Chapter 4

Maximum Independent Set

4.1 Introduction

Given an undirected graph $G = (V, E)$, an independent set in $G$ is a subset $S \subseteq V$ such that no two nodes in $S$ are connected in $G$, i.e. for all edges $\{u, v\} \in E$ we have that $\{u, v\} \not\subseteq S$. The maximum independent set problem is the problem of finding an independent set of maximum cardinality in a given graph. Independent sets are sometimes called stable sets, vertex packings, or node packings.

The complement of $G$ has the same node set, and is denoted by $\bar{G} = (V, \bar{E})$ where $\{u, v\} \in \bar{E}$ if and only if $\{u, v\} \notin E$. A clique in $G$ is a set of nodes $S \subseteq V$ such that for all $\{u, v\} \in S$, $\{u, v\} \in E$. A subset $S \subseteq V$ is an independent set in $G$ if and only if $S$ is a clique in $\bar{G}$. The maximum clique problem is the problem of finding a clique of maximum cardinality in a given graph. Due to the equivalence between independent sets in a graph and cliques in its complement, any algorithm for finding a maximum independent set also gives an algorithm for finding a maximum clique, and vice versa.

Also, independent set problems are related to so-called set packing problems, in which one is given a ground set and a collection of subsets of this ground set, and one has to find a sub-collection of the given subsets that are disjoint and maximal with respect to some objective function. The reduction from the set packing problem to the independent set problem uses a conflict graph that is similar in spirit as the one that is used in Section 3.2.3. For more details on set packing and its relation to independent set problems we refer to Borringer [20] and Marchand, Martin, Weismantel, and Wolsey [82].

4.1.1 Motivation, Literature, and Chapter Outline

Several optimisation algorithms for finding maximum independent sets [101, 106, 81, 80, 85, 126, 48] and maximum cliques [12, 6, 7, 26, 13] have been reported on in the literature. Moreover, several real-life problems can be formulated as independent set problems [100, 2, 117, 129]. The polyhedral structure of the
independent set polytope has been reported on by various authors [90, 86, 87, 112].

Our interest in independent set problems originates from the fact that map labelling problems, which are the subject of Chapter 5, can be reformulated as independent set problems. We develop a number of powerful techniques for the problem that are meant to work well on sparse large-size instances. We do not claim that our independent set code is the best code available for solving independent set problems. It will, however, provide us with the best optimisation algorithm for map labelling problems that is currently known.

In this chapter we explain all the ingredients of our branch-and-cut algorithm for maximum independent set. Other LP-based branch-and-bound algorithms for maximum independent set reported on in the literature are the cutting plane/branch-and-bound algorithm by Nemhauser and Sigismondi [85] and the branch-and-cut algorithm of Rossi and Smriglio [101]. Our approach differs from Nemhauser and Sigismondi in that we use a branch-and-cut approach with a different scheme for lifting odd hole inequalities, different primal heuristics, and stronger implications and pre-processing techniques.

We will first introduce several integer programming formulations for the maximum independent set problem in Section 4.1.2, and recall some basic notions in Section 4.1.3. Using these concepts, we will discuss LP rounding algorithms for the problem in Section 4.2, and then present our branch-and-cut algorithm in Section 4.3. In Section 4.4 we evaluate the performance of our independent set code on a class of randomly generated test instances that is commonly used for generating test instances in the literature.

### 4.1.2 Integer Programming Formulations

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

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

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

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

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

$$\max\{x(V) \mid x \in P_E \cap \{0, 1\}^V\}. \quad (4.1)$$

We call this formulation the *edge formulation* of the maximum 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
\[ x(C) \leq 1 \] (4.2)
is a valid inequality for \( P_{IS} \) for every clique \( C \). If \( C \) is maximal, we call (4.2) a maximal clique inequality. Padberg \cite{padberg1990} 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
\[ P_{\mathcal{C}} = \{ x \in \mathbb{R}^V \mid x(C) \leq 1 \text{ for all } C \in \mathcal{C}, x \geq 0 \}. \]
Then, problem (4.1) can be reformulated as
\[ \max\{ x(V) \mid x \in P_{\mathcal{C}} \cap \{0, 1\}^V \}. \] (4.3)
We call this formulation the clique formulation of the maximum independent set problem.

Finally, we can rewrite the clique formulation by introducing the slack variables \( s \in \{0, 1\}^\mathcal{C} \). This has as advantage that we can use the clique inequalities for GUB branching. Let
\[ P_{\mathcal{C}}^{GUB} = \{ (x, s) \in \mathbb{R}^{V \cup \mathcal{C}} \mid x(C) + s_C = 1 \text{ for all } C \in \mathcal{C}, x \geq 0, s \geq 0 \}. \]
This leads to the following formulation of the maximum independent set problem, that we call the GUB formulation:
\[ \max\{ x(V) \mid (x, s) \in P_{\mathcal{C}}^{GUB} \cap \{0, 1\}^{V \cup \mathcal{C}} \}. \] (4.4)
In our branch-and-cut algorithm we use the GUB formulation.

The sets \( P_E, P_{\mathcal{C}}, \) and \( P_{\mathcal{C}}^{GUB} \) are related as follows:

**Proposition 4.1.** 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, \( P_{IS} \subseteq \{ x_V \mid x \in P_{\mathcal{C}}^{GUB} \} = P_{\mathcal{C}} \subseteq P_E \).

Some of the sub-routines we use in our branch-and-cut algorithm assume as input a vector \( x \in P_E \). By Proposition 4.1 we can apply them to \( x_V \), where \( x \in P_{\mathcal{C}}^{GUB} \) is obtained by maximising over the GUB formulation.

### 4.1.3 Some Basic Notions from Graph Theory

To present the algorithms in this chapter, we will use some basic notions from graph theory. Let \( G = (V, E) \) be a graph. 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.,

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

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

Let \( G = (V, E) \) be a graph. The length of a path \( P \) in \( G \) is the number of edges \( |E(P)| \) in \( P \). The diameter of \( G \), denoted by \( \text{diam}(G) \), is the maximum length of a shortest path connecting two nodes in \( G \), i.e.,

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

For \( S \subseteq V \), the graph induced by \( S \), denoted by \( G[S] \), is the graph with node set \( S \) and edge set \( E(S) \). A connected component in \( G \) is a connected induced graph \( G[S] \) that is maximal with respect to inclusion of nodes, where \( S \subseteq V \).

### 4.2 Heuristics for Maximum Independent Set

Here we consider computationally cheap heuristics for finding hopefully large independent sets, starting from an LP solution in \( P_E \). These heuristics are reported on by Strijk, Verweij and Aardal [107]. In Section 4.2.1 and 4.2.2 we discuss LP rounding heuristics, and in Section 4.2.3 we discuss simple local search neighbourhoods that can be used to increase the cardinality of the independent sets produced by the rounding heuristics.

#### 4.2.1 Simple LP Rounding

Suppose we are given a point \( x \in P_E \). Let \( S \) be the set of nodes that correspond to \( x \)-variables with a value strictly greater than one half, i.e., \( S = \{v \in V \mid x_v > \frac{1}{2}\} \). Because \( x \in P_E \), 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 \( x_S \) up to 1, and \( x_{V \setminus S} \) down to 0 to obtain the vector \( x' = \chi_S \in P_E \cap \{0, 1\}^V \), 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 \( x \in P_E \) with \( x_v = \frac{1}{2} \) for all \( v \in V \) is rounded to 0, corresponding to the empty independent set. On the other hand, the algorithm is capable of producing any optimal solution. For example, if \( x = (1 - \epsilon)\chi_S + \epsilon \chi_{V \setminus S} \) for some maximum independent set \( S^* \) and any \( \epsilon \) such that \( 0 \leq \epsilon < \frac{1}{2} \), then the algorithm produces \( \chi_{S^*} \) as an answer, an optimal solution.

