CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Comparative Study on Optimization Methods for Correlation Clustering

Master's thesis in Computer science and engineering

DRIKVY V. CAPPENBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Comparative Study on Optimization Methods for Correlation Clustering

DRIKVY V. CAPPENBERG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Comparative Study on Optimization Methods for Correlation Clustering
DRIKVY V. CAPPENBERG

Supervisor: Morteza Haghir Chehreghani, Department of Computer Science and Engineering
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Comparative Study on Optimization Methods for Correlation Clustering
DRIKVY V. CAPPENBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Correlation clustering is an optimization problem that aims to create partition of data based on pairwise similarity coefficients that represents the level of similarities between data observations. The thesis focused on the maximization of agreements version of the problem in which to find clustering of data where the data that belong to the same cluster have maximized agreements. The thesis aims to give more details on how different methods are used for correlation clustering problem, how they perform and what are the similarities between these methods. The well known linear programming methods as well as simple iterative algorithms are compared in terms of their runtime and correctness.

Keywords: Correlation Clustering, Maximization of Agreements, Optimization, Comparative Study.

Acknowledgements

I would like to express my sincere gratitude towards the Department of Computer Science and Engineering (CSE), especially for my supervisor that has given me proper assistance and sufficient knowledge to accomplished this thesis project. I would like to thank Indonesia Endowment Fund for Education that has given me full scholarship to complete my study. I would like to give my utmost gratitude to Alberta Maria who has accompanied me through my master study in Sweden. I would also like to thank my family and friends for their support during my master study at Chalmers University of Technology.

Drikvy V. Cappenberg, Gothenburg, February 2019

Contents

List of Figures

List of Tables

1
Introduction

1.1 Correlation Clustering

Clustering of objects into groups is a common task that arises in many applications such as data mining, web analysis, marketing, pattern recognition etc. In a theoretical setting, the objects are usually viewed as points in either a metric space or a general distance matrix, or as vertices in a graph [3]. There are many variants of clustering problem, but this thesis will particularly concentrate on the correlation clustering problem.

Correlation clustering is a clustering problem that can be represented as a complete graph on n vertices (items), where each edge (u, v) is labeled either + or - depending on whether u and v have been deemed to be similar or different [2]. In addition, each edge has real non-negative weight which interpreted as confidence measure of the similarity or dissimilarity of the edge's endpoints (higher weight denotes higher confidence) [3]. The goal is to produce a partition of the vertices (a clustering) that agrees as much as possible with the edge labels [2].

Optimizing such cost function is NP-hard. Therefore, the optimal solution should be approximated [4]. Several approximation method have been formulated such as: PTAS for MAXCUT on dense graphs to maximize agreements with $O(n^2 e^{O(1/\epsilon)})$ time [2] and an $O(log\ n)$-approximation algorithm to minimize disagreements for (weighted) general graphs with linear-programming rounding using the "region-growing" technique. On the other hand, it is also shown that the problem is equivalent to minimum multicut, and therefore it is APX-hard and difficult to approximate better than $\Theta(log\ n)$ [3].

The cost for correlation clustering and min cut are equivalent cost functions, i.e. the cost functions share the same optimal solution. It is proposed to solve the optimization problem for correlation clustering using Shifted Min Cut cost function, with the number of clustering K is specified. A local search method was developed which computes a local minimum of the cost function for $O(tKn^2)$ time [4].

Recently, there is a new approach for correlation clustering optimization based on Frank-Wolfe (FW) framework [1]. This new approach is introduced to the problem based on a natural non-convex relaxation which observed to be well suited to the classical FW or conditional optimization algorithm and develop new algorithms for correlation clustering [1]. Referring to recent advances in the FW method that are proven to be very suitable for large scale optimization, they take advantage of the block coordinate version of the FW algorithm which has fast convergence properties. It is shown that the basic approach leads to a simple and natural local search algorithm with guaranteed convergence [1].

1.2 Contributions

The contribution of this thesis work is to show the differences among well known optimization methods for correlation clustering in terms of runtime scalability and the quality of solutions obtained. The thesis also aims to elaborate in detail regarding the drawbacks that have caused worse performance on some of the optimization methods, bottlenecks, algorithmic similarity and findings that are still relevant in the scope of the thesis. Chapter 2 will elaborate the background theories that are fundamental to the implementations used in the experiments in order to compare different optimization methods. Chapter 3 will discuss the details of implementation of different optimization methods. Chapter 4 will discuss the results obtained by implementation of different optimization methods.

2

Theory

2.1 Correlation Clustering Formulation

2.1.1 Problem Formulation

Correlation clustering is a clustering problem that consists of a complete and dense graph $G = (V, E)$ where V is the set of all nodes in the graph with size n. E is the set of all edges in the graph, where each edge $e \in E$ connects two different nodes $e = (i, j)$ and has positive or negative weights w_{ij}. The aim is to partitioned the nodes such that the weights of edges within a cluster is maximized. The exact formulation of this clustering problem has been specified by Williamson and Shmoys [6] as follows:

$$\begin{aligned} max \sum_{(i,j)\in E} w_{ij}(v_i \cdot v_j) \\ v_i \in \{e_1, ..., e_k\}, \quad i = 1, ..., n \end{aligned} \tag{2.1}$$

The natural convex relaxation of the exact formulation is:

$$\begin{aligned} max \sum_{(i,j)\in E} w_{ij}(v_i \cdot v_j) \\ v_i \in ConvexHull(e_1, ..., e_k), \quad i = 1, ..., n \end{aligned} \tag{2.2}$$

Where v_i is the solution to the clustering optimization. v_i is a vector with length k that represents the clustering assignment for node $i \in V$ where each element $m = 0, ..., k$ of vector v_i denotes the clustering assignment probability for node i in m^{th} cluster. e_k is the k^{th} unit vector where it has a one in the k^{th} coordinate and zero elsewhere.

2.1.2 Randomized Rounding

The exact clustering solution is obtained through randomized rounding over the approximated solution in v_i. The natural randomized rounding algorithm based on the optimal solution to $v_i^*, i = 1, ..., k$. For $i = 1, ..., n$ independently, generate a distribution on 0/1 vectors with exactly one 1 (i.e. a distribution on $e_1, ..., e_k$) with marginals $v_{i,j}^*$, and let $\hat{v_i} = e_k$ with this distribution. A simple way to do this as follows: divide the unit interval into k parts with the j^{th} part having size $v_{i,j}^*$. Then choose U uniformly at random with interval $[0, 1]$. If U falls into the j^{th} interval, set $\hat{v_i} = e_j$.

Note that with this rule,

$$\begin{aligned} P[\hat{v}_i = \hat{v}_j] &= \sum_s P[\hat{v}_i = e_s = \hat{v}_j] \\ &= \sum_s P[\hat{v}_i = e_s] P[\hat{v}_j = e_s] \\ &= \sum_s v^*_{i,s} \cdot v^*_{j,s} \\ &= v^*_i \cdot v^*_j \end{aligned} \tag{2.3}$$

and the expected value of the resulting solution is:

$$\begin{aligned} E\left[\sum_{(i,j)\in E} 1[\hat{v}_i = \hat{v}_j] w_{i,j} \right] &= \sum_{(i,j)\in E} P[\hat{v}_i = \hat{v}_j] w_{i,j} \\ &= \sum_{(i,j)\in E} v^*_i \cdot v^*_j w_{i,j} \end{aligned} \tag{2.4}$$

Which is the optimal solution value of the relaxation.

2.2 Optimization Methods

2.2.1 Quadratic Programming (QP)

Quadratic programming (QP) is the problem of optimizing a quadratic objective function. The objective function can contain bilinear or up to second order polynomial terms, and the constraints are linear and can be both equalities and inequalities. Quadratic programming is a particular type of nonlinear programming. The quadratic programming problem with n variables and m constraints can be specified as follows:

1. a real-valued, n dimensional vector c,
2. an $n \times n$-dimensional real symmetric matrix Q,
3. an $m \times n$-dimensional real matrix A, and
4. an an m-dimensional real vector b,

$$\begin{aligned} min \quad & \frac{1}{2} x^\top Q x + c^\top x \\ subject\ to \quad & Ax \leq b \end{aligned} \tag{2.5}$$

The objective function is arranged such that the vector c contains all of the once-differentiated linear terms and Q contains all of the twice differentiated quadratic terms. In other words, Q is the Hessian matrix of the objective function and c is its gradient. By convention, any constants contained in the objective function are left out of the general formulation. The one-half in front of the quadratic term is included to remove the coefficient that results from taking the derivative of a second-order polynomial.

$x^\top$ denotes the vector transpose of x. The notation $Ax \leq b$ means that every entry of the vector Ax is less than or equal to the corresponding entry of vector b.

If the objective function is convex, then any local optima found is also global optima. To analyze the function's convexity, one can compute its Hessian matrix and verify that the matrix Q is positive definite. This is a sufficient condition, since it is not required to be true for a local optima to be the unique global optima, but will guarantee that this property holds if true. For positive definite Q, it is known that ellipsoid method solves the problem in polynomial time. Whereas only one negative eigenvalue in Q makes the problem NP-hard. Correlation clustering can be formulated as the following quadratic programming problem:

$$
\begin{aligned}
&max \sum_{(i,j)\in E} w_{ij}(v_i \cdot v_j) \\
&subject\ to \\
&\sum_{m=0}^{k} v_{im} = 1, \quad i = 1, ..., n, \\
&v_{im} + v_{jm} \geq 0, \quad \forall\, i \neq j, \quad m \in [0, k[, \\
&v_{im} \in [0, 1], \quad \forall\, i, \quad m \in [0, k[,
\end{aligned}
\tag{2.6}
$$

Matrix Q from the formulation 2.6 is a zero-diagonal symmetric matrix. The trace of matrix Q is zero and since the sum of eigenvalues equals the trace of the matrix, there must be at least one negative eigenvalue of Q. Hence Q is not positive definite and this makes formulation 2.6 NP-hard.

2.2.2 Semidefinite Programming (SDP)

Semidefinite programming uses a symmetric, positive semidefinite matrices. A matrix $X \in \Re^{n\times n}$ is positive semidefinite (PSD) iff for all $y \in \Re^n$, $y^\top Xy \geq 0$. Symmetric PSD matrices have some special properties. If $X \in \Re^{n\times n}$ is a symmetric metrix, then the following properties are equivalent:

1. X is PSD;
2. X has non-negative eigenvalues;
3. $X = V^\top V$ for some $V \in \Re^{m\times n}$ where $m \leq n$.

A semidefinite program (SDP) is similar to a linear program in that there is a linear objective function and linear constraints. In addition, however, a square symmetric matrix of variables can be constrained to be positive semi-definite. Below is an example in which the variables are x_{ij} for $1 \leq i, j \leq n$.

$$\begin{aligned} \textit{max or min} \quad & \sum_{i,j} c_{ij} x_{ij} \\ \textit{subject to} \quad & \sum_{i,j} a_{ijk} x_{ij} = b_k, \quad \forall\, k \\ & x_{ij} = x_{ji}, \qquad \forall\, i,j \\ & X = (x_{ij}) \succeq 0 \end{aligned} \tag{2.7}$$

Semidefinite programming can be represented as vector programming. The variables of vector programs are vectors $v_i \in \Re^n$, where the dimension n of the space is the number of vectors in the vector program. The vector program has an objective function and constraints that are linear in the inner product of these vectors. The inner product of v_i and v_j can be written as $v_i \cdot v_j$. Below is the example of a vector program.

$$\begin{aligned} \textit{max or min} \quad & \sum_{i,j} c_{ij} (v_i \cdot v_j) \\ \textit{subject to} \quad & \sum_{i,j} a_{ijk} (v_i \cdot v_j) = b_k, \quad \forall\, k \\ & v_i \in \Re^n, \qquad i = 1, ..., n \end{aligned} \tag{2.8}$$

Williamson and Shmoys, claimed that SDP 2.6 and vector program 2.7 are equivalent. This follows since a symmetric X is PSD if and only if $X = V^\top V$ for some matrix V. Given a solution to the SDP 2.6, we can take the solution X, compute in polynomial time a matrix V for which $X = V^\top V$ (within small error), and set v_i to be the i^{th} column of V. Then $x_{ij} = v_i \cdot v_j$, and the v_i are a feasible solution of the same value of the vector program 2.7. Similarly, given a solution v_i to the vector program, we construct a matrix V whose i^{th} column is v_i and let $X = V^\top V$. Then X is symmetric ans PSD, with $x_{ij} = v_i \cdot v_j$, so that X is a feasible solution of the same value for the SDP 2.6 [6].

Semidefinite programming is a well known method that can be used to obtain good correlation clustering in an undirected graph. Williamson and Shmoys in 2011 have shown that one can obtain a $\frac{3}{4}$-approximation algorithm by using semidefinite programming [6]. The following is the proposed model of the problem in a vector program:

$$\begin{aligned} \textit{max} \quad & \sum_{(i,j) \in E} \left(w_{ij}^{+} (v_i \cdot v_j) + w_{ij}^{-} (1 - v_i \cdot v_j) \right) \\ & \textit{subject to} \\ & v_i \cdot v_i = 1, \quad \forall\, i, \\ & v_i \cdot v_j \geq 0, \quad \forall\, i,j, \\ & v_i \in \Re^n, \quad \forall\, i. \end{aligned} \tag{2.9}$$

Where w_{ij}^{+} is the degree to which i and j are similar, and w_{ij}^{-} is the degree to which they are different. Note that in this formulation, both w_{ij}^{+} and w_{ij}^{-} are non+negative weights.

