2 to the optimal solution of the LP relaxation, with the parameter t equal to 1 and 2, respectively. The α columns contain the average number of labels placed, the CPU columns contain the average number of CPU seconds that were needed to find the reported independent sets, excluding the time needed to solve the LP relaxations. Finally, the $\alpha(G)$ column contains the average optimal number of labels that can be placed in the test instances.

It is clear from Table 2 that all rounding heuristics are very fast in practice, provided one already has solved the LP relaxation. Their running time is negligible for instances with up to 950 cities. The minimum regret rounding algorithms are the best in terms of solution quality. Also, the mod-k cuts do improve the solution quality slightly, at the cost of a tremendous increase in running time. Using 2-opt after rounding outperforms using 1-opt after rounding. The minimum regret heuristics with parameters $t = 2$ and $t = 1$ both perform equally well. The average number of labels placed by the minimum regret heuristics is within four tenth of the average number of labels in the optimal solutions for all choices of n, and strictly within one tenth for all choices of $n \leq 850$, with the parameter $t = 2$.

	With Mod-k cuts									Without Mod-k cuts									
	LP	L1		L2		R1		R2		LP	L1		L2		R1		R2		
n	CPU	α	CPU	α	CPU	α	CPU	α	CPU	CPU	α	CPU	α	CPU	α	CPU	α	CPU	$\alpha(G)$
100	1.1	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	0.1	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
150	2.9	149.9	0.0	149.9	0.0	149.9	0.0	149.9	0.0	0.1	149.9	0.0	149.9	0.0	149.9	0.0	149.9	0.0	149.9
200	5.3	199.9	0.0	199.9	0.0	199.9	0.0	199.9	0.0	0.2	199.9	0.0	199.9	0.0	199.9	0.0	199.9	0.0	199.9
250	9.4	249.6	0.0	249.6	0.0	249.6	0.0	249.6	0.0	0.2	249.6	0.0	249.6	0.0	249.6	0.0	249.6	0.0	249.6
300	15.2	299.2	0.0	299.2	0.0	299.2	0.0	299.2	0.0	0.3	299.2	0.0	299.2	0.0	299.2	0.0	299.2	0.0	299.2
350	20.9	348.5	0.0	348.5	0.0	348.5	0.0	348.5	0.0	0.4	348.5	0.0	348.5	0.0	348.5	0.0	348.5	0.0	348.5
400	32.1	397.7	0.0	397.7	0.0	397.7	0.0	397.7	0.0	0.5	397.6	0.0	397.7	0.0	397.7	0.0	397.7	0.0	397.7
450	36.9	445.1	0.0	445.1	0.0	445.1	0.0	445.1	0.0	0.6	445.1	0.0	445.1	0.0	445.1	0.0	445.1	0.0	445.1
500	61.0	492.9	0.0	492.9	0.0	492.9	0.0	492.9	0.0	0.8	492.9	0.0	492.9	0.0	492.9	0.0	492.9	0.0	492.9
550	87.3	539.6	0.0	539.7	0.0	539.7	0.0	539.7	0.0	0.9	539.6	0.0	539.7	0.0	539.7	0.0	539.7	0.0	539.7
600	163.0	582.9	0.0	582.9	0.0	582.9	0.0	582.9	0.0	1.2	582.7	0.0	582.9	0.0	582.9	0.0	582.9	0.0	582.9
650	299.2	627.2	0.0	627.2	0.0	627.3	0.0	627.3	0.0	1.6	626.4	0.0	627.1	0.0	627.2	0.0	627.2	0.0	627.3
700	601.8	670.2	0.0	670.2	0.0	670.4	0.0	670.4	0.0	2.6	668.8	0.0	669.8	0.0	670.2	0.0	670.3	0.0	670.4
750	920.4	708.8	0.0	709.0	0.0	709.2	0.0	709.2	0.0	4.3	707.4	0.0	708.6	0.0	709.2	0.0	709.0	0.0	709.2
800	1273.2	748.1	0.0	748.9	0.0	749.2	0.0	749.3	0.0	7.6	746.3	0.0	748.0	0.0	749.1	0.0	749.1	0.1	749.3
850	2348.7	784.7	0.0	785.6	0.0	785.9	0.0	786.0	0.0	14.4	782.2	0.0	784.8	0.1	785.8	0.0	785.9	0.1	786.0
900	3249.4	815.6	0.0	817.4	0.0	818.1	0.0	818.1	0.0	17.9	812.8	0.0	816.2	0.1	817.9	0.0	817.9	0.1	818.2
950	4258.2	849.0	0.0	851.0	0.0	852.3	0.0	852.2	0.0	25.6	845.1	0.0	849.6	0.1	851.6	0.0	851.8	0.2	852.6