Nemhauser and Trotter [86] have shown that if \( x \) is obtained by maximising any objective function over \( P_E \), then \( x \) 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 \( x \) with value 1.
4.2. **Heuristics for Maximum Independent Set**

4.2.2 Minimum Regret Rounding

Denote by $\mathcal{I}$ the collection of all independent sets in $G$. Suppose we are given a vector $x \in P_E$ with some fractional components. We are going to round $x$ by repeatedly choosing an independent set $I \in \mathcal{I}$ with $0 < x_I < 1$, rounding $x_I$ up to 1, and $x_{N(I)}$ down to 0, until $x$ is integral. Let $I \in \mathcal{I}$. The rounding operation above defines a function $f$ that maps $x$ to a vector $x' \in P_E$ using $I$:

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

We say that $f$ rounds up $x_I$.

**Lemma 4.2.** Let $x \in P_E$ and $I \in \mathcal{I}$. Then $f(x, I) \in P_E$.

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

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

$$r(x, I) = \sum_{v \in V} x_v - \sum_{v \in V} (f(x, I))_v = \sum_{v \in V} (x_v - x'_v) = \sum_{v \in N(I)} x_v - |I|$$

Because we will later apply $f$ to a vector $x \in P_E$ that is optimal with respect to $x(V)$, we have that $x(V) \geq x'(V)$, so $r(x, I) \geq 0$ is the degradation of the objective function in that case. Since we do not like degradation, we call $r(x, I)$ the *regret* we have when rounding $x_I$ to 1.

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

$$r(x, I) = x(N(I)) - |I|$$

$$= x(N(I_1)) + x(N(I_2)) - x(N(I_1) \cap N(I_2)) - |I_1| - |I_2|$$

$$= r(x, I_1) + r(x, I_2) - x(N(I_1) \cap N(I_2)).$$

This shows that if

$$x(N(I_1) \cap N(I_2)) = 0,$$  \hspace{1cm} (4.5)

then the regret of rounding $x_I$ to 1 is the same as the combined regret of rounding $x_{I_1}$ and $x_{I_2}$ to 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 (4.5). This is the case if and only if the graph induced by the support of \( x_{N_1(I)} \) is connected.

If we choose \( I \) in such a way that \( 0 < x_I < 1 \), then \( f(x, I) \) has at least \( |I| \) fewer fractional components than \( x \). We will use this to define a greedy rounding algorithm as follows. The algorithm has as input a vector \( x \in P_E \) and an integer \( t > 0 \) and repeatedly replaces \( x \) by \( f(x, I) \) for some set \( I \), rounding \( x_I \) to 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

\[
|I| = k, \quad (4.6)
\]
\[
0 < x_I < 1, \quad \text{and} \quad (4.7)
\]
\[
G[\text{supp}(x_{N_1(I)})] \quad \text{is connected.} \quad (4.8)
\]

During phase \( k \), the next set \( I \in \mathcal{I} \) is chosen so as to minimise the regret \( r(x, I) \) 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 after the choice of \( I \). Note that at any time during the algorithm, \( x_F \) is an optimal LP solution to the maximum independent set problem in \( G[F] \) if the original vector \( x \) was one in \( G \), where \( F = \{ v \in V \mid 0 < x_v < 1 \} \) is the fractional support of \( x \). It follows that the value \( x(V) \) 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 (4.6)–(4.8), where the priority of set \( I \) is the value of \( r(x, I) \). In each iteration, we extract a set \( I \) from \( Q \) with minimum regret. If \( x_I \) has integral components or if the graph induced by the support of \( x_{N_1(I)} \) is not connected, then we proceed with the next iteration. Otherwise we update \( Q \). This is done by decreasing the priorities of all \( I' \in \mathcal{I} \) with \( N_1(I') \cap N_1(I) \neq \emptyset \) by \( x(N_1(I') \cap N_1(I)) \). We replace our current vector \( x \) by \( f(x, I) \) and proceed with the next iteration. Phase \( k \) terminates when \( Q \) is empty.

**Lemma 4.3.** Let \( F \subseteq V \) be the fractional support of \( x \) 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 \( G[F] \),

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

**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 \( I \subseteq F' \). 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 (4.6)–(4.8). This contradicts the termination of phase \(k\).

\[\square\]

**Theorem 4.4.** Let \(x = \chi^S\) be the vector returned by the minimum regret rounding 
algorithm with input \(x'\) and some \(t > 0\), where \(x' \in P_E\). Then \(S\) is an 
independent set.

**Proof.** Since \(x\) is obtained from \(x'\) by iteratively applying \(f\), we have \(x \in P_E\) 
by Lemma 4.2. From Lemma 4.3, we have that upon termination of phase \(k = 1\), the diameter of each 
connected component of the fractional support of \(x\) is strictly less than 0. This implies that the graph induced by the fractional 
support is empty, hence \(x\) 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 (4.8) 
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 \(O(|V|)\) 
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 \(O(|V|)\) time, the total time spent in decreasing priorities is at most

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

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 \(N_1(I)\) for at most \(|\delta(v)|\{-\}|\) 
possible choices of \(I \subseteq X\) 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
has to be updated each time we set \( v \) for any fixed value of \( t \). Rounding algorithm in a brute-force fashion will yield a polynomial algorithm in \( n \) for fixed \( k \). Therefore, a 1-optimal solution can be computed in \( O(W^k) \) 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 \( t \).

### 4.2.3 Iterative Improvement

Here we consider local search neighbourhoods and iterative improvement 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 \( N_k(S) \), 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 \( S' \in N_k(S) \), then \( S' = (S \setminus U) \cup W \), for some \( U \) and \( W \) with \( U \subseteq S \), \( |U| = k - 1 \), \( W \subseteq (V \setminus S) \cup U \), and \( |W| = k \). The \( k \)-opt neighbourhood is undefined if \(|S| < k - 1 \). Because we do not require that \( U \cap W = \emptyset \) we have that, if \( N_k(S) \) is defined, then \( N_j(S) \subseteq N_k(S) \) 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 minimum regret rounding algorithms produce \( k \)-optimal independent sets for any \( k \geq 1 \). This motivates our interest in the \( k \)-opt neighbourhoods of independent sets.

**Proposition 4.5.** 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 \( N_l(S) \subseteq N_k(S) \) 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 \( S' \in N_k(S) \) 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 us 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 4.6.** Let \( S \) be a \((k-1)\)-optimal independent set for some \( k > 1 \), and \( S' \in N_k(S) \). 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 \( S' \in N_k(S) \) 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.

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.