2.2.2.1 Interior point method

SDP is in fact a special case of cone programming and can be solved efficiently with interior point methods. Most code implementations for solving SDP are based on interior point methods. Robust and efficient for general linear SDP problems. Restricted by the fact that the algorithms are second-order methods and need to store and factorize a large matrix. Interior point method for nonlinear optimization can be described as follows, consider the all-inequality version of a nonlinear optimization problem:

$$\begin{aligned} &\min \quad f(x) \\ &\text{subject to} \;\; c_i(x) \geq 0 \text{ for } \; i = 1, ..., m, \; x \in \Re^n, \\ &\text{where } f : \Re^n \to \Re, \; c_i : \Re^n \to \Re \end{aligned} \tag{2.10}$$

The logarithmic barrier function associated with 2.10 is:

$$B(x, \mu) = f(x) - \mu \sum_{i=1}^{m} log(c_i(x)) \tag{2.11}$$

Here μ is a small positive scalar, sometimes called the barrier parameter. As μ converges to zero, the minimum of $B(x, \mu)$ should converge to a solution of 2.10. The barrier function gradient is:

$$g_b = g - \mu \sum_{i=1}^{m} \frac{1}{c_i(x)} \nabla c_i(x) \tag{2.12}$$

Where g is the gradient of the original function $f(x)$, and ∇c_i is the gradient of c_i. In addition to the primal variable x, Lagrange multiplier inspired dual variable $\lambda \in \Re^m$ as follows:

$$c_i(x)\lambda_i = \mu, \forall i = 1, ..., m \tag{2.13}$$

The aim is to try to find those (x_{μ}, μ) for which the gradient of the barrier function is zero. Applying 2.13 to 2.12, we get an equation for the gradient:

$$g - A^T \lambda = 0 \tag{2.14}$$

Where the matrix A is the Jacobian of the constraints $c(x)$. Applying Newton's method to 2.13 and 2.14, we get an equation for (x, λ) update (p_x, p_λ):

$$\begin{pmatrix} W & -A^T \\ \Lambda A & C \end{pmatrix} \begin{pmatrix} p_x \\ p_\lambda \end{pmatrix} = \begin{pmatrix} -g + A^T\lambda \\ \mu - C\lambda \end{pmatrix} \tag{2.15}$$

Where W is the Hessian matrix of $B(x, \mu)$, *Lambda* is a diagonal matrix of λ, and C is a diagonal matrix with $C_{ii} = c_i(x)$. Because of 2.10 and 2.13, the condition $\lambda \geq 0$ should be enforced at each step. This can be done by choosing appropriate α:

$$(x, \lambda) \rightarrow (x + \alpha p_x, \lambda + \alpha p_\lambda) \tag{2.16}$$

2.2.3 Frank-Wolfe Algorithm (FW)

In 1956 Frank and Wolfe developed an algorithm for solving quadratic programming problems with linear constraints. The Frank-Wolfe (FW) or conditional gradient algorithm is one of the oldest methods for nonlinear constrained optimization and has seen an impressive revival in recent years due to its low memory requirement and projection-free iterations [7]. In its classical form, the Frank-Wolfe algorithm can solve problems of the form:

$$max_{x \in D}\ f(x) \tag{2.17}$$

Where f is differentiable with $L-$Lipschitz gradient and the domain D is a convex and compact set. Using the block coordinate version of the Frank-Wolfe algorithm from (Lacoste-Julien et al. 2013), This method applies to problem of the form [1]:

$$max_{v \in M^{(1)} \times M^{(2)} \times ... \times M^{(n)}}\ f(v) \tag{2.18}$$

Where f is a concave function over $M^{(1)} \times M^{(2)} \times ... \times M^{(n)}$ and $M^{(i)} \subseteq \Re^{n_i}$ is convex and compact for $i = 1, ..., k$. This method works well when certain subproblem are computationally cheap to solve when only considering the variables in one variable block $M^{(i)}$ at a time. This method is described below [1]:

Algorithm 1 Block Coordinate Frank-Wolfe Algorithm

1: Let the initial solution $v^0 \in M^{(1)} \times M^{(2)} \times ... \times M^{(n)}$
2: for $t = 1, ..., T$ do
3: Pick object i at random in $\{1, ..., n\}$
4: Find $s_i = argmax_{s_i' \in M^{(i)}}\ s_i' \cdot \nabla_i f(v^{(t)})$
5: Let $\gamma = 2k/(t + 2k)$ or optimize γ by line-search
6: Update $v_{(i)}^{(t+1)} = (1 - \gamma)v_{(i)}^{(t)} + \gamma s_{(i)}$

Block-coordinate version of the algorithm is shown to converge for non-convex functions. By extending the results from (Lacoste-Julien et al. 2013; Reddi et al. 2016), the convergence of Frank-Wolfe algorithms for non-concave problems can be measured by the Frank-Wolfe duality gap [1]:

$$d(v) = max_{s \in M}(s - v) \cdot \nabla f(v) \tag{2.19}$$

Which is zero if and only if v is a stationary point. Let $\tilde{d}_t = min_{0 \leq s \leq t} d(v^{(t)})$. For f continuously differentiable (but not necessarily concave) with finite curvature constant C over M, we have that the Frank-Wolfe iterates with line search satisfy $E\left[\tilde{d}_t\right] \leq Cn/\sqrt{T}$. If f is multilinear in the variable blocks: $E\left[\tilde{d}_t\right] \leq n(f^* - f(v^0))/T$. This convergence rate of $O(1/t)$ in expectation for block coordinate Frank-Wolfe should be compared to the (deterministic) convergence of the duality gap of $O(1/\sqrt{t})$ for general non-concave functions with ordinary Frank-Wolfe developed in (Reddi et al. 2016) [1].

2.2.3.1 Local Search Frank-Wolfe (LSFW)

Using the line search version of the FW algorithm, it is observed that it simplifies to the combinatorial local search algorithm [1]. To see why this holds, consider the so called linear programming oracle (LPO) from FW algorithm:

$$s_i = argmax_{s_i' \in M^{(i)}} \, s_i' \cdot \nabla_i f(v^{(t)}) \tag{2.20}$$

Here $\nabla_i f(v^{(t)}) = \sum_{(i,j) \in E} w_{ij} v_j^{(t)}$. Assuming that all v_i are integers, the k^{th} element of this vector is the cost associated with assigning node i to cluster k. Here, the solution to 2.20 is given by $s_i = e_j^*$ where $j^* = argmax_{j \in [k]}(\nabla_i f)_j$ [1]. Therefore, the LPO returns a 0/1 vector which has the highest objective when all other variables remain fixed. This particular instatiation of FW framework is similar to the greedy method in (Elsner and Schudy 2009), called BOEM [1]. This local search algorithm for correlation clustering is defined as follows [1]:

Algorithm 2 Local Search Frank-Wolfe Algorithm

1: Initialize $v_i \in \{e_1, ..., e_k\}$ randomly for $i = 1, ..., n$
2: while not converged do
3: Select i uniformly at random from [n]
4: Assign i to a cluster which maximizes the correlation clustering objective

2.2.4 Gradient Descent Methods

Gradient descent is a well known iterative method that utilizes gradient and step length to find stationary points of a function. It is usually used for unconstrained optimization problem that can be formulated as,

$$minimize \quad f(\mathbf{x}), \mathbf{x} \in \Re^n \tag{2.21}$$

In this method, one generates a sequence of iterates $\mathbf{x}_j$ that gradually converges towards a local optimum, beginning with a starting point $\mathbf{x}_0$. In essence, the algorithm generates new iterates as,

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \gamma_j \mathbf{d}_j \tag{2.22}$$

where $\mathbf{d}_j$ is the search direction at the point $\mathbf{x}_j$, and γ_j is the step length. The step length is typically adaptive, i.e. its value also depends on $\mathbf{x}$.
Making a Taylor expansion of $f(\mathbf{x})$ at $\mathbf{x} = \mathbf{x}_0$, and neglecting higher-order terms, one obtains,

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) \tag{2.23}$$

Since the aim is to minimize $f(\mathbf{x})$, starting from $\mathbf{x} = \mathbf{x}_0$, one should try to find an x such that $f(\mathbf{x}) < f(\mathbf{x}_0)$. There might be many search directions for which this condition holds, but the direction in which $f(\mathbf{x})$ decreases most is given by the negative gradient at $\mathbf{x}_0$. Thus, given only gradient information as in 2.23, a suitable choice of search direction is given by,

$$\mathbf{d}_0 = -\nabla f(\mathbf{x}_0) \tag{2.24}$$

Returning to 2.22, the iteration now takes the form,

$$\mathbf{x}_{j+1} = \mathbf{x}_j - \gamma_j \nabla f(\mathbf{x}_j) \tag{2.25}$$

The step length γ_j can be a fixed real number or typically optimized e.g. using line search. In cases where the objective function has a very complex shape, line search may be computationally expensive, and one may resort to use a fixed step length γ (typically determined through empirical tests) or a slowly decreasing γ. The gradient descent algorithm typically generates a zig-zag search path towards the local minimum. In fact, under gradient descent, consecutive search directions are orthogonal to each other [13].

3
Methods

3.1 Evaluation of optimization methods for correlation clustering

Several optimization methods are evaluated in terms of their scalability and correctness. The methods that are being evaluated are QP, SDP, LSFW and gradient descent. In chapter 2, it has been discussed regarding the underlying theory behind correlation clustering problem. The formulation of correlation clustering on several optimization methods also presented. This chapter will discussed on how to implement these formulation using off-the-shelf method using academic license solver and basic programming language such as Java or Matlab.

3.1.1 Implementation of QP

Given the quadratic programming formulation 2.6 for correlation clustering, one can construct a model of this formulation using well known solver. In this implementation, academic license IBM ILOG CPLEX optimizer will be used, specifically the Java interface from CPLEX Concert Technology. Concert Technology is a set of libraries offering an API that includes modeling facilities to allow a programmer to embed CPLEX optimizers in C++, Java or .NET applications. The advantage of using CPLEX is that it allows the optimization of a quadratic programming problem even if Q is not positive semidefinite matrix. In this case, it will try to find global optima of the quadratic program and since Q is not a PSD matrix, the quadratic program is NP-hard.

In order to build an optimization model in CPLEX, it is first needed to declare an instance of a CPLEX model using `IloCplex` object. There are three types of parameters that can be used to setup the quality of solution in CPLEX. One can setup CPLEX to find global optima by set the following parameter `IloCplex.Param.OptimalityTarget` with value 3 or a static parameter `IloCplex.OptimalityTarget.OptimalGlobal` after the model has been created.

After the the model is created, variables, constraints and objective function can be specified and later added into the model. The following paragraphs will show the step by step explanations on how CPLEX is used to optimize correlation clustering problem where the number of nodes is $N = 4$ and number of clustering is $K = 2$. The weight matrix is known and has been generated beforehand.

The optimization problem can be constructed in CPLEX by first declaring a CPLEX optimization model. Listing 3.1 below, shows how to do this and in this case the optimality target of the solution is setup to find the global optima.

Listing 3.1: Constructing a CPLEX model to find global optima.

```
IloCplex model = new IloCplex();
model.setParam(IloCplex.Param.OptimalityTarget, 3);
IloLPMatrix lp = model.addLPMatrix();
```

Variables can be added into the model as shown in listing 3.2 below. Take note that the third set of constraints in equation 2.6 where all the elements in the decision vector v_i must be between 0 and 1 is presented as setting up the lower and upper bound arrays. Each element in the array setup the upper or lower bound of a particular variable. In this implementation, we are treating each element in the decision vector as separate variables. Nevertheless, the construction of other constraints will arrange the clustering solution as expected.

Listing 3.2: Construct variables and add them into CPLEX model.

```
// variables
double[] lb = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
double[] ub = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
for(int i = 0; i < N*K; i++) { varX[i] = "x"+i;}
IloNumVar[] x  = model.numVarArray(model.
                    columnArray(lp, N*K), lb, ub, varX);
```

Listing 3.3 shows how both equality and inequality constraints are added into the optimization model. As we can see, the separated variables now depend on each other as dictated by the constraints in 2.6 acting as elements in the decision vector just as we wanted. The first and second set of constraints in 2.6 can be easily added to the model where *addEq* represents creating equality constraints and *addGe* represents inequality constraints "greater equal".

Listing 3.3: Equality and inequality constraints in CPLEX model.

```
// constraints
// the first set of constraints
model.addEq(model.sum(x[0], x[1]), 1.0);
model.addEq(model.sum(x[2], x[3]), 1.0);
...

// the second set of constraints
model.addGe(model.sum(x[0], x[2]), 0.0);
model.addGe(model.sum(x[1], x[3]), 0.0);
...
model.addGe(model.sum(x[4], x[6]), 0.0);
model.addGe(model.sum(x[5], x[7]), 0.0);
```

Finally, the objective that is supposed to be maximized can be constructed as shown by listing 3.4 where in this case an optimization to a correlation clustering objective of $N = 4$ and $K = 2$ is assumed. Below, it is obviously shown that CPLEX model can consists of several smaller models that will built up the entire objective expression that will be solved.

Listing 3.4: Objective maximization in CPLEX.

```
// objective
IloNumExpr x010 = model.prod(1.0, x[0], x[2]);
IloNumExpr x011 = model.prod(1.0, x[1], x[3]);
IloNumExpr q1   = model.prod(model.sum(x010, x011), w[0][1]);
...
IloNumExpr x230 = model.prod(1.0, x[4], x[6]);
IloNumExpr x231 = model.prod(1.0, x[5], x[7]);
IloNumExpr q6   = model.prod(model.sum(x230, x231), w[2][3]);

IloNumExpr Q = model.sum(q1, q2, q3, q4, q5, q6);
model.addMaximize(Q);

cplex.solve();
cplex.exportModel("cc_qp_test.lp");
```

Listing 3.5 shows the exported model, a human-readable representation of the model that is being solved in CPLEX. The exported model helps us to clarify that the constructed model is correct to our QP formulation.

