Università degli Studi di Padova

Dipartimento di Matematica
"Tullio Levi-Civita"

Corso di Laurea Magistrale in Matematica

A computational analysis of projection-free methods

Laureando

Andrea Agnolin

Matricola

1176509

Relatore

Prof. Francesco Rinaldi

26/02/2021 - A.A. 2020/2021

Contents

1 Introduction **1**

2 Algorithms **3**
2.1 Notation and setup . . . 3
2.2 Frank-Wolfe algorithms . . . 4
2.2.1 Active-Set variants . . . 5
2.2.2 Choice of the step size . . . 7
2.3 Short Step Chain iteration . . . 8
2.3.1 Convergence of SSC method . . . 10

3 Problems **13**
3.1 Cliques . . . 13
3.1.1 Continuous formulation of MCP . . . 13
3.2 Pseudo-cliques . . . 14
3.2.1 Continuous formulation of the maximum s-defective clique problem 15

4 Numerical Analysis **17**
4.1 Experiments . . . 17
4.2 Implementation . . . 17
4.3 MCP . . . 18
4.4 MDCP . . . 22

5 Conclusions **29**

Appendix **31**
5.1 Access the code . . . 31
5.2 Notes on the implementation . . . 31

Bibliography **40**

Chapter 1

Introduction

Constrained optimization problems arise in many applications. From engineering to machine learning, from the study of biological system to the analysis of networks, many problems can be expressed as the minimization of a function f on a feasible region $\mathcal{M}$. The feasible region can be quite hard to handle. Hence, finding the projection of a vector on $\mathcal{M}$ may not be practical. Projection-free methods are aimed at solving the minimization problem without the use of the projection on the feasible region. Because of this, projection-free methods are well suited to handle structured constraints and have recently become popular in machine learning applications.

The most known projection-free method is the Frank-Wolfe algorithm [1], many variants of which have been proposed to ensure fast convergence rate towards a solution. Namely, while the classical projected gradient method has a linear convergence rate for some objective, the original Frank-Wolfe algorithm converges with a $O(1/t)$ rate [2]. Nevertheless, Frank Wolfe variants are popular algorithms in many machine learning and data science applications.

A reason that slows down Frank-Wolfe variants is the presence of "short" steps during which the iterate stops at the frontier of the feasible region. In [3] and [4] Rinaldi and Zeffiro proposed a framework for Frank-Wolfe's variants that addresses this issue. The mentioned works proves a linear asymptotical convergence rate under certain conditions.

We implement both the algorithm described in [3] and the classic version, as described in [2], of two Frank-Wolfe variants: namely, the Away-step Frank-Wolfe and Pairwise Frank-Wolfe. Thereafter, we benchmarked every variant against two cluster detection problems in networks. Our goal is to perform a numerical analysis of the actual improvements brought by the framework outlined in [3].

The problems that we considered are the Motzkin-Strauss formulations[5] of the maximum clique problem (as described in [6]) and of the maximum s-defective clique problem (as described in [7] and [8]).

The structure of our work is the following: Chapter 2 presents the algorithms used and the new Short Step Chain framework adjusted to the Frank Wolfe variants that we analyzed; Chapter 3 provides a theoretical description of the problems that we solved and presents their continuous formulations; in Chapter 4, after a brief description of the implementation choices, the results of the numerical analysis are outlined and commented. Thereafter, we present our conclusions. The appendix provides information on the code used.

Chapter 2

Algorithms

2.1 Notation and setup

In this work we analyse and benchmark a group of algorithms based on the conditional gradient method by Frank and Wolfe[1]. We apply them to solve some constrained optimization problems

$$f^* = \min_{x \in \mathcal{M}} f(x). \tag{2.1}$$

We denote with $\|\cdot\|$ and $\langle \cdot, \cdot \rangle$ the norm and the inner product of $\mathcal{M}$ respectively. We assume the domain $\mathcal{M} \subset \mathbb{R}^n$ to be compact, $f : \mathcal{M} \to \mathbb{R}$ to be continuous and differentiable and ∇f to be Lipschitz-continuous. We assume L, the Lipschitz constant of ∇f, to be positive and we call $\tilde{L}$ a positive lower bound for L.

$$L = \sup_{x,y \in \mathcal{M}} \frac{\|\nabla f(x) - \nabla f(y)\|}{\|x - y\|} \geq \tilde{L} > 0$$

We will work with problems where the domain $\mathcal{M}$ is the convex hull $\text{conv}(\mathcal{A})$ of a finite set of vertexes $\mathcal{A} = \{v : v \in \mathbb{R}^n\}$: i.e. $x \in \mathcal{M}$ if and only if (iff) $x = \sum \lambda_i v_i$ for λ such that $\lambda_i \geq 0$ and $\sum \lambda_i = 1$. We will call *vertex* each element $v \in \mathcal{A}$.

A solution x^* of the constrained optimization problem is called *global minimizer*. A *local minimizer* for f is a point x^* such that $f(x^*) \leq f(x)$ for all $x \in U$ for $U \subseteq \mathcal{M}$ neighborhood of x^*. We call $x^* \in \mathcal{M}$ a *stationary point* of f or *local solution* of (2.1) if $\langle \nabla f(x^*), x - x^* \rangle \geq 0$ for all $x \in \mathcal{M}$. Every minimizer is a stationary point. A feasible point $x^* \in \text{conv}(\mathcal{A})$ is stationary iff $\langle -\nabla f(x^*), x^* - v_i \rangle \leq 0$ for $v_i \in \mathcal{A}$.

Given a feasible point x, we call *feasible direction* in x every vector $y - x$ for $y \in \mathcal{M}$. We call *Frank-Wolfe gap* in x (*FW gap* or simply *gap* in x) the supremum of $\langle -\nabla f(x), d \rangle$ for d feasible direction in x. If $x^* \in \mathcal{M}$ is stationary, then the

Frank-Wolfe gap in x^* is zero. For $\mathcal{M} = \text{conv}(\mathcal{A})$, we have:

$$\text{FW gap}(x) = \sup_{y \in \mathcal{M}} \langle -\nabla f(x), y - x \rangle = \max_{v \in \mathcal{A}} \langle -\nabla f(x), v - x \rangle. \tag{2.2}$$

Finally, we assume that we have access to a *linear minimizer oracle* $\mathcal{LMO}$ over the feasible set: i.e. a procedure that, given a feasible direction r, returns a feasible point $x \in \mathcal{M}$ that minimizes the objective $\langle r, x \rangle$. If $\mathcal{M} = \text{conv}(\mathcal{A})$, then the linear minimizer oracle will return a point $v \in \mathcal{A}$:

$$\mathcal{LMO}(r) = \arg\min_{x \in \mathcal{M}} \langle r, x \rangle = \arg\min_{v \in \mathcal{A}} \langle r, v \rangle. \tag{2.3}$$

2.2 Frank-Wolfe algorithms

The FW algorithm produces a sequence of feasible points that converges towards a stationary point of the objective function f of problem (2.1).

Algorithm 1: Frank-Wolfe

Data: A starting feasible point $x_0 \in \mathcal{M}$
Result: A sequence $(x_t)_{t \geq 0}$ that converges to a stationary point

1. $t \leftarrow 0$
2. **repeat**
3. $\quad g \leftarrow -\nabla f(x_t)$
4. $\quad v \leftarrow \mathcal{LMO}(-g)$
5. $\quad gap \leftarrow \langle g, v - x_t \rangle$
6. $\quad d_t \leftarrow (v - x_t)$
7. $\quad$ Choose $\alpha \in (0, 1]$ that reduces $f(x_t + \alpha d_t)$
8. $\quad x_{t+1} \leftarrow x_t + \alpha d_t$
9. $\quad t \leftarrow t + 1$
10. **until** *stopping condition(t, gap)*

The algorithm starts at a given feasible point x_0, repeat the same iteration until a stop condition is met in Line 10, and for every iteration it computes a new element x_t of the converging sequence.

Each iteration of the algorithm computes the gradient for the current point x_t (Line 3) and uses the linear minimizer to find the vertex $v \in \mathcal{A}$ that minimized the product $\langle -g, v \rangle$ (Line 4). Thus, the product $\langle -g, v - x \rangle$ is the FW gap in x as defined in (2.2).

The iteration ends as we move towards the point v, but before we multiply the feasible direction $d_t = v - x_t$ by a parameter α. In Line 7, we choose α using a proper rule. A good rule is one that reduces the value of $f(x + \alpha d_t)$. We describe some criteria for the choice of α later in Subsection 2.2.2.

The stopping condition (Line 10) will check the value of the FW gap: if *gap* is close enough to zero, then the current iterate is a good approximation of a stationary point, hence the algorithm will stop. We can also stop the algorithm after a maximum number of iterations is reached or after a given amount of cpu time has elapsed.

Algorithm 1 converges rather slowly, precisely $f(x_t) - f(x^*) \leq O(1/t)$ [2][9]. When the objective f is strongly-convex, then the accumulation point x^* is the global solution of (2.1).

2.2.1 Active-Set variants

To speed up the convergence, some variants of the original Frank-Wolfe algorithm were proposed in the literature. We work with a pair of variants that use the expression of every point in $\text{conv}(\mathcal{A})$ as a convex combination of elements of $\mathcal{A}$.

Definition 1 (Active Set)**.** Let $x \in \mathcal{M}$ be a feasible point. Let $\lambda \geq 0$ be the vector of coefficients for a convex combination of x (i.e. $x = \sum_{v \in \mathcal{A}} \lambda_v v$). We call *active set* for x and the convex combination λ the set $\mathcal{S}$ of elements v_i of $\mathcal{A}$ such that the corresponding $\lambda_{v_i} > 0$:

$$\mathcal{S} = \{v \in \mathcal{A} : \lambda_v > 0\}$$

By the theorem of Carathéodory, we can always write $x \in \mathbb{R}^n$ as the convex combination of at most $n+1$ points of $\mathbb{R}^n$[10]. Thus, we can write every $x \in \mathcal{M}$ as a convex combination $x = \sum_{v \in \mathcal{S}} \lambda_v v$ with $\mathcal{S}$ active set of cardinality $|\mathcal{S}| \leq n+1$.

We will now present two variants of Algorithm 1. In these variants the algorithm keeps track a convex combination of the current point x_t and the associated active set.

The idea behind Algorithm 2 is to choose between two possible feasible directions: at each iteration we can either perform a *FW-step* following the standard Frank-Wolfe direction $v_{FW} - x_t$ or perform an *Away Step* moving away from the away vertex v_{AS}, which we compute in Line 8 as the vertex that, among the vertexes in the active set $\mathcal{S}$, maximizes the values of the local linearization of the objective function.

Keeping track of the convex combination λ allows the algorithm to compute the maximum feasible step size α such that the direction $\alpha(x_t - v_{AS})$ is feasible (Line 14).

Algorithm 2: Away Step Frank-Wolfe (ASFW)

Data: A starting feasible point $x_0 \in \mathcal{M}$
Result: An approximate stationary point

1 $t \leftarrow 0$
2 $\lambda, \mathcal{S} \leftarrow$ convex combination for x_0 and associated active set
3 **repeat**
4 $g \leftarrow -\nabla f(x_t)$
5 $v_{FW} \leftarrow \mathcal{LMO}(-g)$
6 $v_A \leftarrow \arg\max_{v \in \mathcal{S}} \langle -g, v \rangle$
7 $gap_{FW} \leftarrow \langle g, v_{FW} - x_t \rangle$
8 $gap_{AS} \leftarrow \langle g, x_t - v_A \rangle$
9 **if** $gap_{FW} \geq gap_{AS}$ **then**
10   $d_t \leftarrow v_{FW} - x_t$
11   $\alpha_{max} \leftarrow 1$
12 **else**
13   $d_t \leftarrow x_t - v_{AS}$
14   $\alpha_{max} \leftarrow \frac{\lambda_{v_{AS}}}{1-\lambda_{v_{AS}}}$
15 Choose $\alpha \in (0, \alpha_{max}]$ that reduces $f(x_t + \alpha d_t)$
16 $x_{t+1} \leftarrow x_t + \alpha d_t$
17 Update the vector of coefficient λ and the active set $\mathcal{S}$
18 $t \leftarrow t + 1$
19 **until** *stopping condition(t, gap_{FW})*

Remark 1. Let $\mathcal{S} \subseteq \mathbb{R}^n$ be a finite set and $\hat{v} \in \mathcal{S}$ one of its elements. Let $x \in \text{conv}(\mathcal{S})$ be obtained as the convex combination of coefficients $\lambda \geq 0$ of the points in $\mathcal{S}$ and $y = x + \alpha(x - \hat{v})$ for a given α. Then, for every $\alpha \in \left(0, \frac{\lambda_{\hat{v}}}{1-\lambda_{\hat{v}}}\right]$ we have $y \in \text{conv}(\mathcal{S})$.

Proof. By definition $x = \sum_{v \in \mathcal{S}} \lambda_v v$. Then $y = x + \alpha(x - \hat{v}) = (1+\alpha) \sum_{v \in \mathcal{S}} \lambda_v v - \alpha\hat{v} = (\lambda_{\hat{v}} + \alpha(\lambda_{\hat{v}} - 1))\, \hat{v} + \sum_{v \neq \hat{v}} (1+\alpha)\lambda_v v$. Let $\bar{\lambda}_v = (1+\alpha)\lambda_v$ for $v \neq \hat{v}$ element of $\mathcal{S}$, and $\bar{\lambda}_{\hat{v}} = \lambda_{\hat{v}} + \alpha(\lambda_{\hat{v}} - 1)$. For $\alpha \leq \frac{\lambda_{\hat{v}}}{1-\lambda_{\hat{v}}}$ we have $\bar{\lambda}_{\hat{v}} = \lambda_{\hat{v}} + \alpha(\lambda_{\hat{v}} - 1) \geq 0$. Hence we have $\sum \bar{\lambda} = \lambda_{\hat{v}} + \alpha(\lambda_{\hat{v}} - 1) + \sum_{v \neq \hat{v}} (1+\alpha)\lambda_v = \sum_{\lambda_v} + \alpha \sum_{\lambda_v} - \alpha = 1$. Thus, $y = \sum \bar{\lambda} v \in \text{conv}(\mathcal{S})$. □

Moving away from the away vertex v_{AS} fixes a phenomenon of zigzagging that occurs when the accumulation point of $(x_t)_{t \geq 0}$ is on a facet of $\mathcal{M}$ or close to one [2].

Another algorithm developed to address this problem is the Pairwise FW, which moves alongside the direction $v_{FW} - v_{AS}$, away from the away vertex, towards the vertex returned by the linear minimizer oracle. In this case, the maximum step size is $\lambda_{v_{AS}}$. The proof if similar to the previous one.

