# General $b$-Matching via Cutting Planes

Sara Ahmadian, Mehdi Karimi, André Linhares, Miaolan Xie *

**Abstract**

We implemented a cutting planes algorithm for the general $b$-matching problem. In this report, we explain an exact LP formulation of the problem, and present an exact and a heuristic algorithm for separating blossom inequalities. We also explain how to speed up the algorithm and give instructions on how to use our code. The final section presents some numerical results.

# 1 Introduction

## 1.1 The General $b$-matching Problem

The general $b$-matching problem we consider here is defined as follows: for two index sets $V$ and $E$, a matrix $A \in \{-1, 0, 1\}^{V \times E}$, and vectors $c \in \mathbb{R}^E$, $b \in \mathbb{Z}_+^V$, and $u \in \mathbb{Z}_+^E$, we want to solve

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ \text{s.t.} \quad & \sum_{e \in E} A_{ve} x_e = b_v \quad \forall v \in V \quad \text{(degree constraints)} \\ & 0 \le x_e \le u_e \quad \forall e \in E \quad \text{(edge capacities)} \\ & x \in \mathbb{Z}^E \,, \end{aligned} \tag{1}$$

where we also have an additional property $\sum_{v \in V} |A_{ve}| = 2$ for every $e \in E$. We refer to the convex hull of all the feasible points of Problem (1) as the *$u$-capacitated $b$-matching* polytope. It is known for a long time that there exist efficient combinatorial algorithms for solving the general $b$-matching problem, one can see for example [3, 11, 2]. In summary, we have the following theorem:

**Theorem 1.1.** *There exists a polynomial-time algorithm for the general $b$-matching problem.*

This algorithm, which is generally called the *blossom algorithm*, requires finding, shrinking, and unshrinking structures called blossoms, and is difficult to implement. In this project, we use another approach which is called the *cutting plane method*.

## 1.2 Reduction to the $b$-matching Problem

In this section, we explain how to reduce the general $b$-matching problem to the $b$-matching problem on undirected graphs. The reduction we present is described in [8]. In a bidirected graph, we assign a sign $-$ or $+$ to each endpoint of each edge. To reduce a bidirected graph $B$ to an undirected graph, we perform the following steps for every vertex $v$ of $B$:

1. Replace each vertex $v$ with two vertices $v^-$ and $v^+$.
2. Define $m_v = \sum(u_e : e \text{ is connected to } v \text{ by a } - \text{ sign})$.
3. Connect all the edges that are incident to $v$ by a $-$ sign to $v^-$ and all the edges incident to $v$ by a $+$ sign to $v^+$.

*{sahmadian,alinhare,m8karimi,m8xie}@uwaterloo.ca. Dept. of Combinatorics and Optimization, Univ. Waterloo, Waterloo, ON N2L 3G1.

4. Put vertex demand of $m_v$ for $v^-$ and $m_v + b_v$ for $v^+$.

5. Connect $v^-$ and $v^+$ by an edge with capacity $m_v$ and zero weight.

Figure 1 shows the transformation:

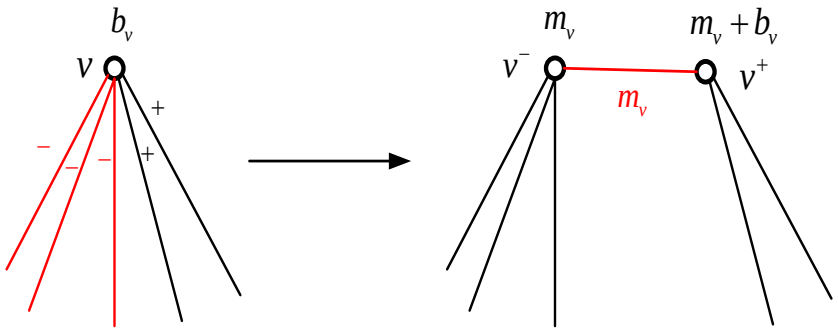



Figure 1: Transformation of a vertex of a bidirected graph.

It can be seen that any $b$-matching in $B$ can be translated into a $b$-matching in the new graph and vice versa.

### 1.3 Outline of the Cutting Planes Algorithm

Doing the reduction described in Section 1.2, from now on, we assume that the matrix $A$ in Problem (1) is the vertex-edge incidence matrix of an undirected graph $G = (V, E)$.

Edmonds [2] and Pulleyblank [11] gave a complete linear description of the $u$-capacitated $b$-matching polytope. They showed that this polytope can be described by replacing the integrality constraint in (1) by the following *blossom constraints*:

$$\sum_{e \in E(W)} x_e + \sum_{e \in F} (u_e - x_e) \leq \frac{b(W) + u(F) - 1}{2} \qquad \forall W \subseteq V, F \subseteq \delta(W) \text{ with } b(W) + u(F) \text{ odd}, \tag{2}$$

where $E(W)$ denotes the set of edges with both endpoints in $W$, $\delta(W)$ denotes the set of edges with exactly one endpoint in $W$, and $b(W)$ and $u(F)$ denote $\sum_{v \in W} b_v$ and $\sum_{e \in F} u_e$, respectively. We cannot simply add all the blossom inequalities to the LP relaxation of (1) (called the *degree LP*) because there are exponentially many of them. In the cutting plane approach, we start by solving the degree LP, and keep adding blossom inequalities as needed. In other words, we find blossom inequalities that are violated by the current solution and add them to the LP until we reach an integral solution. A high-level description of the algorithm is given below.

**Algorithm 1** Cutting Planes Algorithm

**Input:** LP relaxation of the $b$-matching problem (1)

1: **while** the LP is feasible and its optimal solution $\bar{x}$ is not integral **do**
2: find pairs $(W, F)$ such that the blossom constraint (2) is violated at $\bar{x}$
3: add the blossom constraints for such pairs $(W, F)$ to the LP
4: **end while**

In principle, this algorithm has an exponential running time and in the worst case, we may need to add all the blossom inequalities. However, this algorithm works well in practice, as for example reported in [6] for perfect 2-matching. The structure of the report is as follows. In Section 2, we describe an exact algorithm for finding cutting planes. In Section 3, we explain a heuristic algorithm for finding cutting planes. Section 4 presents some techniques to improve the running time of the algorithm. Instructions for using our code can be found in Section 5. In Section 6, we show some numerical results.