Table 2: LP Based Rounding Algorithms

	SA		DS		O1		O2		TS		
n	α	CPU	α	CPU	α	CPU	α	CPU	α	CPU	$\alpha(G)$
100	100.0	0.1	100.0	0.0	97.5	0.0	100.0	0.0	100.0	0.0	100.0
150	149.9	0.2	149.9	0.1	144.2	0.0	149.5	0.0	149.9	0.1	149.9
200	199.9	0.3	199.9	0.1	189.0	0.0	198.8	0.0	199.8	0.1	199.9
250	249.6	0.4	249.6	0.2	232.8	0.0	247.6	0.0	249.5	0.2	249.6
300	299.0	0.6	299.2	0.3	274.9	0.0	294.3	0.0	298.9	0.3	299.2
350	348.3	0.8	348.4	0.6	314.1	0.0	340.1	0.0	348.0	0.4	348.5
400	397.3	1.0	397.6	0.8	354.2	0.0	386.0	0.0	396.6	0.5	397.7
450	444.5	1.3	445.0	0.9	388.7	0.0	428.4	0.0	444.1	0.7	445.1
500	491.7	1.7	492.5	1.1	422.8	0.0	469.8	0.0	490.9	0.9	492.9
550	538.2	2.3	539.4	1.3	456.9	0.0	510.1	0.0	537.6	1.1	539.7
600	580.3	2.7	581.8	1.5	486.0	0.0	546.6	0.0	579.4	1.5	582.9
650	623.8	3.0	625.9	1.7	518.4	0.0	584.6	0.1	624.2	1.9	627.3
700	665.9	3.4	668.9	1.9	545.3	0.0	619.4	0.1	666.2	2.4	670.4
750	703.1	3.8	706.1	2.2	571.8	0.0	651.6	0.1	704.6	3.2	709.2
800	740.4	4.2	746.0	2.4	600.9	0.0	684.1	0.1	743.4	3.6	749.3
850	775.3	4.6	780.9	2.7	622.9	0.0	712.9	0.1	779.0	4.4	786.0
900	804.5	5.1	812.3	3.0	639.9	0.0	738.5	0.1	810.5	5.2	818.2
950	834.4	5.6	845.9	3.3	664.7	0.0	762.7	0.1	844.7	6.6	852.6

Table 3: Local Search Algorithms

The average number of labels placed by our local search algorithms and the corresponding running times are reported in Table 3. The SA and DS columns refer to simulated annealing and diversified neighbourhood search, respectively, applied to a map labelling formulation. The columns O1 and O2 refer to the iterative improvement algorithms that were presented in Section 4.3. These algorithms use the 1-opt and 2-opt neighbourhoods, respectively, starting from $\mathbf{0}$. The column TS refers to the tabu search algorithm applied to an independent set formulation.

The 1-opt and 2-opt algorithms are the fastest but they produce inferior solutions when started from zero. The simulated annealing algorithm works reasonably fast and produces solutions of reasonable quality. However, it is outperformed by the diversified neighbourhood search and the tabu search. The diversified neighbourhood search seems to be the best heuristic among the ones that are not based on LP relaxations, being twice as fast as the tabu search and producing solutions with more labelled cities.