Algorithm 3: Pairwise Frank-Wolfe (PFW)

Data: A starting feasible point $x_0 \in \mathcal{M}$
Result: An approximate stationary point
1 $t \leftarrow 0$
2 $\lambda, \mathcal{S} \leftarrow$ convex combination for x_0 and associated active set
3 **repeat**
4 $g \leftarrow -\nabla f(x_t)$
5 $v_{FW} \leftarrow \mathcal{LMO}(-g)$
6 $v_A \leftarrow \arg\max_{v \in \mathcal{S}} \langle -g, v \rangle$
7 $gap_{FW} \leftarrow \langle g, v_{FW} - x_t \rangle$
8 $d_t \leftarrow v_{FW} - v_{AS}$
9 $\alpha_{max} \leftarrow \lambda_{v_{AS}}$
10 Choose $\alpha \in (0, \alpha_{max}]$ that reduces $f(x_t + \alpha d_t)$
11 $x_{t+1} \leftarrow x_t + \alpha d_t$
12 Update the vector of coefficient λ and the active set $\mathcal{S}$
13 $t \leftarrow t + 1$
14 **until** *stopping condition(t, gap_{FW})*

2.2.2 Choice of the step size

There are many possible criteria for the choice of α in Line 7 of Algorithm 1.

Ideally, the choice of α should minimize the linear function $f(x_t + \alpha d_t)$ with respect to α, although the use of exact methods to find such α is not often the more practical way. Alternative methods to get α include:

1. a constant step size, i.e. $\alpha = k \in (0, 1]$;

2. the Armijo line search[11], described in [12] as

   Algorithm 4: Armijo line search

   Data: The upper bound α_{max}, two parameters $\delta, \gamma \in (0, 1)$
   1 $\alpha \leftarrow \alpha_{max}$
   2 **while** $f(x_k + \alpha d_k) > f(x_k) + \gamma\alpha \langle \nabla f(x_k), d_k \rangle$ **do**
   3 $\alpha \leftarrow \delta\alpha$

   this methods requires to compute the value of the objective function at every iteration of the *while* statement;

3. a step based on the property of the objective f:

$$\alpha = \frac{\langle g, d_t \rangle}{L \left\| d_t \right\|^2} \tag{2.4}$$

this step length criterion is useful to estimate convergence rate and complexity, like in [13], and does not require to compute the objective function again.

We benchmarked the algorithm using the last criterion for Line 7. We used, however, a local lower bound $\tilde{L}$ instead of L.

2.3 Short Step Chain iteration

Within each iteration of the ASFW and PFW algorithms, the steps taken along the descending direction are constrained to remain inside the feasible region. Thus, some steps might be "short" ones, i.e. steps in which $\alpha = \alpha_{max}$ because the iterate x_t reaches the frontier of the feasible set. This implies that some short iterations might "waste" the computational resources to compute again the gradient and linear minimizer even though these have not significantly changed. To address the presence of such "bad steps", a variant of the algorithms described above has been proposed [3][4]. In the following, we omit the instructions that track the convex combination and active set needed by the ASFW and PWF methods. We will assume that the iterate x_t (or y_j) comes together with such information.

The core of the aforementioned method is the *Short Step Chain (SSC)* procedure. Let us call $d(y, g, v_{FW})$ the direction that one of the FW variants compute for y current iterate, g opposite of the gradient, v_{FW} vertex returned by the linear minimizer. In our experiments, $d(y, g, v_{FW})$ will be the direction computed either in Line 8 of Algorithm 3 or in Line 10 and Line 13 of Algorithm 2. Let $\alpha_{max}(y, d)$ be the maximum feasible α in y: either the one computed in Line 9 of Algorithm 3 or the one from Algorithm 2.

This chain of short steps stops either when a null direction is to be followed (Line 7), when y_j reachs the frontier of the feasible region (Line 11), or when the maximum feasible step length α^j_{max} is shorter than a parameter β_j. The latter allows us to know that y_j is close enough to $\bar{x}$ to follow a descent path without the need of a new computation of g and v_{FW}. This parameter $\beta_j = \beta_{max}(g, \bar{x}, d_j, y_j)$ is the maximum β that satisfies a condition that takes into account the property of the gradient of f (namely its Lipschitz constant) and the direction d_j.

Condition 1 (Descent sequence condition)**.** Let y_j and d_j be the current iterate and current direction respectively of the *SSC* procedure. Let $\bar{x}$ be the FW-iterate and $g = -\nabla f(\bar{x})$. We say $\beta \geq 0$ satisfy the descent sequence condition if either:

$$\beta = 0 \qquad \vee \qquad \begin{cases} \|(y_j + \beta d_j) - \bar{x}\| \leq \frac{\langle g, d_j \rangle}{L\|d_j\|} \\ \left\|(y_j + \beta d_j) - \left(\bar{x} + \frac{g}{2L}\right)\right\| \leq \frac{\|g\|}{2L} \end{cases} . \tag{2.5}$$

Algorithm 5: Short step chain

Data: $\bar{x}$ current iterate of FW, $g = -\nabla f(\bar{x})$, v_{FW} result of $\mathcal{LMO}(\nabla f(\bar{x}))$

1 $j \leftarrow 0$
2 $y_0 \leftarrow \bar{x}$
3 **repeat**
4 $\quad d_i \leftarrow d(y_j, g, v_{FW})$
5 $\quad \alpha^j_{max} \leftarrow \alpha_{max}(y_j, d_j)$
6 $\quad$ **if** $d_j = 0$ **then**
7 $\quad\quad$ **return** y_j
8 $\quad \beta_j \leftarrow \beta_{max}(g, \bar{x}, d_j, y_j)$, max β that respects condition Condition 1
9 $\quad \alpha_j \leftarrow \min\{\alpha^j_{max}, \beta_j\}$
10 $\quad$ **if** $\alpha_j = 0$ **then**
11 $\quad\quad$ **return** y_j
12 $\quad y_{j+1} \leftarrow y_j + \alpha_j d_j$
13 $\quad$ **if** $\beta_j \leq \alpha^j_{max}$ **then**
14 $\quad\quad$ **return** y_{j+1}
15 $\quad j \leftarrow j + 1$
16 **until** *a value is returned*

Remark 2. $\beta_{max}(g, \bar{x}, d_j, y_j)$ can be easily computed. Let $s = \bar{x} - y_j$, $z = \bar{x} + \frac{g}{2L} - y_j$, $r_1 = \frac{\|g\|}{2L}$, $r_2 = \frac{\langle g, d_j \rangle}{L\|d_j\|}$, then:

$$\beta_{max}(g, \bar{x}, d_j, y_j) = \begin{cases} 0 & \text{if } \|s\| \geq r_2 \\ \min\left\{ \begin{array}{c} \frac{\langle z, d_j \rangle + \sqrt{\langle z, d_j \rangle^2 + \|d_j\|^2 (r_1^2 - \|z\|^2)}}{\|d_j\|^2}, \\ \frac{\langle s, d_j \rangle + \sqrt{\langle s, d_j \rangle^2 + \|d_j\|^2 (r_2^2 - \|s\|^2)}}{\|d_j\|^2} \end{array} \right\} & \text{otherwise.} \end{cases}$$

Proof. The Condition (2.5) is equivalent to say that $y_j + \beta d_j \in B_{r_1}(\bar{x} + \frac{g}{2L}) \cap B_{r_2}(\bar{x})$ intersection of two closed balls. We call $B_1 = B_{r_1}(\bar{x} + \frac{g}{2L})$ and $B_2 = B_{r_2}(\bar{x})$.
If $\|s\| \geq r_2$, then $\|y_j - \bar{x}\| \geq \frac{\langle g, d_j \rangle}{L\|n_j\|}$, hence the condition 2.5 can hold only if $\beta = 0$.
Otherwise, $y_j \in B_2$. Since β_k satisfies the descent condition for $k \leq j - 1$, then whenever $y_{k-1} \in B_1$ we have $y_{k-1} + \beta d_{k-1} \in B_1$ and thus $y_k \in B_1$. Since $y_0 = \bar{x} \in B_1$ then $y_j \in B_1$.
$y_j \in B_1 \cap B_2$, so, for each ball, the maximum β such that $y_i + \beta d_j$ is inside that ball is that for which $y_i + \beta d_j$ is on the frontier. Let β_1, β_2 such that $\left\|(y_j + \beta_1 d_j) - (\bar{x} + \frac{g}{2L})\right\| = r_1$ and $\|(y_j + \beta_1 d_j) - \bar{x}\| = r_2$, then $\beta_{max} = \min\{\beta_1, \beta_2\}$.
We first look at B_1. $r_2^2 = \left\|(y_j + \beta_1 d_j) - (\bar{x} + \frac{g}{2L})\right\|^2 = \|z - \beta_1 d_j\|^2 = \|z\|^2 +$

$\beta_1^2 \|d_j\|^2 - 2\beta_1 \langle z, d_j \rangle$ leads to the equation $\beta_1^2 \|d_j\|^2 - 2\beta_1 \langle z, d_j \rangle + \|z\|^2 - r_2^2 = 0$ whose only positive solution is $\beta_1 = \frac{\langle z,d_j\rangle + \sqrt{\langle z,d_j\rangle^2 + \|d_j\|^2 (r_1^2 - \|z\|^2)}}{\|d_j\|^2}$ because $r_1 \geq \|z\|$. In the same way, we obtain $\beta_2 = \frac{\langle s,d_j\rangle + \sqrt{\langle s,d_j\rangle^2 + \|d_j\|^2 (r_2^2 - \|s\|^2)}}{\|d_j\|^2}$. □

If $j = 0$, then $y_0 = \bar{x}$ and so $\beta_{max}(g, \bar{x}, d_0, \bar{x}) = \frac{\langle g, d_0\rangle}{L\|d_0\|^2}$ which is equal to the step size in (2.4) for "classic" FW algorithms.

As in classic algorithms we will use $\tilde{L}$ instead of L.

Algorithm 6: Frank-Wolfe with SSC version

Data: A starting feasible point $x_0 \in \mathcal{M}$
Result: An enstimate of a stationary point

1 $t \leftarrow 0$
2 **repeat**
3 $\quad g \leftarrow -\nabla f(x_t)$
4 $\quad v_{FW} \leftarrow \mathcal{LMO}(-g)$
5 $\quad gap_{FW} \leftarrow \langle g, v_{FW} - x_t \rangle$
6 $\quad x_{t+1} \leftarrow SSC(x_k, g, v_{FW})$
7 $\quad t \leftarrow t + 1$
8 **until** *stopping condition(t, gap_{FW})*

In the SSC version of the Frank-Wolfe algorithm, the gradient and the minimizer are computed and then the *SSC* procedure is called to obtain the next point.

2.3.1 Convergence of SSC method

In [3], some convergence properties for Algorithm 6 were proven. Namely, if we call:

$$\pi(\bar{x}, g) = \max \left\{ 0, \sup_{y \in \mathcal{M} \setminus \{\bar{x}\}} \left\langle g, \frac{y - \bar{x}}{\|y - \bar{x}\|} \right\rangle \right\} \qquad \bar{x} \in \mathcal{M}, g \in \mathbb{R}^n$$

then the following theorem holds:

Theorem 1. *Let $\mathcal{X}$ be the set of stationary point of Problem* (2.1). *Assume that for every point $x^* \in \mathcal{X}$, exists $\delta_{x^*} > 0$, $\eta_{x^*} > f(x^*)$ and $M_{x^*} > 0$ such that: for $\varphi_{x^*}(t) = 2M_{x^*} t^{\frac{1}{2}}$, if $x \in B_{\delta_{x^*}}(x^*)$ and $\eta_{x^*} > f(x) > f(x^*)$, then the Kurdyka-Łojiasiewicz inequality*

$$\varphi'\big(f(x) - f(x^*)\big)\pi(x, -\nabla f(x)) \geq 1$$

holds for $\tilde{f} = f + \chi^{\mathcal{M}}$.
Then, it exists $\tau \in (0, 1)$ *and* $M > 0$ *such that, if* $(x_t)_{t \geq 0}$ *is the sequence generated by Algorithm 6 with the ASFW or PFW directions:*

- *the SSC procedure (Algorithm 5) terminates in a finite number of steps;*
- x_t *converges to an element* $x^* \in \mathcal{X}$ *at an asymptotic rate:*

$$\|x_t - x^*\| = O\left(\left(1 + \frac{L^3(1+\tau)^2}{M^2\tau^2}\right)^{-k/2}\right).$$

Moreover, τ *depends only on* $\mathcal{M}$ *and on the choice of the direction (either the ASFW or the PFW one), while* M *depends only on* f *and* $\mathcal{M}$.

Proof. In [3], see Proposition 6.8 and Corollary 6.1 with respect to the ASFW and PFW methods and $\mathcal{M} = \operatorname{conv}(\mathcal{A})$. □

Remark 3. If the objective function f is quadratic, i.e. $f = \frac{1}{2}x^T Q x - b^T x$, then the hypothesis of Theorem 1 is satisfied.

Proof. See corollary 6.1 in [3]. □

These results ensure us a linear convergence under a regularity hypothesis. One of the problems that we considered in our numerical experiments is quadratic and meets the hypothesis. The second one is a cubic problem that will be separated into two quadratic ones.

Chapter 3

Problems

3.1 Cliques

We introduce the problems that were solved in the numerical experiments.

Let $G = (\mathcal{V}, \mathcal{E})$ be a simple graph, with $\mathcal{V} = \{1, 2, \ldots, n\}$ sets of its nodes and $\mathcal{E} = \{\{i, j\} : i, j \in \mathcal{V}\}$ set of its edges. Given $S \subseteq \mathcal{V}$, we denote by $G[S]$ and $\mathcal{E}[S]$ the sub-graph of G and set of edges restricted to S, i.e. $G[S] = (S, \mathcal{E}[S]) = (S, \{\{i, j\} \in \mathcal{E} : i, j \in S\})$.

A *clique* in G is a subset of vertexes $C \subseteq \mathcal{V}$ that produces a sub-graph which is complete, i.e. $\{i, j\} \in \mathcal{E}\ \forall i, j \in C$. Equivalently, $G[C]$ has $\binom{|C|}{2}$ edges. We denote by $\omega(G)$ the cardinality of the largest clique in G. The *maximum clique problem (MCP)* is the following: given G find C such that the cardinality $|C| = \omega(G)$.

The MCP is NP-hard. Nonetheless, a continuous formulation exists and we applied the conditional gradient algorithm to find a local minimum for this formulation in order to obtain an approximate MCP's solution. Our solutions will be *maximal cliques*, i.e. they are not contained in any other larger clique.