Listing 3.5: Exported QP model in CPLEX.

```
Maximize
 obj: [ 2 x0 * x2 - 2 x0 * x4 - 2 x0 * x6 + 2 x1 * x3 -
        2 x1 * x5 - 2 x1 * x7 - 2 x2 * x4 - 2 x2 * x6 -
        2 x3 * x5 - 2 x3 * x7 + 2 x4 * x6 + 2 x5 * x7] / 2
Subject To
 c1:  x0 + x1  = 1    c9:  x1 + x5 >= 0
 c2:  x2 + x3  = 1    c10: x1 + x7 >= 0
 c3:  x4 + x5  = 1    c11: x2 + x4 >= 0
 c4:  x6 + x7  = 1    c12: x2 + x6 >= 0
 c5:  x0 + x2 >= 0    c13: x3 + x5 >= 0
 c6:  x0 + x4 >= 0    c14: x3 + x7 >= 0
 c7:  x0 + x6 >= 0    c15: x4 + x6 >= 0
 c8:  x1 + x3 >= 0    c16: x5 + x7 >= 0
Bounds
 0 <= x0 <= 1    0 <= x4 <= 1
 0 <= x1 <= 1    0 <= x5 <= 1
 0 <= x2 <= 1    0 <= x6 <= 1
 0 <= x3 <= 1    0 <= x7 <= 1
End
```

By default, CPLEX is using branch and cut algorithm to find the global optima to the constructed QP formulation of correlation clustering problem.

3.1.2 Implementation of SDP

There's not much known about academic license optimizer that is able to construct the model for a vector program optimization as shown by the formulation 2.8. In this case, given the vector program for correlation clustering 2.8 and the equivalence between vector program and SDP as shown by Williamson and Shmoys [6], the semidefinite program for correlation clustering is defined as:

$$\begin{aligned} max \quad & \frac{1}{2}\left\langle \boldsymbol{W}, \boldsymbol{X} \right\rangle \\ subject\ to \quad & x_{ij} = 1, \qquad \forall\, i = j \\ & x_{ij} \geq 0, \qquad \forall\, i \neq j \\ & \boldsymbol{X} = (x_{ij}) \succeq 0 \\ & x_{ij} = x_{ji}, \qquad \forall\, i, j \end{aligned} \tag{3.1}$$

Here standard notation is used for the matrix inner product, i.e, for $\boldsymbol{A}, \boldsymbol{B} \in \Re^{m \times n}$ we have $\left\langle \boldsymbol{A}, \boldsymbol{B} \right\rangle = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \boldsymbol{A_{ij}} \boldsymbol{B_{ij}}$. The solution to correlation clustering is a symmetric PSD matrix X. The implementation of SDP program can be carried out using an academic license MOSEK. In this particular evaluation, Fusion API from MOSEK Java interface will be used. Below is the example of creating a model of SDP optimization for correlation clustering with $N = 4$ and $K = 2$ in MOSEK.

Similar to CPLEX, SDP formulation must be constructed as optimization model in MOSEK. Listing 3.6, shows how the model is constructed using `Model` class. There are also several settings for the model such as `setLogHandler()` to turn on logging or printing the output of the optimizer while solving process is taking place.

Listing 3.6: Constructing model in MOSEK using Fusion API.

```
Model model = new Model("cc_sdp_test");
model.setSolverParam("presolveUse", "off");
model.setLogHandler(new PrintWriter(System.out));
```

Constructing variable using Fusion API in MOSEK is very simple especially for SDP. Listing 3.7 shows the construction of 4 by 4 symmetric semidefinite matrix X. Here we can see that the matrix of variables of our correlation clustering problem will grows quadratically as the number of nodes N increased.

Listing 3.7: Construct variables in MOSEK.

```
// setup variables into the model
Variable X = model.variable("X", Domain.inPSDCone(4));
```

Listing 3.8 shows how the SDP objective from 3.1 is constructed using MOSEK Fusion API. The method `objective()` setup the objective to be maximized together with the expression of the objective as the second parameter. We can see that it is almost similar to CPLEX where the objective consists of smaller expressions.

Listing 3.8: Construct SDP objective in MOSEK.

```
// define the objective
model.objective(ObjectiveSense.Maximize,
                    Expr.mul(0.5, Expr.dot(W, X)));
```

Finally, the constraints are constructed as shown by listing 3.9. When all the variables, objective and constraints have been specified for the model, the optimization is then able to be executed simply with the statement `model.solve()`.

Listing 3.9: Construct the constraints in MOSEK.

```
// define the constraints
model.constraint("c1",X.index(0,0),Domain.equalsTo(1.0));
model.constraint("c2",X.index(1,1),Domain.equalsTo(1.0));
model.constraint("c3",X.index(2,2),Domain.equalsTo(1.0));
model.constraint("c4",X.index(3,3),Domain.equalsTo(1.0));
model.constraint("c5",X.index(0,1),Domain.greaterThan(0.0));
model.constraint("c6",X.index(0,2),Domain.greaterThan(0.0));
model.constraint("c7",X.index(0,3),Domain.greaterThan(0.0));
model.constraint("c8",X.index(1,2),Domain.greaterThan(0.0));
model.constraint("c9",X.index(1,3),Domain.greaterThan(0.0));
model.constraint("c10",X.index(2,3),Domain.greaterThan(0.0));

// solve the model
model.solve();
```

When the model is solved successfully, the following instructions can be used to obtain both the decision variables and objective value respectively.

Listing 3.10: Obtain the result of optimization in MOSEK.

```
X.level();
model.primalObjValue();
```

The default setting in MOSEK to solve semidefinite programming is by using interior-point method. In order to get the clustering assignment for each data, the optimization result in X must be transformed into some matrix V where $X = V^\top V$. This can be done by using LU-decomposition either using native programming language or well known math library such as `Apache Commons Math API` for Java. Matrix U as the result of LU-decomposition of X is exactly V.

3.1.3 Implementation of FW

Local search Frank-Wolfe algorithm is an iterative algorithm that converges to a local optima for each random starting points. In contrast, LSFW can be implemented without third-party solver compared to QP and SDP implementation. This is since LSFW itself is just a simple iterative algorithm with greedy rule. The solution to the clustering algorithm can be improved by rerun the algorithm on different starting points, save the objective value and replace it if better objective value is found with different starting points.

As has been mentioned earlier on Chapter 2, that LSFW is actually a block coordinate iterative algorithm. at each step in the iteration, one item is taken randomly, which represents taking a random variable block. Then, the algorithm proceed to update the clustering for that item that yield the most improvement to the objective value. This procedure repeats until the objective value converges. Here, in this implementation, taking an item also can be done by shuffling the order of N items using random permutation and iterate over this shuffled order.

The implementation directly follows algorithm 2. First each decision vector is initialized into random cluster as shown by listing 3.11. Then, compute the objective obtained using this initial configuration to be used as the reference in the optimization process.

Listing 3.11: Initialize the decision vector into random cluster.

```
int[] cluster = new int[Constant.N];
Random random = new Random();

// start with random clustering
for(int i = 0; i < Constant.N; i++) {
    cluster[i] = random.nextInt(Constant.K);
}

// and calculate its cost
double currentObjective = 0.0;
for(int i = 0; i < Constant.N; i++) {
    for(int j = 0; j < i; j++) {
        if (cluster[i] == cluster[j])
            currentObjective = currentObjective + w[i][j];
    }
}
```

The next step is to optimize the clustering with FW until the objective converges. Listing 3.12 shows the procedure that keeps iterating until the gap between the objectives before and after the optimization is below the `EPSILON` which is around 2^{-53} In IEEE 754 arithmetic.

Listing 3.12: Procedure of iteration until the objective converged.

```
double prevObjective = currentObjective - 1;

while((currentObjective - prevObjective) > EPSILON) {
    prevObjective = currentObjective;

    // do maximization of objective until converge
}
```

Listing 3.13 below shows what is inside the `while` loop. On each iteration, one item or node is taken at a time and then the gradient for the variable block which represents the item is computed. Next, using the gradient, objective improvement is calculated for each cluster position where the item will be moved into.

Listing 3.13: Compute gradient of a variable block.

```
// compute the gradients
double[] tempObjs = new double[Constant.K];
for(int j = 0; j < Constant.N; j++) {
    if(j != i) {
        int k = cluster[j];
        tempObjs[k] = tempObjs[k] + w[i][j];
    }
}

int m = cluster[i];
double costOfM = tempObjs[m];

// compute objective changes when cluster for i changes
tempObjs[m] = currentObjective;
for(int k = 0; k < Constant.K; k++) {
    if(k != cluster[i]) {
        tempObjs[k] = currentObjective -
                        costOfM + tempObjs[k];
    }
}
```

Finally, on each iteration, the clustering to a data is updated. The greedy rule in LSFW is to move the data to the cluster that gives the best improvement to the objective. Listing 3.14 shows after the objective is computed for all possible clustering position, the data is moved to the cluster position with the highest objective and then the objective and clustering solution is updated.

Listing 3.14: LSFW chooses to move the item to the cluster with highest objective.

```
// find which cluster to put i that maximizes the objective
int maxIndex = 0;
double max = -Double.MAX_VALUE;
for (int k = 0; k < tempObjs.length; k++) {
    if (tempObjs[k] > max) {
        max = tempObjs[k];
        maxIndex = k;
    }
}

// update the cluster
cluster[i] = maxIndex;
currentObjective = max;
```

Alternatively, one can update the best objective value if better solution has been found as shown by listing 3.15. Here, t is the current restart position and one can try to repeat the entire algorithm on different starting configuration on different restarts with the hope to find better objective maximization.

Listing 3.15: Keep track on the best objective found by LSFW.

```
if (t == 0 || (currentObjective > bestObjective)) {
    bestClustering = cluster;
    bestObjective = currentObjective;
}
```

3.1.4 Gradient Descent Methods

3.1.4.1 Block Coordinate Gradient Descent (BCGD)

Following the formulation of gradient descent method 2.25 in chapter 2, we begin by defining the starting points. In the case for correlation clustering, we can initialize these starting points by randomly assigned each item into a cluster. This can be done by randomly initialize a matrix V with dimension $n \times k$ that consists of vector v_i on each of its row. v_i is initialized as binary unit vector, where e^{th} element is 1 and other elements is 0, denoting that i^{th} item was initially assigned to e^{th} cluster.

The next part is to construct the iterative procedure involving the gradient of the objective function. At each iteration randomly pick i^{th} item and as described by Frank-Wolfe algorithm section in Chapter 2. Then, the i^{th} gradient of the correlation clustering objective is:

$$\nabla_i f = \sum_j v_j.w_{ij} \tag{3.2}$$

By which we will use to update clustering assignment for a particular vector v_i as follows:

$$v_i^{t+1} \leftarrow v_i^t + \gamma \nabla_i f \tag{3.3}$$

Note that its positive $\gamma \nabla_i f$ since it's a maximization problem. This procedure iterates until convergence, where all the gradients are 0. However, notice that since the objective function for correlation clustering is non-convex, the update rule 3.3 will make entries in v_i grows to either $+\infty$ or $-\infty$. Furthermore, since we want our solution v_i to be presented as unit vector, such that we can decide to which cluster does i^{th} item is assigned to, we need to carry out normalization of v_i^t before updating to v_i^{t+1}. Considering the normalization, we can rewrite the update rule of 3.3 as follows:

$$v_i^t = v_i^t + \gamma \nabla_i f \tag{3.4}$$

$$v_{im}^t \begin{cases} 0, & \text{if } v_{im}^t < 0 \\ & \qquad\qquad m \in [0, k[\\ v_{im}^t, & \text{otherwise} \end{cases} \tag{3.5}$$

$$v_{im}^{t+1} \leftarrow v_{im}^t / (|v_i^t|_1), \quad m \in [0, k[\tag{3.6}$$

First, the gradient is add up to the vector $v_i^t = v_i^t + \gamma \nabla_i f$. Next, eliminate all negative elements in v_i^t by overriding each of them with 0 and finally, normalize v_i^t by dividing each element with L1-norm of vector v_i^t. In the case where the norm is 0, $v_i^{t+1} \leftarrow 1/k$, which gives equal probability for each cluster. This iterative procedure is a variant of gradient descent method known as block coordinate gradient descent (BCGD), since we are using the gradient $\nabla_i f$ from i^{th} block coordinate to update v_i at each iteration.

3.1.4.2 Stochastic Block Coordinate Gradient Descent (SBCGD)

Recall that in 3.2 to compute the gradient of i^{th} block coordinate, it is required to sum over all other vectors v_j where j is an item index other than i, or $\forall j,\ j \neq i$. Assume that a set S contains all these $n-1$ items whose vectors needed to compute i^{th} gradient $\nabla_i f$, stochastic block coordinate gradient descent aims to compute $\nabla_i f$ using vectors of some randomly chosen items in S. The formulation of the update rule for SBCGD is the following:

$$\nabla_i f = \sum_{j \in S}^{m} v_j . w_{ij} \tag{3.7}$$

Where m is an arbitrary size of random sample chosen from S. Once the gradient is obtained this way, the update rule is the same as 3.5 and 3.6.

3.1.4.3 Batch Gradient Descent (BGD)

Batch gradient descent in principle is similar to the block coordinate version above except its update rule. In block coordinate version, only gradient from i^{th} block coordinate is used to update clustering assignment vector v_i at each iteration. In batch gradient descent, a batch of gradients with fixed size m is used to update v_i instead of individual block coordinate. The batch size m does not necessarily to be all existing number of items n but can be arbitrarily defined, as long as $m \leq n$. Thus, m gradients are computed and used altogether to update v_i^{t+1} at each iteration. Following the update rule 3.3, batch update can be formulated as follows:

$$v_i^{t+1} \leftarrow v_i^t + \gamma \sum_i^m \nabla_i f \tag{3.8}$$

Surely the normalization procedure 3.5 and 3.6 also apply for BGD.

3.1.4.4 Stochastic Gradient Descent (SGD)

Stochasticity or randomness of a gradient descent method depends on random samples of vectors to be included in order to compute the gradient as in the case for SBCGD. Thus, stochastic gradient descent is a variant of gradient descent method that uses both random samples of j^{th} vector for computing gradient, and batch update rule as in BGD.

LSFW is an iterative method that uses the information of gradients and step length to optimize correlation clustering. It is interesting to see its performance and correctness compared to other well known iterative method such as gradient descent. More importantly, the similarities and differences of behaviors of both algorithms that could potentially provides the answer of what makes one of them better than the other.