### 4.3 A Branch-and-Cut Algorithm

Recall the definitions of \( P_E, P_C, P_{GUB} \), and \( P_{IS} \) from Section 4.1.2. To solve independent set problems, we use the standard branch-and-cut algorithm of Section 3.3 starting with \( P_{GUB} \) in the root node for some collection \( C \) of maximal cliques that covers all edges in \( E \). The formulation is improved by applying some pre-processing, which is discussed in Section 4.3.1. We use the following valid inequalities for \( P_{IS} \) to strengthen our formulation of the maximum independent set problem: maximal clique inequalities, lifted odd hole inequalities, and mod-\( k \) inequalities. These are the subjects of Sections 4.3.2, 4.3.3, and 4.3.4, respectively. In addition, we use the strengthened conditions for setting variables based on reduced cost from Section 3.2.3. Whenever we set variables either by reduced cost or by branching, we try to set more variables by using logical implications. These are described in Section 4.3.5. In each node of the branch-and-bound tree, we may use the rounding algorithms of Section 4.2 to find integer solutions.

#### 4.3.1 Preprocessing Techniques

In this section we consider ways to recognise nodes that belong to a maximum independent set, or that do not belong to any maximum independent set at all. Nodes that belong to a maximum independent set can be set to one, and nodes that do not belong to any maximum independent set can be set to zero before starting the branch-and-cut algorithm.
CHAPTER 4. MAXIMUM INDEPENDENT SET

58

\[ v \text{ is node-dominated by } u \]

\[ \forall \quad v \text{ is set-dominated by } N(v) \]

Figure 4.1: Node- and Set-Dominance Criteria

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 [87, Theorem 1].

**Proposition 4.7.** 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 \( I \).

**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 \( I \).

A *simplicial node* is a node whose neighbours form a clique. As a corollary to Proposition 4.7 we have that 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 4.1, are special cases of the dominance criteria reported by Zwaneveld, Kroon and van Hoesel [129].

**Proposition 4.8.** (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. 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 \( N_1(u) \subseteq N_1(v) \),
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 4.9 we can set $x_v$ to zero if $v$ is set-dominated.
4.3.2 Maximal Clique inequalities

Let \( x \in P_E \) be a fractional solution to the maximum independent set problem. Our procedure to identify violated maximal clique inequalities is a combination of two greedy heuristics described by Nemhauser and Sigismondi [85].

We look for a violated maximal clique inequality starting from each node \( v \) in the fractional support from \( x \). For any node \( v \) this is done as follows. We initialise \( C := \{v\} \) and set \( N := N_1(v) \setminus v \). We maintain as invariant that \( N \) contains all nodes in \( G \) that are adjacent to each node in \( C \). As long as \( N \) is not empty, we extract a node \( u \) from \( N \), add this node to \( C \), and remove all nodes from \( N \) that are not adjacent to \( u \). We use the following two criteria to choose \( u \), each defining its own greedy heuristic:

\[
  u = \arg \min_{u \in N} \{|x_u - \frac{1}{2}|\}, \quad \text{and} \quad u = \arg \max_{u \in N} \{x_u\}.
\]

When upon termination \( N \) is empty, \( C \) is a maximal clique by our invariant.

Because \(|N| \leq \delta(v)\), the procedure that looks for a maximal clique starting from a given node can be implemented in \( O(\delta(v)^2) \) (using an incidence matrix to determine in \( O(1) \) time whether two nodes in \( G \) are adjacent). As a consequence, the whole procedure can be made to work in time

\[
\sum_{v \in V} O(\delta(v)^2) \leq |V| \sum_{v \in V} O(\delta(v)) = O(|V||E|).
\]

4.3.3 Lifted Odd Hole Inequalities

Let \( x \in P_E \) be a fractional solution. A hole in \( G \) is a cycle \( G \) without chords. 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

\[
  x(V(H)) \leq \lfloor |V(H)|/2 \rfloor.
\]

It was shown by Padberg [90] that the odd hole inequality is valid for \( P_{IS} \), and facet-defining for \( P_{IS} \cap \{x \mid x_{V \setminus V(H)} = 0\} \). Given an odd hole \( H \), a lifted odd hole inequality is of the form

\[
  x(V(H)) + \sum_{v \in V \setminus V(H)} \alpha_v x_v \leq \lfloor |V(H)|/2 \rfloor \quad (4.9)
\]

for some suitable vector \( \alpha \in \mathbb{R}^V \). We compute values \( \alpha \in \mathbb{N}^V \) using sequential lifting (Theorem 3.3), 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 \( x \) 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.
4.3. A BRANCH-AND-CUT ALGORITHM

Identifying a (Nearly) Violated Odd Hole Inequality

We start by describing the separation algorithm for the basic odd hole inequalities \( x(V(H)) \leq \lfloor |V(H)|/2 \rfloor \) given the vector \( x \). We first find an odd cycle starting from some node \( v \in V \) using the construction described by Grötschel, Lovász, and Schrijver [55]. 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 \( c \in [0,1]^E \) and \( \tilde{c} \in [0,1]^{\tilde{E}} \) 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 \( c(u,v) = \tilde{c}(u_1,v_2) = \tilde{c}(u_2,v_1) = 1 - x_u - x_v \). 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 \( c(E(H)) < 1 \), 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 paths with length at most 1.125. Shortest paths in a graph with non-negative edge lengths can be found using Dijkstra’s algorithm [42]. Hence, we can find a closed walk

\[ C := (v = v_0, e_1, v_1, e_2, v_2, \ldots, v_k = v) \]

in \( G \) with odd \( k \) that is minimal with respect to \( c \) and \( |C| \) by using Dijkstra’s algorithm to find a shortest path (with respect to \( \tilde{c} \)) 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.

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 4.2. We proceed by showing that $H$ is indeed an odd hole.

**Proposition 4.11.** Let $H$ be constructed as above. Then, $H$ is an odd hole in $G$.

**Proof.** Because $\{v^1, v^2\} \not\in \tilde{E}$, 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,

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

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

$$c(\{e_{i+1}, \ldots, e_j\}) = (j - i) - (2 \sum_{p=i+1}^{j-1} x_{vp}) - x_{v_i} - x_{v_j}.$$ 

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

$$\sum_{p=i+1}^{j-1} x_{vp} \leq (j - i - 1)/2.$$ 

Therefore,

$$c(\{e_{i+1}, \ldots, e_j\}) = (j - i) - (j - i - 1) - x_{v_i} - x_{v_j} = c_{\{v_i, v_j\}},$$

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

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

**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 $\{v_1, v_2, \ldots, v_{|V\setminus V(H)|}\}$. By Theorem 3.3 a lifted odd hole induces a facet if we choose

$$\alpha_v = \lfloor |V(H)|/2 \rfloor - \max\{x(V(H)) + \sum_{j=1}^{i-1} \alpha_v x_{v_j} \mid x \in X_{IS}^i\},$$

where

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

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

Nemhauser and Sigismondi [85] 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 $\alpha$ 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 $\alpha$, we will make use of a path decomposition (see Bodlaender [19], and de Fluiter [46]) of the graph induced by the nodes in the hole and the nodes we already lifted.
4.3. A BRANCH-AND-CUT ALGORITHM

**Definition 4.3.** A path decomposition of a graph $G = (V, E)$ is a sequence $(S_i)_{i=1}^n$ satisfying the following conditions:

\[\bigcup_{i=1}^n S_i = V,\]

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

\[S_i \cap S_j \subseteq S_k \quad \text{for all } i, j, k \text{ with } 1 \leq i < j < k \leq n \text{ we have } S_i \cap S_k \subseteq S_j.\]

(4.10) (4.11) (4.12)

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 $S_1 = S_n = \emptyset$. 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

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

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 $(S_i)_{i=1}^{2|V'|+1}^{2|V'|+1}$ is a path decomposition of the graph induced by $V'$. A path decomposition of $G[V' \setminus N_1(v)]$ can be obtained from $(S_i)_{i=1}^{2|V'|+1}^{2|V'|+1}$ 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 (4.10)–(4.12) 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.