3.1.1 Continuous formulation of MCP

We denote with $e \in \mathbb{R}^n$ the vector of ones (i.e. $(e)_i = 1\ \forall 1 \leq i \leq n$), with $A(G)$ the adjacency matrix for G (i.e. $(A(G))_{ij} = 1$ if $\{i, j\} \in \mathcal{E}$, $(A(G))_{ij} = 0$ otherwise), and with $\Delta = \{x \in \mathbb{R}^n : \langle e, x \rangle = 1, x \geq 0\}$ the n-dimensional simplex.

We recall the following theorem by Motzkin and Straus[5]:

Theorem 2.

$$\max_{x \in \Delta} x^T A(G) x = \left(1 + \frac{1}{\omega(G)}\right) \tag{3.1}$$

And, for every C such that $|C| = \omega(G)$, we have that $x \in \Delta$, with $x_i = \frac{1}{\omega(G)}$, is a maximizer.

Proof. By induction on $n = |\mathcal{V}|$, see [5]. □

This theorem suggests a path to the solution of the MCP by finding a maximum for $x^T A(G)x$. Two problems arise: firstly, $A(G)$ is not a semi-definite matrix so we might get with a local minimum instead of a global one; moreover, some local minima are spurious solutions that do not represent any clique of G. In [6] some regularizers are considered to address these issues.

We consider the following problem for $\gamma \in (0,1)$:

$$\max_{x\in\Delta} f(x) = x^T A(G)x + \gamma \|x\|^2. \tag{3.2}$$

Remark 4. If $C \subseteq \mathcal{V}$ is a maximal clique and $x \in \Delta$, such that $x_i = \frac{1}{|C|}$ for $i \in C$, then $xA(G)d \leq 0$ for every d feasible direction from x.

Remark 5. A point $x \in \Delta$ is a local maximizer of f in (3.2) iff $x_i = \frac{1}{|C|}$ for $i \in C$ for some maximal clique C.

Proof. See lemma 2.3 and proposition 2.5 in [6] and choose $\Phi(x) = \gamma \|x\|^2$. □

If we choose f as above and use our constrained optimization methods to find a local maximizer, then $S = supp(x) = \{i \in \mathcal{V} : x_i \neq 0\}$ is a maximal clique for G.

3.2 Pseudo-cliques

In some applications finding the maximum clique is not the more practical solution. Two alternative clique relaxations or pseudo-cliques were defined in [7] to address cluster detection problems in networks.

Definition 2. Given $G = (\mathcal{V}, \mathcal{E})$, $S \subseteq \mathcal{V}$ and $s \in \mathbb{N}$, we say that:

- S is a *s-defective clique* if $G[S]$ has at least $\binom{|S|}{2} - s$ edges, i.e. S would be a clique by including s missing egdes at most;

- S is a *s-plex* if each vertex is adjacent to at least $|S| - s$ other vertexes in S.

A clique is a 0-defective clique and a 1-plex.

We focused on the the maximum s-defective clique problem (*MDCP*).

Let $\omega_s^d(G)$ denotes the maximum cardinality of a s-defective clique in G. We call maximal a s-defective clique that is not contained in any other larger s-defective clique. We follow the variation of the Motzkin-Strauss(*MS*) formulation for proposed in [7].

3.2.1 Continuous formulation of the maximum s-defective clique problem

Let $\bar{\mathcal{E}}$ be the complementary of $\mathcal{E}$ and $\bar{G} = (\mathcal{V}, \bar{\mathcal{E}})$ the complementary graph of G. Let $m = |\bar{\mathcal{E}}|$ be the number of edges missing in G. We introduce a new variable $y \in \mathbb{R}^m$ such that for every edge $\{i,j\} \in \bar{\mathcal{E}}$ we have $y_{ij} \in [0,1]$. We short $A_G = A(G)$ and we denote by $A_{\bar{G}}(y)$ the $n \times n$ matrix in which $(A_{\bar{G}}(y))_{ij} = 0$ if $\{i,j\} \in \mathcal{E}$ and $(A_{\bar{G}}(y))_{ij} = y_{ij}$ if $\{i,j\} \in \bar{\mathcal{E}}$. Let:

$$D_{s,m} = \{y \in [0,1]^m : \langle e_m, y \rangle \leq s\}. \tag{3.3}$$

We consider the following problem:

$$\max_{(x,y) \in \Delta \times D_{s,m}} f(x,y) = x^T (A_G + A_{\bar{G}}(y))x. \tag{3.4}$$

Theorem 3. *Let (x^*, y^*) be a solution of (3.4), then:*

$$x^{*T}(A_G + A_{\bar{G}}(y^*))x^* = \max_{(x,y) \in \Delta \times D_{s,m}} x^T (A_G + A_{\bar{G}}(y))x = 1 - \frac{1}{\omega_s^d(G)}.$$

Moreover, for S maximum s-defective clique, (x^*, y^*) such that:

$$x_i^* = \begin{cases} \frac{1}{|S|} & i \in S \\ 0 & i \notin S \end{cases} \qquad y_{ij}^* = \begin{cases} 1 & \{i,j\} \in \bar{\mathcal{E}} \wedge i,j \in S \\ 0 & \textit{otherwise} \end{cases}$$

is a solution of (3.4).

These results, whose proof can be found in [7], provide a way to describe the *MDCP* in a continuous formulation which, nevertheless, is affected by the same problems of the original *M.-S. MCP* formulation. Namely, using our methods we will not find a global solution and these local solutions might not be associated to maximal cliques. So, we consider this problem instead:

$$\max_{(x,y) \in \Delta \times D_{s,m}} x^T (A_G + A_{\bar{G}}(y))x + \gamma \|x\|^2 + \frac{\mu}{2} \|y\|^2 \tag{3.5}$$

Given an s-defective clique $S \subseteq \mathcal{V}$, we denote by $s[S]$ the number of *actually missing edges*, i.e. $s[S] = \binom{|S|}{2} - |\mathcal{E}[S]| = |\bar{\mathcal{E}}[\mathcal{S}]|$.

Theorem 4. *Assume $\gamma \in (0,2)$ and $\mu > 0$. $(x^*, y^*) \in \Delta \times D_{s,m}$ is a local maximizer for (3.5) iff, for S s-defective clique, we have $x_i^* = \frac{1}{|S|}$ for $i \in S$, $y_e \in \{0,1\}$, $\langle e, y^* \rangle = s$ and $y_{ij} = 1$ for $\{i,j\} \in \bar{\mathcal{E}}[S]$.*
Moreover, S is a maximal clique in $G' = (\mathcal{V}, \mathcal{E} \cap supp(y))$ and S is a $s[S]$-defective maximal clique in G.

Proof. The proof of the first equivalence can be found in [8].
We can prove that S is a maximal $s[S]$-defective clique by contradiction. □

The Theorem 4 does not guaranties that the s-defective cliques associated to local minima are maximal.

An issue with this formulation is the big dimension of the feasible region $\Delta \times D_{s,m}$. This issues will be addressed in Section 4.4.

Chapter 4

Numerical Analysis

4.1 Experiments

We now describe the numerical experiments conducted and their results.

The algorithms were implemented using python code with the numpy libraries[1] and the scripts run on a HP 15-da0xxx series notebook featuring an Intel i7-7500U processor and 16GB of ddr4 memory.

In the experiments conducted, we considered a set of standard graphs for benchmark and ran our code to solve the *MCP* and the *MDCP* with $s \in \{5, 20, 50\}$. The graphs are a subset of those proposed in the DIMACS maximum-clique implementation challenge[14][15]. For each instance of the considered problems (that is: for each graph with respect to the *MCP*; for each graph and for each s with respect to the *MDCP*) 10 feasible points are chosen randomly and passed to our algorithm as starting points.

4.2 Implementation

The feasible region $\text{conv}(\mathcal{A})$ is implemented through a matrix-like object that at the i-th row returns the i-th element of $\mathcal{A} = \{v_i : i \in I\}$.

The convex combination associated to a point x_t is described by an object that keeps tack of the indexes of the vertexes and coefficients λ of the combination. Moreover, it also keeps track of a matrix A of the active set. At each iteration of the algorithms, the convex combination object is modified and x_{t+1} is computed as the product $A\lambda$.

We logged the cpu time used in every iteration and in the whole execution on a given test with the `time.process_time` function.

[1] Python version: 3.8.6. Libraries versions: numpy v1.19.5; scipy v1.4.1; bottleneck v.1.3.2

Starting point

The starting points are generated randomly through the `numpy.random` library. For a given instance of the problem, to generate the i-th starting point the function `numpy.random.seed(`i`)` is called.

Lipschitz constant

We prove a lower bound $\tilde{L}$ for the Lipschitz-constant L of $\nabla f(x)$:

Remark 6. Let $x_t, x_{t+1} \in \mathcal{M}$, then, we have $\hat{L}(x_t, x_{t+1}) \leq L$ for:

$$\hat{L}(x_t, x_{t+1}) = 2\frac{f(x_{t+1}) - f(x_t) - \langle \nabla f(x_t), x_{t+1} - x_t \rangle}{\|x_t - x_{t+1}\|^2}.$$

Proof. Let $h : [0,1] \to \mathbb{R}^n$ be such that $h(\alpha) = \nabla f(x+\alpha d) - \nabla f(x)$, for $d = x_{t+1} - x_t$. By definition of L, $\|h(\alpha)\| \leq \|\alpha d\| \, L = \alpha L \|d\|$. Thus, $\langle h(\alpha), d \rangle \leq \alpha L \|d\|^2$. We consider $f(x_{t+1}) - x_t = \int_0^1 \langle \nabla f(x_t + \alpha d), d \rangle \, d\alpha = \int_0^1 \langle \nabla f(x_t) + h(\alpha), d \rangle \, d\alpha \leq \int_0^1 \langle \nabla f(x_t), d \rangle \, d\alpha + \int_0^1 \alpha L \|d\|^2 \, d\alpha = \langle \nabla f(x_t), d \rangle + \frac{1}{2} L \|d\|^2$. The thesis follows. □

In our algorithm, we will keep track of a lower bound $\tilde{L}$ for L. Before the first iterate, we will set $\tilde{L} = \hat{L}(x_{\mathcal{M}}, x_0)$ where $x_{\mathcal{M}}$ is a feasible point fixed for every test. After the t-th iteration of the algorithm, we will update $\tilde{L}$ by setting $\tilde{L} = \max\{\tilde{L}, \hat{L}(x_t, x_{t+1})\}$.

In the *MDCP* experiment, we encountered a problem: for the first few iterations $\hat{L}$ was very high. This would had slow down the following iteration. To address this we capped $\tilde{L}$ at 100.

4.3 MCP

In the first experiment, we consider the following formulation of problem (2.1):

$$\min_{x \in \Delta} f(x)$$

with the following objective function:

$$\begin{aligned} f(x) &= -x^T A(G) x - \frac{1}{2} \|x\|^2 \\ \nabla f(x) &= -\left(2A(G) + 1_n\right) x. \end{aligned}$$

This is a formulation of (3.2) as a minimization problem.

We run the *SSC* and "classic" versions of both the Away-Step FW and Pairwise FW algorithm to find x^* approximate local solution. Given such x^*, as proven in

Remark 5, $C = supp(x^*)$ is a maximal clique. We choose as a stopping condition the following: the number of iteration $t = 10^4$ or the gap is lower than $\epsilon = 10^{-6}$. After the algorithms have ended, we checked $|\mathcal{E}[C]| = \binom{|C|}{2}$ to be sure that the C is indeed a clique.

The vertexes $\mathcal{A}$ of the simplex Δ are the unitary vectors e^i. As such, the linear minimizer oracle $\mathcal{LMO}(g)$ was implemented by simply selecting the lower coordinate in g. In fact, $\langle e^i, g\rangle = (g)_i$.

Each one of the 10 random feasible starting points were chosen as $x_0 = \frac{w}{\langle e,w\rangle}$ for a vector $w \in [0,1]^n$ randomly generated with the `numpy.random.rand` function.

Table 4.1: Results of the *MCP* tests

Instance	Algorigthm	clique number $\lvert C\rvert$				cpu time			
		min	mean	max	std	min	mean	max	std
brock200_2	Classic Away-step	6	7.8	9	0.75	0.01	0.25	0.36	0.12
	Classic Pairwise	7	8.4	10	0.80	0.25	0.30	0.36	0.04
	SSC Away-step	6	6.7	7	0.46	0.01	0.02	0.05	0.01
	SSC Pairwise	6	7.8	10	1.25	0.38	0.43	0.50	0.04
brock200_4	Classic Away-step	12	13.1	14	0.70	0.28	0.34	0.40	0.05
	Classic Pairwise	12	13.2	14	0.75	0.26	0.31	0.36	0.04
	SSC Away-step	12	13.7	15	0.90	0.79	1.19	2.17	0.37
	SSC Pairwise	12	12.6	14	0.66	0.46	0.54	0.59	0.04
brock400_2	Classic Away-step	19	20.3	22	0.78	0.80	1.02	1.42	0.18
	Classic Pairwise	19	21.3	23	1.35	0.72	0.94	1.18	0.15
	SSC Away-step	17	19.4	20	0.92	1.19	1.99	4.14	0.93
	SSC Pairwise	19	20.2	22	0.87	1.06	1.34	1.71	0.18
brock400_4	Classic Away-step	19	19.7	21	0.78	0.89	1.32	1.99	0.31
	Classic Pairwise	19	20.6	23	1.20	0.82	1.14	1.65	0.28
	SSC Away-step	18	20.3	22	1.10	2.48	3.18	4.53	0.70
	SSC Pairwise	20	20.9	23	0.94	1.32	1.49	1.75	0.15
brock800_2	Classic Away-step	13	15.3	17	1.27	5.08	6.52	9.19	1.29
	Classic Pairwise	15	16.5	19	1.36	4.86	5.90	7.23	0.75
	SSC Away-step	15	16.5	18	0.92	5.15	6.98	10.37	1.39
	SSC Pairwise	15	16.5	18	0.81	3.93	4.78	5.95	0.67
brock800_4	Classic Away-step	14	16.0	18	1.10	5.17	7.17	8.86	1.17
	Classic Pairwise	15	16.2	18	1.25	4.76	6.70	8.37	0.98
	SSC Away-step	13	15.3	16	0.90	5.98	11.24	15.92	2.92
	SSC Pairwise	15	16.1	18	0.94	3.96	5.56	7.33	0.92
C125.9	Classic Away-step	28	30.4	32	1.50	0.32	0.35	0.40	0.03
	Classic Pairwise	26	29.7	32	1.73	0.21	0.27	0.31	0.03
	SSC Away-step	28	30.2	33	1.54	0.68	1.22	1.93	0.40
	SSC Pairwise	27	29.3	31	1.27	0.48	0.57	0.67	0.06
C250.9	Classic Away-step	37	39.0	43	2.19	0.34	0.67	0.84	0.12
	Classic Pairwise	37	38.4	40	1.20	0.53	0.59	0.67	0.05
	SSC Away-step	36	38.4	41	1.36	1.35	1.47	1.97	0.18
	SSC Pairwise	37	39.1	43	1.76	1.05	1.21	1.37	0.10
C500.9	Classic Away-step	46	48.2	52	1.60	1.45	1.77	2.94	0.47
	Classic Pairwise	44	48.3	52	2.28	1.25	1.50	2.65	0.47
	SSC Away-step	45	48.4	53	2.20	1.85	2.25	3.63	0.54
	SSC Pairwise	45	47.2	49	1.33	1.70	1.89	2.15	0.13
C1000.9	Classic Away-step	51	53.6	57	1.85	9.48	9.64	9.78	0.08
	Classic Pairwise	52	55.7	59	1.95	7.91	8.05	8.35	0.13
	SSC Away-step	51	54.3	58	2.37	8.35	8.54	8.68	0.11
	SSC Pairwise	53	56.0	59	1.48	7.65	7.92	8.64	0.32
C2000.5	Classic Away-step	10	11.9	13	0.94	0.18	42.95	69.48	28.12
	Classic Pairwise	11	12.1	14	0.94	48.18	50.35	57.36	3.21
	SSC Away-step	11	11.6	12	0.49	0.21	4.75	44.41	13.22
	SSC Pairwise	10	12.0	13	0.89	40.56	42.38	46.78	1.89

continues...