## 2 Exact Separation Algorithm

In this section, we explain an exact separation algorithm. First note that we can rewrite the blossom inequalities (2) as:

$$\sum_{e\in\delta(W)-F} x_e + \sum_{e\in F}(u_e - x_e) \geq 1 \qquad \forall W \subseteq V, F \subseteq \delta(W) \text{ with } b(W)+u(F) \text{ odd.} \tag{3}$$

In our approach, we solve the current LP to get a solution $\bar{x}$, and if $\bar{x}$ is not integral, we can find a pair $(W, F)$ for which $\bar{x}$ violates (3). Padberg and Rao [10] proved that finding such a pair $(W, F)$ reduces to finding a *minimum odd cut* in an auxiliary graph. Let $G$ be a graph whose vertices have been labeled odd and even such that the total parity of vertices is even. An odd cut is a cut that partitions the vertices into two sets with odd parity.

### 2.1 Padberg and Rao's Auxiliary Graph

To construct Padberg and Rao's auxiliary graph, take the solution of the current LP $\bar{x}$, and let $G(\bar{x})$ be the graph induced by edges $e$ with $\bar{x}_e > 0$. The vertices are labeled with the parities of $b_v$'s. Then we make another graph $G(\bar{x}, u)$ as follows:

- Scan through the edges of $G(\bar{x})$ one by one and replace each edge $e = (i, j)$ with two edges $(i, i_e)$ and $(i_e, j)$.
- Vertex $i_e$ is labeled by the parity of $u_e$.
- Vertex $j$ retains its current label.
- Vertex $i$ gets an even label if its current label and the label of $i_e$ are equal; it gets an odd label otherwise.

Note that the parity of a vertex may change multiple times during the construction of the graph $G(\bar{x}, u)$,

**Remark 2.1.** *Padberg and Rao in [10] considered the problem in which we have an inequality for the degree constraints in* (1)*. However, in this report (and also in our code) we solve problem* (1) *with equality constraints. The difference with the auxiliary graph constructed above is that we need to add a special vertex $S$ to $G(\bar{x}, u)$ that accounts for the slacks of the inequality constraints, and connect this vertex to all the vertices via edges whose capacities are equal to the slacks. Our code can be updated to solve a combination of equality and inequality constraints.*

Padberg and Rao [10] proved the following theorem:

**Theorem 2.2.** *Given a feasible solution $\bar{x}$, there exists $W \subseteq V$ and $F \subseteq \delta(W)$ with $b(W) + u(F)$ odd that violates* (3) *if and only if there exists an odd cut in $G(\bar{x}, u)$ with capacity less than 1.*

### 2.2 Letchford et al.'s Auxiliary Graph

To construct the above auxiliary graph, we need to add a vertex for every edge $e$ with $\bar{x}_e > 0$. In the worst case, it can increase the number of vertices by $O(n^2)$. Letchford et al. [9] constructed another auxiliary graph on the same set of vertices as the original graph. To do that, we take an optimal solution $\bar{x}$ of the current LP and again let $G(\bar{x})$ be the graph induced by edges $e$ with $\bar{x}_e > 0$. To each edge $e$ of $G(\bar{x})$, we assign a capacity $\min\{\overline{x}_e, u_e - \overline{x}_e\}$.

Note that in this auxiliary graph we ignore the parity of vertices. To find a possible violated cut, we find a minimum cut in this auxiliary graph. If the capacity of the cut $\delta(W)$ is less that 1, then we construct a set $F$ as follows:

$$F := \{e \in \delta(W) : u_e - \overline{x}_e < \overline{x}_e\}. \tag{4}$$

Now two cases can happen based on the parity of $b(W) + u(F)$.

1. If $b(W) + u(F)$ is odd, then $(W, F)$ defines a violated inequality.

2. If $b(W) + u(F)$ is even, then we pick an odd-capacity edge $e \in \delta(W)$ (if there exists one) minimizing $|(u_e - x_e) - x_e|$ and define $\bar{F} := F\Delta\{e\}$, where $\Delta$ denotes the symmetric difference. Then we check if $(W, \bar{F})$ defines a violated inequality.

Ignoring the parity of the vertices and designing the above algorithm is based on the following remark, which is also the base of our heuristic separation algorithm.

**Remark 2.3.** *In view of the blossom inequality* (3)*, for any given subset of vertices $W$, we can easily check if there exists a corresponding $F \subseteq \delta(W)$ such that the pair $(W, F)$ defines a violated inequality. Every edge $e \in \delta(W)$ is either in $F$ and contributes $x_e$ to the left-hand side (LHS) of* (3)*, or it is in $\delta(W) - F$ and contributes $u_e - x_e$. To find a violated cut, we want the LHS to be as small as possible. Hence, the best strategy is putting every edge with $u_e - x_e < x_e$ in $F$. If for the set of edges $F$ constructed in this way, the LHS of* (3) *is less than 1 and $b(W) + u(F)$ is odd, then we have a violated inequality. Otherwise, if the LHS is less than 1 and $b(W) + u(F)$ is even, then the best way to update $F$ is to switch an odd-capacity edge that minimizes the increase on the LHS, i.e., an odd-capacity edge minimizing $|(u_e - x_e) - x_e|$.*

The above remark implies that we can ignore the parity of vertices and look for the minimum cuts in the auxiliary graph. Then if we find a cut with capacity less than 1, we can easily check if we can make a violated pair $(W, F)$ out of it. To find cuts with capacity less than 1, we use the *Gomory-Hu* tree.