**Weighted Independent Sets by Path Decomposition**

Suppose we are given a graph $G = (V, E)$, together with a normalised path decomposition $(S_i)_{i=1}^{2|V|+1}$ of $G$. Let $I$ again denote the collection of all inde-
pendent sets in $G$, and let $\mathcal{I}_i = \{I \in \mathcal{I} \mid I \subseteq S_i\}$ be all independent sets in $G[S_i]$. Finally, let $V_i = \bigcup_{j=1}^{i} S_j$. Since the path decomposition $(S_i)_{i=1}^{2|V|+1}$ is normalised, for all $i > 1$ there is a node $v \in V$ such that either $S_i = S_{i-1} \cup \{v\}$ or $S_i = S_{i-1} \setminus \{v\}$. As a consequence the sets $\mathcal{I}_i$ satisfy the following recursive relation:

$$
\mathcal{I}_i = \begin{cases}
\{\emptyset\}, & \text{if } i = 1, \\
\mathcal{I}_{i-1} \cup \{(I \cup \{v\}) \in \mathcal{I} \mid I \in \mathcal{I}_{i-1}\}, & \text{if } i > 1 \text{ and } S_i = S_{i-1} \cup \{v\}, \text{ and } \\
\{I \in \mathcal{I}_{i-1} \mid v \notin I\}, & \text{if } i > 1 \text{ and } S_i = S_{i-1} \setminus \{v\}. 
\end{cases}
$$

(4.13)

Given a path decomposition $(S_i)_{i=1}^{2|V|+1}$ of $G$ and a weight vector $\alpha \in \mathbb{N}^V$, we can compute the weight of a maximum weight independent set in $G$ with respect to $\alpha$ using a dynamic programming-like algorithm. To describe this algorithm, we use the functions

$$
z_i : \mathcal{I}_i \to \mathbb{N} : I \mapsto \max\{\alpha(I') \mid I' \text{ is independent set in } G[V_i], I' \cap S_i = I\}.
$$

It follows from condition (4.10) that the value of $z_{2|V|+1}(\emptyset)$ is the maximum weight of an independent set of $G$, which is what we want to compute. From the definition of $z_i$ we find $z_i(\emptyset) = 0$. Now suppose $i > 1$, let $I \in \mathcal{I}$ be an independent set in $S_i$, and let $I^*$ be the maximum weight independent set in $G[V_i]$ with $I^* \cap S_i = I$. In the following, we will relate $I^*$ to a maximum weight independent set $I'$ in $G[V_{i-1}]$ and characterise $I' \cap S_{i-1}$. There are four cases to consider.

**Case (i):** $S_i = S_{i-1} \cup \{v\}$ and $v \notin I$. Clearly, $I^*$ also is a maximum weight independent set in $G[V_{i-1}]$ with $I^* \cap S_{i-1} = I$. Hence we can take $I' = I^*$, and $I' \cap S_{i-1} = I$.

**Case (ii):** $S_i = S_{i-1} \cup \{v\}$ and $v \in I$. We claim that $u \in S_i$ for all $\{u, v\} \in E(V_i)$. It follows from this claim that $I' = I^* \setminus \{v\}$ is a maximum weight independent set in $G[V_{i-1}]$ (or $I^*$ would not be optimal either), and $I' \cap S_{i-1} = I \setminus \{v\}$. To prove our claim suppose by way of contradiction that $\{u, v\} \in E(V_i)$ but $u \notin S_i$. From this it follows that $u \in S_k$ for some $k < i$. By condition (4.11) there exists $j \in \{1, \ldots, 2|V| + 1\}$ such that $\{u, v\} \subseteq S_j$. Since $S_i = S_{i-1} \cup \{v\}$ and by condition (4.12) $v$ occurs only in consecutive sets in the path decomposition, we find that $j > i$. However, condition (4.12) together with $u \in S_k$ and $u \in S_j$ implies that $u \in S_i$, a contradiction, completing the proof of the claim.

**Case (iii):** $S_i = S_{i-1} \setminus \{v\}$ and $I \cup \{v\}$ is an independent set in $G[S_{i-1}]$. From $V_i = V_{i-1}$ it follows that $I' = I^*$ is a maximum weight independent set in $G[V_{i-1}]$ with $I' \cap S_i = I$. Either $v \in I'$ or $v \notin I'$. Hence, $I' \cap S_{i-1} = I \cup \{v\}$ or $I' \cap S_{i-1} = I$.
4.3. A BRANCH-AND-CUT ALGORITHM

Case (iv): $S_i = S_{i-1} \setminus \{v\}$ and $I \cup \{v\}$ is not an independent set in $G[S_{i-1}]$.
Since $I^* \cap S_i = I$ implies $(I^* \cup \{v\}) \cap S_{i-1} = I \cup \{v\}$, and $I^*$ is an independent set, we have $v \notin I^*$. It follows that we can take $I' = I^*$, and $I' \cap S_{i-1} = I$.

The case analyses above leads to the following recurrence for $z_i(I)$ with $i > 1$:

$$
\begin{align*}
    z_i(I) = \begin{cases}
    z_{i-1}(I), & \text{if } S_i = S_{i-1} \cup \{v\} \text{ and } v \notin I, \\
    z_{i-1}(I \setminus \{v\}) + \alpha_v, & \text{if } S_i = S_{i-1} \cup \{v\} \text{ and } v \in I, \\
    \max(z_{i-1}(I), z_{i-1}(I \cup \{v\})), & \text{if } S_i = S_{i-1} \setminus \{v\} \text{ and } I \cup \{v\} \in \mathcal{I}_{i-1}, \\
    z_{i-1}(I), & \text{if } S_i = S_{i-1} \setminus \{v\} \text{ and } I \cup \{v\} \notin \mathcal{I}_{i-1}.
    \end{cases}
\end{align*}

(4.14)

The dynamic programming algorithm works in $2|V| + 1$ iterations, numbered $1, \ldots, 2|V| + 1$. In iteration $i$, the algorithm computes the domain of $z_i$, together with its function values. The domain of $z_i$ is $\{\emptyset\}$, and $z_1(\emptyset) = 0$. For $i > 1$, the domains and function values are computed unfolding the recursive definitions (4.13) and (4.14), respectively.

To implement the dynamic programming algorithm, we maintain a list $\mathcal{L}$ of tuples $(I, z_i(I))$ with $I \in \mathcal{I}_i$, where $i$ is the iteration number. Iteration 1 is implemented by setting $\mathcal{L} = \{(\emptyset, 0)\}$. Iteration $i > 1$ is implemented as follows. Either $S_i = S_{i-1} \cup \{v\}$ or $S_i = S_{i-1} \setminus \{v\}$. If $S_i = S_{i-1} \cup \{v\}$ we update $\mathcal{L}$ by adding the tuple $(I \cup \{v\}, z + \alpha_v)$ to $\mathcal{L}$ for each tuple $(I, z) \in \mathcal{L}_{i-1}$ that satisfies $N_i(v) \cap I = \emptyset$. If $S_i = S_{i-1} \setminus \{v\}$, we update $\mathcal{L}$ by removing all tuples $(I, z)$ that have $v \in I$. After removing a tuple $(I, z)$ from $\mathcal{L}$, we replace the tuple $(I \setminus \{v\}, z') \in \mathcal{L}$ by the tuple $(I \setminus \{v\}, \max(z, z'))$.

