
UNIVERSITY OF GOTHENBURG

Finding the Densest Common Subgraph with Linear Programming

Bachelor of Science Thesis in Computer Science and Engineering

Alexander Reinthal Anton Törnqvist Arvid Andersson
Erik Norlander Philip Stålhammar Sebastian Norlin

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 1, 2016

Bachelor of Science Thesis

Finding the Densest Common Subgraph with Linear Programming

Alexander Reinthal Anton Törnqvist Arvid Andersson
Erik Norlander Philip Stålhammar Sebastian Norlin

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Göteborg, Sweden, 2016

Finding the Densest Common Subgraph with Linear Programming
Alexander Reinthal
Anton Törnqvist
Arvid Andersson
Erik Norlander
Philip Stålhammar
Sebastian Norlin

Supervisor: Birgit Grohe
Examiner: Niklas Broberg, Department of Computer Science and Engineering

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2016

Finding the Densest Common Subgraph with Linear Programming

Alexander Reinthal
Anton Törnqvist
Arvid Andersson
Erik Norlander
Philip Stålhammar
Sebastian Norlin

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

Bachelor of Science Thesis

Abstract

This thesis studies the concept of *dense subgraphs*, specifically for graphs with multiple edge sets. Our work improves the running time of an existing Linear Program (LP) for solving the *Densest Common Subgraph* problem. This LP was originally created by Moses Charikar for a single graph and extended to multiple graphs (DCS_LP) by Vinay Jethava and Niko Beerenwinkel. This thesis shows that the run time of the DCS_LP can be shortened considerably by using an interior-point method instead of the simplex method. The DCS_LP is also compared to a greedy algorithm and a Lagrangian relaxation of DCS_LP. We conclude that the greedy algorithm is a good heuristic while the LP is well suited to problems where a closer approximation is important.

Keywords: Linear Programming, Graph theory, Dense Subgraphs, Densest Common Subgraph

Sammanfattning

Denna kandidatuppsats studerar *täta delgrafer*, speciellt för grafer med flera kantmängder. Den förbättrar körningstiden för ett befintligt Linjärprogram (LP) som löser *Densest common subgraph*-problemet. Detta LP skapades ursprungligen av Moses Charikar för en graf och utvidgades till flera grafer (DCS_LP) av Vinay Jethava och Niko Beerenwinkel. Uppsatsen visar att körtiden för DCS_LP kan förkortas avsevärt genom att använda en Interior-point metod i stället för simplex. DCS_LP jämförs med en girig algoritm och med en Lagrangian relaxation av DCS_LP. Slutligen konstaterades att DCS_GREEDY är en bra heuristik och att DCS_LP är bättre anpassat till problem där en närmare approximation sökes.

Nyckelord: Linjärprogrammering, Grafteori, Täta delgrafer, Tätaste gemensamma delgrafen

ACKNOWLEDGMENTS

We would like to thank Birgit Grohe for all the help she has provided when writing this paper and for going above and beyond what can be expected of a supervisor.

We would also like to thank Dag Wedelin for acquiring a version of CPLEX we could use, Ashkan Panahi for starting us off in the right direction and Simon Pedersen for his guidance on interior-point methods. Thanks also to all those who gave feedback and helped form the report along the way.

Contents

1 Introduction

In 2012, the IDC group expected a biennial exponential growth of the world's recorded data between 2012 and 2020 [1]. In 2015, Cisco reported a 500% increase in IP traffic over the last five years [2]. Data is getting bigger, and this calls for more efficient algorithms and data structures. Graphs are data structures well suited for relational data such as computer networks, social platforms, biological networks, and databases. For example, when modeling a social network as a graph, individuals are represented by vertices and friendships by edges connecting the vertices. In such a context it may be interesting to find a part of the social network — a subgraph — where a lot of people are friends. A subgraph where the ratio of edges to vertices is high is called a *dense subgraph*. A dense subgraph has the following relevant characteristics

- It has a low minimum distance between any two nodes. That is useful in routing or when planning travel routes to minimize the number of transfers [3].
- It is robust, which means that several edges need to be removed to disconnect the graph, e.g., the network is less likely to have a single point of failure [4].
- It has a high probability of propagating a message through the graph, which is useful for targeted commercials in social networks among other applications [3, 4].

It should be clear that finding these dense subgraphs are of interest for an array of applications. Therefore the problem of finding the *densest* subgraph is of great interest and has been studied since the 1980s.

A History of Algorithms

In 1984, A.V. Goldberg formulated a max flow algorithm for finding the densest subgraph of a single graph. His method determines an exact solution in $O(n^3 \log n)$ time for undirected graphs and therefore does not scale to large graphs. In 2000, Moses Charikar [5] presented a greedy $O(n+m)$ 2-approximation for the same problem. Charikar also presented the first Linear Programming (LP) formulation for finding the densest subgraph of a single graph in the same paper, a formulation which he proved optimal. His LP was later used to develop the Densest Common Subgraph LP (DCS_LP) by V. Jethava and N. Beerenwinkel [6]. Work has also been done in size restricted variants of the problems DkS (densest k-size subgraph), DalkS (densest at least k-size subgraphs) and DamkS (densest at most k-size subgraphs) [7, 8].

When one Graph is not Enough

Finding dense subgraphs has been used in recent years in the analysis of social networks [3, 4], in bioinformatics for analyzing gene-expression networks [9] and in machine learning [3, 10]. There are however limitations in finding a dense subgraph from a single graph. Measurements made in noisy environments, like those of microarrays in genetics research, will contain noise. Graphs created from this data will include edges that are the result of such noise. Attempting to find a dense subgraph might give a subgraph that contains noise and thus the subgraph may not exist in reality. One way to filter out those erroneous edges is to make multiple measurements. Dense Subgraphs that appear in most of the resulting graphs are probably real. Such subgraphs are called dense common subgraphs. One way to find a dense common subgraph is to aggregate the graphs into a single graph and then look for a dense subgraph. Aggregating graphs can, however, produce structures that are not present in the original graphs. Therefore, aggregating graphs can result in corrupted data [9].