Table 4.1: (continues)

Instance	Algorigthm	clique number $\|C\|$				cpu time			
		min	mean	max	std	min	mean	max	std
C2000.9	Classic Away-step	57	60.5	64	2.01	61.08	64.92	77.21	4.81
	Classic Pairwise	61	63.1	65	1.30	48.91	51.82	58.13	3.09
	SSC Away-step	59	61.8	64	1.78	46.21	52.43	85.66	11.58
	SSC Pairwise	59	63.1	66	2.02	43.03	47.17	54.22	2.76
DSJC500_5	Classic Away-step	8	9.1	11	0.70	0.02	0.40	1.32	0.59
	Classic Pairwise	9	10.4	11	0.80	1.22	1.25	1.34	0.04
	SSC Away-step	9	9.0	9	0.00	0.03	0.04	0.06	0.01
	SSC Pairwise	8	9.7	11	0.90	1.32	1.40	1.49	0.06
DSJC1000_5	Classic Away-step	9	10.4	12	1.02	0.06	6.00	9.10	3.89
	Classic Pairwise	10	10.7	12	0.64	7.31	7.42	7.78	0.16
	SSC Away-step	9	9.0	9	0.00	0.08	0.10	0.17	0.03
	SSC Pairwise	9	11.0	13	1.10	6.62	6.74	7.01	0.10
gen200_p0.9_44	Classic Away-step	35	36.2	38	0.87	0.37	0.40	0.49	0.05
	Classic Pairwise	32	34.2	37	1.47	0.29	0.30	0.32	0.01
	SSC Away-step	33	35.3	39	1.85	0.77	1.19	1.65	0.28
	SSC Pairwise	32	34.2	37	1.40	0.57	0.62	0.67	0.03
gen200_p0.9_55	Classic Away-step	35	37.8	44	2.36	0.36	0.37	0.41	0.01
	Classic Pairwise	35	37.1	39	1.37	0.30	0.32	0.41	0.03
	SSC Away-step	35	37.2	39	1.08	0.82	1.36	1.90	0.35
	SSC Pairwise	34	36.7	42	2.00	0.57	0.64	0.70	0.04
gen400_p0.9_55	Classic Away-step	43	45.2	47	1.08	0.86	0.88	0.91	0.02
	Classic Pairwise	42	44.6	47	1.36	0.74	0.79	0.88	0.05
	SSC Away-step	43	44.7	46	1.00	1.38	1.82	2.43	0.38
	SSC Pairwise	43	44.7	46	1.10	1.13	1.25	1.37	0.09
gen400_p0.9_65	Classic Away-step	42	43.6	47	1.43	0.68	0.89	0.93	0.07
	Classic Pairwise	40	42.9	46	2.02	0.75	0.80	0.88	0.04
	SSC Away-step	40	42.3	48	2.37	1.43	1.93	3.31	0.55
	SSC Pairwise	41	44.2	47	1.89	1.23	1.35	1.47	0.06
gen400_p0.9_75	Classic Away-step	44	46.1	50	2.07	0.85	0.93	1.01	0.05
	Classic Pairwise	40	42.6	45	1.85	0.73	0.77	0.88	0.04
	SSC Away-step	41	44.4	46	1.91	1.49	2.01	2.66	0.44
	SSC Pairwise	41	44.9	50	3.05	1.16	1.30	1.43	0.07
hamming8-4	Classic Away-step	13	15.7	16	0.90	0.40	0.41	0.41	0.01
	Classic Pairwise	11	15.1	16	1.81	0.37	0.38	0.39	0.01
	SSC Away-step	11	14.3	16	2.15	0.94	1.85	2.93	0.63
	SSC Pairwise	6	11.0	16	3.19	0.59	0.64	0.81	0.06
hamming10-4	Classic Away-step	30	31.7	35	1.55	9.43	9.72	9.97	0.15
	Classic Pairwise	29	30.3	32	1.19	8.12	8.37	8.48	0.10
	SSC Away-step	27	30.3	34	1.95	8.83	12.96	17.59	2.44
	SSC Pairwise	29	31.7	36	1.95	7.82	8.03	8.45	0.22
keller4	Classic Away-step	7	7.4	9	0.66	0.01	0.04	0.26	0.07
	Classic Pairwise	7	9.1	11	0.94	0.23	0.24	0.25	0.00
	SSC Away-step	7	7.4	9	0.66	0.03	0.10	0.71	0.21
	SSC Pairwise	7	7.9	9	0.70	0.36	0.40	0.46	0.03
keller5	Classic Away-step	17	18.8	22	1.66	4.65	4.78	4.92	0.08
	Classic Pairwise	19	19.9	23	1.22	4.27	4.39	4.53	0.07
	SSC Away-step	15	18.8	21	1.54	5.00	5.82	6.77	0.57
	SSC Pairwise	18	19.1	20	0.83	3.74	3.93	4.56	0.22
keller6	Classic Away-step	37	40.0	45	2.10	338.78	370.15	554.09	61.99
	Classic Pairwise	35	38.0	42	2.14	298.79	336.37	346.63	13.91
	SSC Away-step	35	38.7	42	2.00	303.58	322.16	336.23	10.27
	SSC Pairwise	31	36.7	40	2.53	245.90	252.30	259.60	4.38
MANN_a27	Classic Away-step	117	117.0	117	0.00	0.87	1.01	1.06	0.05
	Classic Pairwise	117	117.1	118	0.30	0.69	0.78	0.86	0.06
	SSC Away-step	117	117.0	117	0.00	3.68	4.91	6.24	0.83
	SSC Pairwise	117	117.0	117	0.00	1.58	2.03	2.42	0.26

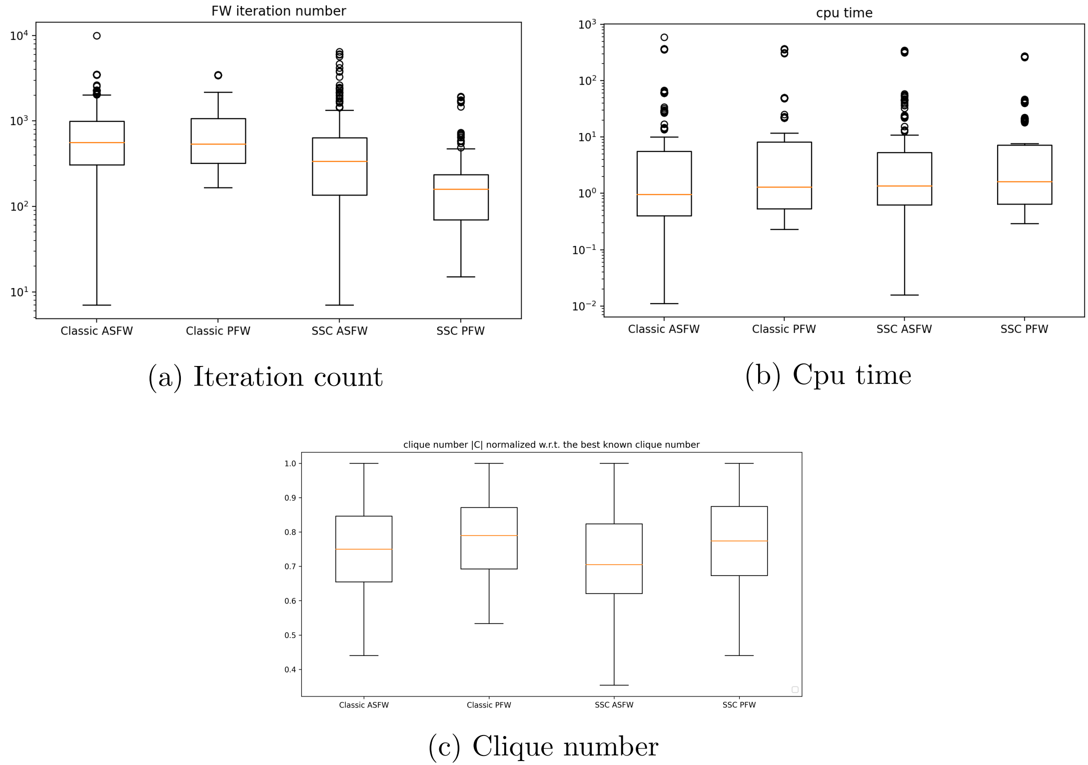
continues...

Table 4.1: (continues)

Instance	Algorigthm	clique number $\|C\|$				cpu time			
		min	mean	max	std	min	mean	max	std
MANN_a45	Classic Away-step	330	330.0	330	0.00	13.39	14.03	15.95	0.69
	Classic Pairwise	330	330.0	330	0.00	9.10	9.51	10.21	0.41
	SSC Away-step	330	330.0	330	0.00	37.47	49.24	60.51	8.61
	SSC Pairwise	330	330.0	330	0.00	14.23	19.26	23.32	2.59
p_hat300-1	Classic Away-step	6	6.6	8	0.66	0.44	0.46	0.51	0.02
	Classic Pairwise	6	6.5	8	0.67	0.42	0.43	0.47	0.01
	SSC Away-step	5	6.7	8	0.90	0.82	0.93	1.03	0.07
	SSC Pairwise	6	6.8	8	0.60	0.64	0.67	0.76	0.04
p_hat300-2	Classic Away-step	11	17.8	25	4.64	0.05	0.40	0.71	0.26
	Classic Pairwise	19	21.3	24	1.19	0.44	0.45	0.47	0.01
	SSC Away-step	11	12.2	13	0.98	0.08	0.10	0.14	0.02
	SSC Pairwise	11	19.3	23	4.36	0.66	0.70	0.80	0.04
p_hat300-3	Classic Away-step	18	24.5	32	5.89	0.08	0.31	0.55	0.21
	Classic Pairwise	29	31.7	33	1.27	0.45	0.49	0.59	0.04
	SSC Away-step	18	22.0	32	4.69	0.16	0.56	3.04	0.87
	SSC Pairwise	30	31.8	33	0.87	0.73	0.77	0.82	0.03
p_hat700-1	Classic Away-step	7	7.7	9	0.64	3.58	3.68	3.83	0.08
	Classic Pairwise	6	7.4	9	0.80	3.37	3.43	3.48	0.05
	SSC Away-step	6	7.4	9	0.92	3.15	3.42	3.73	0.17
	SSC Pairwise	7	8.0	9	0.77	2.74	2.83	2.99	0.07
p_hat700-2	Classic Away-step	37	40.4	43	1.56	3.89	4.06	4.17	0.09
	Classic Pairwise	38	39.5	41	1.02	3.42	3.48	3.56	0.05
	SSC Away-step	38	39.2	42	1.33	4.25	4.65	6.10	0.52
	SSC Pairwise	39	40.8	43	1.40	3.08	3.17	3.27	0.06
p_hat700-3	Classic Away-step	31	35.2	38	2.27	0.32	0.39	0.53	0.06
	Classic Pairwise	56	58.0	60	1.48	3.38	3.53	3.67	0.08
	SSC Away-step	31	34.9	38	1.92	0.44	0.56	0.67	0.07
	SSC Pairwise	55	58.2	61	1.99	3.11	3.20	3.31	0.07
p_hat1500-1	Classic Away-step	6	7.3	10	1.42	0.09	5.72	28.65	11.20
	Classic Pairwise	8	8.7	9	0.46	21.33	23.02	30.08	2.68
	SSC Away-step	6	6.6	7	0.49	0.10	0.12	0.16	0.02
	SSC Pairwise	8	9.2	10	0.75	18.37	19.20	21.25	0.90
p_hat1500-2	Classic Away-step	55	57.2	60	1.40	27.91	29.67	32.27	1.41
	Classic Pairwise	56	57.7	60	1.35	21.71	22.45	28.05	1.87
	SSC Away-step	23	47.5	59	14.29	0.86	17.81	30.53	11.16
	SSC Pairwise	58	59.1	61	1.04	19.80	20.50	21.69	0.49
p_hat1500-3	Classic Away-step	77	82.3	88	3.29	26.72	27.48	29.60	0.79
	Classic Pairwise	78	80.8	84	2.32	21.97	22.17	22.33	0.10
	SSC Away-step	76	80.4	84	2.42	22.05	22.97	23.84	0.60
	SSC Pairwise	82	85.1	89	2.66	21.05	21.45	21.97	0.25

For every instance, Table 4.1 has two groups of columns: one for the cardinality of the maximal clique found; one for the cpu times elapsed in the execution. For each group of columns, we report the minimum, maximum and average value among the 10 executed tests, plus the standard deviation.

The Figure 4.1 presents three box plots that aggregate the results from the first experiment. Figure 4.1a displays the number of iterations (i.e. the number of times the gradient and linear minimizer has been computed) performed by the algorithms. Figure 4.1b displays the cpu time elapsed during each execution of the algorithm. Figure 4.1c displays the clique number of the solutions found: in this case, we normalized the result of each instance with respect to the maximum clique ever found for that instance that we know of. The cardinality of the maximum clique can be found in [15].



(a) Iteration count

(b) Cpu time

(c) Clique number

Figure 4.1: Comparison of the results

Looking at Figure 4.1b, we see that the SSC versions are not noticeably faster than the classic versions. Nevertheless, in Figure 4.1a, we clearly see that using a SSC version spares a large percentage of iterations. Moreover, the cliques number displayed in Figure 4.1c also very similar: we see both PFW versions finding slightly larger cliques than ASFW variants.