4
Results

4.1 Performance evaluation

The optimization methods that has been mentioned earlier in Chapter 2 along with their implementations in Chapter 3 are measured. The methods are tested using synthetic generated data and measured in terms of their runtime and correctness of finding the solution for correlation clustering problem. Synthetic data generator generates positive-negative weight matrix that will be used as the input to the correlation clustering problem. The data generator uses the parameters regarding the number of nodes n, number of clusters k and noise p to generate the synthetic data. Each weight value in the matrix is any random number with normal distribution on which with probability p the sign of the weight is flipped. The system specification that is used for conducting the experiments consists of Intel core i5-6200U processor with 4 cores @2.3GHz and a 4GB RAM. The operating system is Windows 10 64-bit.

4.1.1 Runtime evaluation of QP, SDP and LSFW

In order to compare the runtime among optimization methods, synthetic data are generated each with fixed number of clusters $k = 4$, noise $p = 0.3$, and the increasing data size n. to compare the performance between QP and SDP, the data size parameter n used are 10, 12, 16, 18 and 20, while the number of clusters and noise parameter are fixed $k = 4$ and $p = 0.3$ respectively. The runtime measurement between QP and SDP are shown by table 4.1 and figure 4.1, where the runtime for QP grows exponentially compared to the runtime of SDP that is hardly seems to make any changes as data size getting larger. Since CPLEX is set to find global optima for QP, it is evidently NP-hard as has been elaborated in Chapter 2. Meanwhile, interior-point is used to solve SDP problem and SDP relaxation is a type of approximation algorithm that solves the correlation clustering problem in polynomial time with approximation ratio 0.75 of the optimal solution [6].

Table 4.1: Runtime comparison between QP and SDP.

RUNTIME. k = 4, p = 0.3						
n	8	10	12	16	18	20
QP (s)	0.06	0.17	0.48	3.54	51.43	17204
SDP (s)	0.01	0.01	0.02	0.04	0.05	0.06

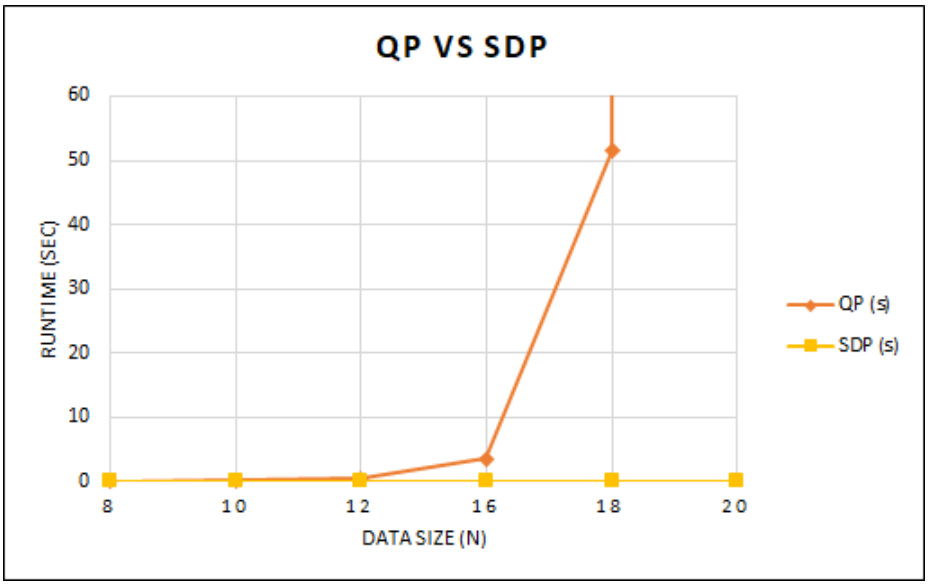

Figure 4.1: Runtime comparison between QP and SDP.

The runtime evaluation proceed to observe on the performance comparison between SDP and LSFW. The runtime comparison between SDP and LSFW is shown by table 4.2 and figure 4.2. This result can be expected as in the case for SDP, the number of constraints and the size of SDP matrix grows quadratically. Moreover, interior point method as the efficient method to solve SDP required to store and factorize a large(and often dense) matrix for its computation that may caused bottleneck in the memory while optimizing SDP. MOSEK returns "error insufficient space" when SDP is ran with $n = 300$. Karmarkar's algorithm which is the most efficient algorithm in the class of interior point methods requires $O(n^{3.5}L)$ operations on $O(L)$ digit numbers.

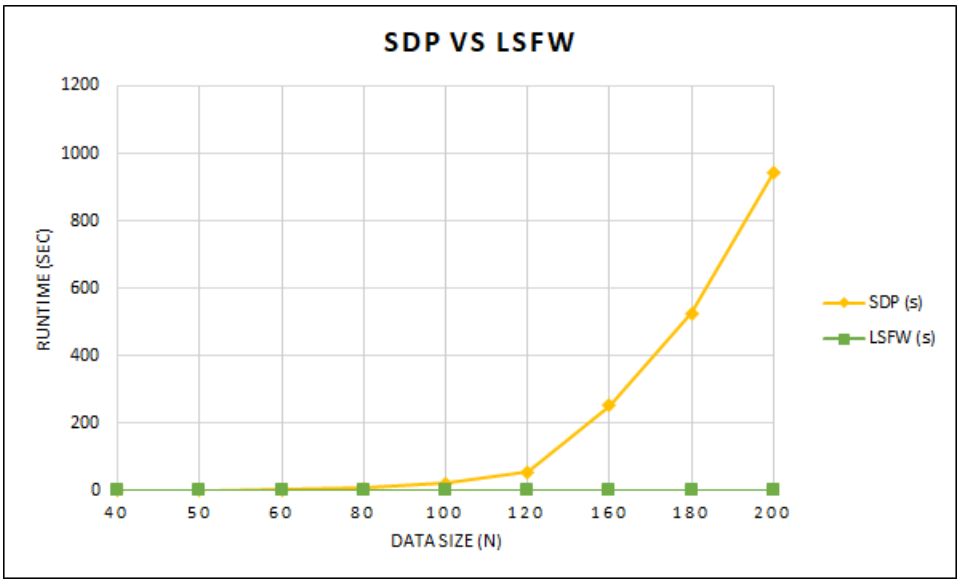

Figure 4.2: Runtime comparison between SDP and LSFW.

Meanwhile, LSFW is operating on the concept of local search where initially random starting point is given and the algorithm will iterate until converge to a local optima. In order to improve the solution, it is simply needed to rerun the algorithm with different starting points which may lead to better local optima. Furthermore, 10 tests are carried out for each optimization methods and the average time spent from these tests is considered into the observation.

Table 4.2: Runtime comparison between SDP and LSFW.

RUNTIME. k = 4, p = 0.3									
n	40	50	60	80	100	120	160	180	200
SDP (s)	0.37	0.97	1.97	8.39	22.81	55.22	252.71	526.02	941.83
FW (s)	0.006	0.012	0.011	0.024	0.039	0.042	0.082	0.086	0.113

All the observation results provided in the graphs are represented as the growth of runtime in seconds against the number of items to be partitioned n. Figure 4.3 shows the runtime observation for all optimization methods QP, SDP and LSFW. The growth of LSFW is much slower compared to well known optimization methods such as QP and SDP. Thus, Figure 4.4 shows how much LSFW grows in terms of runtime up to $N = 1000$.

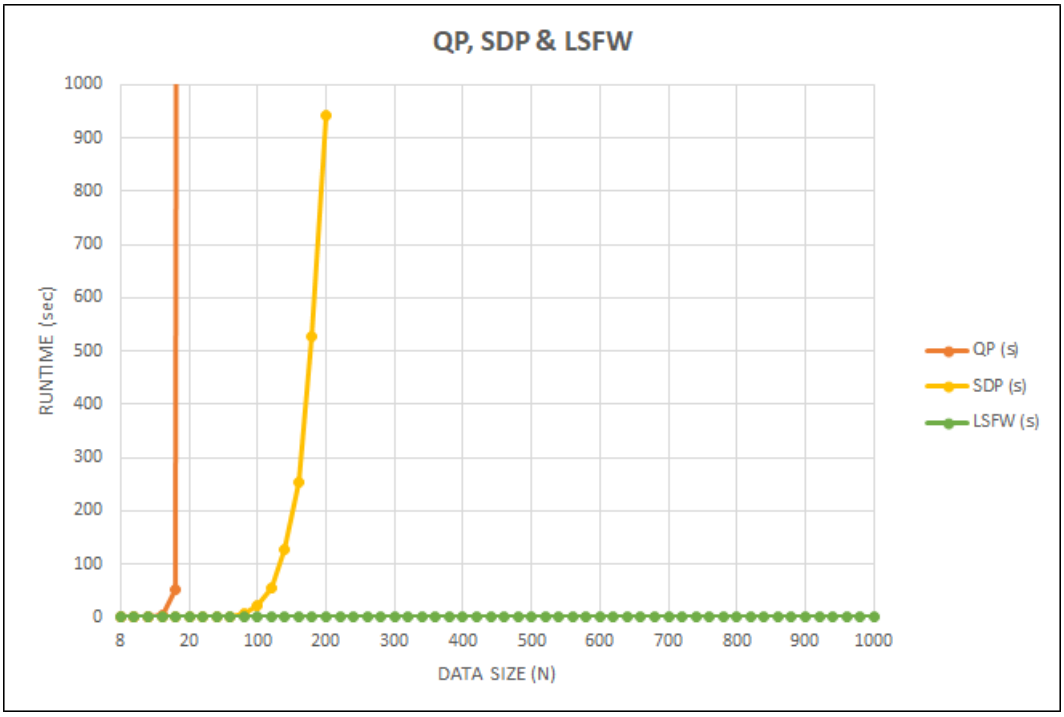

Figure 4.3: Runtime comparison between QP, SDP and LSFW.

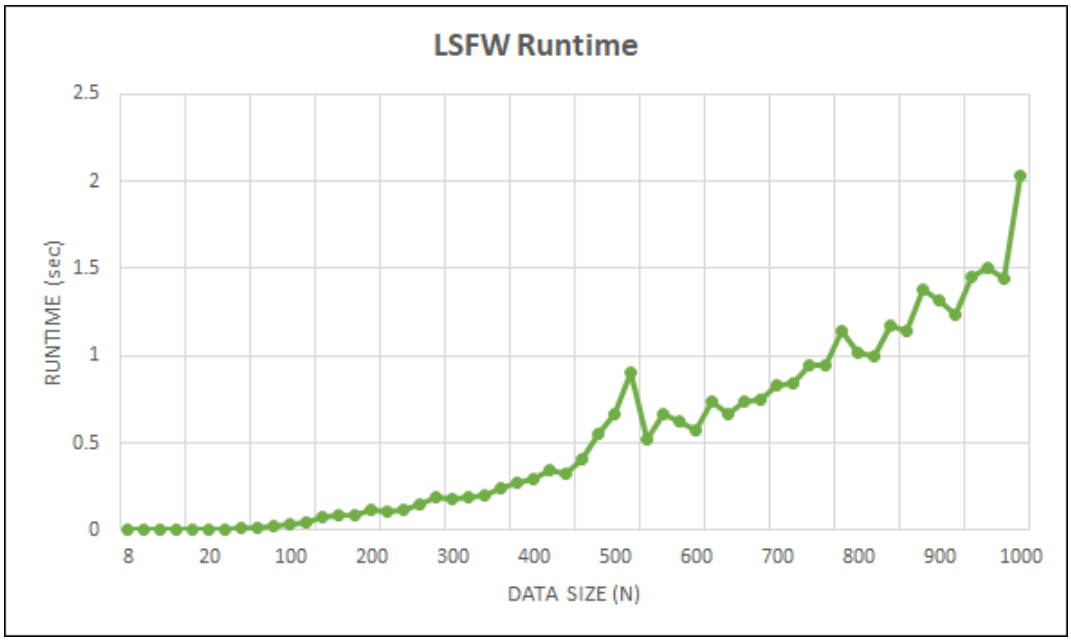

Figure 4.4: Close up LSFW runtime growth shown in figure 4.3. Note that for comparison with QP and SDP, LSFW ran with 10 restarts for each test data.

4.1.2 Runtime evaluation of gradient descent and LSFW

Several types of gradient descent methods are tested, which have been mentioned in Chapter 3. The result shown in figure 4.5, where BCGD and LSFW has similar runtime behavior compared to other gradient descent variant. SBCGD is constant time slower compared to BCGD since it takes constant time to check its convergence due to more oscillating objective when converged. BCGD, SBCGD and LSFW have linear runtime growth $O(tn)$ where t is the number of iterations needed to reach convergence.

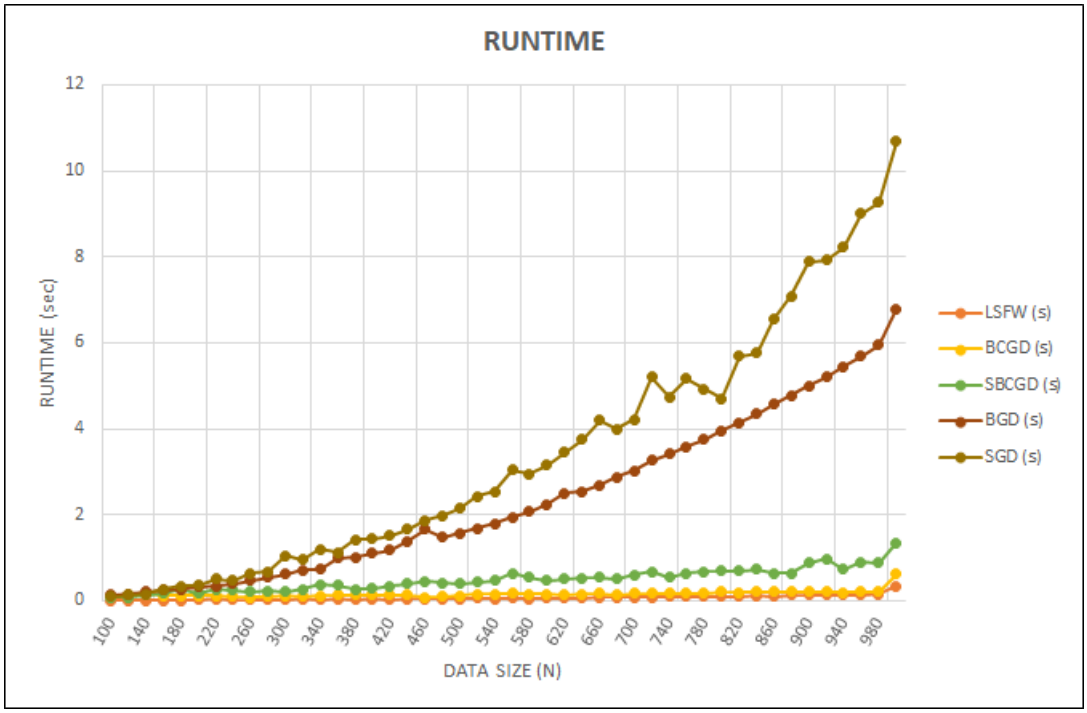

Figure 4.5: Runtime comparison between gradient descent methods and LSFW.