This dilemma calls for a way of finding the densest common subgraph (DCS) from multiple edge sets. In late 2015 Jethava and Beerenwinkel presented two algorithms in [6], DCS_GREEDY and DCS_LP, for finding the densest common subgraph of a set of graphs. DCS_GREEDY is a fast heuristic that scales well to large graphs. DCS_LP is significantly slower but finds a tighter upper bound. In this thesis, we look at ways to make DCS_LP competitive with DCS_GREEDY in terms of speed.

1.1 Purpose

The densest common subgraph problem is: Given a set of graphs

$$\mathbb{G} = \{G^m = (V,\ E^m)\}_{m=1}^{M}$$

over the same set of vertices V, find the common subgraph $S \subseteq V$ with the maximum density over all the graphs. This problem is suspected to be NP-Hard[1] but no proof exists at this time [6].

There already exists algorithms for solving the problem, such as the Linear Program (LP) presented by Jethava and Beerenwinkel (see section 2.7 for further detail). Their results show that this LP does not scale well to large graphs. The purpose of this thesis is to find if there are ways to improve the scalability of the LP presented by Jethava and Beerenwinkel. There are two ways of doing this: using a better solving method or formulate a better LP. This paper considers both these approaches. To determine if this new LP is successful its solution will be compared in quality to the solutions of other algorithms, the old LP and an approximate greedy algorithm which section 2.8 introduces. How much time it takes to produce a solution will also be an important point of comparison and ultimately decide the feasibility of the algorithm in question.

1.2 Delimitations

This thesis has the following delimitations:

- The focus of this thesis is not to find the fastest or the most efficient way of finding the densest common subgraph, but to formulate an LP that is scalable to larger graphs. Therefore we do not attempt to find new algorithms that do not use Linear Programming.
- We will use existing LP solvers and known methods for solving and relaxing LPs.
- Only undirected graphs will be used.
- Only unweighted graphs will be used because algorithms for weighted graphs are more complex than those for unweighted graphs.
- Only the average degree definition of density will be utilized in finding the densest common subgraph.

[1] NP-hard problems are computationally expensive and do not scale to large inputs. NP-hard problems are at least as hard as all other problems in NP, see [11] for a list of NP-complete problems and some information on NP-hardness.

2 Theory

This section gives a presentation of the mathematical theory that is required to understand this thesis fully. Sections 2.3 and 2.5.1 are less formal than other and more intuitive explanations are presented. Some parts assume prior knowledge of multivariable calculus.

First, the graph terminology and notation that is used in this thesis is presented along with a few different definitions of density. We then give a summary and examples of our definitions. The density definitions are used for comparing the resulting subgraphs from our algorithms later on.

Subsequently, Linear Programming is introduced along with an example and an explanation of LP duality. A short explanation to Lagrangian relaxation follows, which is a way of simplifying a Linear Program. Then two separate solving methods for Linear Programming problems are then presented: the simplex method and interior-point method.

The last subsections present algorithms for solving the densest common subgraph problem. They contain the formulation of our Linear Programs and also a greedy algorithm.

2.1 Graph Terminology

An undirected graph $G = (V,\ E)$ consists of a set V with vertices $v_1,\ v_2,\ ...,\ v_n$, and a set E of edges $e_{ij} = \{v_i,\ v_j\}$ connecting those vertices. Sometimes the shorter notation i and ij will be used for v_i and e_{ij} respectively. The degree of a vertex is the number of edges connected to that vertex.

A set of M graphs over the same set of vertices but with different edge sets is written $\mathbb{G} = \{G^m = (V,\ E^m)\}_{m=1}^{M}$.

A graph $H = (S,\ E(S))$ is a subgraph to G if $S \subseteq V$ and if E(S) is the set of induced edges on S, i.e. all edges e_{ij} such that both $v_i,\ v_j \in S$. A common subgraph is a subgraph that appears in all graphs $\mathbb{G}$.

Given two graphs $(V,\ E^1)$ and $(V,\ E^2)$, a common subgraph will have vertex set $S \subseteq V$ and $E^1(S) \subseteq E^1,\ E^2(S) \subseteq E^2$. See figure 1 for an example of a graph with one vertex set and two edge sets (red/blue).

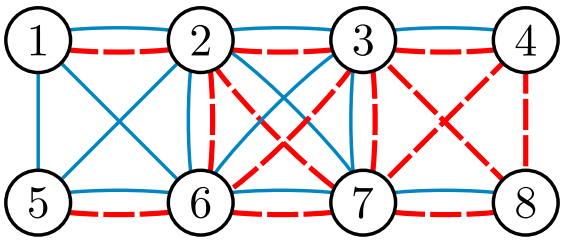


Figure 1: *A set of nodes* $\{1,\ ...,\ 8\}$ *and two edge sets shown in solid blue and dashed red respectively. The blue edges give a densest subgraph* $\{1,\ 2,\ 3,\ 5,\ 6,\ 7\}$ *while the red give* $\{2,\ 3,\ 4,\ 6,\ 7,\ 8\}$*, both with density (see section* 2.2.6*)* $11/6 \approx 1.8$*. The densest common subgraph is* $\{2,\ 3,\ 6,\ 7\}$ *with a density of* $6/4 = 1.5$*.*

The diameter of a graph $G = (V,\ E)$ is the length of the longest of all shortest paths i to j between any pair of nodes $(i,\ j) \in V \times V$. Less formally, the diameter is an upper bound to the length of the path one has to take to get from any node in the graph to any other node.

2.2 Definitions of Density

There are multiple formulations of density for graphs. These formulations differ to better suit the application for which they are used, but all of them measure in some way how tightly connected a set of vertices are. In this section, the different definitions of density used in this thesis are presented.

2.2.1 Relative and Absolute Density

There are two main classes of density, relative and absolute density. Relative density is defined by comparing the density of the subgraph with external components. Dense areas are defined by being separated by low-density areas. Relative density is often used in graph clustering [4] and will not be further discussed in this paper.