4.4 MDCP

The second experiment involves the solution of the *maximum s-defective clique problem*. We looked for local minima of (3.5) that correspond to s-defective clique for Theorem 4. The solution of (3.5) with the algorithm given in Chapter 2 proved to require lots of memory and computational resources. To perform our experiments, we used a variants of our algorithms that works separately on the simplex Δ and on the y feasible region $D_{s,m}$.

For $f : \Delta \times D_{s,m} \to \mathbb{R}$ objective function of the main problem we define two objective functions for the x, y sub-problems:

$$f_y : \Delta \to \mathbb{R} \text{ s.t. } f_y(x) = f(x,y) \quad \text{and} \quad f_x : D_{s,m} \to \mathbb{R} \text{ s.t. } f_x(y) = f(x,y)$$

We outline in Algorithm 7 the modified algorithm. On Line 5 and Line 8 the iteration of either the "Classic" or *SSC* FW variants are performed and the gap is returned. Each iteration in Line 5 works with the f_y objective and each one in Line 8 with f_x. A target ϵ is provided, that is a ϵ for which the iteration of Frank-Wolfe would not change the current iterate when the FW gap is lower than ϵ, and for which the algorithm will exit whenever global gap $\max\{gap_x, gap_y\} \leq \epsilon$.

Algorithm 7:

Data: A starting feasible point $(x_0, y_0) \in \Delta \times D_{s,m}$ and a target gap ϵ
Result: An approximation of a stationary point (x^*, y^*)

1 $t \leftarrow 0$
2 $f_x(\cdot) \leftarrow f(x_0, \cdot)$
3 $f_y(\cdot) \leftarrow f(\cdot, y_0)$
4 **repeat**
5     $x_{t+1}, gap_x \leftarrow$ iteration on x
6   **if** $gap_x \geq \epsilon$ **then**
7         $f_x(\cdot) \leftarrow f(x_{t+1}, \cdot)$
8     $y_{t+1}, gap_y \leftarrow$ iteration on y
9   **if** $gap_y \geq \epsilon$ **then**
10         $f_y(\cdot) \leftarrow f(\cdot, y_{t+1})$
11     $gap \leftarrow \max\{gap_x, gap_y\}$
12     $t \leftarrow t + 1$
13 **until** $gap \leq \epsilon \ \vee \ t = t_{max}$

The objective function and gradient are the opposite of those in (3.5) as we solved a minimization problem:

$$\begin{aligned} f(x,y) &= -x^T(A_G + A_{\bar{G}}(y))x - \frac{1}{2}\|x\|^2 - \frac{10^{-4}}{2}\|y\|^2 \\ \nabla_x f(x,y) &= -\left(2A_G + 2A_{\bar{G}}(y) + 1_n\right)x \\ \nabla_y f(x,y) &= d - 10^{-4}y \qquad \text{with } \ d_{ij} = -2x_i x_y \ \forall\{i,j\} \in \bar{\mathcal{E}} \end{aligned}$$

The parameters used in the stopping condition in Line 13 were $\epsilon = 10^{-4}$ and $t_{max} = 5 \cdot 10^5$. When (x^*, y^*) are returned, we check $s[S] = (supp(y^*))[S]$ to confirm $S = supp(x^*)$ to be an s-defective clique. Some solutions did not represented a defective clique, in those cases we run the algorithm again with $(x_0, y_0) = (x^*, y^*)$, $\epsilon = 10^{-4.5}$ and $t_{max} = 2 \cdot 10^5$, and checked again. In these occurrences, the reported cpu time is the sum of the two executions.

Linear minimizer

The x iterations of the algorithm can use the linear minimizer on Δ described previously.

The y iterations need a linear minimizer to solve the problem:

$$\mathcal{LMO}_{D_{s,m}}(g) = \arg\min_{y \in D_{s,m}} \langle g, x \rangle$$

$D_{s,m}$ has many vertexes: for every $I \subseteq \{1 \dots m\}$ with $|I| \leq s$ the characteristic vector χ^I(i.e. $(\chi^I)_i = 1$ if $i \in E$, $(\chi^I)_i = 0$ otherwise) is a vertex. As such, we cannot solve the linear problem by computing products.

$D_{s,m}$ can be rewritten as

$$D_{s,m} = \{y \in \mathbb{R}^m : y \geq 0, \langle y, \chi^I \rangle \leq \beta(I) \, \forall I \subseteq \{1 \dots m\}\}$$

for χ^I characteristic vector and $\beta : \mathcal{P}(\{1 \dots m\}) \to \mathbb{N}$ with $\beta(I) = \min\{|I|, s\}$. β is a crescent, positive, sub-modular function and thus, $D_{s,m}$ is a polimatroid. As such, we can solve linear programs on $D_{s,m}$ with the greedy algorithm.

Hence, the $\mathcal{LMO}_{D_{s,m}}$ will find up to s negative entries in g with the largest absolute values and return the vertex χ^I with I set of those entries.

We will store every vertex returned from the linear minimizer inside the matrix-like object that describes the feasible set vertexes $\mathcal{A}$ alongside the 0 vertex and the starting point y_0. In this way, we track the active set with low memory usage.

Starting point

For the choice of the starting point (x_0, y_0), we followed the same strategy detailed above when it comes to x_0. For the choice of y_0, we chose a random integer $k \leq \min\{s, m\}$ and then $I \subseteq \{1 \dots m\}$ such that $|I| = k$, finally we set $y_0 = \chi^I$.

Results

Table 4.2: Results of the *MDCP* tests

	s	Algorigthm	clique number $\lvert S \rvert$				cpu time				missing edges $s[S]$			
			min	mean	max	std	min	mean	max	std	min	mean	max	std
brock200_2	5	Classic ASFW	9	10.2	11	0.75	3.76	35.69	88.83	33.92	2.0	4.1	5.0	1.2
		Classic PFW	9	10.3	12	0.90	3.23	23.65	53.95	18.66	3.0	4.5	5.0	0.8
		SSC ASFW	6	9.6	12	1.91	0.28	4.03	8.71	3.05	0.0	3.3	5.0	1.9
		SSC PFW	8	9.8	11	0.87	0.82	1.24	1.59	0.22	2.0	4.2	5.0	1.1
	20	Classic ASFW	12	13.4	16	1.28	4.42	62.72	133.88	50.21	15.0	17.5	19.0	1.3
		Classic PFW	12	13.3	15	1.00	4.55	44.25	74.52	27.60	15.0	18.1	20.0	1.5
		SSC ASFW	6	12.3	14	2.93	0.28	4.07	9.05	3.14	0.0	14.9	20.0	7.6
		SSC PFW	11	12.8	14	0.98	1.18	1.60	2.28	0.39	12.0	16.7	20.0	2.2
	50	Classic ASFW	15	18.4	20	1.28	5.89	104.34	198.29	77.08	27.0	45.6	50.0	6.3
		Classic PFW	17	18.5	20	0.81	6.74	76.90	126.14	41.25	44.0	47.1	50.0	1.8
		SSC ASFW	6	15.5	20	4.61	0.33	3.50	6.38	1.78	0.0	34.8	49.0	17.6
		SSC PFW	11	17.3	20	2.33	1.41	2.40	3.14	0.56	23.0	43.6	50.0	7.6

continues...

Table 4.2: (continues)

	s	Algorigthm	clique number $\|S\|$ min	mean	max	std	cpu time min	mean	max	std	missing edges $s[S]$ min	mean	max	std
brock200_4	5	Classic ASFW	14	14.4	15	0.49	2.97	21.92	118.45	33.93	3.0	4.5	5.0	0.7
		Classic PFW	13	14.6	16	1.02	2.92	10.28	33.76	9.91	4.0	4.9	5.0	0.3
		SSC ASFW	10	14.4	17	2.06	0.34	3.65	8.69	2.35	0.0	4.1	5.0	1.6
		SSC PFW	12	14.1	16	1.30	0.94	1.70	3.16	0.62	4.0	4.7	5.0	0.5
	20	Classic ASFW	17	18.1	20	0.94	4.29	62.64	152.32	55.75	17.0	18.9	20.0	0.9
		Classic PFW	17	18.5	20	0.81	5.73	36.55	86.71	32.05	17.0	19.0	20.0	1.0
		SSC ASFW	10	17.1	21	2.66	0.38	3.42	6.59	2.02	0.0	15.1	20.0	5.4
		SSC PFW	16	18.1	20	1.14	1.26	2.24	3.92	0.89	19.0	19.2	20.0	0.4
	50	Classic ASFW	22	23.3	25	0.90	5.47	87.72	214.81	79.11	33.0	44.9	50.0	4.9
		Classic PFW	23	23.7	24	0.46	7.80	59.80	141.78	49.96	45.0	48.5	50.0	1.5
		SSC ASFW	10	20.8	24	3.76	0.41	4.31	8.02	2.23	0.0	34.3	47.0	13.0
		SSC PFW	22	23.9	25	0.94	1.66	3.38	5.51	1.05	42.0	48.2	50.0	2.4
brock400_2	5	Classic ASFW	19	21.0	23	1.18	11.90	61.11	347.81	97.95	0.0	2.7	5.0	2.0
		Classic PFW	21	22.2	23	0.60	14.60	87.22	218.63	80.66	4.0	4.8	5.0	0.4
		SSC ASFW	19	20.9	22	1.04	4.89	12.48	27.62	8.14	3.0	4.2	5.0	0.7
		SSC PFW	20	22.0	23	1.10	3.23	4.52	6.12	1.10	3.0	4.7	5.0	0.6
	20	Classic ASFW	19	22.6	27	2.46	12.02	102.17	439.43	160.85	1.0	10.2	18.0	6.2
		Classic PFW	25	25.5	27	0.67	18.91	170.51	249.32	92.78	15.0	17.8	20.0	1.8
		SSC ASFW	22	23.5	26	1.50	5.28	10.92	19.48	5.49	7.0	12.4	18.0	3.7
		SSC PFW	23	24.8	27	1.25	3.04	5.06	7.23	1.51	17.0	18.7	20.0	0.9
	50	Classic ASFW	19	23.6	27	2.11	13.02	162.22	532.59	215.29	4.0	15.0	30.0	8.8
		Classic PFW	27	29.5	32	1.69	21.22	262.72	335.60	88.02	33.0	37.4	43.0	4.0
		SSC ASFW	22	23.8	26	1.47	4.24	9.13	17.14	4.55	14.0	18.7	26.0	3.7
		SSC PFW	27	30.2	33	1.54	4.43	7.06	9.80	1.93	33.0	41.4	50.0	4.5
brock400_4	5	Classic ASFW	20	21.4	23	0.80	12.96	58.86	333.27	95.19	1.0	3.4	5.0	1.3
		Classic PFW	20	22.4	24	1.11	19.92	104.60	216.67	87.57	3.0	4.7	5.0	0.6
		SSC ASFW	20	21.5	23	0.92	4.11	7.53	14.91	3.43	3.0	4.5	5.0	0.7
		SSC PFW	21	22.7	24	0.78	2.78	4.00	4.75	0.53	4.0	4.7	5.0	0.5
	20	Classic ASFW	22	24.4	27	1.36	13.43	130.40	413.61	175.93	4.0	12.8	19.0	4.3
		Classic PFW	24	25.9	28	1.30	72.23	207.00	264.86	64.20	16.0	17.9	20.0	1.4
		SSC ASFW	22	24.1	27	1.70	3.89	7.28	13.41	2.68	7.0	12.4	19.0	4.1
		SSC PFW	23	26.4	29	1.56	3.34	4.87	7.57	1.31	16.0	19.0	20.0	1.2
	50	Classic ASFW	22	24.3	29	2.37	15.47	154.31	541.18	210.42	10.0	17.8	38.0	7.5
		Classic PFW	27	29.5	31	1.12	83.67	275.25	355.85	84.67	29.0	35.2	49.0	6.1
		SSC ASFW	22	24.1	27	1.58	4.37	6.03	8.62	1.36	10.0	17.9	23.0	3.7
		SSC PFW	28	30.7	32	1.42	4.30	6.13	7.71	0.93	34.0	41.9	48.0	4.2
C125.9	5	Classic ASFW	29	31.4	34	1.50	1.73	22.96	98.18	34.73	5.0	5.0	5.0	0.0
		Classic PFW	30	31.9	36	1.87	1.07	9.33	50.88	15.38	4.0	4.9	5.0	0.3
		SSC ASFW	30	32.4	35	1.43	1.45	2.47	4.02	0.75	3.0	4.7	5.0	0.6
		SSC PFW	29	31.4	34	1.43	0.83	0.99	1.68	0.26	4.0	4.8	5.0	0.4
	20	Classic ASFW	31	33.0	36	1.48	15.84	39.72	103.05	29.79	19.0	19.9	20.0	0.3
		Classic PFW	32	34.9	37	1.37	5.00	18.40	59.79	19.91	19.0	19.9	20.0	0.3
		SSC ASFW	31	36.0	39	2.00	1.66	2.59	4.34	0.75	15.0	18.4	20.0	1.6
		SSC PFW	33	35.2	37	1.08	0.91	1.26	2.11	0.38	19.0	19.9	20.0	0.3
	50	Classic ASFW	36	37.6	40	1.02	16.63	75.12	105.02	27.83	43.0	48.4	50.0	2.3
		Classic PFW	38	40.4	44	1.62	5.55	35.88	72.16	22.03	46.0	49.1	50.0	1.4
		SSC ASFW	35	40.3	44	2.76	1.42	2.25	4.62	0.97	32.0	46.2	50.0	5.0
		SSC PFW	38	40.8	43	1.40	0.96	1.30	2.31	0.40	44.0	47.6	50.0	2.1
C250.9	5	Classic ASFW	38	40.1	43	1.51	6.02	119.62	315.15	121.84	2.0	3.9	5.0	1.1
		Classic PFW	37	40.1	42	1.45	4.23	47.49	156.25	56.63	3.0	4.1	5.0	0.7
		SSC ASFW	38	39.2	41	0.98	2.72	4.40	7.03	1.51	1.0	3.5	5.0	1.3
		SSC PFW	38	39.8	41	1.08	1.94	2.56	3.87	0.67	3.0	4.1	5.0	0.7
	20	Classic ASFW	40	43.2	46	1.60	26.43	226.53	343.72	110.75	14.0	16.7	20.0	2.0
		Classic PFW	40	42.5	45	1.43	7.94	95.65	164.06	61.88	11.0	15.5	18.0	2.0
		SSC ASFW	39	41.6	43	1.11	2.50	3.55	6.25	1.38	6.0	9.5	14.0	2.1
		SSC PFW	41	42.1	44	1.14	1.80	2.43	3.97	0.72	13.0	15.5	18.0	1.7
	50	Classic ASFW	42	45.1	48	1.97	103.31	318.45	426.38	86.38	40.0	45.4	50.0	3.1
		Classic PFW	42	46.2	49	2.32	27.70	149.13	259.08	64.76	30.0	40.6	47.0	5.3
		SSC ASFW	43	44.4	46	1.28	3.08	5.51	12.19	3.10	16.0	22.5	26.0	3.1
		SSC PFW	45	46.7	49	1.19	2.65	3.53	5.72	1.09	26.0	36.7	46.0	5.1
C500.9	5	Classic ASFW	45	48.1	51	1.87	17.93	320.85	831.72	320.71	0.0	1.4	3.0	0.9
		Classic PFW	47	49.0	51	1.10	21.34	195.07	437.78	171.45	1.0	1.8	4.0	1.1
		SSC ASFW	45	47.6	51	2.01	6.15	8.96	16.94	3.81	0.0	1.1	3.0	0.9
		SSC PFW	47	49.0	52	1.55	4.79	6.19	10.04	1.45	0.0	2.5	5.0	1.4
	20	Classic ASFW	46	49.0	52	1.95	18.28	468.05	909.17	378.44	1.0	5.4	8.0	2.2
		Classic PFW	46	49.2	51	1.40	81.70	322.84	485.32	145.24	3.0	5.4	8.0	1.5
		SSC ASFW	45	48.4	51	1.85	5.67	7.45	14.67	2.53	2.0	4.3	8.0	1.8
		SSC PFW	47	50.3	53	1.73	4.27	5.95	10.04	1.55	5.0	10.0	14.0	2.6
	50	Classic ASFW	47	49.8	51	1.47	20.21	530.25	1023.52	430.87	7.0	11.7	16.0	3.4
		Classic PFW	48	50.3	53	1.49	87.53	358.92	546.75	156.78	7.0	11.3	15.0	2.7
		SSC ASFW	46	49.1	53	1.92	6.49	9.25	18.92	3.55	3.0	7.4	14.0	3.2
		SSC PFW	50	52.4	54	1.28	6.32	7.69	12.49	1.67	11.0	17.8	26.0	5.3

continues...