BGD takes slower time compared to BCGD and SBCGD since it computes all the gradients on each iteration, and it needs to iterate over all items every time it computes each gradient. Likewise, SGD also need to compute all gradients and takes constant time more for checking convergence. BGD and SGD both have quadratic runtime growth $O(tn^2)$. For the evaluation between gradient descent methods and LSFW, LSFW was executed once for each test data without using constant number of restarts.

4.1.3 Correctness of QP, SDP and LSFW

In order to compare the correctness among optimization methods, synthetic data are generated with fixed number of nodes n, clusters $k = 4$ and the increasing noise parameter p. The performance of each optimization methods is measured against the increasing noise in the weight matrix data from $p = 0.0$ until $p = 0.9$. Figure 4.6 below shows the objective values obtained from each optimization methods over the noisy data. For this observation, LSFW is allowed to restart on 10 different starting points because its runtime is much faster that QP or SDP.

The average objective values obtained from 10 different observations for each noise parameter p are considered. Table below shows that QP and LSFW relatively come up with similar objective value for each noise parameter, and are much better compared to objective values obtained by SDP. The details of objectives obtained by QP, SDP and LSFW over increasing noise in the data can be seen on table 4.3 and 4.4. The data size tested is small $n = 16$ so that the size is still feasible for QP.

Table 4.3: Correctness of QP, SDP and LSFW. $p = 0.0 \sim 0.4$.

CORRECTNESS. n = 16, k = 4					
p	0	0.1	0.2	0.3	0.4
QP	16.848	14.622	15.579	14.789	17.558
SDP	16.848	12.243	11.66	8.635	12.949
FW	16.848	14.622	15.579	14.789	17.558

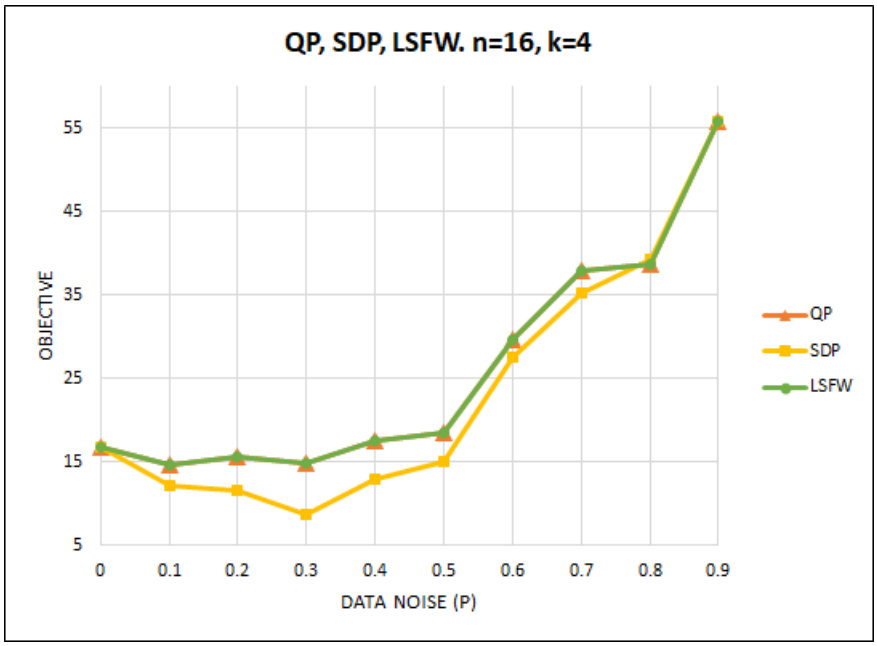

Figure 4.6: Objective values obtained using QP, SDP and LSFW. LSFW and QP obtained the exact same objective most of the time.

Table 4.4: Correctness of QP, SDP and LSFW. $p = 0.5 \sim 0.9$.

CORRECTNESS. n = 16, k = 4					
p	0.5	0.6	0.7	0.8	0.9
QP	18.425	29.671	37.98	38.654	55.672
SDP	15.076	27.467	35.171	39.217	55.672
FW	18.425	29.671	37.98	38.654	55.672

Figure 4.7 shows the objectives obtained between SDP and LSFW. This time with larger data size $n = 100$ that is feasible for SDP. As can be seen, LSFW can perform better maximization over noisy data. The objective decreased as the noise in the data increased until reached the lowest objective obtained when the noise is $p = 0.4$ and then the objective raise again when $p > 0.4$.

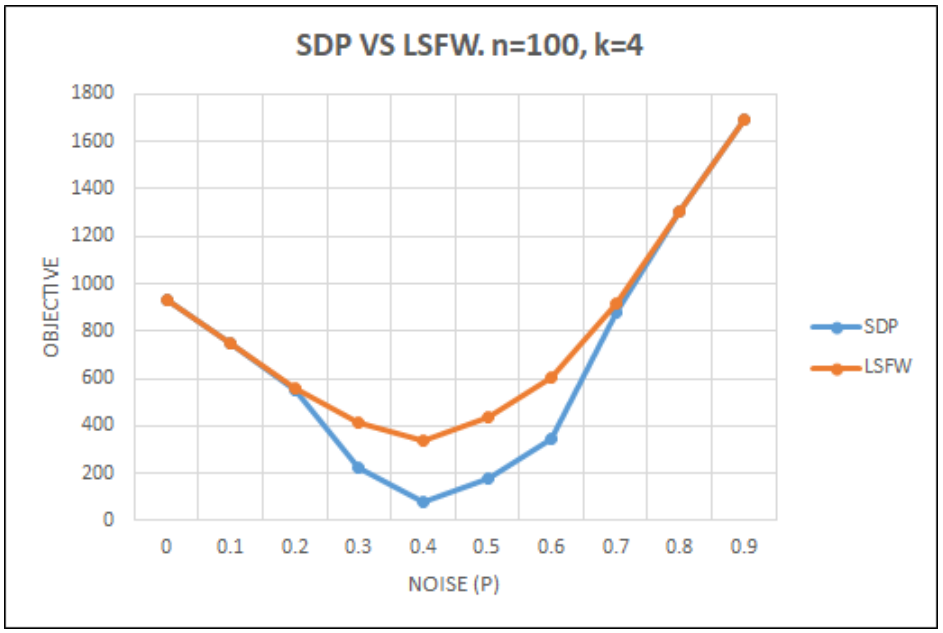

Figure 4.7: Objective values for larger data size $n = 100$ using SDP and LSFW.

4.1.4 Correctness of Gradient descent and LSFW

The objective maximization obtained by gradient descent and LSFW is measured using fixed number of nodes, but with varying noise level. Figure 4.8 and 4.9 show the result of objectives obtained using gradient descent and LSFW. The results show that the objective maximization obtained by LSFW is higher compared to gradient descent variants. The results also show that most variants of gradient descent namely BCGD and BGD converge towards negative objective when the noise in the data is low for current number of nodes. When comparing with gradient descent variants, LSFW only ran once without restarts. The sample size for SBCGD is 0.4.

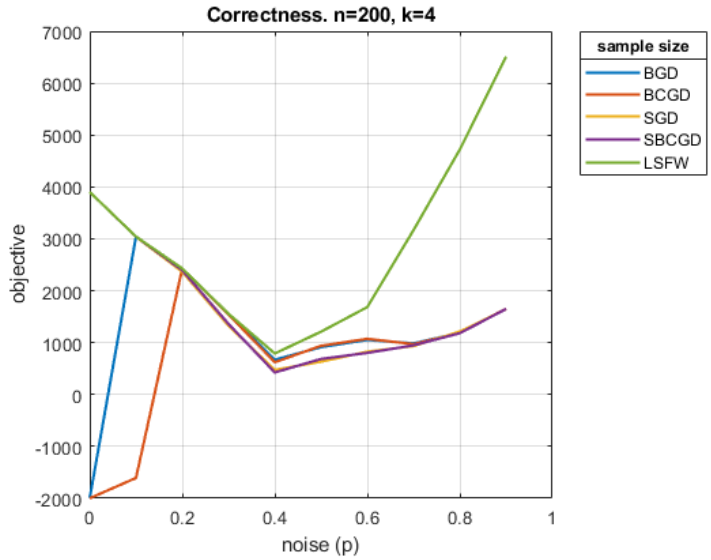

Figure 4.8: Objectives obtained for data size $n = 200$ and noise $p = 0 \sim 0.9$.

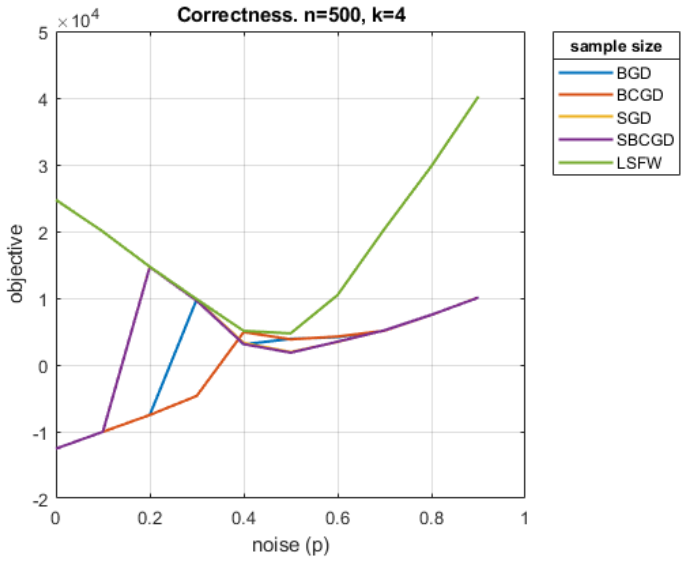

Figure 4.9: Objectives obtained for data size $n = 500$ and noise $p = 0 \sim 0.9$.

4.2 Similarity of iterative methods

One practical difference between LSFW and other well known iterative method such as gradient descent is that while the gradient descent methods use both gradient and step length (γ), LSFW simplifies the optimization of step length and instead used greedy rule to maximize the correlation clustering objective. LSFW algorithm uses the gradient as the reference and chooses which cluster to put the item that yields the maximum improvement for the correlation clustering objective.

Let us turn the attention to the update rule used by gradient descent methods 3.3. It is intuitively easy to suggest that gradient plays the important role in determining the clustering assignment that maximizes the objective because it gives the information of clustering agreement on all clusters. We can also see that from 3.2 at each iteration, the gradient itself is influenced by the current cluster assignment of each item and their relations to each other represented by weights w_{ij}. To be more simple, the gradient $\nabla_i f$ will have larger value on its m^{th} element, if the there are more items $j \neq i$ in m^{th} cluster that have positive weights w_{ij} than any other cluster. Likewise, the gradient $\nabla_i f$ will have smaller value on its m^{th} element, if the there are less items $j \neq i$ in m^{th} cluster that have positive weights w_{ij}.

In this case, we can conclude that the gradient will give the information on how the items should be re-arranged for the next iteration based on the current clustering situation in order to improve the objective. The information could be to move the items to another cluster, to make the items stay in their current cluster or inconclusive which means that all elements have $1/k$ probability.

Given the role of gradient and the update rule 3.3, we can see that the step length (γ) is just amplifying the information already contained in the gradient $\nabla_i f$ before being used to update v_i^t. Several experiments has been carried out to see how various step lengths affect the improvement of objective. Figure 4.10 and 4.11, shows the step length chosen by BCGD at each iteration for a particular data. The data tested have $n = 100$ and $k = 4$, each with increasing noise parameter $p = 0 \sim 0.9$.

Figure 4.10 and 4.11 show that only 2 step lengths $\gamma = 0.1$ and $\gamma = 10$ that are chosen along the entire iterations. It is not the case that for low noise data, gradient descent prefers smaller step lengths more than larger step lengths, since there are ties of objectives obtained by larger step lengths too, i.e. when $\gamma = 10$ is chosen, $\gamma = 100$, $\gamma = 1e3$ and $\gamma = 1e4$ all yield the same objective.