Absolute density instead uses a specific formula to calculate the density and every graph has an explicit value. Examples of this are cliques and relaxations thereof. The average degree that is used in this project is also an example of absolute density.

2.2.2 Cliques

A well-known measurement of density is cliques. A clique is a subset of vertices such that their induced subgraph is a complete graph. That means that every vertex has an edge to every other vertex in the subgraph. The problem of finding a clique with a given size is an NP-Complete problem [12]. The largest clique of a graph is therefore computationally hard to find, and it is also very hard to find a good approximation of the largest clique [13].

2.2.3 Quasi-Cliques

Many of the other absolute definitions of density are relaxations of cliques — so called quasi-cliques. These relaxations typically focus on a certain attribute of the relaxed clique: average degree, some measure of density or the diameter [4]. Because of the difficulty in finding cliques, many of the quasi-cliques are also computationally hard to find [13].

There are several definitions of quasi-cliques. A common definition is a restriction on their minimum degree: the induced subgraph must contain at least a fraction α of all the possible edges [13]. More formally: $S \subseteq V$ is an α-quasi-clique if

$$|E(S)| \geq \alpha \binom{|S|}{2}, \quad 0 \leq \alpha \leq 1.$$

2.2.4 Triangle Density

Density can also be expressed in how many triangles there are in a graph. A triangle is a clique of size 3 [3], see figure 2 for a visualization. Let $\mathrm{t}(S)$ be all triangles of the subgraph $(S,\ E(S))$. The maximum number of triangles in the node set S is $\binom{|S|}{3}$. The triangle density [13] is expressed as:

$$\frac{t(S)}{\binom{|S|}{3}}$$

Which is the percentage of triangles in the subgraph in comparison to the maximum amount of triangles possible.

2.2.5 Clustering Coefficient

The clustering coefficient of a graph has, like triangle density, a close relation to the number of triangles in the graph. Instead of taking the ratio of the number of triangles with the total number of possible triangles, the clustering coefficient is the ratio of triplets that are triangles and triplets that are not triangles. A triplet is an ordered set of three nodes. One of the nodes, the *middle* node, has edges to the other two nodes. If the two nodes that are not the middle node also are connected, the three nodes will be a triangle, and it will consist of three different connected triplets since each of the nodes can be a middle node.

$$\mathrm{C} = \frac{3 \cdot \text{ number of triangles}}{\text{number of connected triplets of nodes}} \tag{1}$$

2.2.6 Average Degree

Most commonly the density of a subgraph $S \subseteq V$ is defined by the average degree [4].

$$f(S) = \frac{|E(S)|}{|S|} \tag{2}$$

This definition has very fast approximate algorithms as well as exact algorithms [5] and will be the one used in this thesis. Restrictions such as a maximum or minimum size of S may be set but are uncommon and makes the problem NP-hard [4].

2.2.7 Summary of Definitions

In general, finding a quasi-clique is computationally more difficult than finding a dense subgraph with high average degree [4, 13]. However, quasi-cliques give a smaller, more compact subgraph with a smaller diameter and larger triangle density than those found by average degree methods [13].

To get a better feeling for what the definitions look like they are shown in figure 2.

- Graph density: 1.25 (5 edges, 4 nodes)
- Clique: $\{1,\ 2,\ 3\}$ and $\{2,\ 3,\ 4\}$
- Quasi-clique: 0.8, $\{1,\ 2,\ 3,\ 4\}$ ($\frac{5}{6}$ edges)
- Diameter: 2 (length from node 1 to 4)
- Triangle density: 0.5 (2 triangles out of 4)
- Clustering Coefficient: $\frac{2 \cdot 3}{8} = 0.75$

2.3 Linear Programming

Linear Programming is an optimization method. If a problem can be expressed as a Linear Program (LP), then it can be solved using one of several LP solution methods. There are several programs called *solvers* that implement these algorithms and can give a solution to a given LP problem. An LP minimizes (or maximizes) an objective function, which is a linear combination of variables, for instance

$$f(\mathbf{x}) = \mathbf{c}^\intercal \mathbf{x} = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$$

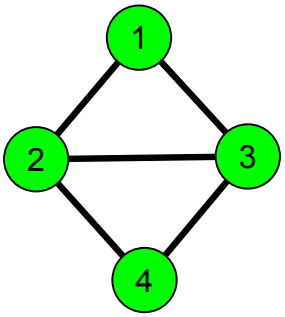


Figure 2: *A graph showing some examples of density definitions such as cliques* $\{1,\ 2,\ 3\}$ *(also a triangle) and quasi-clique* $\{1,\ 2,\ 3,\ 4\}$.

The LP model also contains a number of constraints that are usually written in matrix form $\mathbf{Ax} \leq \mathbf{b}$, which is equivalent to writing it out in the following way (for m number of constraints):

$$\begin{matrix} a_{11}x_1 + \ldots + a_{1n}x_n \leq b_1 \\ a_{21}x_1 + \ldots + a_{2n}x_n \leq b_2 \\ \vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m \end{matrix} \Leftrightarrow \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \tag{3}$$

An LP can be written in vector form like this:

Maximize	$\mathbf{c}^\intercal\mathbf{x}$	*(or minimize)*
where	$\mathbf{x} \in \mathbb{R}^n$	
subject to	$\mathbf{Ax} \leq \mathbf{b}$	
	$\mathbf{x} \geq 0$	*(element-wise,* $x_i \geq 0, \quad \forall x_i \in \mathbf{x}$*)*

If $\mathbf{x}$ has two elements, an LP can be visualized in two dimensions, see figure 3. Each constraint is drawn as a half-plane and the solution must be inside the convex polygon shown in gray. This polygon is called the *feasible region* and is bounded by the constraints. With more variables the feasible region becomes a polytope. A polytope is any geometrical figure with only flat sides. The feasible region still retains many of the same characteristics such as being convex. Being convex means that a line can be drawn from any point of the polytope to any other point of the polytope, and the line will only be drawn within the polytope, i.e. all points on the line will also be inside the polytope [14].
If the feasible region is convex and the objective function is linear we can make the following observations:

Observation 1 *Any local minimum or maximum is a global minimum or maximum [14].*

Observation 2 *If there exists at least one optimal solution, then at least one optimal solution will be located in an extreme point of the polytope [14].*

2.3.1 Example of a Linear Program

Here is an example of how Linear Programming can be used. A furniture factory produces and sells tables and chairs. The furniture is made from a limited supply of materials. The factory manager wants to produce the optimal amount of tables and chairs to maximize profit. Assume

that a table gives a profit of 10 and requires one large piece of wood and two small ones. A chair gives profit 6, with material cost one large and one small piece of wood. Furthermore, the factory's supply of materials is limited to seven large and ten small pieces. The problem is, what should the factory produce to maximize profit. If tables are x, and chairs are y, the profit can be written as a function: $profit = 10x + 6y$. This function is the objective function for the LP and it should be maximized. We also have constraints from the limited materials, and these are $x + y \leq 7$ from large pieces and $2x + y \leq 10$ from small pieces.

$$\begin{aligned} \text{Maximize} \quad & z = 10x + 6y \\ \text{subject to} \quad & x + y \leq 7 \\ & 2x + y \leq 10 \\ & x,\ y \geq 0 \end{aligned}$$

Since this LP is two-dimensional, it can be graphically visualized as shown in figure 3. In the figure, we see our constraints (the lines) and the feasible region they create (the gray area). A solution to the LP can exist only inside the feasible region and an optimal solution exists where the constraints cross each other. Those points (labeled 1–4 in the figure) are known as extreme points.

In this problem, we have four extreme points. One way to find the optimal solution is to consider all extreme points. This follows directly from observation 2. The extreme point labeled 1 can be immediately dismissed as the goal was to maximize profit. If nothing is sold, no profit is made. So now we are left with three possible solutions that we input to the objective function, and we pick the solution that gives the highest value. In this problem, it was extreme point 3 at position (3, 4) which gives a profit of 54. That means that the most profitable alternative for the factory is to make three tables and four chairs.

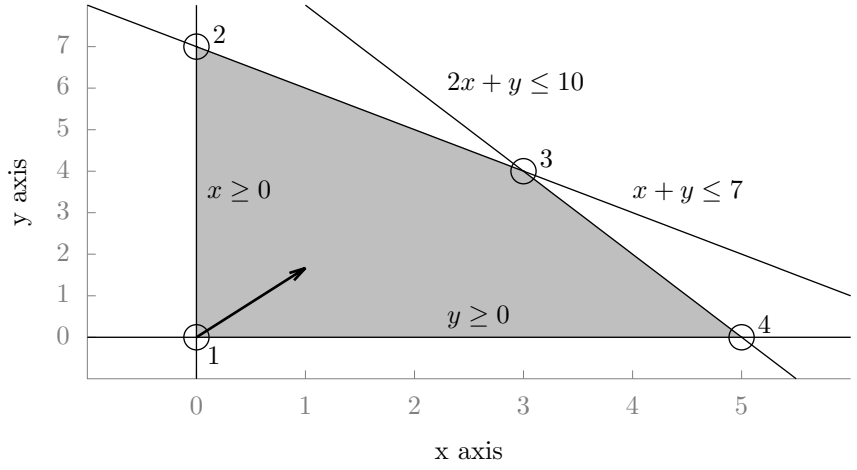


Figure 3: *This is a graphical representation of the LP for the furniture factory example. The feasible region is shaded, and the constraints are the various lines. The extreme points are circled and numbered. The arrow represents the direction in which the objective function grow*

2.3.2 Duality of Linear Programs

Consider the example of the furniture factory, presented again for clarity:

$$\begin{aligned} \text{Maximize} \quad & z = 10x + 6y && \text{(4a)} \\ \text{subject to} \quad & x + y \leq 7 && \text{(4b)} \\ & 2x + y \leq 10 && \text{(4c)} \\ & x,\ y \geq 0 && \text{(4d)} \end{aligned}$$

It is immediately clear from observing constraints (4a) and (4b) coupled with the fact that both x and y have to be non-negative, that an upper bound to the profit is 70. This can sometimes be a very useful observation, but can this upper bound be made tighter? Constraints (4b) and (4c) may be added together to obtain the much tighter upper bound of 56.7 which is not far off the optimal 54.

$$3x + 2y = 17 \Rightarrow 10x + 6.7y = 56.7$$

The equivalent problem of finding a tighter and tighter upper bound is called the *dual* of the original *primal* problem [15]. All LPs have a corresponding dual problem. If the primal is a maximization, then the dual is a minimization and vice versa. The optimal solution to the primal and the dual are equivalent and can therefore be obtained from solving either of them [15]. Some methods, as shown in section 2.5.2, use information from both the dual and the primal to more quickly converge to an optimal solution.

The relationship between a primal and dual formulation is shown in equation (5). To the left, the primal problem is shown and to the right, the equivalent dual problem is shown.

$$\begin{array}{cc|cc} \multicolumn{2}{c|}{\text{Primal}} & \multicolumn{2}{c}{\text{Dual}} \\ \text{Maximize} & \mathbf{c}^\intercal\mathbf{x} & \text{Minimize} & \mathbf{b}^\intercal\mathbf{y} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} & \text{subject to} & A^\intercal\mathbf{y} \geq \mathbf{c} \\ & \mathbf{x} \geq 0 & & \mathbf{y} \geq 0 \end{array} \tag{5}$$

where $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$.

2.4 Lagrangian Relaxation

Lagrangian relaxation is a general relaxation method applicable to constrained optimization problems. The general idea is to relax a set of constraints by adding penalizing multipliers which punish violations of these constraints. The objective is to minimize the penalty multipliers so that no constraint is violated.