Table 4.2: (continues)

	s	Algorigthm	clique number $\|S\|$				cpu time				missing edges $s[S]$			
			min	mean	max	std	min	mean	max	std	min	mean	max	std
DSJC500_5	5	Classic ASFW	9	11.7	14	1.35	0.97	91.99	360.80	132.10	0.0	3.5	5.0	1.7
		Classic PFW	11	12.6	14	0.80	27.60	117.87	200.98	75.86	3.0	4.3	5.0	0.6
		SSC ASFW	9	11.8	13	1.60	0.92	18.39	43.67	14.82	0.0	3.6	5.0	1.9
		SSC PFW	11	12.2	13	0.75	3.08	4.43	6.61	1.00	3.0	4.6	5.0	0.7
	20	Classic ASFW	9	13.1	15	2.12	0.92	125.66	412.65	159.82	0.0	11.4	19.0	7.3
		Classic PFW	14	15.2	17	0.87	36.10	192.55	256.31	71.86	14.0	18.0	20.0	2.0
		SSC ASFW	9	14.0	16	2.57	0.85	25.13	53.60	18.12	0.0	13.6	20.0	7.0
		SSC PFW	13	14.5	16	1.12	3.41	4.68	7.93	1.38	11.0	16.3	20.0	3.1
	50	Classic ASFW	9	14.5	18	3.75	0.96	147.50	474.24	188.76	0.0	23.0	38.0	15.0
		Classic PFW	19	19.9	21	0.70	46.69	262.78	356.47	111.01	46.0	47.2	49.0	1.3
		SSC ASFW	9	15.5	19	3.44	0.91	18.50	53.19	17.85	0.0	22.6	35.0	12.0
		SSC PFW	18	19.2	20	0.75	4.50	7.54	12.14	2.19	34.0	44.9	50.0	4.3
gen200_p0.9_44	5	Classic ASFW	33	35.5	38	1.86	3.45	81.56	202.99	83.92	4.0	4.7	5.0	0.5
		Classic PFW	36	37.0	38	0.63	3.11	42.33	115.14	45.83	4.0	4.9	5.0	0.3
		SSC ASFW	30	36.0	40	2.97	0.78	3.43	8.59	2.02	0.0	3.3	5.0	2.0
		SSC PFW	31	36.3	40	2.41	1.40	1.81	2.55	0.31	1.0	3.9	5.0	1.3
	20	Classic ASFW	38	40.6	44	1.85	5.89	103.30	211.34	79.58	14.0	19.3	20.0	1.8
		Classic PFW	40	42.4	45	1.62	6.12	64.10	126.23	49.77	16.0	18.2	20.0	1.2
		SSC ASFW	32	39.7	44	4.03	0.78	3.68	4.90	1.54	0.0	13.6	20.0	6.8
		SSC PFW	31	41.2	45	3.66	1.03	1.85	2.70	0.43	1.0	16.4	20.0	5.4
	50	Classic ASFW	37	42.6	48	2.50	3.78	101.54	211.81	81.90	28.0	47.2	50.0	6.5
		Classic PFW	43	46.0	50	2.14	5.55	72.93	133.16	47.97	40.0	46.6	49.0	3.0
		SSC ASFW	31	41.1	45	4.89	0.82	4.13	7.66	2.23	0.0	28.2	49.0	15.1
		SSC PFW	31	44.6	50	5.10	1.12	2.04	3.59	0.60	1.0	39.0	50.0	14.6
gen400_p0.9_55	5	Classic ASFW	40	45.2	47	2.09	51.39	295.19	620.42	201.21	2.0	3.9	5.0	0.9
		Classic PFW	41	45.2	48	1.72	17.08	107.23	320.61	107.95	2.0	2.9	4.0	0.7
		SSC ASFW	36	43.8	47	3.09	1.28	7.14	14.21	3.53	0.0	2.1	4.0	1.4
		SSC PFW	44	45.5	48	1.36	3.30	4.57	7.03	0.96	2.0	2.9	4.0	0.7
	20	Classic ASFW	45	48.2	52	2.32	54.83	545.04	738.17	214.43	9.0	13.6	20.0	3.4
		Classic PFW	45	46.9	49	1.14	17.17	233.39	364.19	118.79	8.0	10.2	14.0	2.1
		SSC ASFW	36	44.3	49	3.38	1.15	6.43	13.66	3.03	0.0	7.2	13.0	3.2
		SSC PFW	46	49.0	53	1.67	3.86	4.54	6.50	0.71	6.0	12.1	15.0	2.8
	50	Classic ASFW	46	49.4	53	1.96	54.85	718.27	1094.18	321.73	17.0	22.5	35.0	6.0
		Classic PFW	45	48.4	52	2.50	17.05	321.63	535.25	185.15	14.0	17.9	23.0	2.9
		SSC ASFW	36	45.0	50	4.82	1.40	7.48	14.08	3.72	0.0	12.3	19.0	5.3
		SSC PFW	48	50.4	52	1.20	4.31	6.52	9.50	1.36	21.0	25.2	29.0	2.8
gen400_p0.9_75	5	Classic ASFW	41	43.6	48	1.80	13.20	319.60	719.65	288.19	3.0	4.5	5.0	0.7
		Classic PFW	41	45.5	47	1.96	11.54	132.41	317.13	118.12	1.0	3.9	5.0	1.2
		SSC ASFW	42	45.0	50	2.00	5.97	12.68	19.20	4.07	1.0	3.7	5.0	1.2
		SSC PFW	42	46.1	52	3.05	4.23	4.89	5.52	0.36	2.0	3.8	5.0	1.1
	20	Classic ASFW	48	51.1	54	1.70	15.42	580.92	822.38	220.02	8.0	16.8	20.0	3.6
		Classic PFW	46	50.2	53	2.44	96.89	237.40	395.79	94.61	9.0	13.4	17.0	2.6
		SSC ASFW	47	49.3	51	1.42	5.54	10.77	18.44	4.49	3.0	9.3	17.0	3.8
		SSC PFW	48	50.9	56	2.26	3.93	5.30	6.63	0.78	8.0	13.7	18.0	2.6
	50	Classic ASFW	47	60.7	72	7.48	22.75	703.31	1080.61	270.57	12.0	36.7	50.0	12.7
		Classic PFW	45	50.4	58	3.38	171.94	378.29	576.02	141.87	10.0	15.1	24.0	4.4
		SSC ASFW	46	48.9	52	1.87	6.08	15.99	30.69	9.88	7.0	11.4	16.0	2.6
		SSC PFW	51	52.9	56	1.76	5.24	7.99	10.92	1.94	15.0	19.6	26.0	4.0
hamming8-4	5	Classic ASFW	11	13.0	15	1.18	5.19	46.02	160.11	57.50	4.0	4.8	5.0	0.4
		Classic PFW	11	13.6	15	1.50	6.07	33.69	83.94	30.94	4.0	4.9	5.0	0.3
		SSC ASFW	11	12.1	14	0.94	2.38	4.14	6.88	1.23	4.0	4.9	5.0	0.3
		SSC PFW	9	12.0	14	1.55	1.32	1.93	3.26	0.59	3.0	4.7	5.0	0.6
	20	Classic ASFW	15	17.2	19	1.25	4.86	42.03	190.19	60.33	19.0	19.5	20.0	0.5
		Classic PFW	16	17.2	19	0.75	12.78	57.32	103.81	34.71	17.0	18.7	20.0	1.1
		SSC ASFW	15	17.1	18	0.94	3.39	9.29	17.60	4.59	18.0	19.4	20.0	0.8
		SSC PFW	15	17.0	19	1.10	2.17	2.65	3.54	0.37	17.0	19.2	20.0	1.0
	50	Classic ASFW	16	21.8	24	2.27	7.35	78.61	285.22	109.05	34.0	43.5	49.0	4.4
		Classic PFW	23	24.1	25	0.94	21.64	126.51	191.30	52.10	48.0	48.4	50.0	0.8
		SSC ASFW	17	21.3	23	1.79	4.75	9.52	13.74	2.75	26.0	39.1	47.0	5.8
		SSC PFW	22	23.7	25	0.90	2.78	3.88	5.41	0.71	45.0	48.0	50.0	1.8
keller4	5	Classic ASFW	7	9.7	12	1.85	0.23	3.56	18.88	5.23	0.0	3.4	5.0	2.2
		Classic PFW	10	10.8	12	0.60	2.12	14.44	49.74	17.49	5.0	5.0	5.0	0.0
		SSC ASFW	7	9.9	13	1.87	0.22	2.09	3.67	1.34	0.0	3.4	5.0	2.2
		SSC PFW	10	11.3	12	0.78	0.76	1.10	1.35	0.21	5.0	5.0	5.0	0.0
	20	Classic ASFW	7	12.6	15	3.47	0.22	4.34	23.60	6.57	0.0	13.5	20.0	8.7
		Classic PFW	15	16.0	17	0.63	4.36	27.26	66.70	24.03	18.0	19.6	20.0	0.7
		SSC ASFW	7	13.5	17	4.10	0.25	1.88	3.82	1.22	0.0	12.9	20.0	8.4
		SSC PFW	15	16.0	18	0.77	0.93	1.40	1.96	0.28	18.0	19.4	20.0	0.7
	50	Classic ASFW	7	16.1	22	5.89	0.20	16.10	134.16	39.40	0.0	28.9	48.0	19.0
		Classic PFW	22	22.9	24	0.70	6.60	55.09	96.46	33.40	47.0	48.7	50.0	0.9
		SSC ASFW	7	16.1	21	5.79	0.21	2.71	7.01	2.08	0.0	28.4	48.0	19.0
		SSC PFW	20	22.6	24	1.28	1.03	1.94	3.04	0.57	43.0	47.4	50.0	2.2

continues...

Table 4.2: (continues)