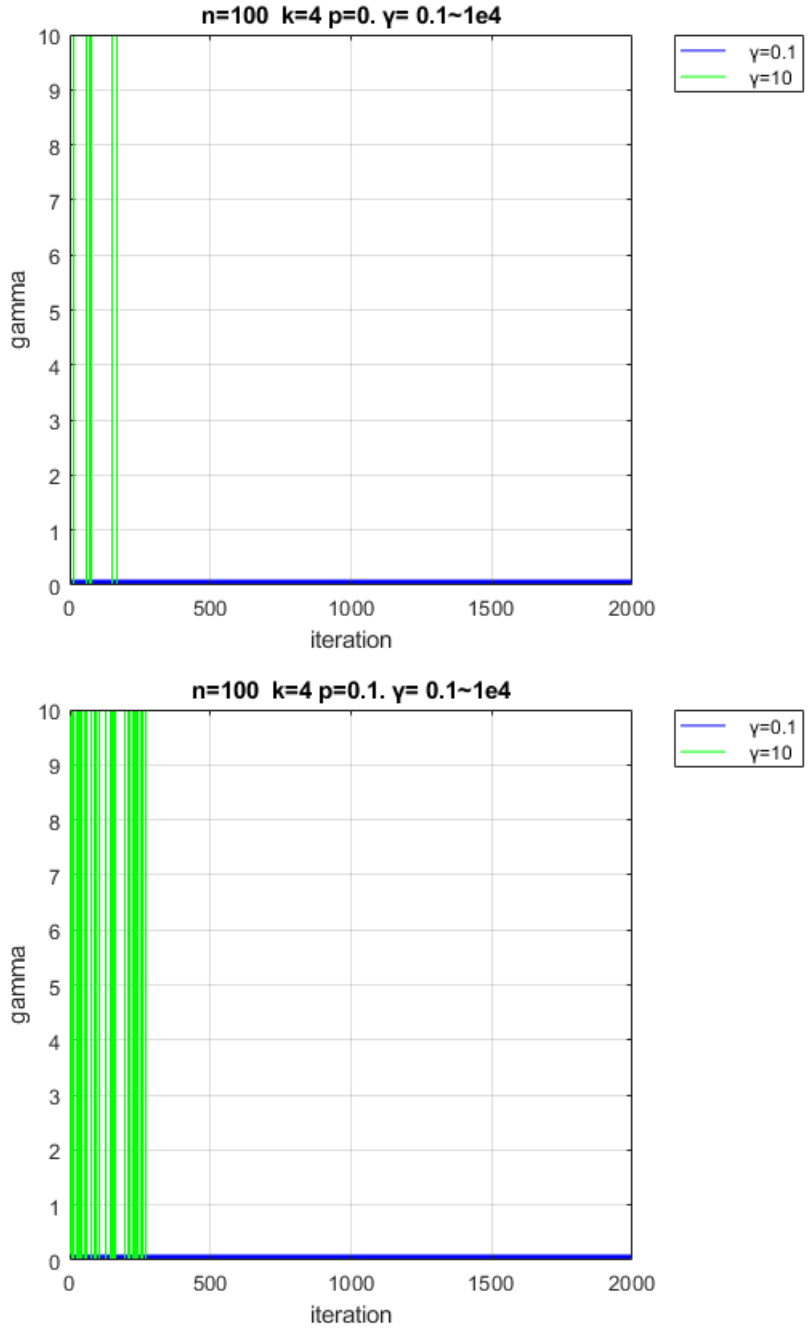

Figure 4.10: The step length chosen at each iteration for data with noise $p = 0$ and $p = 0.1$. Only $\gamma = 0.1$ and $\gamma = 10$ that are chosen along the entire iterations.

The reason behind this, is because starting from $\gamma = 10$, the step lengths gave the same update to the decision vector thus larger step length yield equal improvement for the objective as shown by table 4.6. Moreover, a step length is chosen because it is able to give the highest probability to the decision in v_i to move the item to the cluster with the largest agreement. Table 4.5 and 4.6 show the example where $\gamma = 0.1$ and $\gamma = 10$ are chosen. The tables show that gradient descent always prefers to move the item to the cluster with the largest agreement.

Table 4.5: The objectives obtained by different step lengths. Here, the smallest step length ($\gamma = 0.1$) yields the best improvement.

$n = 100,\ k = 4,\ p = 0$			step length				
	start	gradient	0.1	10	1.0e+02	1.0e+03	1.0e+04
v_i	1	-4.2615	1	0.25	0.25	0.25	0.25
	0	-12.8145	0	0.25	0.25	0.25	0.25
	0	-9.3145	0	0.25	0.25	0.25	0.25
	0	-19.2915	0	0.25	0.25	0.25	0.25
objective			-577.11875	-584.27775	-584.27775	-584.27775	-584.27775

Table 4.6: The objectives obtained by different step lengths. Here, the step length ($\gamma = 10$) is chosen.

$n = 100,\ k = 4,\ p = 0.1$			step length				
	start	gradient	0.1	10	1.0e+02	1.0e+03	1.0e+04
v_i	0	-16.651	0	0	0	0	0
	0	6.842	0.6222	1	1	1	1
	0	-16.093	0	0	0	0	0
	1	-5.845	0.3778	0	0	0	0
objective			250.032	254.8255	254.8255	254.8255	254.8255

Table 4.7 shows that starting from ($\gamma = 100$), the improvement to the objective is equal because all the step lengths give the same update to v_i. Table 4.8 shows that the largest step length is chosen because it is able to give the highest probability to the decision to move the item to the cluster with the largest agreement.

Table 4.7: The objectives obtained by different step lengths. Here, the step length ($\gamma = 100$) is chosen.

$n = 100,\ k = 4,\ p = 0.3$			step length				
	start	gradient	0.1	10	1.0e+02	1.0e+03	1.0e+04
v_i	0	0.0831	0.0098	0.8772	1	1	1
	0.4892	-1.5206	0.3956	0	0	0	0
	0	-15.5701	0	0	0	0	0
	0.5108	-0.0394	0.5947	0.1228	0	0	0
objective			98.2086	98.9008	98.9159	98.9159	98.9159

Table 4.8: The objectives obtained by different step lengths. Here, the largest step length ($\gamma = 1e4$) is chosen.

$n = 100,\ k = 4,\ p = 0.2$			step length				
	start	gradient	0.1	10	1.0e+02	1.0e+03	1.0e+04
v_i	0	-21.112	0	0	0	0	0
	0	-7.1147	0	0	0	0	0
	1	0.4249	0.7366	0.1234	0.1045	0.1025	0.1023
	0	3.7288	0.2634	0.8766	0.8955	0.8975	0.8977
objective			-169.82436	-167.79864	-167.73604	-167.72963	-167.72898

Figure 4.11 shows that as there are more noise in the data, larger step lengths started to appear more along the iteration. This can also be seen in figure 4.12 which shows that the percentage of choice for the largest step length is increased as there are more noise in the data.

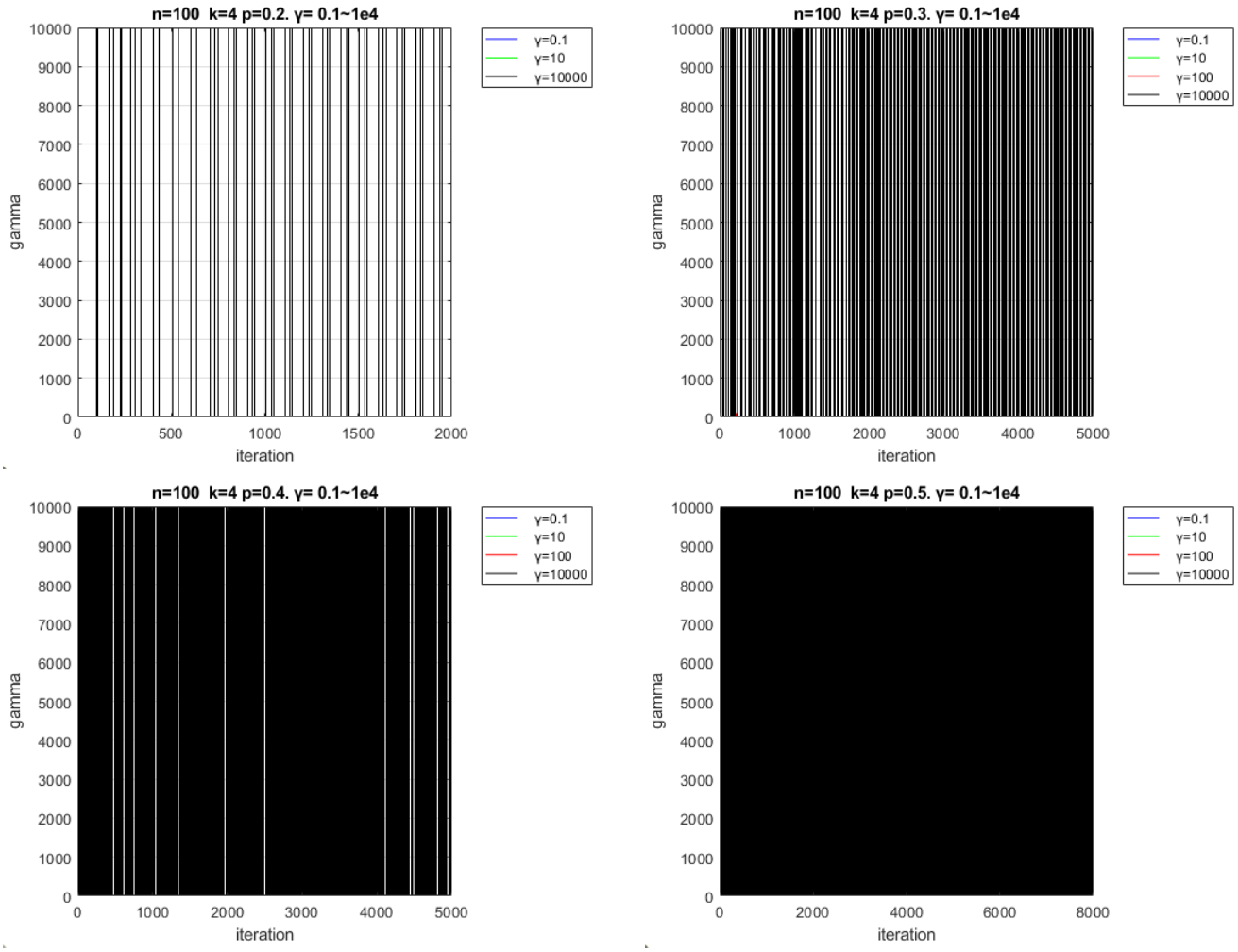

Figure 4.11: The largest step length $\gamma = 1e4$ is chosen more as the noise in the data increased.

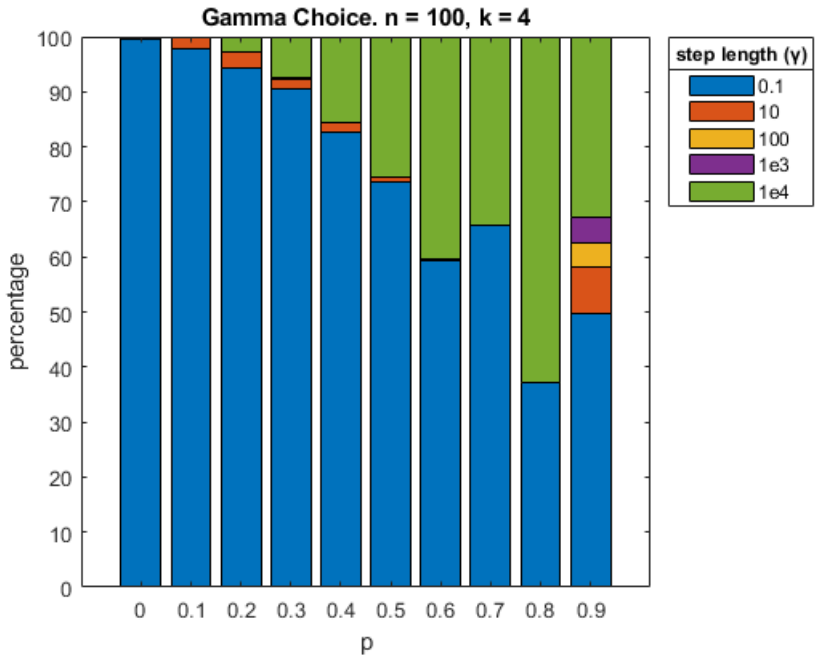

Figure 4.12: Percentage of each step length chosen compared to noise in the data.

This is because as noise is increased in the data, more clustering assignments will become probabilistic which also means that more decision vectors will become non-binary. Since it is known that higher probability in the decision vector gives better improvement for the objective, and larger step lengths give higher probability for the decision in the decision vector, thus more and more larger step lengths are chosen. Notice that the percentage is increasing progressively as the noise is increased from 0 to 0.5. When the noise is $p > 0.5$, the percentage goes up and down because as noise approach 1, the data started to become the inverse of itself meaning that all the positive weights become negatives and all the negatives become positives.

Finally, the results suggest that on each iteration, gradient descent will always try to move the item to the cluster with the largest agreement and this is similar to the greedy rule found in LSFW. Moreover, gradient descent prefers larger step lengths to move the item to other cluster because they give higher probability to the decision in v_i to move the item to the cluster with the largest agreement, resulting in higher improvement for the objective.

4.3 Suboptimality of gradient descent

Gradient descent clearly has similarity with LSFW on how the items are arranged on each iteration that gives most improvement to the objective. However, using the normalization scheme elaborated in Chapter 3 will make gradient descent at some point converged to a suboptimal solution such that all or most of the elements in matrix V have the same value $1/k$ when converged.

This phenomenon is shown at figure 4.13, where increasing data size with noise parameter $p = 0$ is tested.

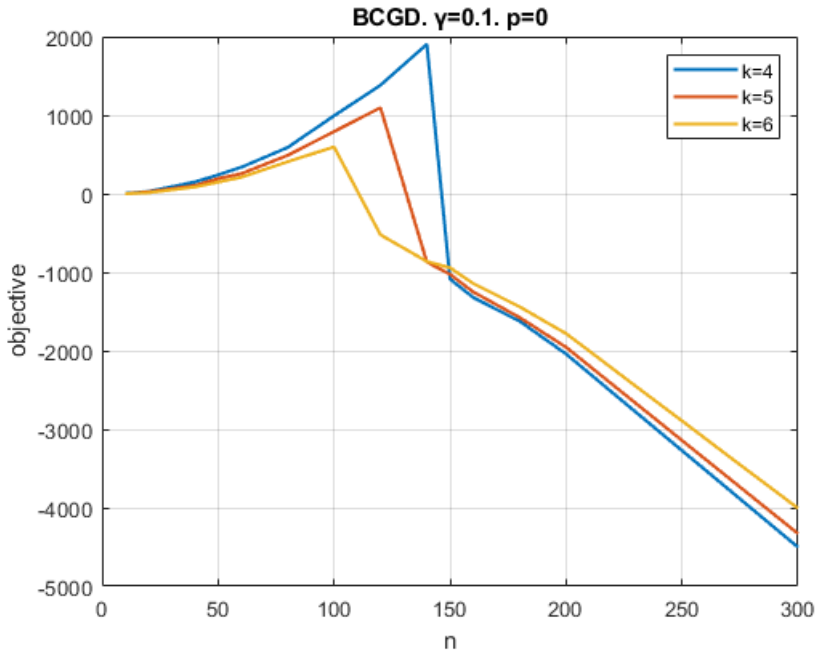

Figure 4.13: The sudden collapse of objective values obtained by gradient descent.