If we order $\mathcal{L}$ such that for all pairs of tuples $(I_1, z_1(I_1)), (I_2, z_2(I_2)) \in \mathcal{L}$ with $I_2 \subseteq I_1$ we have that the tuple containing $I_1$ precedes the tuple containing $I_2$ in $\mathcal{L}$, each iteration can be implemented by a single pass over $\mathcal{L}$, using two pointers into $\mathcal{L}$. If the path decomposition $(S_i)_{i=1}^{2|V|+1}$ has width $w$, then the independent sets in the domains of each of the functions $z_i$ can be implemented using bit vectors of length $w$, and the ordering can be implemented using the default “greater than” comparison operator on bit vectors. If we restrict ourselves to path decompositions with width at most 32, then bit vectors can be implemented using integers. In that case the dynamic programming algorithm can be implemented using a double-linked list of tuples of integers as most complicated data structure.

**Proposition 4.12.** A maximum weight independent set can be computed using a path decomposition of width $w$ in $O(w 2^w |V| + |E|)$ time.

**Proof.** When adding a node $v$, we have to check whether $I \cup \{v\} \in \mathcal{I}$ for each $(I, z) \in \mathcal{L}$. This can be done in constant time for each such $I$ after we mark the neighbours of node $v$. Making a new independent set takes at most $O(w)$ time. It follows that iteration $i$ in which a node is added can be implemented in $O(w |\mathcal{I}_i| + |\delta(v)|)$, where $v$ is the node we add. If we delete a node in iteration
all checks and updates can be done in constant time, so the iteration can be implemented in $O(|\mathcal{I}_i|)$ time. Since $|\mathcal{I}_i| \leq 2^w$ for all $i$, the dynamic programming algorithm can be implemented to work in

$$
\sum_{i=1}^{2^{|V|}+1} O(w 2^w + |\delta(v)|) = O(w 2^w |V| + |E|)
$$
time.

The factor $2^w$ in the time complexity of the dynamic programming algorithm looks rather ominous. However, the actual running time depends on the actual size that $\mathcal{L}$ attains. In iteration $i$, this size equals the number of independent sets in $G[S_i]$. In our implementation, we restricted ourselves to path decompositions with width at most 24, and it rarely occurred that the path width exceeded 24. On those rare occasions, however, we do not compute the true lifting coefficients but use a coefficient of 0 instead. Counting to $2^{24}$ each time we want to compute a lifting coefficient would definitely stall our algorithm. Although we cannot exclude the possibility that this occurs, we would like to stress that this did not occur in our computational experiments. We see this as an indication that the worse case behaviour is unlikely to manifest itself on the classes of problem instances that we are interested in.

4.3.4 Maximally Violated Mod-$k$ Cuts
Suppose $x^* \in P_E$ is a fractional solution to the maximum independent set problem. We use the algorithm of Section 3.3.5 to search maximally violated mod-$k$ cuts. As input to the mod-$k$ separation algorithm, we use all (globally) valid inequalities for $P_{GUB}$ that are present in the formulation of the linear programming relaxation and are satisfied with equality by $x^*$. The inequalities we use are the following: maximal clique inequalities (4.2), lifted odd hole inequalities (4.9), non-negativity constraints on $x_V$, upper bound constraints of 1 on components of $x_V$, and mod-$k$ cuts that were found at an earlier stage of the algorithm.

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

$$
z^* = \max\{z(x) = x(V) \mid Ax = b, l \leq x \leq u\},
$$

for some $l, u \in \{0, 1\}^V$ where $A, b$ are obtained from the constraint matrix of $P_{GUB}$ and 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 3.2.2.

We start with the most elementary of all logical implications:
Proposition 4.13. 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 \( z^* \).

Proof. Directly from the definition of \( P_E \).

Proposition 4.14. Let \( v, W \) be as in Proposition 4.13. 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 \( z^* \).

Proof. Because \( z^* = z(x^*) \) for some optimal solution \( x^* \) to model (4.15), and \( x^*_W = 0 \) together with optimality of \( x^* \) implies that \( x^*_v = 1 \), 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 4.13. In a formulation of the independent set problem with explicit slack variables, such as model (4.4), 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 4.13 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 4.14.

After applying Proposition 4.13 to all nodes that are set to one, we can restate the linear programming problem 4.15 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 0 and an upper bound of 1). Therefore, setting variables to zero can be interpreted as removing nodes from the graph.

Proposition 4.15. Let \( v \in G, U = N(v) \) the set of neighbours of \( v \), and \( W = N(N_1(v)) \) the set of neighbours of the 1-neighbourhood of \( v \). When removing \( v \) from \( G \), 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 4.7, 4.8, and 4.9.

After setting a variable to zero, we check whether any of the three cases of Proposition 4.15 occurs, and if so, we take the appropriate action.

4.3.6 Branching Scheme

We complete the description of our branch-and-cut algorithm by giving the branching scheme we use. Since we use the formulation of the maximum independent set problem with explicit slack variables (4.4) all maximal clique
CHAPTER 4. MAXIMUM INDEPENDENT SET

Table 4.1: Heuristics for the Maximum Independent Set Problem

<table>
<thead>
<tr>
<th>Name</th>
<th>Heuristic</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>1-Opt starting from zero</td>
<td>4.2.3</td>
</tr>
<tr>
<td>O2</td>
<td>1-Opt followed by 2-opt starting from zero</td>
<td>4.2.3</td>
</tr>
<tr>
<td>L1</td>
<td>Simple LP rounding followed by 1-opt</td>
<td>4.2.1, 4.2.3</td>
</tr>
<tr>
<td>L2</td>
<td>Simple LP rounding followed by 1-opt and 2-opt</td>
<td>4.2.1, 4.2.3</td>
</tr>
<tr>
<td>R1</td>
<td>Minimum regret rounding with parameter $t = 1$</td>
<td>4.2.2</td>
</tr>
<tr>
<td>R2</td>
<td>Minimum regret rounding with parameter $t = 2$</td>
<td>4.2.2</td>
</tr>
</tbody>
</table>

inequalities are GUB constraints. We try to take advantage of this by using branching on variables and branching on GUB constraints. Deciding whether to branch on a variable or a GUB constraint is done based on pseudo-cost as described in Section 3.2.5.

4.4 Computational Results

In this section we report on the computational behaviour of the algorithms proposed in this chapter, which we implemented using the graph data structure from LEDA [83]. We tested our algorithms on independent set problems that are similar to those reported on in the literature. An interesting class of independent set problem instances that can be found in the literature is the class defined on so-called uniform random graphs, or URGs. URGs are graphs on $n$ nodes in which each of the $n(n - 1)/2$ possible arcs is present with a fixed probability $p$. All averages reported in this section are taken over 25 randomly generated instances for every choice of $n$ and $p$. We will first consider the performance of our heuristic algorithms, and then proceed by discussing the behaviour of our optimisation algorithms.