When comparing the minimum regret rounding with parameter $t = 2$ without the mod-k cuts with the diversified neighbourhood search, we notice that the minimum regret rounding is faster and produces better solutions for instances with up to 650 cities. For larger instances minimum regret rounding is slower, but still produces a higher solution quality.

6.2 Optimisation Algorithms

We tested our cutting plane/branch-and-bound algorithm and our branch-and-cut algorithm for the map labelling problem on the same set of instances that we used to evaluate the performance of our heuristics. The average running times and branch-and-bound tree sizes

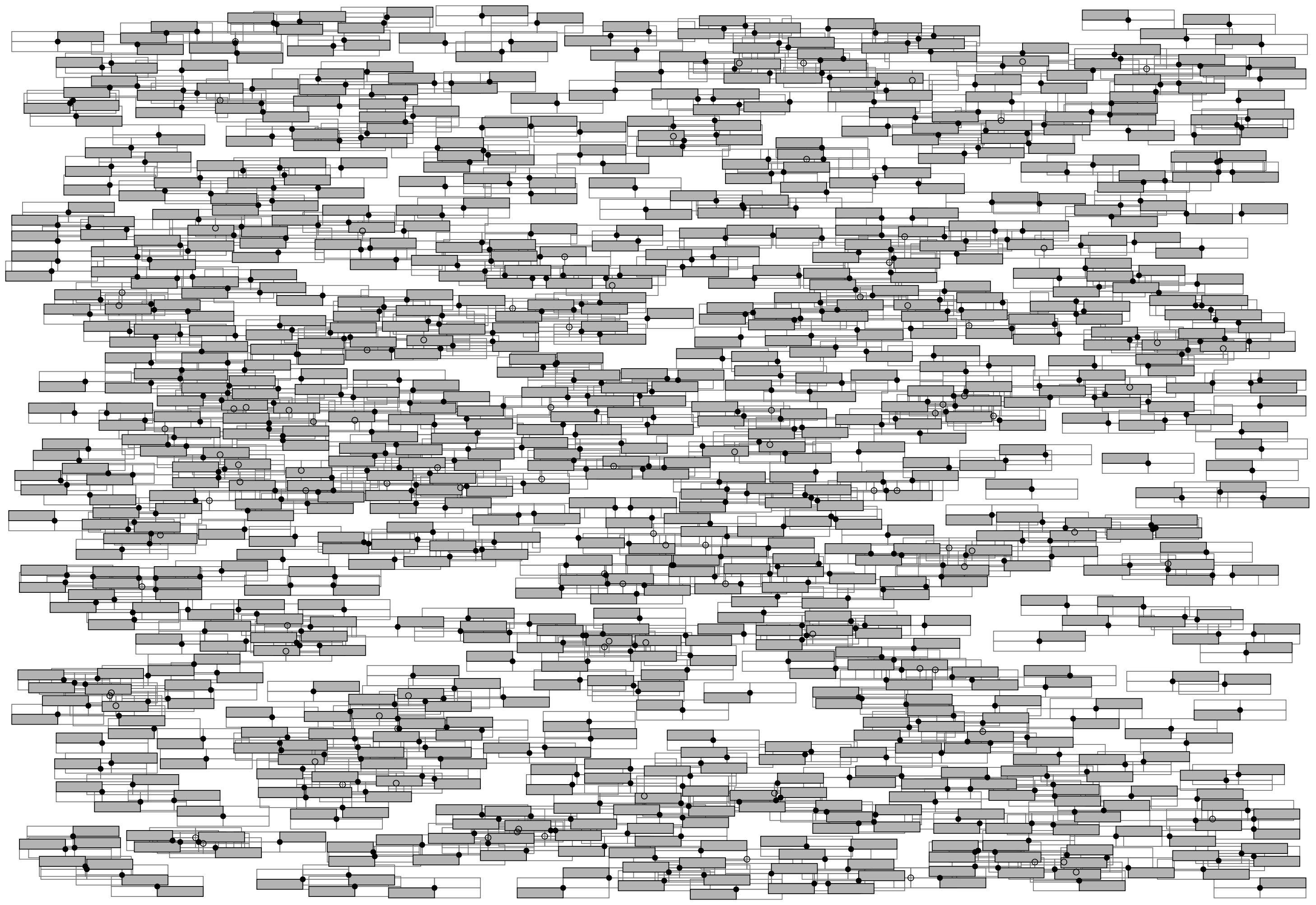


Figure 5: An Optimal Solution to a Map Labelling Problem on 950 Cities

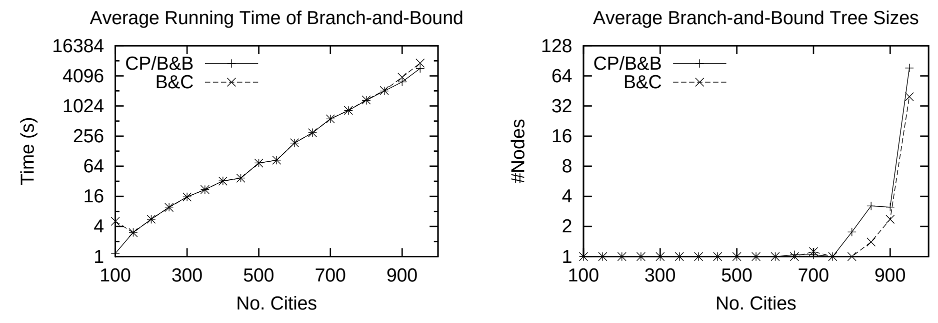


Figure 6: Average Branch-and-Bound Running Time and Tree Sizes

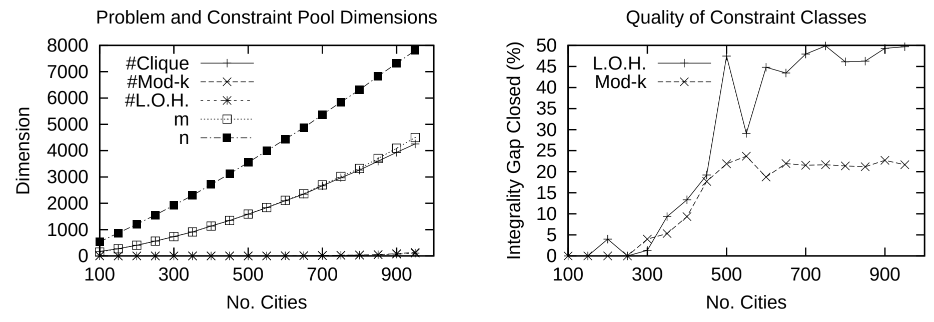


Figure 7: Problem Statistics and Constraint Performance

of these algorithms can be found in Figure 6. For the branch-and-cut algorithm, problem statistics and the gaps closed by the various valid inequalities are depicted in Figure 7, run time profiling data is given in Figure 8, and finally the average number of variables set by pre-processing, reduced cost, and logical implication sub-routines and the average number of the mod-k cuts found for each value of k are given in Figure 9.

The first thing to note from Figure 6 is that the branch-and-bound tree does not start to grow until the number of cities to label is approximately 750, and then it starts to grow at a tremendous rate. There is an obvious explanation for this, namely, that we do increase the number of cities that we label but we do not increase the map, so the problems become more dense, and 750 cities seems to be a critical number. Taking this into account, the steady exponential growth of the running time needed seems to be remarkable. The exponential growth of the running time demonstrates that the cutting plane algorithm itself behaves exponentially.

For the purpose of testing the influence of SRS we did some experiments in which we scaled the size of the map in order to keep the density of the graphs constant. We discuss these experiments in detail later in this section. What is important here is to note that they also show that the tremendous growth does not occur if the density is kept constant; instead we see a more modest exponential growth (see Figure 10).