Upon reaching certain data size, the objective suddenly collapsed into negative and it continues to degrade as the data size increased. This is the result when the data is large enough that makes the gradient more likely to be negatives because there are more disagreements or negative weights w_{ij} on every cluster than the positive ones, and as elaborated in Chapter 3, negative gradient is penalized to 0 and thus all the elements in v_i will be updated with equal probability $1/k$.

In order to tackle this problem, one well known method is to add stochasticity to prevent gradient descent converging into suboptimal solution. This can be done with several implementations such as, adding noise to the data, random sampling for computing gradient and adding perturbation to the data.

4.3.1 Increase noise in the data

The first plot in figure 4.14 shows the objectives obtained by gradient descent over data with different noise level from $p = 0$ to $p = 0.2$. It shows small noise in the data may prevent convergence to saddle points but still when certain data size reached, saddle points may occur again. When the noise level is increased $p = 0.3 \sim 0.5$ significant changes can be seen that gradient descent is not converged to saddle points anymore at least until $n = 300$. The quality of the objectives however are degrading as shown by the gaps among objective plots.

Another experiment is also support these findings. As can be seen in figure 4.15, where the result of clustering assignment using gradient descent is compared to the ground truth of the data using rand index (RI) and normalized mutual information (NMI). The data tested is generated using the following parameters $n = 100$, $k = 4$ and $p = 0$; the same data is used and different noise level is added to generate new data. A ground truth is the clustering assignment used when generating the data with no noise $p = 0$.

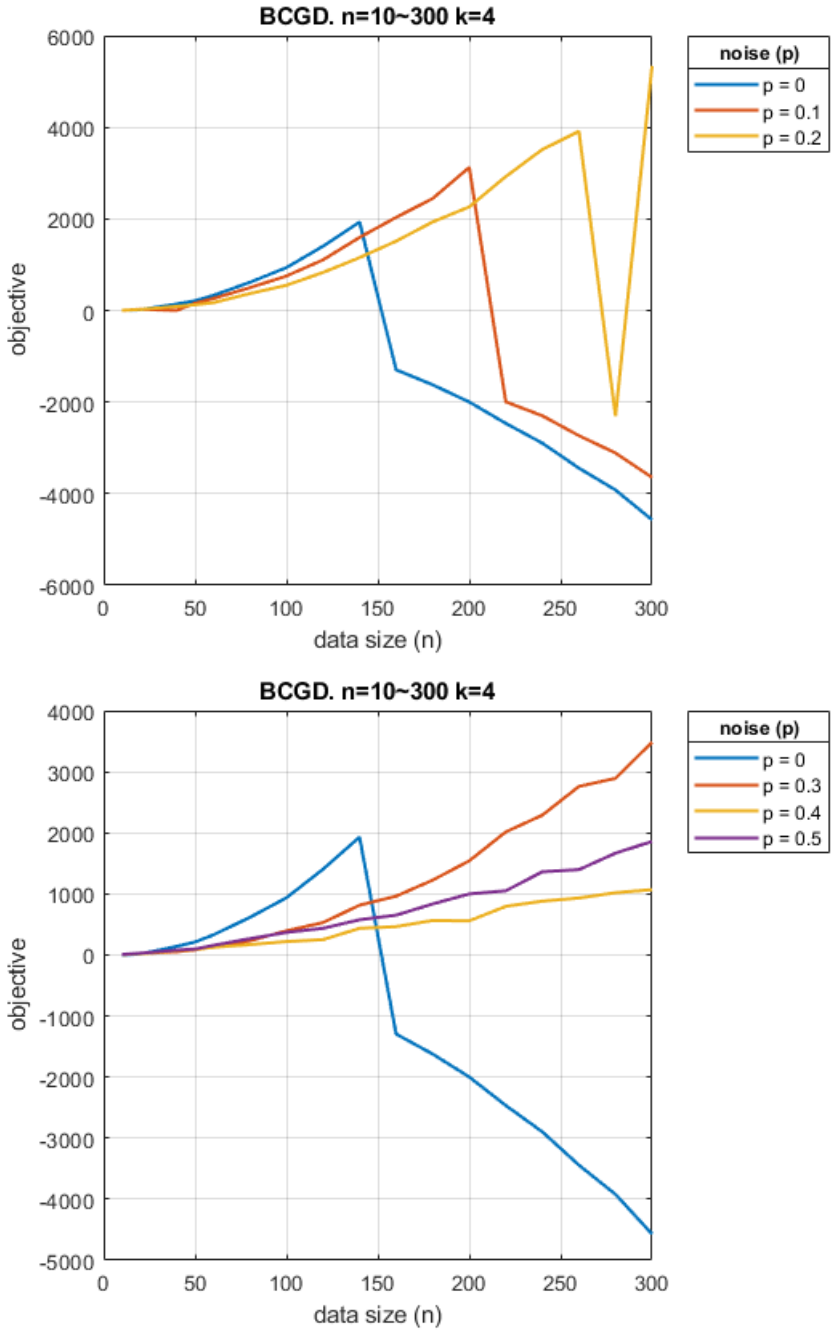

Figure 4.14: More noise level may prevents gradient descent from converging into saddle points but at the same time it also degrades solution quality.

The result of RI and NMI over different noise levels shows that as the noise in the data increased, the clustering solution is getting more random compared to the ground truth. This is clearly shown when NMI is approaching 0 as the noise is $p = 0.5$, meaning the clustering solution of data with high noise shared no mutual information with the ground truth. Likewise, the value of RI also decreased when noise level in the data increased, which means that the frequency of occurrence of agreements between ground truth and clustering solution is getting down.

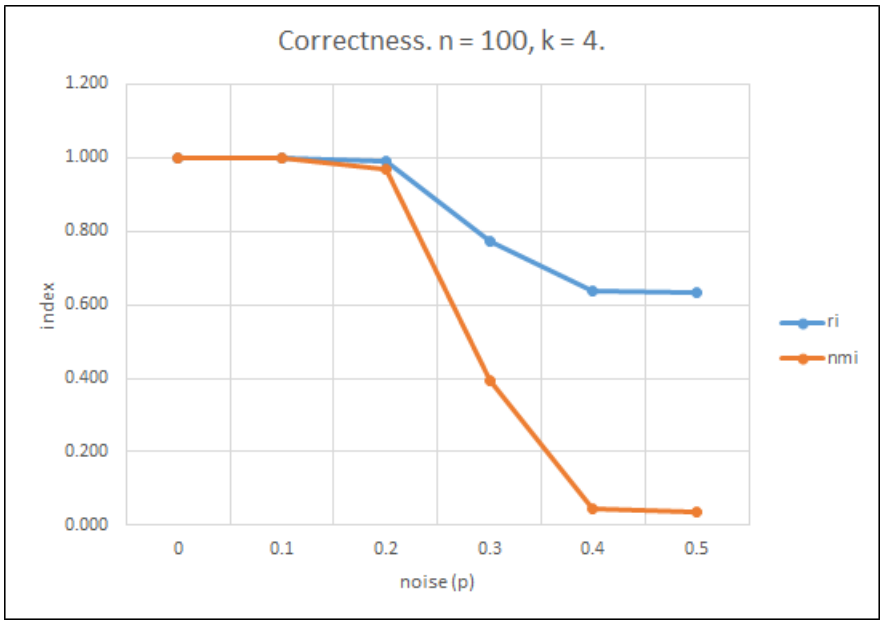

Figure 4.15: Increasing the noise takes the solution away from the ground truth of the data.

4.3.2 Compute gradient with random sampling

Adding stochasticity to avoid suboptimal solution can also be done by random sampling of data while computing the gradient, which is exactly what SBCGD is. Chapter 3 has elaborated in detail that with SBCGD, not all data is involved in order to compute the gradient $\nabla_i f$. SBCGD can avoid saddle points better than by adding noise to the data since there's no modification whatsoever to the data resulting in better objective and closer clustering solution to ground truth.

Figure 4.16 shows the result of SBCGD using different sample size. The experiment results show that using larger sample size $0.5 \sim 0.9$ gradient descent still end up in saddle points, where the largest sample size 0.9 starts end up in saddle points when data size is $n \geq 500$. Meanwhile, with small sample size $0.1 \sim 0.4$ shown in 4.17, none of the objectives obtained were found converged to saddle points at least until the size of data is $n = 1000$. This happens because stochasticity is increased as sample size getting smaller and also with smaller sample size, smaller number of items involved in computing the gradient which may prevent the elements in the gradient to become dominated by negatives.

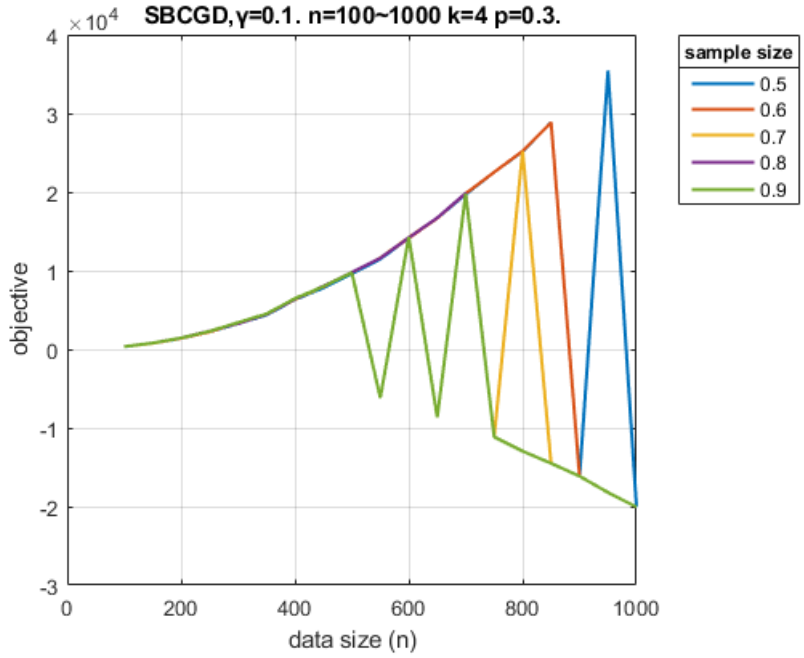

Figure 4.16: Big sample size $0.5 \sim 0.9$ still converge to saddle points.

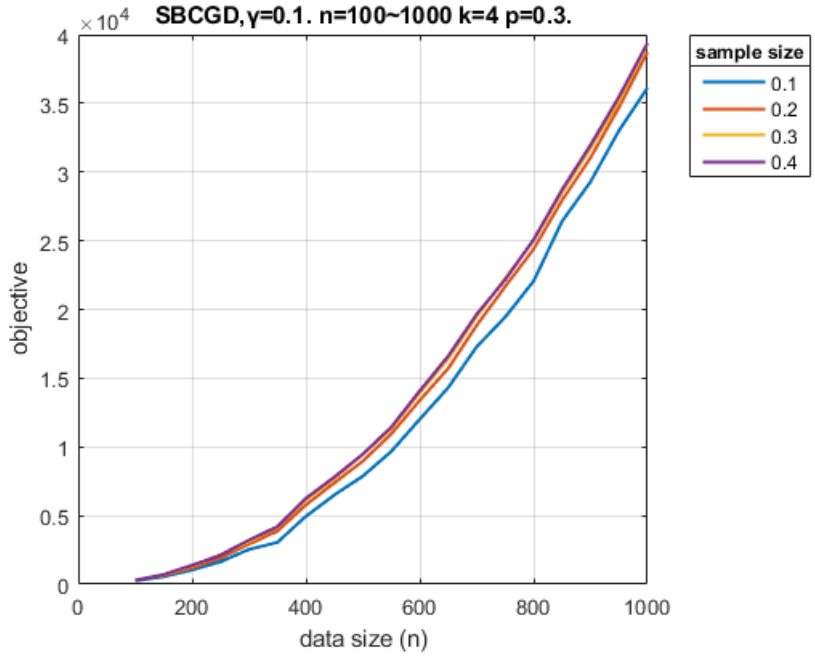

Figure 4.17: Smaller sample size $0.1 \sim 0.4$ means more stochasticity and as the result prevents saddle points at least until $n = 1000$.

Moreover, it is shown by figure 4.18 that when compared to objectives obtained by BCGD represented as blue plot in the graph, SBCGD with smaller sample sizes have lower objectives. However, sample sizes 0.3 and 0.4 are shown to have the smallest gap with objectives obtained by BCGD.