## 2.3 Using Gomory-Hu Tree

Given a weighted undirected graph $G$, the Gomory-Hu tree ([5]) of $G$ is a weighted tree $T$ with the same vertex set that represents the minimum $s$-$t$ cuts for all $s$-$t$ pairs in $G$. To find a minimum $s$-$t$ cut, we find the unique path in $T$ that connects $s$ and $t$, and pick an edge $f$ with minimum weight in this path. The weight of $f$ is the capacity of the minimum $s$-$t$ cut in $G$, and the cut itself is the bipartition of the vertices we get by removing $f$ from $T$.

In the conventional discussions of Gomory-Hu tree, the algorithm for constructing the tree $T$ requires vertex contraction operations, which are difficult to implement. Gusfield [7] showed that one can construct the Gomory-Hu tree by adding only five simple lines of code to any program that produces a minimum cut. That is the algorithm we are using in our code. Note that in our implementation, we do not wait for the tree $T$ to be completed and then look for the violated cuts. At each step of the construction of $T$, we need to find the minimum cut for a pair of vertices. That is when we check if the found set $W$ induces a violated cut.

Our complete exact separation algorithm is as follows:

**Algorithm 2** Exact Separation Algorithm

**Input:** A fractional solution $\bar{x}$ to a relaxation of the $b$-matching problem (1)

$G(\bar{x}) \leftarrow$ the auxiliary graph made from $\bar{x}$ as in Subsection 2.2
ViolatedCuts $\leftarrow \emptyset$
**for** each of the $n-1$ steps of the Gomory-Hu algorithm applied on $G(\bar{x})$ **do**
  $W \leftarrow$ minimum cut found by Gomory-Hu algorithm
  $F \leftarrow \{e \in \delta(W) : u_e - x_e < x_e\}$
  **if** $b(W) + u(F)$ is even and $\exists e \in \delta(W) : u_e$ is odd **then**
    $e \leftarrow \arg\min_{e \in \delta(W): u_e \text{ is odd}} |x_e - (u_e - x_e)|$
    $F \leftarrow F\Delta\{e\}$
  **end if**
  **if** $b(W) + u(F)$ is odd and $\sum_{e \in \delta(W) - F} x_e + \sum_{e \in F}(u_e - x_e) < 1$ **then**
    ViolatedCuts $\leftarrow$ ViolatedCuts $\cup \{(W, F)\}$
  **end if**
**end for**
**return** ViolatedCuts

For finding the maximum flow in the Gomory-Hu algorithm, we tried both Edmonds-Karp [4] and Dinitz [1] maximum flow algorithms. Our implementation of Dinitz algorithm is based on the code available at

`http://www.prefield.com/algorithm/graph/dinic.html` . We also used other techniques to speed up this exact algorithm that we discuss in Section 4. In the next section, we explain our heuristic algorithm for finding violated cuts.

# 3 Heuristic Separation Algorithm

Instead of trying to find the violated constraints by the exact separation algorithm, we consider using a heuristic. In this way, hopefully we can reduce the overall running time.

The algorithm we used is based on the heuristic separation algorithm for the minimum-weight 2-matching problem with unit capacities by Grötschel and Holland [6]. The intuitive ideas are the following. We are looking for a pair $(W, F)$ with $b(W) + u(F)$ odd that violates the inequality

$$\sum_{e \in \delta(W) - F} x_e + \sum_{e \in F} (u_e - x_e) \geq 1.$$

In other words, we want to find a set of vertices $W$ such that we can find a set of edges $F \subseteq \delta(W)$ such that the sum $\sum_{e \in \delta(W)-F} x_e + \sum_{e \in F}(u_e - x_e)$ is small. Note that this requires edges $e$ in the boundary of $W$ to have $x$ values either close to 0 or close to $u_e$. If an edge has $x$ value close to 0, we want to put it in $\delta(W) - F$. Otherwise, we want to put it in the edge set $F$. In this way, we make the left-hand side of the blossom inequality to be small, and have a higher chance of finding a violated cut.

We can achieve this by simply first removing all the edges $e$ with $x$ values either close to 0 or close to $u_e$, and then considering the remaining connected components as our potential $W$'s. By construction, the edges in the boundary of each of the components will have $x$ values close to 0 or $u_e$.

We also need to take care of the parity of the sum $b(W) + u(F)$. If by construction the sum is odd, we do nothing. If this sum turns out to be even, we can fix it by either shifting an odd-capacity edge in $\delta(W) - F$ to $F$, or taking an odd-capacity edge out of $F$ to $\delta(W) - F$. We want to make this choice so that the increase of the left-hand side of the inequality is minimized. Hence, we can simply pick the odd-capacity edge $e \in \delta(W)$ that minimizes the value $|x_e - (u_e - x_e)|$.

For each connected component $W$, we check whether the pair $(W, F)$ obtained in this way provides a violated blossom inequality.

The heuristic algorithm discussed above is as follows:

**Algorithm 3** Our Modified Heuristic Separation Algorithm [$\alpha = 0.3$]

**Input:** A fractional solution $\bar{x}$ to a relaxation of the $b$-matching problem (1)

$G' \leftarrow (V, \{e \in E : \min(x_e, u_e - x_e) \geq \alpha\})$
ViolatedCuts $\leftarrow \emptyset$
**for each** connected component $W$ of $G'$ **do**
  $F \leftarrow \{e \in \delta(W) : u_e - x_e < x_e\}$
  **if** $b(W) + u(F)$ is even and $\exists e \in \delta(W) : u_e$ is odd **then**
    $e \leftarrow \arg\min_{e \in \delta(W): u_e \text{ is odd}} |x_e - (u_e - x_e)|$
    $F \leftarrow F \Delta \{e\}$
  **end if**
  **if** $b(W) + u(F)$ is odd and $\sum_{e \in \delta(W)-F} x_e + \sum_{e \in F}(u_e - x_e) < 1$ **then**
    ViolatedCuts $\leftarrow$ ViolatedCuts $\cup \{(W, F)\}$
  **end if**
**end for**
**return** ViolatedCuts

# 4 Speed-up Techniques

## 4.1 Techniques Implemented

To further improve the overall performance of our algorithm, we implemented the following speed-up techniques:

- Cut pool

  We store the constraints found by the separation algorithms in a cut pool. Each time we solve the LP, we check if the constraints in the cut pool are tight. We remove the constraints that are not tight for a number of consecutive iterations (by default, 30 iterations). In this way, we can reduce the size of the LP while keeping the essential constraints.