From Figure 7 it is clear that although the number of lifted odd hole inequalities is small, their contribution to the quality of the LP formulation is significant. The same holds for mod-k cuts, although to a lesser extent. From Figure 8 it is clear that most of the running

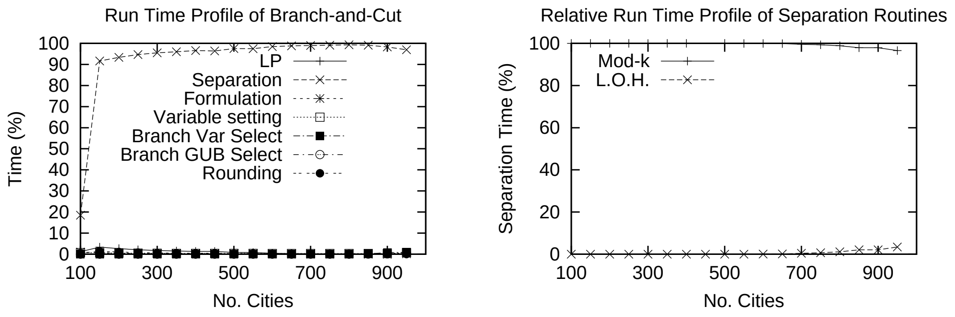


Figure 8: Run Time Profiling Data

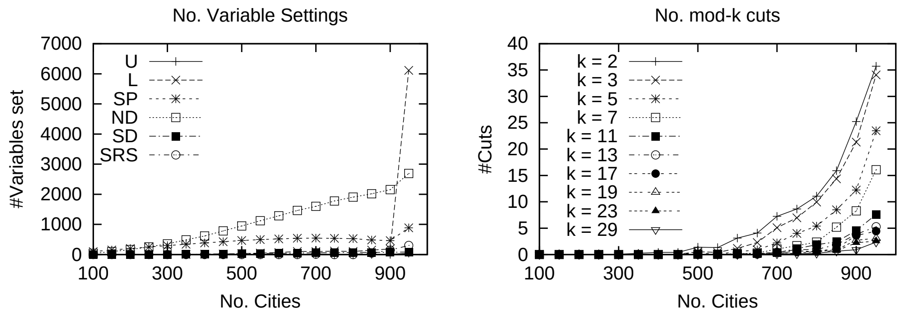


Figure 9: Variable Setting and Mod-k Statistics

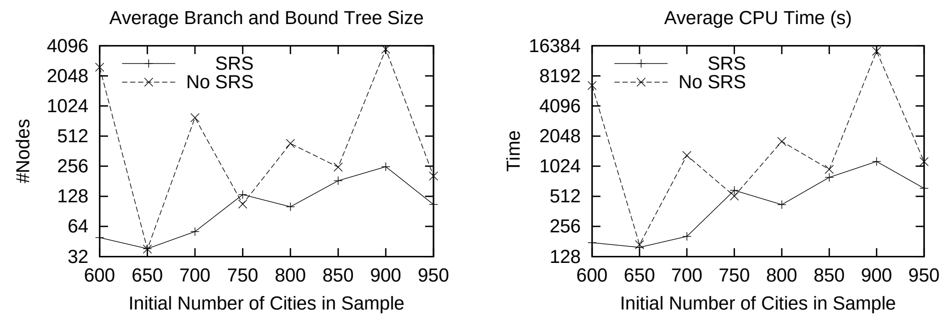


Figure 10: Influence of SRS on the Behaviour of the Branch-and-Cut Algorithm

time is spent in the separation of mod-k cuts. We feel that this is due to the fact that we do not exploit the similarity between consecutive LP solutions in the branch-and-cut algorithm as we do with the lifted odd hole separation as discussed in Section 5.9.

We end the discussion of our optimisation algorithm by considering Figure 9. It shows that the average number of times that a variable can be set. Here U, L, SP, D, SD, and SRS indicate setting by reduced cost to upper bound, to lower bound, setting of simplicial nodes, dominated nodes, set dominated nodes and SRS, respectively. The spike in the graph of setting variables to their lower bound is caused by a single run in which this could be applied frequently in the lower parts of the tree. SRS only starts to play a role for the larger instances.