Heuristics. The heuristics for the maximum independent set problem presented in Section 4.2 are summarised in Table 4.1. The average sizes of the maximum independent sets computed by our heuristics and the corresponding running times are reported on in Table 4.2. The columns O1 and O2 refer to the iterative improvement algorithms that were presented in Section 4.2.3. These algorithms use the 1-opt and 2-opt neighbourhoods, respectively, starting from 0. The L1, L2, R1 and R2 columns all refer to LP-based rounding algorithms that first compute the optimal solution to an LP relaxation of the independent set polytope that is strengthened by our cutting plane routines. The L1 and L2 columns refer to the algorithms that first apply the simple LP rounding algorithms from Section 4.2.1 to obtain an integer solution. Next, they 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 4.2.2 to the optimal solution of the LP relaxation, with the parameter $t$ equal to 1.
4.4. COMPUTATIONAL RESULTS

Table 4.2: Heuristics on Uniform Random Graphs (URGs)
and 2, respectively. The $\alpha$ columns contain the average size of the resulting independent set, the CPU columns contain the average number of CPU seconds that were needed to find the reported independent sets, the time needed to solve the LP relaxations excluded. Finally, the $\alpha(G)$ column contains the average size of the maximum independent set in the test instances.

On uniform random graphs the 2-opt algorithm consistently performs better than the 1-opt algorithm (on average). Using a rounded LP solution as a starting point of an iterative improvement algorithm improves the quality of the reported independent sets for sparse problems ($p \leq .2$) and for dense problems ($p = .9$). The minimum regret heuristics perform consistently better (on average) than the simple rounding plus iterative improvement heuristics. All rounding algorithms are fast in practice (CPU time < .1s on average, with the exception of minimum regret rounding with $t = 2$ that runs within 1.5s), the CPU time needed for the rounding phase is dominated by the CPU time needed for solving the linear programs. This makes these algorithms well suited for use from within an LP based branch-and-bound algorithm. Note that the minimum regret heuristic with parameter $t = 2$ often achieves an average value of $\alpha$ that is greater than $\alpha(G) - 1$. This can only happen if the algorithm reports optimal solutions on some of the instances.

**Optimisation Algorithms.** The average performance of our branch-and-cut and cutting plane/branch-and-bound algorithms are presented in Tables 4.3 and 4.4, respectively. In our cutting plane/branch-and-bound algorithm we solve the root node in exactly the same way as we do in our branch-and-cut algorithm, and then invoke the MIP solver from CPLEX with the strongest possible formulation and the best primal bound we have available. In these tables, the Nodes column contains the average number of nodes in the branch-and-bound tree, and the CPU column contains the average CPU time it took for the algorithm to terminate in seconds. The remaining columns report on the effectiveness of the valid inequalities we use, and their corresponding separation algorithms. For each class of valid inequalities, the # column contains the average number of distinct valid inequalities that were separated, the Gap column contains the average percentage of duality gap closed in the root node, and the Prof. column contain the average amount of CPU time that was spent in the corresponding separation algorithm (Prof. is an abbreviation of Profiling Data).

When comparing the branch-and-cut algorithm to the cutting plane/branch-and-bound algorithm, it is clear that the faster of the two algorithms is the cutting plane/branch-and-bound algorithm. We believe that this is due to the fact that we use the MIP solver of CPLEX 6.5 for the branch-and-bound phase, which is a highly optimised commercial package. The branch-and-cut algorithm does achieve the smallest branch-and-bound tree sizes of on almost all of the runs. From the Gap columns it is clear that using our cutting plane routines from each node in the branch-and-bound tree does improve the formulation. Here, maximum clique inequalities become more important when problems are
## 4.4. COMPUTATIONAL RESULTS

<table>
<thead>
<tr>
<th>$n$</th>
<th>$p$</th>
<th>Performance</th>
<th>Max Clique</th>
<th>Lifted Odd Hole</th>
<th>Mod-k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$n$</td>
<td>$p$</td>
<td>Nodes</td>
<td>CPU</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>9.2</td>
<td>55.6</td>
<td>221.7</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>87.3</td>
<td>187.2</td>
<td>425.4</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>139.6</td>
<td>235.4</td>
<td>702.0</td>
<td>11.8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>110.2</td>
<td>309.5</td>
<td>997.2</td>
<td>21.0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>67.8</td>
<td>350.8</td>
<td>1223.3</td>
<td>34.2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>42.2</td>
<td>378.6</td>
<td>1278.6</td>
<td>48.2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>18.4</td>
<td>363.8</td>
<td>1302.5</td>
<td>62.2</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4.5</td>
<td>178.2</td>
<td>10388.8</td>
<td>75.6</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1.0</td>
<td>15.6</td>
<td>634.4</td>
<td>75.6</td>
</tr>
<tr>
<td>90</td>
<td>1</td>
<td>16.9</td>
<td>108.7</td>
<td>273.6</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>272.4</td>
<td>464.6</td>
<td>590.6</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>354.5</td>
<td>656.1</td>
<td>1095.9</td>
<td>13.3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>218.0</td>
<td>748.8</td>
<td>1590.7</td>
<td>21.8</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>131.9</td>
<td>737.2</td>
<td>1794.7</td>
<td>34.4</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>72.3</td>
<td>1023.6</td>
<td>2223.4</td>
<td>47.1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>30.0</td>
<td>861.2</td>
<td>2002.0</td>
<td>61.4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8.0</td>
<td>520.1</td>
<td>1513.3</td>
<td>73.6</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1.0</td>
<td>29.5</td>
<td>886.5</td>
<td>78.5</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>65.5</td>
<td>339.2</td>
<td>333.4</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>766.6</td>
<td>1000.0</td>
<td>844.5</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1008.0</td>
<td>1965.0</td>
<td>1675.2</td>
<td>12.3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>712.8</td>
<td>3042.3</td>
<td>2767.4</td>
<td>21.9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>244.8</td>
<td>1802.0</td>
<td>2813.0</td>
<td>34.5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>133.6</td>
<td>2047.8</td>
<td>3085.8</td>
<td>47.1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>54.8</td>
<td>1749.1</td>
<td>2867.4</td>
<td>58.6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.3</td>
<td>1255.6</td>
<td>2087.6</td>
<td>74.4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1.3</td>
<td>183.9</td>
<td>1289.4</td>
<td>86.1</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>213.6</td>
<td>808.4</td>
<td>395.4</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2542.4</td>
<td>3399.3</td>
<td>1210.0</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2528.7</td>
<td>5290.9</td>
<td>2370.5</td>
<td>12.3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1291.7</td>
<td>4984.3</td>
<td>3392.5</td>
<td>22.3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>564.9</td>
<td>5564.0</td>
<td>4763.7</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>210.4</td>
<td>3935.6</td>
<td>4699.5</td>
<td>46.9</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>82.5</td>
<td>3770.2</td>
<td>4562.8</td>
<td>58.0</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>14.4</td>
<td>2625.8</td>
<td>3174.0</td>
<td>71.6</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1.6</td>
<td>691.3</td>
<td>1633.1</td>
<td>83.6</td>
</tr>
</tbody>
</table>

Table 4.3: Run-Time Behaviour of Branch-and-Cut on URGs
Table 4.4: Run-Time Behaviour of Cutting-Plane/Branch-and-Bound on URGs