- Branch-and-cut

  Even though the blossom inequalities give an exact description of the $u$-capacitated $b$-matching polytope, for hard instances, our algorithm may need to run for a large (possibly exponential) number of iterations until it finds an integral solution. If the algorithm does not find an integral solution and the lower bound given by the LP does not change for several iterations (by default, 60 iterations), we start branching. We pick an edge $e$ such that $\bar{x}_e$ is fractional, and recursively solve two subproblems, obtained from the current one by adding the constraints: $x_e \leq \lfloor \bar{x}_e \rfloor$, $x_e \geq \lceil \bar{x}_e \rceil$. After the first branching step, whenever the lower bound does not improve for a number of iterations (by default, 3 iterations), we perform another branching step. As usual, we discard a node of the branch-and-cut tree if the optimal value of the LP is greater than the cost of the best integral solution found so far.

- Variable fixing

  We use reduced costs to shrink the range of the variables. Consider a moment during the execution of the branch-and-cut algorithm immediately after solving the LP. Suppose the optimal solution $\bar{x}$ to the LP is not integral, and let $x_f$ be a nonbasic variable with range $l_f \leq x_f \leq u_f$ and reduced cost $\pi_f \neq 0$. An arbitrary feasible solution $x$ to the current LP has cost

  $$\sum_{e \in E} c_e \bar{x}_e + \sum_{e \in E} \pi_e (x_e - \bar{x}_e) \geq \sum_{e \in E} c_e \bar{x}_e + \pi_f (x_f - \bar{x}_f) \ .$$

  Let UB be the weight of the best integral solution found so far. In the case where $\bar{x}_f = l_f$ (and $\pi_f > 0$), such a solution can only improve on that solution if

  $$\sum_{e \in E} c_e \bar{x}_e + \pi_f (x_f - l_f) < UB \ ,$$

  which yields

  $$x_f < l_f + \frac{UB - \sum_{e \in E} c_e \bar{x}_e}{\pi_f} \ .$$

  Since we need $x$ to be integral, we actually need

  $$x_f \leq \underbrace{l_f + \left\lfloor \frac{UB - \sum_{e \in E} c_e \bar{x}_e}{\pi_f} \right\rfloor}_{u'_f} \ .$$

  If $u'_f < u_f$, then we can set $u_f \leftarrow u'_f$, thus reducing the range of the variable $x_f$. A symmetric argument can be used to strengthen the lower bound $l_f$ in the case where $\bar{x}_f = u_f$ (and $\pi_f < 0$).

- Strenghtening the LP-relaxation lower bound

  Let LB denote the lower bound on the optimal value of an (integral) $b$-matching in the current node, given by the LP solver. Let the ranges of the variables be $L \leq x \leq U$. Since we are only interested in

integral solutions, the following IP gives a valid (and potentially stronger) lower bound:

$$\begin{aligned} \mathrm{LB}' = \min \quad & \sum_{e \in E} c_e x_e \\ & \sum_{e \in E} c_e x_e \geq \mathrm{LB} \\ & 2 \sum_{e \in E} x_e = b(V) \\ & L \leq x \leq U \\ & x \text{ integral} \end{aligned}$$

We consider computing $\mathrm{LB}'$ using CPLEX. If $\mathrm{LB}' > \mathrm{LB}$, the current node is then more likely to be pruned. As pointed out by Prof. Cook, it is risky to use this improved lower bound, since CPLEX is not guaranteed to find the optimal solution to this auxiliary IP. If the solution returned is not optimal, its cost may not be a valid lower bound. We have disabled the use of this auxiliary IP by default. If the user chooses to use it, we ignore the value found by CPLEX if the status code after solving the IP is `CPXMIP_OPTIMAL_TOL` (which means that the solution found is optimal within some tolerance). We only use a lower bound when the status is `CPXMIP_OPTIMAL` (which means that the solution found is optimal). Even in that case, it is important to point out that this may still not be safe, as it is possible that CPLEX incorrectly claims optimality of a solution due to precision limitations.

### 4.2 More Speed-up Techniques

In addition to the techniques we implemented as discussed above, we list below some speed-up ideas that may be helpful:

- Fine-tuning of parameters

  There are quite a few parameters we used in the program, some of which are listed in Section 5.2. Changing these parameters can lead to significantly different running times.

- Reducing the auxiliary graph

  We know the edges of weight greater or equal to 1 in the auxiliary graph constructed for the exact separation algorithm can never contribute to any violated cut. We can contract these edges before constructing the Gomory-Hu tree, thus reducing the size of the auxiliary graph.

- Post-processing the cuts returned by the separation algorithms

  After the violated cuts are returned by the separation algorithms, we may perform a local search to seek cuts that have higher violations. Furthermore, we may choose to add only a subset of those cuts to the LP, say, the ones with largest violations.

- Column generation

  Instead of starting the LP with variables for all the edges, we can start with a subset of the edges (say, the cheapest edges incident to each vertex), and add more edges as needed. For large and dense graphs this technique will be very helpful.

- Refinement of the branching/searching strategies

  We may further refine the branching and searching strategies. Using a heuristic to find a integral $b$-matching, then using its cost as an upper bound for the branch-and-cut process may be worthwhile as well.

## 5 How to Use Our Code

### 5.1 Compiling

The makefile can be used to compile our code, and has been tested on Mac machines. Before using it, the user needs to modify the paths to CPLEX.