6.2.1 SRS

To evaluate the influence that SRS can have on the performance of our algorithm, we conducted a second set of experiments. These experiments were conducted with an earlier version of our code that did not feature the strengthened reduced cost setting from Section 5.4, the pre-processing from Section 3.4, the complete GUB formulation of Section 3.2 and the separation of mod-k inequalities. In these experiments the density of the problems was kept constant by making the map size a function of the number of cities. For a problem with n cities, we use a map of size $\lfloor 792/\sqrt{750/n} \rfloor \times \lfloor 612/\sqrt{750/n} \rfloor$. For each $n \in \{600, 650, \ldots, 900, 950\}$ we randomly generated 50 maps. From each generated map, we selected its largest connected component and used that as the input for our algorithm both with and without SRS. Figure 10 shows the average branch-and-bound tree sizes and running times for these experiments. The reported branch-and-bound tree sizes for the case with SRS includes the nodes in branch-and-bound trees of recursive calls. Considering the logarithmic scale on the vertical axis, the potential savings from using SRS are clear.

References

[1] P.K. Agarwal, M. van Kreveld, and S. Suri. Label placement by maximum independent set in rectangles. *Computational Geometry: Theory and Applications*, 11:209–218, 1998.

[2] E.M.L. Beale and J.A. Tomlin. Special facilities in a general mathematical programming system for nonconvex problems using ordered sets of variables. In J. Lawrence, editor,

Proceedings of the Fifth International Conference on Operations Research, pages 447–454. Tavistock Publications, London, 1970.

[3] H.L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Mathematical Foundations of Computer Science 1997, Proceedings of the 22nd International Symposium, MFCS '97*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer-Verlag, Berlin, 1997.

[4] A. Caprara, M. Fischetti, and A. Letchford. On the separation of maximally violated mod-k cuts. In G. Cornuéjols, R.E. Burkard, and G.J. Woeginger, editors, *Integer Programming and Combinatorial Optimization, Proceedings of the 7th International IPCO Conference*, volume 1610 of *Lecture Notes in Computer Science*, pages 87–98. Springer-Verlag, Berlin, 1999.

[5] A. Caprara, M. Fischetti, and A. Letchford. On the separation of maximally violated mod-k cuts. *Mathematical Programming*, 87:37–56, 2000.

[6] J. Christensen, J. Marks, and S. Shieber. Placing text labels on maps and diagrams. In P. Heckbert, editor, *Graphics Gems IV*, pages 497–504. Academic Press, Boston, Massachusetts, 1994.

[7] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.

[8] R.G. Cromley. An LP relaxation procedure for annotating point features using interactive graphics. In *AUTO-CARTO 7, Proceedings, Digital Representations of Spatial Knowledge*, pages 127–132, 1985.

[9] H. Crowder, E.L. Johnson, and M.W. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[10] G.B. Dantzig and M.N. Thapa. *Linear Programming, 1: Introduction.* Springer-Verlag, Berlin, 1997.

[11] L. Danzer and B. Grünbaum. Intersection properties of boxes in $\mathbb{R}^d$. *Combinatorica*, 2(3):237–246, 1982.

[12] S. van Dijk, D. Thierens, and M. de Berg. Robust genetic algorithms for high quality map labeling. Technical Report UU-CS-1998-41, Department of Computer Science, Utrecht University, 1998.

[13] S. van Dijk, D. Thierens, and M. de Berg. On the design of genetic algorithms for geographical applications. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '99)*, pages 188–195. Morgan Kaufmann, San Fransisco, California, 1999.

[14] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] B. de Fluiter. *Algorithms for Graphs of Small Treewidth.* PhD thesis, Department of Computer Science, Utrecht University, Utrecht, 1997.