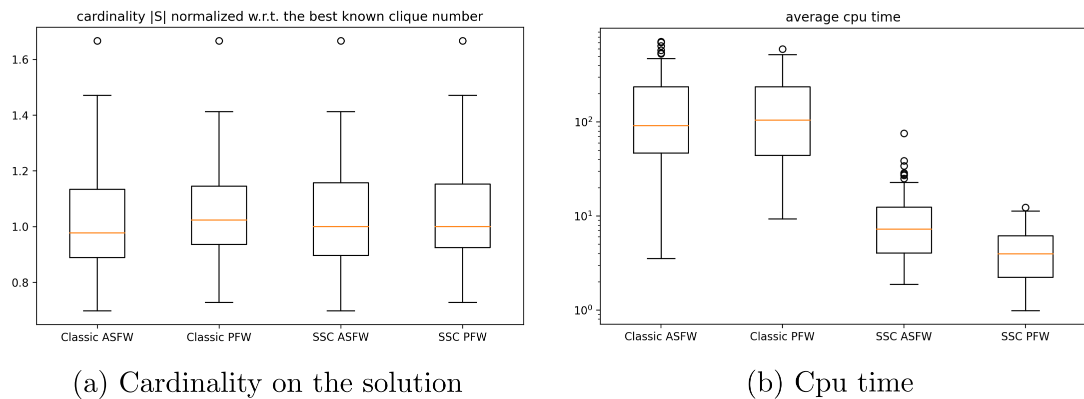
	s	Algorigthm	clique number $\|S\|$				cpu time				missing edges $s[S]$			
			min	mean	max	std	min	mean	max	std	min	mean	max	std
keller5	5	Classic ASFW	18	19.0	21	1.00	1.93	45.23	57.57	21.65	0.0	1.8	4.0	1.5
		Classic PFW	19	20.8	22	0.98	99.59	322.64	507.89	160.66	4.0	4.6	5.0	0.5
		SSC ASFW	17	19.7	22	1.68	1.89	28.84	63.69	18.80	0.0	2.6	5.0	1.6
		SSC PFW	18	20.3	22	1.10	6.29	10.79	12.69	2.30	1.0	3.8	5.0	1.2
	20	Classic ASFW	15	20.0	23	2.14	1.73	46.85	65.63	22.68	0.0	5.9	12.0	3.9
		Classic PFW	22	24.9	27	1.30	146.42	522.10	613.05	130.09	15.0	17.1	20.0	1.6
		SSC ASFW	15	20.6	24	2.62	1.62	27.86	65.67	17.61	0.0	7.3	16.0	5.0
		SSC PFW	22	23.9	26	1.14	7.74	11.41	13.21	1.88	13.0	14.7	18.0	1.8
	50	Classic ASFW	15	21.8	25	3.03	1.53	49.49	65.12	23.97	0.0	14.8	27.0	9.9
		Classic PFW	26	27.1	28	0.70	166.72	595.62	696.62	149.85	21.0	27.8	37.0	4.9
		SSC ASFW	15	20.9	23	2.26	1.68	34.15	64.33	22.73	0.0	9.9	18.0	6.0
		SSC PFW	24	27.0	29	1.41	7.98	12.43	14.79	2.44	18.0	25.2	34.0	5.2
MANN_a27	5	Classic ASFW	118	119.8	121	0.87	48.28	257.24	707.37	205.16	1.0	2.8	4.0	0.9
		Classic PFW	119	120.0	121	0.77	14.88	94.43	309.26	87.74	1.0	2.8	4.0	0.9
		SSC ASFW	118	120.5	122	1.20	14.28	18.92	24.43	3.15	1.0	3.5	5.0	1.2
		SSC PFW	119	120.6	122	1.11	6.05	6.86	7.36	0.42	2.0	3.6	5.0	1.1
	20	Classic ASFW	124	126.7	131	2.00	48.59	475.55	718.08	270.82	7.0	10.6	15.0	2.7
		Classic PFW	124	126.6	131	1.96	15.46	246.39	506.12	178.25	7.0	10.5	15.0	2.7
		SSC ASFW	124	131.0	135	3.13	17.06	38.85	85.88	19.95	7.0	14.5	19.0	3.4
		SSC PFW	124	130.1	136	3.21	6.06	6.58	7.01	0.32	7.0	13.4	20.0	3.6
	5	Classic ASFW	135	139.6	149	3.58	487.44	644.59	898.02	104.84	20.0	24.1	34.0	3.9
		Classic PFW	135	139.9	150	3.81	255.97	435.55	648.93	115.06	21.0	24.6	38.0	4.7
		SSC ASFW	143	149.6	158	5.04	15.66	75.72	161.37	44.76	27.0	34.1	44.0	6.1
		SSC PFW	141	147.6	158	5.33	5.87	6.93	12.13	1.77	25.0	32.4	44.0	6.5
p_hat300-1	5	Classic ASFW	5	7.4	9	1.62	0.56	6.22	10.29	3.76	0.0	2.9	5.0	2.4
		Classic PFW	7	8.1	9	0.54	7.71	11.40	25.17	5.33	2.0	4.1	5.0	1.0
		SSC ASFW	5	7.1	9	1.58	0.52	4.63	10.10	3.23	0.0	2.9	5.0	2.0
		SSC PFW	7	7.5	8	0.50	1.42	1.68	2.05	0.21	3.0	4.0	5.0	0.6
	20	Classic ASFW	5	8.4	11	2.62	0.56	7.19	11.69	4.42	0.0	10.5	20.0	8.5
		Classic PFW	10	10.9	12	0.54	11.14	41.59	107.00	31.13	14.0	17.3	20.0	1.8
		SSC ASFW	5	9.2	12	2.79	0.56	5.25	11.77	3.96	0.0	12.1	19.0	8.2
		SSC PFW	8	10.2	12	1.08	1.54	2.24	3.40	0.62	7.0	15.4	20.0	4.4
	50	Classic ASFW	5	10.7	16	4.36	0.56	8.18	12.64	5.07	0.0	25.1	49.0	20.4
		Classic PFW	14	15.5	17	0.81	13.63	105.10	156.67	50.38	43.0	46.7	50.0	2.2
		SSC ASFW	5	11.1	16	4.32	0.57	4.52	11.57	3.73	0.0	23.6	47.0	18.2
		SSC PFW	7	13.9	17	3.08	1.68	3.37	5.06	1.17	3.0	36.5	50.0	16.0
p_hat300-2	5	Classic ASFW	11	21.1	26	4.95	0.95	19.30	43.61	13.73	0.0	3.2	5.0	2.2
		Classic PFW	22	23.9	27	1.58	7.57	30.97	118.99	36.89	4.0	4.7	5.0	0.5
		SSC ASFW	11	18.1	25	5.24	0.90	3.01	7.59	2.08	0.0	1.5	5.0	2.0
		SSC PFW	12	22.2	26	3.87	1.57	2.79	4.61	0.83	0.0	3.3	5.0	1.7
	20	Classic ASFW	11	23.4	30	6.48	1.02	125.04	347.90	127.03	0.0	11.8	19.0	8.1
		Classic PFW	27	28.8	30	0.87	12.86	88.58	201.35	62.15	10.0	17.6	20.0	2.7
		SSC ASFW	11	20.1	29	7.06	0.92	4.48	12.49	3.96	0.0	6.5	19.0	7.4
		SSC PFW	12	25.7	29	5.35	1.56	3.34	5.91	1.15	0.0	14.7	20.0	7.4
	50	Classic ASFW	11	25.8	33	7.83	1.24	216.87	468.37	190.25	0.0	26.6	48.0	19.8
		Classic PFW	28	31.1	34	1.70	51.03	187.05	301.40	85.70	19.0	26.5	36.0	5.4
		SSC ASFW	11	21.7	32	8.32	1.00	5.96	14.08	5.29	0.0	11.5	31.0	11.9
		SSC PFW	12	29.4	35	7.31	1.65	4.31	7.98	1.77	0.0	27.7	43.0	14.5
p_hat300-3	5	Classic ASFW	32	33.3	35	1.00	12.63	78.19	339.12	91.36	4.0	4.8	5.0	0.4
		Classic PFW	32	34.0	37	1.48	7.85	27.79	144.55	39.36	4.0	4.6	5.0	0.5
		SSC ASFW	4.0	4.6	5.0	0.49	1.18	7.58	10.18	2.83	0.0	3.8	5.0	1.6
		SSC PFW	0.0	3.8	5.0	1.60	2.16	2.91	5.47	0.90	3.0	4.2	5.0	0.7
	20	Classic ASFW	36	37.9	40	1.30	49.96	238.94	433.38	146.52	14.0	18.4	20.0	1.9
		Classic PFW	37	38.8	41	1.40	15.65	92.90	246.18	78.34	13.0	16.6	20.0	1.7
		SSC ASFW	23	34.8	38	4.19	1.47	9.25	14.83	4.13	0.0	11.3	17.0	4.9
		SSC PFW	36	38.2	41	1.66	2.99	3.80	8.00	1.43	13.0	16.5	20.0	2.5
	50	Classic ASFW	38	40.2	43	1.78	64.24	428.58	582.53	136.69	35.0	45.5	50.0	4.4
		Classic PFW	37	39.9	43	1.76	24.79	206.76	334.09	81.46	14.0	21.6	30.0	5.7
		SSC ASFW	23	35.3	39	4.43	1.82	12.38	22.69	5.30	0.0	14.6	25.0	7.3
		SSC PFW	37	41.2	43	1.94	3.20	4.55	9.13	1.67	21.0	26.9	35.0	4.8
p_hat700-1	5	Classic ASFW	5	8.5	10	1.63	1.17	52.57	76.33	26.00	0.0	2.5	5.0	2.2
		Classic PFW	8	9.4	11	0.92	78.38	142.73	261.09	66.70	1.0	3.5	5.0	1.2
		SSC ASFW	5	9.1	11	1.87	1.12	18.12	40.35	12.63	0.0	3.5	5.0	1.9
		SSC PFW	8	9.3	10	0.64	5.47	7.04	10.46	1.44	0.0	3.4	5.0	1.6
	20	Classic ASFW	5	9.6	13	2.37	1.13	55.16	79.72	27.31	0.0	8.6	19.0	7.5
		Classic PFW	11	12.8	14	1.08	217.18	320.39	366.18	49.79	13.0	17.5	20.0	2.2
		SSC ASFW	5	10.8	14	2.79	1.11	22.86	49.71	16.74	0.0	12.7	19.0	6.8
		SSC PFW	11	12.0	13	0.77	6.53	8.17	13.05	1.84	12.0	16.3	19.0	2.2
	50	Classic ASFW	5	10.7	16	3.44	1.18	60.32	89.05	30.07	0.0	16.6	40.0	14.7
		Classic PFW	15	16.4	17	0.66	350.92	493.56	548.90	56.57	40.0	43.9	49.0	2.4
		SSC ASFW	5	12.7	16	4.12	1.14	27.24	65.48	21.06	0.0	24.4	42.0	15.7
		SSC PFW	14	15.9	17	1.22	7.59	11.39	21.56	4.58	32.0	42.4	50.0	6.7

Table 4.2 shows the results of the second experiment. It contains, for every instance and for every s the same data of Table 4.1 plus a third group of columns with the number $s[S]$ of missing edges.

We can notice that the cpu time elapsed during the SSC versions is much lower than those of the Classic versions. The cpu usage is especially low for the SSC PFW algorithm. We attribute this difference to the large computational cost of the objective function and linear minimizer.

When it comes to the cardinality of the s-defective clique found, we do not see any large differences. We only observe that the ASFW behaves on average worse than the PFW on the p_hat problems.

Looking at the columns of the missing edges, we see some instances where the maximum $s[S]$ is significantly lower than the parameter s. This might be due to the structure of the graph (both the brock_400 graphs present this issue) or might be the fact that we do not find maximal s-defective cliques.



(a) Cardinality on the solution (b) Cpu time

Figure 4.2: Comparison of the results

Figure 4.2 portraits two box plots regarding the cpu time and the cardinality of $|S|$. The data therein are the average values from Table 4.2 normalized as described above for Figure 4.1. In this case, Figure 4.2a presents values larger than 1 because the cardinality of some s-defective cliques found is greater than that of the best known clique.

The content of Figure 4.2 confirms that the SSC versions, and SSC PFW in particular, end significantly before than the classic versions. Moreover, they produce comparable average results in terms of cardinality of S.

Chapter 5

Conclusions

In our study, we implemented four projection-free methods based on the FW algorithms. We tested the two variants that use the SSC frameworks against the respective classic ones. With these methods, we solved a set of instances of the *MCP* and *MDCP*. These cases are very interesting because they both present non-convex objective functions. Moreover, the objective and linear minimizer for the *MDCP* need more computational resources than those of the *MCP*. This difference gives us a way to understand how the different algorithms scale when the complexity of the computations, performed at the beginning of each FW iteration, increases.

Firstly, we have found that, on average, the cardinality of the solutions obtained in the SSC and in the Classic versions do not change much. This fact ensures us that, on average, the "goodness"(i.e. how low the local minimum is) of the solutions obtained will not change.

Secondly, we observed that both SSC-ASFW and SSC-PFW perform very well when f, ∇f and $\mathcal{LMO}_{\mathcal{M}}$ have a greater computational complexity. We clearly saw that, in the *MDCP*, both SSC versions out-performed both Classic versions.

A third fact, that was outside of the original scope of our study, is that, on average, the PFW variant is faster than the ASFW one.

Although, we also unveiled some un-answered questions and new possible matters of interest.

Firstly, we may analyze how the SSC behaves in comparison to classic methods when different line-search criteria are used to determine the step size. For example, we may want to try the Armijo rule.

Secondly, we may want to test how each algorithm performs on the solution of the maximum s-plex problem.

Thirdly, looking at the *MDCP*, we may want to look for an objective function that ensures us that every local minimum corresponds to a maximal s-defective clique. Such an objective might be able to find larger s-defective cliques.

Finally, we may ask ourself how to find a good local lower bound for the Lipschitz constant. We capped our bound $\tilde{L}$ to address the occurrence of a very large gradient slope in the first few iterations. This might be a matter for further investigation.

To conclude, we can say that the Short Step Chain reduces the number of iterations. As such, this procedure is a good method to obtain a fast convergence. Besides, the SSC improves its convenience as the objective and linear minimizer become more complex.

Appendix

5.1 Access the code

The code can be found at: `https://github.com/Agno94/frankwolfe_thesis`

The tests can be run following these steps:

- Clone the repository: `git clone https://github.com/Agno94/frankwolfe_thesis.git`
- Checkout to specific branch (optional): `git checkout thesistest`
- Download the DIMACS instances: `wget -c -i dimacs/list -P dimacs`
- Run the MCP tests: `python3 clique_test_script.py`
- Run the MDCP tests: `python3 defective_clique_test_script.py`

Depending on the python installation some library might be needed. They can be installed through the `pip3`(or simply `pip`) utility.

5.2 Notes on the implementation

Frank-Wolfe and SSC

Each algorithm was implemented with a subclass of the following:

```
class FW_Base_Algorithm:
    label = "Base FW Active-Set Model"
    def __init__(self, epsilon, zero_threshold, lipschitz_cap, ... **kwargs):
        ...
    def set_parameters(self, dim, vertices):
        ...
    def print(self, *args, level = 1):
        ...
    def set_objective(self, func, grad, lipschitz = 0, start_point = None, **kwargs):
        ...
    def linear_minimizer(self, g):
```

```
        ...
    def log(self, key, entry):
        ...
    def track_time(self, gap):
        ...
    def initialize_log(self,):
        ...
    def initialize(self, start_point_convcomb = None, start_lipschitz = 0, **kwargs):
        ...
    def update_lipschitz(self, gradiente, other_x_t, f_other_x):
        ...
    def run(self, max_iter = 1000, start_point_convcomb = None, **kwargs):
        ...
    def iteration(self, grad):
        pass
    def shrink_active_set(self):
        ...
```

The SSC algorithms were implemented with this parent class:

```
class FW_Base_SSC_Algorithm(fw.FW_Base_Algorithm):
    def __init__(self, multi_step_max_iter, **kwargs):
        ...
    def get_auxiliary_setsize(self, g, d, n=0):
        n = n or linalg.norm(d)
        if (n != linalg.norm(d)):
            print(" !! Error")
        if (n < self.zero_threshold**2):
            return 0
        L = self.lipschitz
        y, x = self.y_j, self.x_t
        if linalg.norm(x-y):
            s = y - x
            norm_of_s = linalg.norm(s)
            gap = g.T @ d
            # stop if norm(s) >= (g @ (d / norm(d))) / L
            if (norm_of_s * n * L > gap):
                return 0
            y2center = x + g / 2 / L - y
            r = linalg.norm(g / 2 / L)
            p1 = y2center.T @ d
            p2 = n**2 * (r**2 - y2center.T @ y2center)
            bound1 = ( p1 + (p1**2 + p2)**0.5 ) / n**2
            # BOUND1: Center x + g/2L and radius |g|/2L
            if (bound1 != bound1):
                print("   bound1 calc: p1, p2 =", p1, p2, "n=", n, "- d @ yc =", - d @ yc , "
            p1 = - s.T @ d
            p2 = ( gap**2 / L**2 - n**2 * norm_of_s**2)
            bound2 = ( p1 + (p1**2 + p2)**0.5 ) / n**2
```

```
            # BOUND2: Center x and radius g@d/2nL
        else:
            bound2 = (d.T @ g) / L / n**2
            bound1 = bound2
        self.log("bound ratio", bound2/bound1)
        self.log("bound1", bound1)
        self.log("bound2", bound2)
        beta = min(bound1, bound2)
        new_y = y + beta * d
        return min(bound1, bound2)
    def short_step_chain(self, g, e_FW_index, e_FW):
        self.e_j = self.active_set.copy()
        self.y_j = self.e_j.value()
        for j in range(0, self.multi_step_max_iter):
            r = self.small_step(g, e_FW_index, e_FW)
            if (r):
                self.print("Terminating multistep loop with code %d"%r, level=3)
                self.log("MultiStepTermination", r)
                break
        self.log("MultiStepsNumber", j+1)
        if (j == self.multi_step_max_iter - 1):
            self.print("SSCMultistepProcedure: max number of iterations reached", level=2)
    def iteration(self, grad):
        e_FW_index, e_FW = self.linear_minimizer(grad)
        gap = -grad.T @ (e_FW - self.x_t)
        if gap <= self.epsilon:
            return gap
        self.short_step_chain(-grad, e_FW_index, e_FW)
        self.active_set = self.e_j
        del self.e_j
        del self.y_j
        return gap
```