## 5.2 Settings of the Solver

An object of the class `Settings` carries the settings that can be passed to the solver. The table below explains the different fields of such an object.

| Parameter | Possible values | Default value | Meaning |
|---|---|---|---|
| `useHeuristic` | true, false | false | use heuristic separation algorithm |
| `alpha` | $\geq 0$ | 0.3 | parameter $\alpha$ used in the heuristic |
| `useAuxiliaryIP` | true, false | false | use auxiliary IP to improve lower bounds |
| `maxIterationsAtRootBeforeBranching` | >= 1 | 60 | maximum number of iterations at the root node without improvement on the lower bound before branching |
| `maxIterationsBeforeBranching` | >= 1 | 3 | maximum number of iterations at a non-root node without improvement on the lower bound before branching |
| `maxCutPoolSize` | $\geq 1$ | 8000 | maximum number of constraints in the cut pool |
| `maxAge` | $\geq 1$ | 30 | maximum number of iterations a constraint can be slack before being dropped |
| `useEdmondsKarp` | true, false | false | use Edmonds-Karp algorithm instead of Dinitz algorithm |

## 5.3 Format of Instance Files

### 5.3.1 Bidirected Instances

The first line contains the number of vertices $n$ and the number of edges $m$. Then, each of the following $m$ lines lists the fields of an edge: the first endpoint and the sign at that vertex, the second endpoint and the sign at that vertex, the weight, and the capacity. Each of the following $n$ lines contains the demand of a vertex. Below is an example:

```
4 5
0 -1 1  1  2.5 2
0  1 2  1  3.7 1
0 -1 3 -1  2.0 1
1  1 3  1  2.5 1
2 -1 3 -1 -3.2 3
0
1
1
-1
```

### 5.3.2 Undirected Instances

The first line contains the number of vertices $n$ and the number of edges $m$. Then, each of the following $m$ lines lists the fields of an edge: the first endpoint, the second endpoint, the weight, and the capacity. Each of the following $n$ lines gives the demand of a vertex. Below is an example:

```
4 5
0 1  2.5 2
0 2  3.7 1
0 3  2.0 1
1 3  2.5 1
2 3 -3.2 3
0
1
1
1
```

## 5.4 Command-line Interface

Some of the functionalities are available via the command line. If the user launches `./main` without additional parameters, the program prints the following instructions:

```
Usage: ./main [-see below-] [prob_file]
   -b     bidirected
   -r     random instance
   -s d   random seed
   -n d   number of vertices
   -d f   density
   -h     enable heuristic
   -a f   alpha (used by heuristic)
   -A     enable auxiliary IP
   -v     disable variable fixing
   -R d   max. number of iterations at root before branching
   -B d   max. number of iterations before branching
   -c d   max. age for a cut in the cut pool
   -e     use Edmonds-Karp instead of Dinitz algorithm
```

We list below some usage examples:

- Solve a bidirected instance in the file `instance.txt`, using the heuristic separation algorithm, and using Edmonds-Karp algorithm.

  ```
  ./main -b -h -e instance.txt
  ```

- Solve a random bidirected instance, where the graph is complete and has 200 vertices. Use 37 as a seed. Set the maximum number of iterations without improvement at the root to 150.

  ```
  ./main -b -r -n 200 -s 37 -R 150
  ```

- Solve a random undirected instance with 150 vertices and density 0.4. Use 43 as a seed.

  ```
  ./main -r -n 150 -d 0.4 -s 43
  ```

## 5.5 Constructing an instance

### 5.5.1 Bidirected instances

The first step is creating a bidirected graph `G`. Below we list 4 ways in which this can be done, and give examples.

1. Create a graph, then add edges one at a time.

   ```
   BidirectedGraph G(4); // create a graph with 4 vertices
   // add an edge with endpoints 0 (sign -) and 1 (sign +), weight 2.5, and capacity 2
   G.addEdge(BidirectedEdge(0, -1, 1, 1, 2.5, 2));
   // add an edge with endpoints 0 (sign +) and 2 (sign +), weight 3.7, and capacity 1
   G.addEdge(BidirectedEdge(0, 1, 2, 1, 3.7, 1));
   ```

2. Create a random complete graph.

   ```
   // Create a random complete graph with 100 vertices, using 37 as a seed.
   // Vertex demands, edge weights, and edge capacities are chosen uniformly at
   // random in the ranges [1, 3], [10, 1000], and [1, 2] respectively.
   // Each edge has sign - or + in each of its endpoints, with equal probability.
   BidirectedGraph G = randomCompleteBidirectedGraph(37, 100, 1, 3, 10, 1000, 1, 2);
   ```

3. Create a random graph with specified density.

```
// Create a random graph with 100 vertices and density 0.5, using 37 as a seed.
// Vertex demands, edge weights, and edge capacities are chosen uniformly at
// random in the ranges [1, 3], [10, 1000], and [1, 2] respectively.
// Each edge has sign - or + in each of its endpoints, with equal probability.
BidirectedGraph G = randomBidirectedGraph(37, 100, 0.5, 1, 3, 10, 1000, 1, 2);
```

4. Read a graph from a file `instance.txt`.

```
ifstream fin(``instance.txt'');
BidirectedGraph G;
fin >> G;
fin.close();
```

After creating the graph, the user can create an instance using the default settings of the solver as follows:

```
BidirectedMatchingInstance instance(G);
```

The user may want to change the default settings (see Section 5.2 for details). They can do so as follows:

```
Settings settings;
settings.useHeuristic = true;
settings.maxAge = 30;
BidirectedMatchingInstance instance(G, settings);
```

### 5.5.2 Undirected instances

The first step is creating a graph `G`. Below we list 4 ways in which this can be done, and give examples:

1. Create a graph, then adding edges one at a time.

```
CapacitatedGraph G(4); // create a graph with 4 vertices
// add an edge with endpoints 0 and 1, weight 2.5, and capacity 2
G.addEdge(CapacitatedEdge(0, 1, 2.5, 2));
// add an edge with endpoints 0 and 2, weight 3.7, and capacity 1
G.addEdge(CapacitatedEdge(0, 2, 3.7, 1));
```

2. Create a random complete graph.

```
// Create a random complete graph with 100 vertices, using 37 as a seed.
// Vertex demands, edge weights, and edge capacities are chosen uniformly at
// random in the ranges [1, 3], [10, 1000], and [1, 2] respectively.
CapacitatedGraph G = randomCompleteCapacitatedGraph(37, 100, 1, 3, 10, 1000, 1, 2);
```