<table>
<thead>
<tr>
<th>$n$</th>
<th>$p$</th>
<th>Performance</th>
<th>Max Clique</th>
<th>Lifted Odd Hole</th>
<th>Mod-$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$n_p$ Nodes</td>
<td>CPU</td>
<td>#</td>
<td>Gap</td>
</tr>
<tr>
<td>80</td>
<td>.1</td>
<td>21.6 26.3</td>
<td>221.1 0.3</td>
<td>0.4</td>
<td>101.3</td>
</tr>
<tr>
<td></td>
<td>.2</td>
<td>314.2 34.5</td>
<td>348.0 4.7</td>
<td>0.2</td>
<td>6.9</td>
</tr>
<tr>
<td></td>
<td>.3</td>
<td>356.6 59.1</td>
<td>470.2 10.6</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>.4</td>
<td>231.1 95.5</td>
<td>589.1 18.6</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.5</td>
<td>162.7 136.3</td>
<td>706.3 31.2</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.6</td>
<td>96.0 179.4</td>
<td>792.6 45.0</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.7</td>
<td>66.8 256.3</td>
<td>895.9 59.8</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.8</td>
<td>18.8 226.4</td>
<td>874.1 74.3</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.9</td>
<td>1.1 59.3</td>
<td>634.3 75.6</td>
<td>0.3</td>
<td>0.0</td>
</tr>
<tr>
<td>90</td>
<td>.1</td>
<td>75.5 42.3</td>
<td>271.8 0.5</td>
<td>0.3</td>
<td>105.6</td>
</tr>
<tr>
<td></td>
<td>.2</td>
<td>954.7 74.8</td>
<td>429.8 4.7</td>
<td>0.1</td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td>.3</td>
<td>754.6 134.9</td>
<td>601.5 11.6</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.4</td>
<td>402.3 173.2</td>
<td>749.4 19.4</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.5</td>
<td>200.8 232.6</td>
<td>892.7 31.4</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.6</td>
<td>117.9 351.4</td>
<td>1044.5 44.1</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.7</td>
<td>89.5 471.4</td>
<td>1139.5 58.7</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.8</td>
<td>44.9 506.0</td>
<td>1149.9 72.5</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.9</td>
<td>1.4 118.2</td>
<td>886.5 78.5</td>
<td>0.3</td>
<td>0.0</td>
</tr>
<tr>
<td>100</td>
<td>.1</td>
<td>415.2 63.3</td>
<td>327.6 0.6</td>
<td>0.2</td>
<td>96.0</td>
</tr>
<tr>
<td></td>
<td>.2</td>
<td>1769.2 146.7</td>
<td>539.0 4.9</td>
<td>0.1</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>.3</td>
<td>1426.6 269.2</td>
<td>740.0 10.6</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.4</td>
<td>679.0 314.6</td>
<td>929.5 19.8</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.5</td>
<td>323.6 417.5</td>
<td>1118.0 31.2</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.6</td>
<td>179.1 582.2</td>
<td>1255.2 43.2</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.7</td>
<td>122.3 798.3</td>
<td>1386.9 55.5</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.8</td>
<td>70.7 864.6</td>
<td>1427.4 72.5</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.9</td>
<td>1.3 431.0</td>
<td>1261.3 86.0</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td>110</td>
<td>.1</td>
<td>901.9 99.2</td>
<td>384.4 0.7</td>
<td>0.2</td>
<td>88.9</td>
</tr>
<tr>
<td></td>
<td>.2</td>
<td>3786.0 324.1</td>
<td>641.2 4.2</td>
<td>0.0</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>.3</td>
<td>3070.0 496.6</td>
<td>877.5 10.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.4</td>
<td>1505.6 541.4</td>
<td>1130.8 19.6</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.5</td>
<td>566.1 615.9</td>
<td>1337.4 31.1</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.6</td>
<td>266.3 777.9</td>
<td>1501.2 42.6</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.7</td>
<td>149.4 1107.2</td>
<td>1723.9 55.0</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.8</td>
<td>118.7 1256.0</td>
<td>1790.6 69.6</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>.9</td>
<td>12.0 828.1</td>
<td>1542.6 83.3</td>
<td>0.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>
more dense, lifted odd hole inequalities become more important for problems with lower densities, and mod-$k$ cuts do well for $p = .1$ and $p \geq .5$.

When comparing the branch-and-bound tree sizes of our branch-and-cut algorithm to those reported in the literature by Mannino and Sassano [80] and Sewell [106], it is clear that we achieve the smallest branch-and-bound tree sizes for very sparse problems ($p \leq .2$) and very dense problems ($p \geq .8$), even though we do not employ a branching scheme that is tailor-made for the independent set problem. We believe that this is due to the quality of the bound obtained from the strengthened LP relaxations that we use. This quality does have its cost, though, because we spend a lot of time in each node of the branch-and-bound tree to obtain it. As a consequence, our algorithm is outperformed in terms of CPU time by those reported in the literature.

Run time profiles, branching decisions, and the average number of variables that is set by our pre-processing, reduced cost and logical implication subroutines are reported on in Table 4.5. The LP, F, VP, GP, Sep, Set, and Heur columns contain the percentage of CPU time spent in solving linear programs, formulating subproblems, initialising and maintaining variable pseudo costs, GUB pseudo costs, variable setting and our rounding heuristics, respectively. The GUB column contains the percentage of branches that were GUB branches, all other branches were variable branches. A minus indicates that all runs of the algorithm that were averaged over were solved in the root node. The U and L columns contain the average number of variables that were set by strengthened reduced cost fixing to their upper and lower bound, respectively. The SP, ND, and SD the average number of nodes that were set because they became simplicial, node-dominated, and set-dominated, respectively.

The GUB column clearly shows that GUB branches are competitive with variable branches when compared using pseudo costs. It was to be expected that GUB branching would be very useful on the dense instances. Surprisingly, it also is very competitive for the instances with $p = .1$ and $n \geq 90$. The relative large amount of time spent in initialising the GUB pseudo costs for problems with higher density ($p \geq .5$) is due to the fact that the branch-and-bound tree sizes are small on those classes of instances. Finally, we note that our variable setting criteria do not have a particularly large yield. On the other hand, they do not take up a lot of CPU time.
### CHAPTER 4. MAXIMUM INDEPENDENT SET