The followings are the `small_step` methods for the ASFW and PFW variants:

```
#ASFW
def small_step(self, g, e_FW_index, e_FW):
    e_Away_setIndex = np.argmax(self.e_j.matrix @ -g)
    e_Away_vertexIndex = self.e_j.indexes[e_Away_setIndex]
    e_Away = self.e_j.matrix[e_Away_setIndex]
    d_FW = e_FW - self.y_j
    n_FW = linalg.norm(d_FW)
    d_Away = self.y_j - e_Away
    n_Away = linalg.norm(d_Away)
    if ((n_Away * g.T @ d_FW) >= (n_FW * g.T @ d_Away)): #FW vs Away criterion
        step_type = "FW"
        d_j, n_j = d_FW, n_FW
        gammamax = 1
    else:
        step_type = "Away"
```

```
            d_j, n_j = d_Away, n_Away
            gammamax = self.e_j.weights[e_Away_setIndex]/(1 - self.e_j.weights[e_Away_setIndex])
        gamma_bound = self.get_auxiliary_setsize(g, d_j, n_j)
        if (gamma_bound <= 0):
            return 1
        gamma = min(gammamax, gamma_bound)
        if (step_type == "Away"):
            alpha_t = self.e_j.weights[e_Away_setIndex]
            self.e_j.weights = (1 + gamma) * self.e_j.weights
            self.e_j.weights[e_Away_setIndex] = alpha_t - gamma * (1 - alpha_t)
        else:
            self.e_j.weights = (1-gamma) * self.e_j.weights
            if (e_FW_index in self.e_j.indexes):
                i, = np.where(self.e_j.indexes == e_FW_index)
                setIndex = i[0]
                self.e_j.weights[setIndex] += gamma
            else :
                if (gamma == 1):
                    self.e_j.indexes[0] = e_FW_index
                    self.e_j.matrix[0] = e_FW
                    self.e_j.weights[0] = 1
                else:
                    self.e_j.indexes = np.hstack((self.e_j.indexes, e_FW_index))
                    self.e_j.matrix = np.vstack((self.e_j.matrix, e_FW))
                    self.e_j.weights = np.hstack((self.e_j.weights, gamma))
                    if (self.e_j.size() > (self.dim + 1)):
                        self.e_j = caratheodory(self.e_j)
        zero_weights = np.where(
            np.abs(self.e_j.weights) <= self.zero_threshold/self.e_j.size())
        if zero_weights[0].size:
            self.e_j.drop(zero_weights)
        summ = self.e_j.weights @ np.ones(self.e_j.size())
        self.e_j.weights = self.e_j.weights / summ
        y_jplus1 = self.e_j.value()
        self.y_j = y_jplus1
        return (gamma < gammamax) and 2

    # PFW
    def small_step(self, g, e_FW_index, e_FW):
        e_Away_setIndex = np.argmax(self.e_j.matrix @ -g)
        e_Away_vertexIndex = self.e_j.indexes[e_Away_setIndex]
        e_Away = self.e_j.matrix[e_Away_setIndex]
        step_type = "PFW"
        d_j = e_FW - e_Away
        n_j = linalg.norm(d_j)
        if (n_j == 0):
            return 3
        if (e_FW_index in self.e_j.indexes):
            i, = np.where(self.e_j.indexes == e_FW_index)
```

```
            setIndex = i[0]
            y_FW_j = self.e_j.weights[setIndex]
        else:
            y_FW_j = 0
        gammamax = min(1 - y_FW_j, self.e_j.weights[e_Away_setIndex])
        gamma_bound = self.get_auxiliary_setsize(g, d_j, n_j)
        if (gamma_bound <= 0):
            return 1
        gamma = min(gammamax, gamma_bound)
        self.e_j.weights[e_Away_setIndex] -= gamma
        if (y_FW_j):
            self.e_j.weights[setIndex] += gamma
        else:
            if (abs(self.e_j.weights[e_Away_setIndex]) <= 1e-12):
                self.e_j.drop(e_Away_setIndex)
            self.e_j.indexes = np.hstack((self.e_j.indexes, e_FW_index))
            self.e_j.matrix = np.vstack((self.e_j.matrix, e_FW))
            self.e_j.weights = np.hstack((self.e_j.weights, gamma))
            if (self.e_j.size() > (self.dim + 1)):
                self.e_j = caratheodory(self.e_j)
        zero_weights = np.where(
            np.abs(self.e_j.weights) <= self.zero_threshold/self.e_j.size())
        if zero_weights[0].size:
            self.e_j.drop(zero_weights)
        summ = self.e_j.weights @ np.ones(self.e_j.size())
        self.e_j.weights = self.e_j.weights / summ
        y_jplus1 = self.e_j.value()
        self.y_j = y_jplus1
        return (gamma < gammamax) and 2
```

Objective functions

The objective function for problem (3.2) is the following. To check that x represents a clique, we find every entries i of x for which $x_i \geq 10^{-3}$ and then sum the $Q = -(2A_G + 1_n)$ for those entries.

```
def create_functions(Q, c, N):
    def objective(x):
        return 1/2 * x.T @ (Q @ x) - c @ x
    def gradient(x):
        return Q @ x - c
    return objective, gradient

def check_clique(Q, x):
    findx, = np.where(abs(x) > 1e-3)
    c = findx.size
    v = abs(sum(sum(Q[findx][:, findx])) + c*(2*(c-1)+1))
    print("Check clique:", c, v)
    return v<0.1, c
```

For problem the *MDCP*. we generate the function f_x, f_y and their gradients. We implemented this we a python cycle of lenght $O(|\mathcal{V}|)$. Another possible implementation would have been to find the positive entries in y and cycle through those entries. In this case the length of the cycle would have been $O(|supp(y)|)$: this would be $O(|\bar{\mathcal{E}}|)$ when y was full of positive entries, and $O(s)$ when y is binary.

```
def create_functions(A_G, E_bar_list, c, N, M, l_yparameter = 1):
    Mbar = len(E_bar_list)
    dim = N
    E_bar_array = np.array(E_bar_list)
    edge_second_node_dict = { v1: [] for v1 in range(0,N)}
    edge_two_node_dict = {}
    i = 0
    for v1,v2 in E_bar_list:
        edge_second_node_dict[v1].append(i)
        edge_two_node_dict[(v1,v2)] = i
        edge_two_node_dict[(v2,v1)] = i
        i += 1
    for v1 in range(0,N):
        edge_second_node_dict[v1] = np.array(edge_second_node_dict[v1])
    def A_Gbar(y):
        A_y = np.zeros((N,N))
        for v1 in range(0,N):
            e_list = edge_second_node_dict[v1]
            if not len(e_list):
                continue
            try:
                v2s = E_bar_array[e_list,1]
            except IndexError:
                print(v1, e_list)
            A_y[v1,v2s] = y[e_list]
            A_y[v2s,v1] = y[e_list]
        return A_y
    def generate_with_y(y):
        A_y = A_Gbar(y)
        Q = - (A_G + A_y + 0.5 * np.eye(N))
        y_sq = y.T @ y
        def obj_of_x(x):
            f = x.T @ Q @ x - l_yparameter / 2 * y_sq
            return f
        def grad_x_of_x(x):
            return 2 * Q @ x
        return obj_of_x, grad_x_of_x
    def generate_with_x(x):
        gx = np.zeros(Mbar)
        for v1 in range(0,N):
            e_list = edge_second_node_dict[v1]
            if not len(e_list):
                continue
```

```
            try:
                v2s = E_bar_array[e_list,1]
            except IndexError:
                print(v1, e_list)
            gx[e_list] = - 2 * x[v1] * x[v2s]
        x_Q_x = - x.T @ (A_G + 0.5 * np.eye(N)) @ x
        def obj_of_y(y):
            f = x_Q_x
            nonzero, = np.where(y >= 1e-9)
            for e in nonzero:
                v1, v2 = E_bar_list[e]
                f -= 2 * x[v1] * x[v2] * y[e]
            f = f - l_yparameter / 2 * y.T @ y
            return f
        def grad_y_of_y(y):
            return gx - l_yparameter * y
        return obj_of_y, grad_y_of_y
    def check_clique(xy, S, display = 0):
        x = xy[:N]
        y = xy[N:]
        if (abs(sum(x)-1) > 1e-6):
            print("ERROR sum(x) == %.2f"%sum(x))
        A_y = A_Gbar(y)
        A_y[np.where(abs(A_y) >= 1e-5)] = 1
        findx, = np.where(abs(x) > 1e-3/N)
        findy, = np.where(abs(y) > 1e-5)
        print("x, y, x.size, y.size, findx", x, y, x.size, y.size, findx, findy)
        c = findx.size
        vx = sum(sum(A_G[findx][:, findx]))
        vy = sum(sum(A_y[findx][:, findx]))
        missing_egdes = vy / 2
        if missing_egdes > S+1e-8:
            print("ERROR", S)
        delta = (vx + vy) - (c)*(c-1)
        result = abs(delta) < 0.1
        print("Check is", result,
            " (clique size:%d, vx:%.1f, vy:%.1f, delta:%.2f)"%(c, vx, vy, delta))
        if display:
            A_show = 0.8 * A_G + 0.5 * A_y
            A_show[findx][:, findx] += 1.1 * A_G[findx][:, findx]
            plt.matshow(A_show[findx][:, findx])
        if display > 1:
            y_vertices = listOfVertices(findy)
            print("y's edges' vertices:",y_vertices)
            indexes = list(findx) + sorted(list(set(y_vertices) - set(findx)))
            print("  indexes=",indexes)
            tot_vertices = np.array([int(x) for x in indexes])
            plt.matshow(A_show[tot_vertices][:, tot_vertices], cmap='plasma')
        plt.show()
```

```
        return (result, c, missing_egdes)
    return generate_with_y, generate_with_x, check_clique
```

Starting points generation

The following is used to generate the starting point of the *MDCP*. For the i-th test of a given instance, the value i was passed.

```
def generate_random_start_01y(i = None):
    start_indexes = np.arange(0,N)
    if (i != None):
        np.random.seed(i)
    w = np.random.rand(N)
    xstart_weights = w / sum(w)
    xstart = (x_vertices[start_indexes], start_indexes, xstart_weights)
    num_y_tochoose = np.random.randint(1,S)
    chosen_edges = np.random.choice(np.arange(0,Mbar), size=num_y_tochoose)
    ystart_list = [sorted(list(chosen_edges))]
    ystart_matrix = np.zeros((1, Mbar))
    ystart_matrix[0, chosen_edges] = 1
    return xstart, ystart_matrix, ystart_list, False
```

Linear Minimizer for $D_{s,m}$

The following snippet code was used in the linear liminizer of `gy` over $D_{s,m}$. `threaded_multiple_arg_min` is a function that find s minimum entry using the processor multi-core capabilities.

```
neg_y_direction, = np.where(gy <= 0)
min_neg_dir = threaded_multiple_arg_min(gy[neg_y_direction], self.S)
indexes_y = neg_y_direction[min_neg_dir]
index_sorted = sorted(list(indexes_y))
index_y = self.get_vertex(index_sorted)
```

Bibliography

[1] M. Frank and P. Wolfe, "An algorithm for quadratic programming," *Naval Research Logistics Quarterly*, vol. 3, no. 1-2, pp. 95–110, 1956.

[2] S. Lacoste-Julien and M. Jaggi, "On the global linear convergence of frank-wolfe optimization variants," 2015.

[3] F. Rinaldi and D. Zeffiro, "A unifying framework for the analysis of projection-free first-order methods under a sufficient slope condition," 2020.

[4] F. Rinaldi and D. Zeffiro, "Avoiding bad steps in frank wolfe variants," 2021.

[5] T. S. Motzkin and E. G. Straus, "Maxima for graphs and a new proof of a theorem of turán," *Canadian Journal of Mathematics*, vol. 17, p. 533–540, 1965.

[6] J. T. Hungerford and F. Rinaldi, "A general regularized continuous formulation for the maximum clique problem," *Mathematics of Operations Research*, vol. 44, no. 4, pp. 1161–1173, 2019.

[7] V. Stozhkov, A. Buchanan, S. Butenko, and V. Boginski, "Continuous cubic formulations for cluster detection problems in networks," 2020.

[8] I. M. Bomze, F. Rinaldi, and D. Zeffiro, "Continuous formulation for s-defective clique," 2021. in preparation.

[9] P. Wolfe, "Convergence theory in nonlinear programming," *Integer and nonlinear programming*, pp. 1–36, 1970.

[10] C. Carathéodory, "Über den variabilitätsbereich der koeffizienten von potenzreihen, die gegebene werte nicht annehmen," 1907.

[11] L. Armijo, "Minimization of functions having lipschitz continuous first partial derivatives.," *Pacific J. Math.*, vol. 16, no. 1, pp. 1–3, 1966.

[12] A. Cristofari, M. De Santis, S. Lucidi, and F. Rinaldi, "An active-set algorithmic framework for non-convex optimization problems over the simplex," *Computational Optimization and Applications*, vol. 77, p. 57–89, May 2020.

[13] I. M. Bomze, F. Rinaldi, and D. Zeffiro, "Active set complexity of the away-step frank-wolfe algorithm," 2019.

[14] D. Johnson and M. Trick, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993.* Center for Discrete Mathematics and Theoretical Computer Science New Brunswick, NJ: DIMACS series in discrete mathematics and theoretical computer science, American Mathematical Society, 1996.

[15] F. Mascia, "Dimacs benchmark set." `http://iridia.ulb.ac.be/~fmascia/maximum_clique/`. Accessed: 2021-02-16.