3. Create a random graph with specified density.

```
// Create a random complete graph with 100 vertices and density 0.5, using 37 as a seed.
// Vertex demands, edge weights, and edge capacities are chosen uniformly at
// random in the ranges [1, 3], [10, 1000], and [1, 2] respectively.
CapacitatedGraph G = randomCapacitatedGraph(37, 100, 0.5, 1, 3, 10, 1000, 1, 2);
```

4. Read a graph from a file `instance.txt`.

```
ifstream fin(``instance.txt'');
CapacitatedGraph G;
fin >> G;
fin.close();
```

After creating the graph, the user can create an instance using the default settings as follows:

```
CapacitatedbMatchingInstance instance(G);
```

The user may want to change the default settings of the solver (see Section 5.2 for details). They can do so as follows:

```
Settings settings;
settings.useHeuristic = true;
settings.maxAge = 30;
CapacitatedbMatchingInstance instance(G, settings);
```

### 5.6 Solving an instance

If `instance` is an instance of bidirected or undirected $b$-matching constructed as described in Section 5.5, the following commands can be used to solve it.

```
bool feasible; // (true if the instance is feasible)
vector<double> x; // optimal solution
double weight; // value of optimal solution
instance.solve(feasible, x, weight);
```

## 6 Numerical results

In this section, we present some results produced by our code on several instances. We start the section with two figures of the output of the code for pr76, a complete graph on 76 vertices. Figure 2 shows the minimum $b$-matching for pr76 with $b_v = 2$ for all vertices and $u_e = 1$ for all edges. Figure 3 shows the minimum $b$-matching for pr76 with $b_v = 3$ for all vertices and $u_e = 2$ for all edges. Black edges are those with $x_e = 2$ and red edges are those with $x_e = 1$.

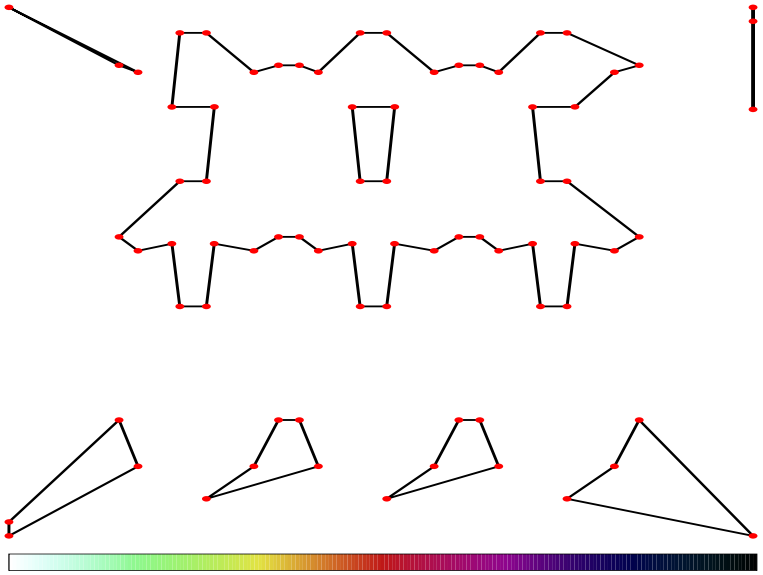


Figure 2: Minimum $b$-matching for pr76 with $b_v = 2$ for all vertices and $u_e = 1$ for all edges.

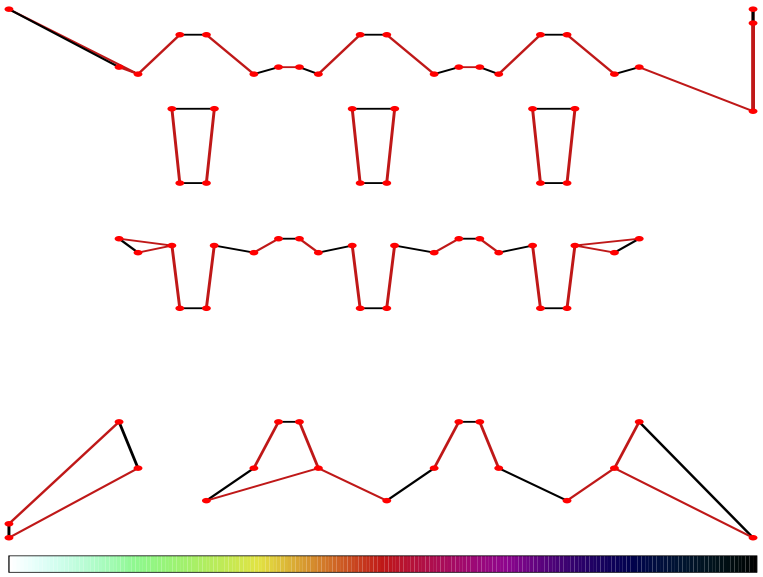


Figure 3: Minimum $b$-matching for pr76 with $b_v = 3$ for all vertices and $u_e = 2$ for all edges.

Figure 4 shows the output of our code for an instance made of Stephen Colbert's picture. Initially, the graph was a complete graph on 5952 vertices which our code could not solve. We made the graph sparser with $|V| = 5952$ and $|E| = 134505$. Before using the speed-up techniques, our code solved his problem in around 90 minutes. By using the speed-up techniques we explained in Section 4, we reduced the running time to less than 10 minutes.