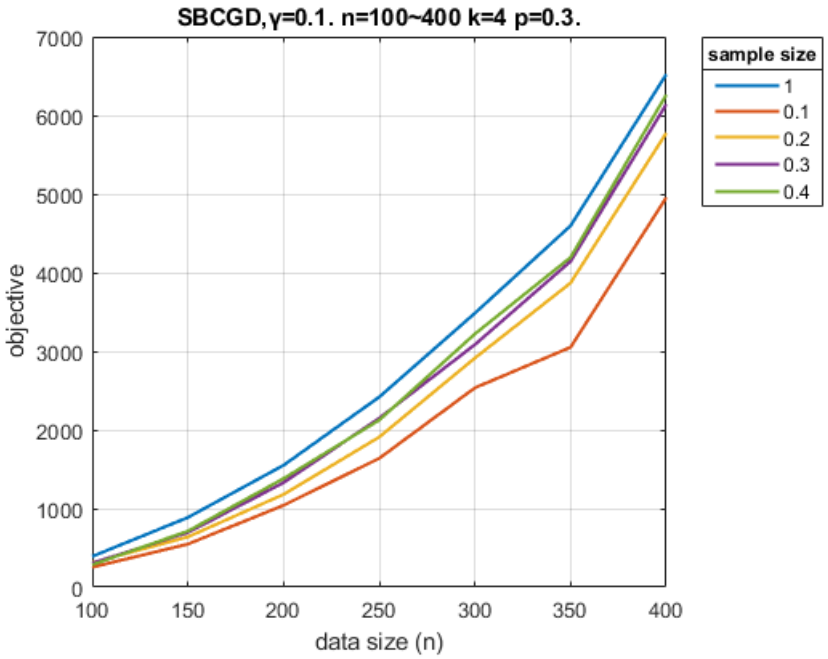

Figure 4.18: The difference of objectives obtained using smaller sample size.

4.3.3 Adding perturbation to saddle points

Adding perturbation in the data is another way to add stochasticity during optimization process. This can be done by adding each element in vector v_i with an independent random variable e.g. α, β, γ etc. Recall the normalization procedure in 2.23 and 2.24, in the case where L1-norm of v_i is 0 then each element is replaced with $\frac{1}{k} + \alpha$ instead of overriding all the elements in v_i with only $1/k$. Each random variable has normal distribution with fixed mean set as $\mu = 0$ and custom value for standard deviation (σ). Figure 4.19 shows the result of adding different levels of perturbation to the data with standard deviation $\sigma = 0, 1, 2$.

Figure 4.20 shows the result with even larger data range $n = 100 \sim 1000$. This result shows that bigger perturbation may provide better performance for gradient descent to avoid saddle points. For example, when $\sigma = 1$ for data size $n \leq 300$ gradient descent is able to obtained satisfying objective, but starts experiencing saddle points again once the data size is approaching $n = 1000$.

Despite advantages achieved by adding perturbation to data, it also suffers another side effect. Figure 4.21 shows that larger perturbation may lead to slower convergence and difficulties to decide convergence state during optimization process. Perturbation may also sensitive with the noise level in the data such data the more noise in the data.

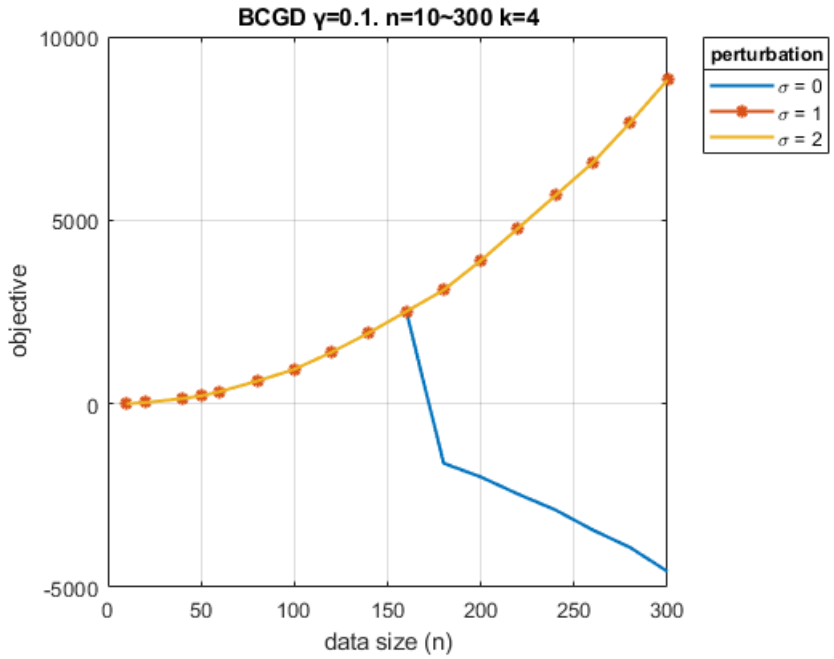

Figure 4.19: Using the same range of data size as in figure 4.13, this shows that by adding perturbation with certain magnitude of σ could prevent gradient descent from converging into saddle points.

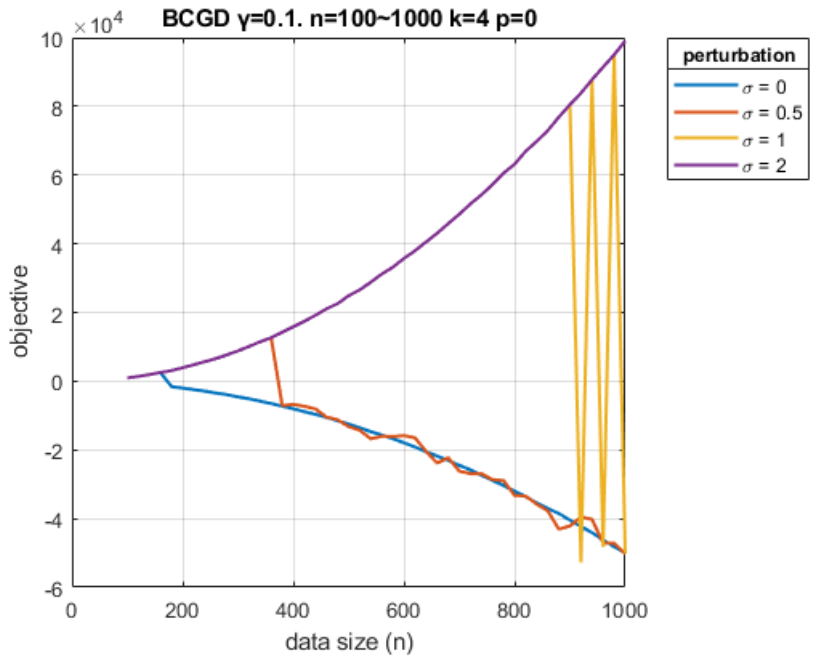

Figure 4.20: Correlation clustering objective obtained using different perturbation level. This suggests that larger perturbation may yields better objectives.

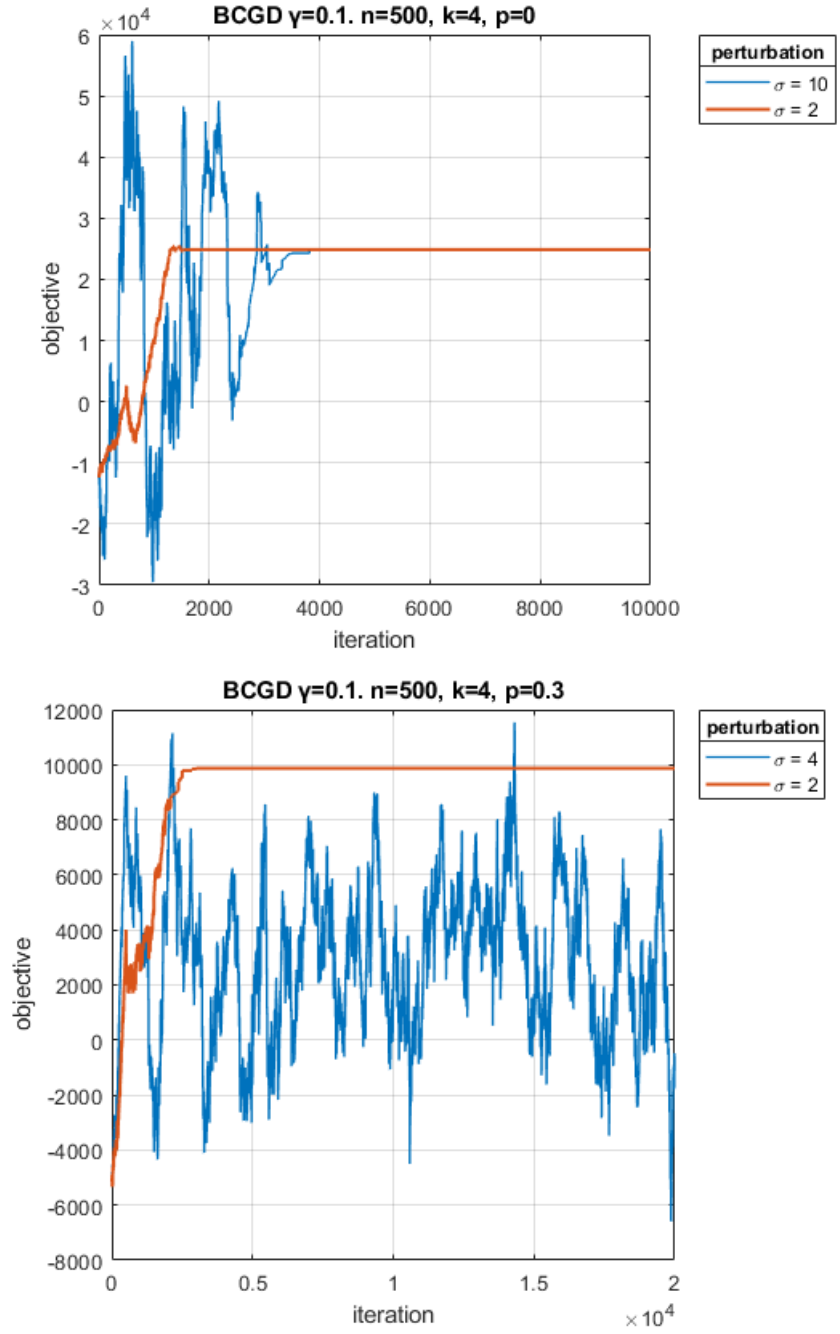

Figure 4.21: Increased perturbation may caused gradient descent to take longer time to converge and also more difficult to check convergence.

5
Conclusion

5.1 Performance evaluation

Since QP is NP-hard thus it cannot be solved in polynomial time. SDP can solve correlation clustering in polynomial time but with approximation ratio 0.75 from optimal solution and also suffers bottleneck in terms of memory capacity because it requires large and often dense matrix for computation. LSFW has the best runtime complexity and the best objective maximization compared to QP, SDP and all gradient descent methods. BCGD has similar runtime complexity with LSFW and the fastest among other gradient descent methods. The gradient descent variants: BCGD, SBCGD, BGD and SGD all suffers from saddle points $1/k$ when the data size is $n \geq 150$.

5.2 Algorithmic similarity

BCGD is a variant of gradient descent that is most similar to LSFW. LSFW simplifies optimization of step length and uses greedy rule to put the item in the cluster that gives the best improvement to the objective. Whereas BCGD moves the item to the cluster which has the highest agreement resulting in the best improvement for the objective.

5.3 Stochasticity in gradient descent

Gradient descent has drawback in which it is likely end up converge in saddle point if the data is large enough. Adding stochasticity can avoid this convergence into suboptimal solution and this can be done by either adding noise into the correlation data, random sampling of data to compute gradient and perturbation of data. From all three ways of adding stochasticity, adding noise to data is the worst method since it alters the underlying data and larger noise will make the clustering solution deviates away from the ground truth. Random sampling for gradient is a good approach since it doesn't change the correlation data and for sample size 0.3 or 0.4, both yield the closest objective maximization to the ground truth while at the same time avoiding saddle points at least until the data size is $n = 1000$.

Adding saddle points with perturbation during optimization is also a good approach and with small enough parameter $\mu = 0$ and $\sigma = 2$ for each independent random variable, this is able to prevent convergence to saddle point at least until the data size reaches 1000.

5.4 Conclusion

For the maximization agreement of correlation clustering problem, LSFW has the best performance both in terms of runtime and solution quality compared with other well known optimization methods such as QP, SDP and gradient descent methods.

Bibliography

[1] Erik Thiel, Morteza Haghir Chehreghani, Devdatt Dubhashi, *A Non-convex Optimization Approach to Correlation Clustering*, Thirty-Third AAAI Conference on Artificial Intelligence (AAAI), 2019.

[2] N. Bansal, A. Blum and S. Chawla, *Correlation clustering*, Machine Learning, 56(1-3):89-113, 2004.

[3] E.D. Demaine, D. Emanuel, A. Fiat and N. Immorlica, *Correlation clustering in general weighted graphs*, Theor. Comput. Sci., 361(2-3):172-187, 2006.

[4] M.H. Chehreghani, *Clustering by shift*, IEEE International Conference on Data Mining (ICDM), https://doi.org/10.1109/ICDM.2017.94, 2017.

[5] A.A. Goshtasby, *Similarity and dissimilarity measures*, Image Registration-Principles, Tools and Methods. Advances in Computer Vision and Pattern Recognition. Springer. pp.7-61, 2012.

[6] D.P. Williamson, D.B. Shmoys, *The Design of Approximation Algoritms*. Cambridge University Press. 2011.

[7] Jaggi, M. (2013) Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, 427-435.

[8] Lacoste-Julien, S.; Jaggi, M.; Schmidt, M. W.; and Pletscher, P. 2013. Block-coordinate frank-wolfe optimization for structural svms. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, 53-61.

[9] Mathieu, C., and Schudy, W. (2010) Correlation clustering with noisy input. In *Proceedings of the 21st Ánnual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, 712-728.

[10] Reddi, S. J.; Sra, S.; Póczos, B.; and Smola, A. J. 2016. Stochastic frank-wolfe methods for nonconvex optimization. In *54th Annual Allerton Conference on Communication, Control and Computing, Allerton 2016, Monticelli, IL, USA, September 27-30, 2016*, 1244-1251.

[11] Demaine, E. D.; Emanuel, D.; Fiat, A.; and Immorlica, N. 2006. Correlation clustering in general weighted graphs. *Theor. Comput. Sci.*, 361(2-3):172-187.

[12] Wang, H.; Banerjee, A. 2014. Randomized Block Coordinate Descent for Online and Stochastic Optimization.

[13] Wahde, M. 2008. Biologically Inspired Optimization Methods, An Introduction. WIT Press, ISBN: 978-1-84564-148-1.