[16] M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry*, pages 281–288, 1991.

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

[18] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, Berlin, 1988.

[19] A. Hertz, E. Taillard, and D. de Werra. Tabu search. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 121–136. John Wiley & Sons Ltd, Chichester, 1997.

[20] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of Algorithms*, 4:310–323, 1983.

[21] E. Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975.

[22] K.G. Kakoulis and I.G. Tollis. A unified approach to labeling graphical features. In *Proceedings of the 14th Annual ACM Symposium on Computational Geometry*, pages 347–356, 1998.

[23] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[24] G.W. Klau and P. Mutzel. Optimal labelling of point features in the slider model. In D-Z. Du, P. Eades, and X. Lin, editors, *Computing and Combinatorics, Proceedings of the 6th Annual International Conference (COCOON'2000)*, Lecture Notes in Computer Science, 2000. To appear.

[25] M. van Kreveld, T.W. Strijk, and A. Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999.

[26] L. Kučera, K. Mehlhorn, B. Preis, and E. Schwarzenecker. Exact algorithms for a geometric packing problem. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *STACS 93, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 317–322. Springer-Verlag, Berlin, 1993.

[27] J.T. Linderoth and M.W.P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2), 1999.

[28] G.L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operations Research Society*, 43(5):443–457, 1992.

[29] G.L. Nemhauser and L.E. Trotter, Jr. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6:48–61, 1974.

[30] G.L. Nemhauser and L.E. Trotter, Jr. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[31] G. Neyer. Map labeling with application to graph labeling. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models.* Teubner, Stuttgart, 2000. To appear.

[32] M.W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[33] M.W. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[34] Z. Qin, A. Wolff, Y. Xu, and B. Zhu. New algorithms for two-label point labeling. In M. Paterson, editor, *Algorithms — ESA '00, Proceedings of the 8th Annual European Symposium*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000. To appear.

[35] A.M. Verweij. *Selected Applications of Integer Programming: A Computational Study.* PhD thesis, Department of Computer Science, Utrecht University, Utrecht, 2000.

[36] A.M. Verweij and K.I. Aardal. An optimisation algorithm for maximum independent set with applications in map labelling. In J. Nešetřil, editor, *Algorithms — ESA '99, Proceedings of the 7th Annual European Symposium*, volume 1643 of *Lecture Notes in Computer Science*, pages 426–437. Springer-Verlag, Berlin, 1999.

[37] F. Wagner and A. Wolff. An efficient and effective approximation algorithm for the map labeling problem. In P. Spirakis, editor, *Algorithms — ESA '95, Proceedings of the Third Annual European Symposium*, volume 979 of *Lecture Notes in Computer Science*, pages 420–433. Springer-Verlag, Berlin, 1995.

[38] F. Wagner, A. Wolff, V. Kapoor, and T. Strijk. Three rules suffice for good label placement. *Algorithmica Special Issue on GIS*, 2000. To appear.

[39] A. Wolff and T.W. Strijk. The map labeling bibliography. `http://www.math-inf.uni-greiswald.de/map-labeling/bibliography`.

[40] L.A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

[41] L.A. Wolsey. Facets and strong valid inequalities for integer programs. *Operations Research*, 24:367–372, 1975.

[42] L.A. Wolsey. *Integer Programming.* John Wiley and Sons, New York, 1998.

[43] Pinhas Yoeli. The logic of automated map lettering. *The Cartographic Journal*, 9:99–108, 1972.

[44] S. Zoraster. Integer programming applied to the map label placement problem. *Cartographica*, 23(3):16–27, 1986.

[45] S. Zoraster. The solution of large 0-1 integer programming problems encountered in automated cartography. *Operations Research*, 38(5):752–759, 1990.

[46] P.J. Zwaneveld, L.G. Kroon, and C.P.M. van Hoesel. Routing trains through a railway station based on a node packing model. Technical Report RM/97/030, Maastricht University FdEWB, Maastricht, 1997. To appear in European Journal on Operational Research.