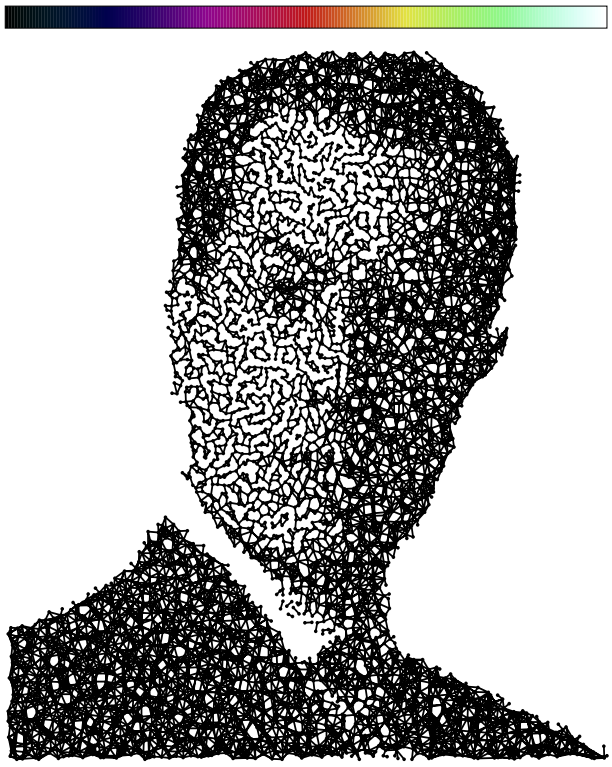
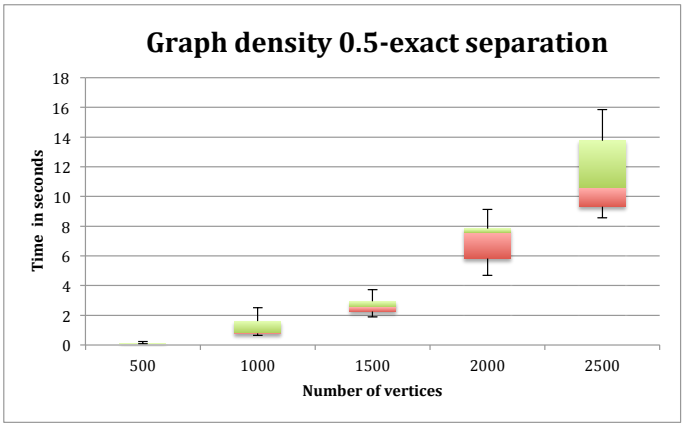


Figure 4: Picture of Colbert on a graph with $|V| = 5952$ and $|E| = 134505$.

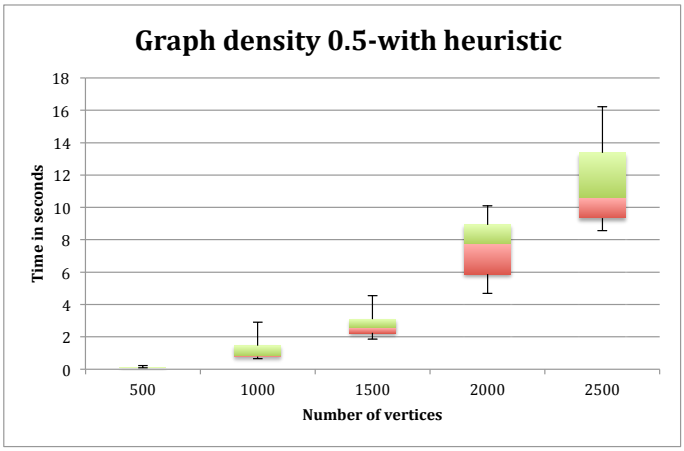
By running our code on several instances, we have some rule-of-thumb results as follows:

- The heuristic separation algorithm is fast and finds many violated cuts. However, it does not significantly improve the overall running time of the algorithm.
- Running the Gomory-Hu algorithm until the end to find all the cuts is faster compared to stopping it after finding a few cuts.
- Finding the violated cuts is fast in our code and the bottleneck is solving the LP.

Figures 5 – 7 show the box plots of the running times for heuristic and exact separation obtained for random instances on 500 – 2500 vertices, for vertex demands $b_v = 2$, edge capacities $u_e = 1$, and densities in the range 0.5–1.0.

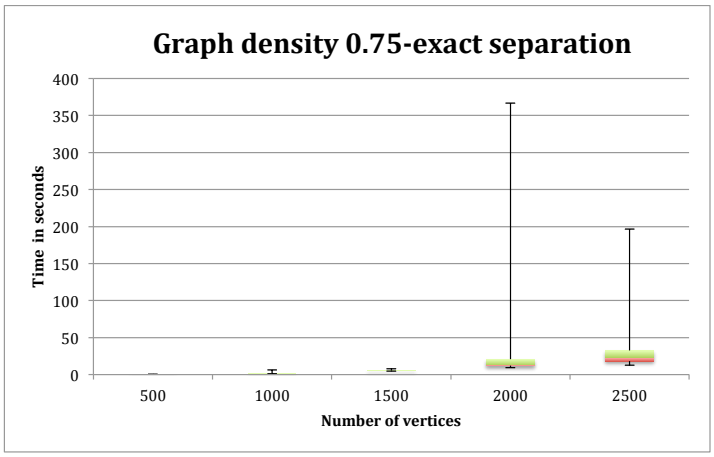



(a)

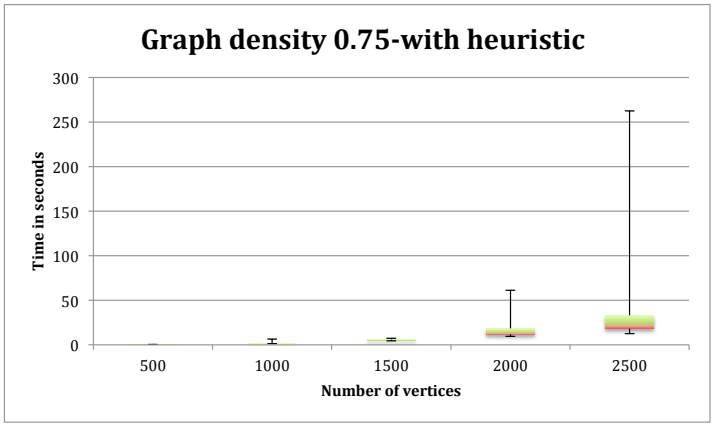



(b)

Figure 5: Results for graphs with density 0.5.