Consider the general LP:

$$\begin{aligned} \text{Maximize} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ & D\mathbf{x} \leq \mathbf{d} \\ & \mathbf{x} \geq 0 \end{aligned}$$

where $f(\mathbf{x})$ is linear, and $\mathbf{Dx} \leq \mathbf{d}$ are considered *difficult* constraints. These difficult constraints can for example be constraints that are computationally harder than the others. For instance, compare the two constraints $x_1 \geq 0$ and $\sum_{i=0}^{n} x_i \leq 4$ for some large n. The second constraint obviously takes more computing power to check than the first one and is therefore *harder* than

the first constraint and probably suited to relax. It is generally hard to define what makes constraints difficult since this varies for different models. Thus, an analysis of the given model is needed.

A Lagrangian relaxation of an LP removes these difficult constraints and moves them to the objective function paired with a Lagrangian multiplier [14]. This multiplier acts as a penalty for when the constraint is violated.

For the formerly mentioned LP we obtain the following Lagrangian relaxation:

$$\begin{array}{ll} \text{Maximize} & f(\mathbf{x}) + \boldsymbol{\lambda}^{\intercal}(\mathbf{d} - D\mathbf{x}) \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array} \tag{6}$$

Assuming $\boldsymbol{\lambda}$ is constant we now have an easier LP to solve, now the problem is to find good values for $\boldsymbol{\lambda}$. This is called the Lagrangian dual problem, and can be written as the following unconstrained optimization problem:

$$\begin{array}{ll} \text{Minimize} & Z(\boldsymbol{\lambda}) \\ \text{subject to} & \boldsymbol{\lambda} \geq 0 \end{array}$$

Where $Z(\boldsymbol{\lambda})$ is the whole LP shown in (6). There are a few methods for solving this problem, but only one is presented in this thesis: the subgradient method.

2.4.1 Subgradient Method

The subgradient method is an iterative method that solves a series of LPs and updates the values of $\boldsymbol{\lambda}$ by the following procedure:

1. Initialize $\boldsymbol{\lambda}^0$ to some value.
2. Compute $\boldsymbol{\lambda}^{k+1} = \min\{0,\ \boldsymbol{\lambda}^k - \alpha^k(\mathbf{d} - \mathbf{D}\mathbf{x}^k)\}$ for the kth iteration.
3. Stop when k is at some set iteration limit.

Here $\mathbf{x}_k$ is the solution obtained in iteration k and α^k the step size. There are different ways to calculate the step size, but the formula proven most efficient in practice is the following [16]:

$$\alpha^k = \mu^k \frac{Z(\boldsymbol{\lambda}^k) - Z^*}{||\mathbf{d} - \mathbf{D}\mathbf{x}^k||^2} \tag{7}$$

Where $||\,.\,||$ is the Euclidean norm, Z^* is the objective value,i.e. the vale of the objective function, of the best known feasible solution, often computed by some heuristics. The scalar μ^k is initiated to the value 2 and halved when whenever the solution to $Z(\boldsymbol{\lambda})$ failed to improve for a given number of iterations.

2.5 Methods for Solving Linear Programs

The approach of considering all extreme points to solve a Linear Program shown in section 2.3.1 becomes unfeasible for larger problems. This due to the fact that the the number of vertices that must potentially be examined is:

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

Where n is the dimension of the solution, i.e. the amount of variables, and m is the number of constraints [17]. Therefore, a better method of solving the LP must be used.

Today there exists multiple commercial and free programs that solve Linear Programs [18]. These are henceforth called solvers. These solvers use different algorithms to find the solution to the optimization problem.

There are two main groups of algorithms used by solvers today: variations of the simplex method and various types of interior-point algorithms. These two groups of algorithms have a fundamental difference in their approach to solving linear optimization problems. The simplex algorithm considers the extreme points of the LP's feasible region and traverses those. The interior-point methods start inside the feasible region and follows gradients to the optimal solution. Interior-point algorithms use similar techniques to those used in unconstrained optimization [19].

The following sections give an overview of the algorithms and guide the curious reader to additional material for a more complete understanding of the methods.

2.5.1 The Simplex Method

The simplex algorithm is a frequently used and well-known algorithm originally designed by George Dantzig during the second world war [14]. Today it is one of the most used algorithms for linear optimization, and almost all LP solvers implement it [18]. It locates the minimum value of the problem by traversing the feasible region's vertices until a minimum is found. See figure 4 for a visualization of an example in three dimensions.

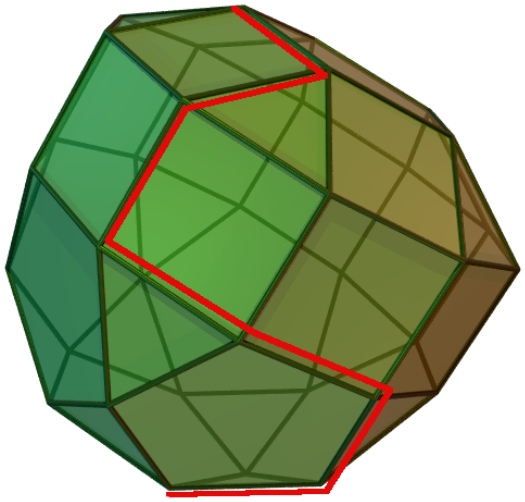


Figure 4: *A graphical interpretation of how the simplex algorithms works. It traverses the extreme points, i.e. corners of the feasible region, continuously improving the value of the objective function until it finds the optimal solution [20]. Image by wikimedia, user Sdo, licensed under CC BY-SA 3.0.*

The simplex method is most easily explained using an example. We will reuse the example from chapter 2.3.1 and obtain the optimal value with the simplex method. For clarity the problem

is presented again below:

$$\begin{aligned} \text{Maximize} \quad & z = 10x + 6y && \text{(8a)} \\ \text{subject to} \quad & x + y \leq 7 && \text{(8b)} \\ & 2x + y \leq 10 && \text{(8c)} \\ & x,\ y \geq 0 && \text{(8d)} \end{aligned}$$

on matrix form the matrix A and vectors $\mathbf{c}, \mathbf{b}$ have the following values for the above example:

$$A = \begin{pmatrix} 1 & 1 \\ 2 & 1 \end{pmatrix} \quad \mathbf{c} = (1, 2) \quad \mathbf{b} = (7, 10) \tag{9}$$