#### Run-Time Profiles

<table>
<thead>
<tr>
<th>n</th>
<th>p</th>
<th>LP</th>
<th>F</th>
<th>VP</th>
<th>GP</th>
<th>Sep</th>
<th>Set</th>
<th>Heur</th>
<th>GUB</th>
<th>U</th>
<th>L</th>
<th>SP</th>
<th>ND</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>.1</td>
<td>0.3</td>
<td>3.7</td>
<td>40.5</td>
<td>12.6</td>
<td>36.5</td>
<td>1.9</td>
<td>1.7</td>
<td>7.8</td>
<td>0.0</td>
<td>2.2</td>
<td>0.2</td>
<td>0.4</td>
<td>0.0</td>
</tr>
<tr>
<td>.2</td>
<td>1.8</td>
<td>6.2</td>
<td>15.9</td>
<td>24.4</td>
<td>37.9</td>
<td>4.2</td>
<td>3.8</td>
<td>22.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.3</td>
<td>3.4</td>
<td>8.8</td>
<td>15.0</td>
<td>25.3</td>
<td>31.1</td>
<td>5.6</td>
<td>4.4</td>
<td>18.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.4</td>
<td>2.8</td>
<td>10.3</td>
<td>15.8</td>
<td>29.2</td>
<td>27.4</td>
<td>5.6</td>
<td>3.4</td>
<td>24.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.5</td>
<td>1.9</td>
<td>10.0</td>
<td>18.3</td>
<td>37.2</td>
<td>21.9</td>
<td>4.1</td>
<td>2.5</td>
<td>25.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.6</td>
<td>1.2</td>
<td>7.9</td>
<td>20.3</td>
<td>44.4</td>
<td>18.8</td>
<td>2.9</td>
<td>1.7</td>
<td>19.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.7</td>
<td>0.6</td>
<td>5.8</td>
<td>23.6</td>
<td>47.3</td>
<td>17.7</td>
<td>2.3</td>
<td>1.0</td>
<td>21.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.8</td>
<td>0.2</td>
<td>4.3</td>
<td>21.0</td>
<td>52.0</td>
<td>17.8</td>
<td>2.3</td>
<td>0.6</td>
<td>31.8</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>8.8</td>
<td></td>
</tr>
<tr>
<td>.9</td>
<td>0.1</td>
<td>9.1</td>
<td>0.0</td>
<td>0.0</td>
<td>56.4</td>
<td>12.2</td>
<td>1.2</td>
<td>–</td>
<td>0.0</td>
<td>15.2</td>
<td>0.0</td>
<td>2.5</td>
<td>24.7</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>.1</td>
<td>0.4</td>
<td>4.1</td>
<td>33.7</td>
<td>18.2</td>
<td>35.8</td>
<td>1.9</td>
<td>2.3</td>
<td>33.2</td>
<td>0.0</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td>.2</td>
<td>3.2</td>
<td>9.5</td>
<td>7.7</td>
<td>13.5</td>
<td>45.7</td>
<td>6.7</td>
<td>5.7</td>
<td>30.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.3</td>
<td>4.9</td>
<td>13.5</td>
<td>8.0</td>
<td>16.2</td>
<td>34.3</td>
<td>8.7</td>
<td>5.5</td>
<td>34.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.4</td>
<td>3.5</td>
<td>15.4</td>
<td>10.6</td>
<td>23.8</td>
<td>28.1</td>
<td>7.2</td>
<td>4.0</td>
<td>32.6</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.5</td>
<td>2.5</td>
<td>14.7</td>
<td>14.3</td>
<td>34.4</td>
<td>20.9</td>
<td>5.1</td>
<td>2.7</td>
<td>20.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.6</td>
<td>1.3</td>
<td>11.0</td>
<td>15.3</td>
<td>47.5</td>
<td>16.4</td>
<td>3.6</td>
<td>1.6</td>
<td>22.6</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.7</td>
<td>0.7</td>
<td>7.0</td>
<td>19.0</td>
<td>53.7</td>
<td>14.4</td>
<td>2.4</td>
<td>1.0</td>
<td>21.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>.8</td>
<td>0.2</td>
<td>4.6</td>
<td>18.8</td>
<td>56.6</td>
<td>16.0</td>
<td>1.8</td>
<td>0.5</td>
<td>28.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>7.4</td>
<td></td>
</tr>
<tr>
<td>.9</td>
<td>0.1</td>
<td>9.9</td>
<td>0.0</td>
<td>0.0</td>
<td>59.7</td>
<td>12.6</td>
<td>0.8</td>
<td>–</td>
<td>0.1</td>
<td>19.9</td>
<td>0.0</td>
<td>1.7</td>
<td>26.0</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>.1</td>
<td>0.7</td>
<td>6.5</td>
<td>13.1</td>
<td>13.0</td>
<td>52.2</td>
<td>3.1</td>
<td>2.9</td>
<td>71.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td>.2</td>
<td>5.9</td>
<td>12.9</td>
<td>4.3</td>
<td>9.6</td>
<td>39.9</td>
<td>8.9</td>
<td>8.7</td>
<td>25.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.3</td>
<td>7.4</td>
<td>20.1</td>
<td>4.2</td>
<td>12.0</td>
<td>27.8</td>
<td>10.4</td>
<td>6.7</td>
<td>21.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.4</td>
<td>6.2</td>
<td>24.3</td>
<td>5.7</td>
<td>16.8</td>
<td>22.3</td>
<td>9.6</td>
<td>3.9</td>
<td>16.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.5</td>
<td>3.1</td>
<td>18.9</td>
<td>11.5</td>
<td>32.6</td>
<td>18.3</td>
<td>6.3</td>
<td>2.6</td>
<td>21.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.6</td>
<td>1.7</td>
<td>14.2</td>
<td>14.1</td>
<td>43.4</td>
<td>16.1</td>
<td>4.7</td>
<td>1.5</td>
<td>21.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.7</td>
<td>0.7</td>
<td>9.0</td>
<td>16.9</td>
<td>53.2</td>
<td>14.6</td>
<td>2.6</td>
<td>0.9</td>
<td>21.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.8</td>
<td>0.1</td>
<td>3.5</td>
<td>17.0</td>
<td>67.6</td>
<td>9.6</td>
<td>1.2</td>
<td>0.4</td>
<td>46.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>4.7</td>
<td></td>
</tr>
<tr>
<td>.9</td>
<td>0.0</td>
<td>4.3</td>
<td>10.5</td>
<td>54.4</td>
<td>22.4</td>
<td>4.0</td>
<td>0.3</td>
<td>100.0</td>
<td>0.0</td>
<td>15.0</td>
<td>0.0</td>
<td>0.6</td>
<td>26.1</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>.1</td>
<td>1.2</td>
<td>7.9</td>
<td>5.7</td>
<td>7.7</td>
<td>60.4</td>
<td>4.0</td>
<td>3.6</td>
<td>63.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>.2</td>
<td>8.7</td>
<td>16.5</td>
<td>1.8</td>
<td>6.8</td>
<td>32.2</td>
<td>10.8</td>
<td>11.3</td>
<td>18.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.3</td>
<td>10.9</td>
<td>24.1</td>
<td>2.3</td>
<td>10.0</td>
<td>18.6</td>
<td>12.6</td>
<td>7.6</td>
<td>19.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.4</td>
<td>8.7</td>
<td>27.0</td>
<td>4.4</td>
<td>16.0</td>
<td>16.1</td>
<td>10.3</td>
<td>4.8</td>
<td>20.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.5</td>
<td>4.8</td>
<td>24.5</td>
<td>8.0</td>
<td>26.1</td>
<td>15.7</td>
<td>8.5</td>
<td>2.3</td>
<td>13.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.6</td>
<td>2.2</td>
<td>17.2</td>
<td>12.9</td>
<td>39.8</td>
<td>15.5</td>
<td>5.6</td>
<td>1.4</td>
<td>17.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.7</td>
<td>0.8</td>
<td>10.7</td>
<td>15.8</td>
<td>51.5</td>
<td>14.3</td>
<td>3.4</td>
<td>0.8</td>
<td>21.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>.8</td>
<td>0.2</td>
<td>4.2</td>
<td>14.7</td>
<td>68.6</td>
<td>9.8</td>
<td>1.3</td>
<td>0.4</td>
<td>40.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>.9</td>
<td>0.0</td>
<td>2.1</td>
<td>6.5</td>
<td>79.0</td>
<td>9.1</td>
<td>1.6</td>
<td>0.1</td>
<td>87.5</td>
<td>0.0</td>
<td>4.0</td>
<td>0.0</td>
<td>0.3</td>
<td>26.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Profiles, Branching, and Setting Statistics of Branch-and-Cut