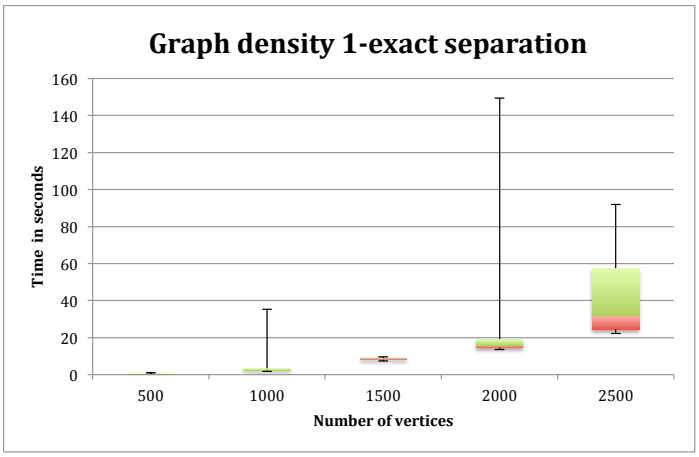



(a)

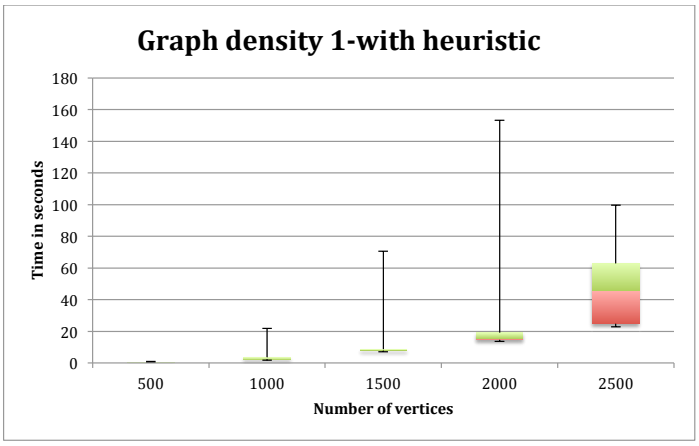



(b)

Figure 6: Results for graphs with density 0.75.




(a)




(b)

Figure 7: Results for graphs with density 1.0.

Figure 8 shows the geometric mean of the ratios of the running time of the heuristic separation algorithm and that of the exact separation algorithm for random graphs generated as described above.

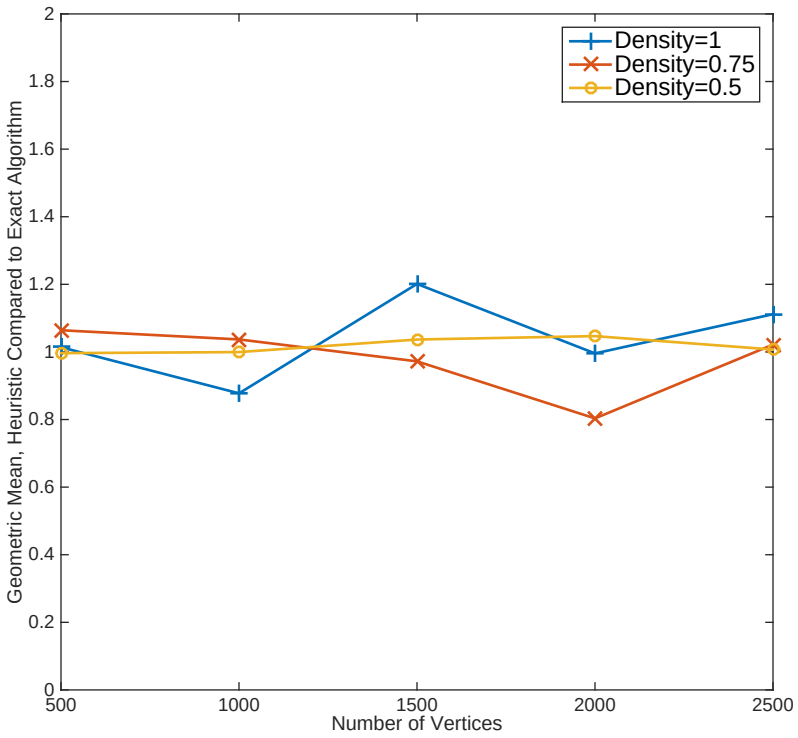



Figure 8: Comparison between heuristic and exact separation algorithms.

# References


[1] E. A. Dinitz, *Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[2] J. Edmonds, *Maximum matching and a polyhedron with 0-1 vertices*, J. Res. Nat. Bur. Standards Sect. B, (1965), pp. 125–130.

[3] J. Edmonds and E. L. Johnson, *Matching: A well-solved class of integer linear programs*, Edited by G. Goos, J. Hartmanis, and J. van Leeuwen, (1970), p. 27.

[4] J. Edmonds and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of the ACM (JACM), 19 (1972), pp. 248–264.

[5] R. E. Gomory and T. C. Hu, *Multi-terminal network flows*, Journal of the Society for Industrial & Applied Mathematics, 9 (1961), pp. 551–570.

[6] M. Grötschel and O. Holland, *A cutting plane algorithm for minimum perfect 2-matchings*, Computing, 39 (1987), pp. 327–344.

[7] D. Gusfield, *Very simple methods for all pairs network flow analysis*, SIAM Journal on Computing, 19 (1990), pp. 143–155.

[8] E. L. Lawler, *Combinatorial optimization: networks and matroids*, Courier Corporation, 1976.

[9] A. N. Letchford, G. Reinelt, and D. O. Theis, *Odd minimum cut sets and b-matchings revisited*, SIAM Journal on Discrete Mathematics, 22 (2008), pp. 1480–1487.

[10] M. W. Padberg and M. R. Rao, *Odd minimum cut-sets and b-matchings*, Mathematics of Operations Research, 7 (1982), pp. 67–80.

[11] W. R. Pulleyblank, *Faces of Matching Polyhedra*, PhD thesis, University of Waterloo, 1973.