Before initializing the simplex method the LP is rewritten in standard form. This is because it will significantly simplify parts of the algorithm [14]. Both the constraints (8b) and (8c) are of the form $A\mathbf{x} \leq \mathbf{b}$ and will be reformulated on the form $A\mathbf{x} = \mathbf{b}$. This is done by introducing slack variables s_1 and s_2 so that the new, but equal, LP becomes:

$$\begin{aligned} \text{Maximize} \quad & z = 10x + 6y && \text{(10a)} \\ \text{subject to} \quad & x + y + s_1 = 7 && \text{(10b)} \\ & 2x + y + s_2 = 10 && \text{(10c)} \\ & x,\ y,\ s_1,\ s_2 \geq 0 && \text{(10d)} \end{aligned}$$

This is equal to the following matrix where the values are the coefficients in front of the variables:

$$\begin{array}{cccccc} z & x & y & s_1 & s_2 & b \\ \begin{pmatrix} 1 & -10 & -6 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 7 \\ 0 & \underline{2} & 1 & 0 & 1 & 10 \end{pmatrix} \end{array} \tag{11}$$

where the header row describes which variable is represented in each column. The slack variables $s_1,\ s_2$ are called *basic variables* and $x,\ y$ are called *non-basic variables* and z is the value of the objective function. In this case the variables are initialized to $x = y = 0$ and $s_1 = 7,\ s_2 = 10$. This is called the *basic* solution as all non-basic variables are 0. This is the standard starting point of the simplex algorithm [14]. Geometrically these values represent point 1 in figure 3.

The algorithm terminates when all coefficients on the first row of the non-basic variables are non-negative. This yields an optimal solution, and a proof of this is given in [14]. As long as there are negative coefficients we need to pivot on one of the non-basic variables x or y. The process of pivoting is explained below.

A pivot variable is chosen among those non-basic variables with negative coefficients. This can be done either arbitrarily, or by choosing the lowest coefficient. In this case, we will pivot on x as it has the lowest coefficient at -10.

The row is then chosen to specify a specific element in the matrix. The row with the smallest ratio of $\frac{b_i}{a_{ij}}$ should be used, where a_{ij} is the ith row and jth column of A, and $A, \mathbf{b}$ are shown in equation 9. Intuitively this can be understood as the jth non-basic variable in constraint i. In this case we need to consider $\frac{b}{x}$ and $\frac{10}{2}$ is smallest on row 3, so that row will be used and the chosen pivot variable a_{ij} is underlined in (11). The reason the lowest value of $\frac{b_i}{a_{ij}}$ is chosen is that it gives the highest value that the pivot variable can take without invalidating any of the constraints [14].

A pivot is performed by dividing the chosen row so the coefficient of the pivot variable is one. Then elementary matrix operations are performed to reduce the coefficient of the chosen non-basic variable to 0 in all other rows. These operations are only valid when the LP has constraints on the form $\mathbf{Ax} = \mathbf{b}$ and is the reason inequality constraints are reformulated. After a pivot our matrix looks as follows:

$$\begin{matrix} z & x & y & s_1 & s_2 & b \end{matrix}$$
$$\begin{pmatrix} 1 & 0 & -1 & 0 & 5 & 50 \\ 0 & 0 & \underline{0.5} & 1 & -0.5 & 2 \\ 0 & 1 & 0.5 & 0 & 0.5 & 5 \end{pmatrix} \tag{12}$$

The values of our variables in (12) are $x = 5$, $y = s_2 = 0$, $s_1 = 2$ which corresponds to the point 4 in figure 3. Pivoting on the underlined element in (12), chosen by the same logic as before, we obtain the matrix:

$$\begin{matrix} z & x & y & s_1 & s_2 & b \end{matrix}$$
$$\begin{pmatrix} 1 & 0 & 0 & 2 & 4 & 54 \\ 0 & 0 & 1 & 2 & -1 & 4 \\ 0 & 1 & 0 & -1 & 1 & 3 \end{pmatrix} \tag{13}$$

This matrix has only positive coefficients of the non-basic variables on the first row and is the optimal solution. The variables have the values: $x = 3$, $y = 4$, $s_1 = 0$, $s_2 = 0$ and the value of the objective function is 54. This corresponds to point 3 in figure 3. The simplex method has thus, by pivoting, traversed the route of extreme points $1 \to 4 \to 3$ in figure 3.

2.5.2 Interior-Point Methods

Recall the standard Linear Programming problem:

$$\begin{aligned} \text{Maximize} \quad & \mathbf{c}^\intercal \mathbf{x} && \text{(14a)} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} && \text{(14b)} \\ & \mathbf{x} \geq 0 && \text{(14c)} \end{aligned}$$

Interior-point methods are a subgroup of barrier methods. Instead of having constraints that are separate to the objective function a new composite objective function, say B, is created that significantly reduces the value of B if any of the old constraints are not satisfied [21]. This way the algorithm will stay inside the old feasible region. One such barrier function is the logarithmic barrier function written as follows:

$$B(\mathbf{x},\ \mu) = \mathbf{c}^\intercal \mathbf{x} + \mu \sum_{i=1}^{m} \ln\left(\mathbf{b} - \mathbf{a}_i^\intercal \mathbf{x}\right) \tag{15}$$

Where a_i is the ith row of A and μ is a scalar that is used to weight the penalty given by the constraints. $B(\mathbf{x},\ \mu)$ behaves like $\mathbf{c}^\intercal \mathbf{x}$ when $\mathbf{x}$ lies within the feasible region but as the function gets closer to the constraints the second term will approach negative infinity. This is what creates the *barrier* that makes solutions outside of the constraints implausible and keeps the algorithm from moving outside of the original feasible region.

For large μ the constraints will have a noticeable effect on the value of B further from the constraint boundary. This makes the interior region smoother. As μ decreases, the shape of the interior region will resemble the original feasible region. The algorithm starts with a large μ in

order to start in the center of the polytope [15]. As μ is decreased and the algorithm follows the gradient it will follow the so-called *central path* to the unique optimal solution [15].

Below is a more mathematical, though very short, explanation of a primal-dual interior-point method for LPs. This is important in order to get a full understanding of why interior-point methods work particularly well for the DCS problem. Interior-point methods are more mathematically complex than the simplex method. Because of the limited scope of this thesis we are not able to give a full explanation of the underlying mathematical theory nor do we give any proofs. We recommend reading chapter 7.2 of [15] which gives a clear and expanded explanation of the below material. For some mathematical background knowledge we recommend [21] in its entirety or at least chapter 1.4, 2.1, 2.2, 3.1, 4.1, 4.2. For an implementation of the algorithm partially presented below see [22].

When solving an LP using interior-point methods, Newtons method will be utilized. This cannot take inequality constraints so the LP shown in equation (14) will be rewritten in the form:

$$\text{Maximize} \qquad \mathbf{c}^\intercal \mathbf{x} + \mu \sum_{j=1}^{n} \ln x_j \tag{16a}$$

$$\text{subject to} \qquad A\mathbf{x} = \mathbf{b} \tag{16b}$$

where $\mu > 0$ is a scalar and where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{x} > 0$. The solution will also be approximate as $\mathbf{x} > 0$. The constraints may also be moved into the objective function with λ as the Lagrange multiplier to obtain the Lagrangian:

$$\text{Maximize} \qquad L(\mathbf{x},\ \boldsymbol{\lambda}) = \mathbf{c}^\intercal \mathbf{x} + \mu \sum_{j=1}^{n} \ln x_j + \boldsymbol{\lambda}(A\mathbf{x} - \mathbf{b}) \tag{17}$$

the gradient of L is split into two parts, the x and λ variables and is:

$$-\nabla_{\mathbf{x}} L(\mathbf{x},\ \boldsymbol{\lambda}) = -\mathbf{c} + A^\intercal \boldsymbol{\lambda} + \mu X^{-1}\mathbf{1} \tag{18a}$$

$$-\nabla_{\lambda} L(\mathbf{x},\ \boldsymbol{\lambda}) = \mathbf{b} - A\mathbf{x} \tag{18b}$$

where X denotes the diagonal matrix of $\mathbf{x}$ where $X_{ii} = \mathbf{x}_i$ and $\mathbf{1}$ is a vector of 1's. Uppercase letters will be used for this type of diagonal matrix throughout the rest of the chapter as is the standard for LP texts [21]. LP (16) is called the barrier subproblem and it satisfies optimality conditions:

$$\nabla_{\boldsymbol{\lambda}} L(\mathbf{x},\ \boldsymbol{\lambda}) = 0 \tag{19a}$$

$$\nabla_{\mathbf{x}} L(\mathbf{x},\ \boldsymbol{\lambda}) = 0 \tag{19b}$$

. We split (18a) into:

$$\mathbf{c} = A^\intercal \boldsymbol{\lambda} + \mathbf{z} \tag{20}$$

$$X\mathbf{z} = \mu\mathbf{1} \tag{21}$$

with $\mathbf{z} = \mu X^{-1}\mathbf{1}$.

We then have three equations: 18b, 20, 21 and given (19) we can construct the so called *central path conditions* shown in equation (22). The central path conditions ensure that the algorithm converges to an optimal solution x^*. They are explained in greater detail in section 3.4 of [21].

$$A\mathbf{x} = \mathbf{b},\ \mathbf{x} > 0,\ \mathbf{c} = A^\intercal\boldsymbol{\lambda} + \mathbf{z},\ \mathbf{z} > 0,\ X\mathbf{z} = \mu\mathbf{1} \tag{22}$$

We seek three sets of variables: the primal variables $\mathbf{x}$, dual variables for equality constraints $\boldsymbol{\lambda}$, and the dual variables for inequality constraints $\mathbf{z}$ [23]. To do this we solve for the newton steps in $x,\ y$ and λ according to:

$$\begin{pmatrix} A & 0 & 0 \\ 0 & A^\intercal & I \\ Z & 0 & X \end{pmatrix} \begin{pmatrix} \mathbf{p}_x \\ \mathbf{p}_\lambda \\ \mathbf{p}_z \end{pmatrix} = \begin{pmatrix} \mathbf{b} - A\mathbf{x} \\ \mathbf{c} - \mathbf{A}^\intercal\mathbf{x} + \mathbf{z} \\ \mu\mathbf{1} - XZ\mathbf{1} \end{pmatrix} \tag{23}$$

One way to solve this large system of 2n+m equations is by breaking out $\mathbf{p}_\lambda$ and solving for it in the linear equation below. This result can then be used to find $\mathbf{p}_x$ and $\mathbf{p}_z$ [23].

$$AZ^{-1}XA^\intercal\mathbf{p}_\lambda = AZ^{-1}X(\mathbf{c} - A^\intercal\mathbf{y} - \mathbf{z}) + \mathbf{b} - A\mathbf{x} \tag{24}$$

$\mathbf{p}_x$, $\mathbf{p}_z$ and $\mathbf{p}_\lambda$ are calculated in every iteration of the algorithm and used together with a scalar α to update $\mathbf{x}, \mathbf{z}, \boldsymbol{\lambda}$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}_x$$
$$\mathbf{z}_{k+1} = \mathbf{z}_k + \alpha\mathbf{p}_z$$
$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k + \alpha\mathbf{p}_\lambda$$

No full algorithm is presented as it is out of the scope of this thesis but a very clear presentation of a primal-dual interior-point algorithm is presented in [22]. Most important to note however is that each iteration of primal-dual interior-point algorithm has cubic complexity from the matrix multiplications in equation (24) [22].

2.6 Finding Dense Subgraphs Using Linear Programming

The problem of finding the densest subgraph S in an undirected graph $G = (V, E)$ can be expressed as a Linear Program. In this section a derivation of Charikar's LP [5] for finding the densest subgraph is presented, although all details are not rigorously proven because of the scope of this thesis but can be found in [5]. Recall the definition of density of a graph in section 2.2.6, meaning the density of S is $f(S) = \frac{|E(S)|}{|S|}$.

We define the following variables by assigning a value y_i for every node in V:

$$y_i = \begin{cases} 1, & i \in S \\ 0, & \text{otherwise} \end{cases} \tag{25}$$

That means that if node v_i is chosen to be in the subgraph, then the corresponding LP variable y_i will have value 1. Now, analogously, assign each edge in E with a value x_{ij}.

$$x_{ij} = \begin{cases} 1, & i, j \in S \\ 0, & \text{otherwise} \end{cases} \Leftrightarrow x_{ij} = \min\{y_i,\ y_j\} \tag{26}$$

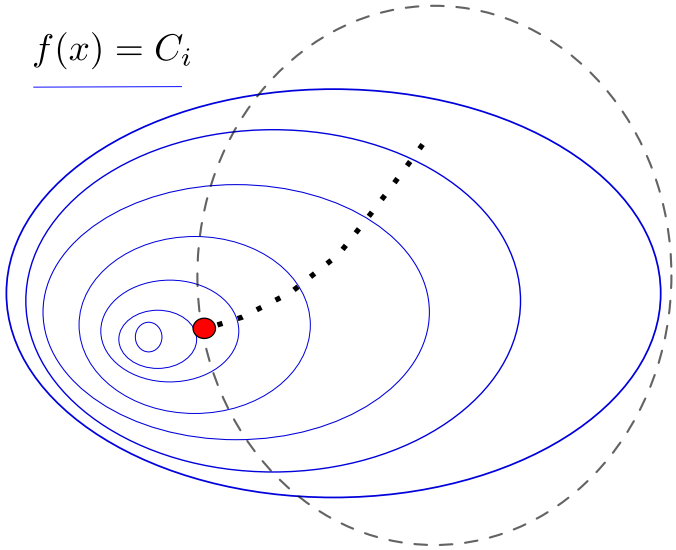


Figure 5: *The area within the dotted circle represents the feasible region of the objective function and the blue ellipses are different contour lines. The interior-point method first finds a feasible solution, and then moves towards the optimal value (the red circle). Since the step lengths become smaller and smaller it retrieves an approximation of the optimal value.*

Which means that all edges connecting two chosen nodes will have their corresponding LP variable assigned value 1. Using these equations the density of the subgraph S is simply $\frac{\sum x_{ij}}{\sum y_i}$.

Finding the maximum density of some subgraph $H = (S,\ E(S))$ can thus be expressed as an optimization problem, where the objective function is the following[2]:

$$\max_{S \subseteq V} f(S) \Leftrightarrow \max_{S \subseteq V} \frac{|E(S)|}{|S|} \Leftrightarrow \max \frac{\sum x_{ij}}{\sum y_i} \tag{27}$$

subject to the following constraints:

$$\begin{aligned} x_{ij} &= \min\{y_i,\ y_j\} \\ y_i &\in \{0,\ 1\} \end{aligned} \tag{28}$$

The function in (27) is not a linear function since it is a fraction of two sums, also both y_i and x_{ij} can only take discrete values. What that means is that a few changes need to be made to make this a Linear Program. The first step is to relax the binary variables by changing the constraint $y_i \in \{0, 1\}$ into $0 \leq y_i \leq 1$. And secondly the constraint $x_{ij} = min\{y_i, y_j\}$ must be described in terms of inequality constraints. These two steps then gives the following Linear Fractional Program:

[2]Observe that the denominator is only 0 when no nodes are chosen so this is not of concern since the objective is to maximize.

$$\begin{aligned} \text{Maximize} \quad & \frac{\sum x_{ij}}{\sum y_i} \\ \text{subject to} \quad & x_{ij} \leq y_i \\ & x_{ij} \leq y_j \\ & 0 \leq y_i \leq 1 \end{aligned} \tag{29}$$

Here the requirement $x_{ij} = min\{y_i,\ y_j\}$ is described in the two first inequality constraints. This program can then be transformed into an LP by a Charnes-Cooper transformation [24], that is shown below.

Introduce a new variable z defined as:

$$z = \frac{1}{\sum y_i}$$

Substituting in z by Charnes-Cooper transformation gives the following LP:

$$\begin{aligned} \text{Maximize} \quad & \sum z x_{ij} \\ \text{subject to} \quad & \sum z y_i = 1 \\ & z x_{ij} \leq z y_i \\ & z x_{ij} \leq z y_j \\ & 0 \leq y_i \leq 1 \end{aligned} \tag{30}$$

A constraint has been added to deal with the non-linearity. To further reduce the program two more substitutions are made:

$$\begin{aligned} x'_{ij} &= z x_{ij} \\ y'_i &= z y_i \end{aligned}$$

The Linear Program is transformed into the following:

$$\begin{aligned} \text{Maximize} \quad & \sum x'_{ij} \\ \text{subject to} \quad & \sum y'_i = 1 \\ & x'_{ij} \leq y'_i \\ & x'_{ij} \leq y'_j \end{aligned}$$

Finally the notation using prime ($'$) is dropped since it was only used to clarify the variable substitution. The final Linear Program is written as:

$$\begin{aligned} \text{Maximize} \quad & \sum x_{ij} \\ \text{subject to} \quad & \sum y_i = 1 \\ & x_{ij} \leq y_i \qquad \forall ij \in E \\ & x_{ij} \leq y_j \qquad \forall ij \in E \\ & x_{ij}, y_i \geq 0 \end{aligned} \tag{31}$$

This is the same LP formulation that Charikar gives. Thus the derivation is complete.

Observe that formulation (31) is a relaxation of the original formulation. A proof that formulation (31) recovers exact solutions to the maximum dense subgraph problem, in other words $OPT\ (31) = \max_{S \subseteq V} f(S)$, is shown in [5].

2.7 Finding the Densest Common Subgraph of a Set of Graphs

As shown by [6], LP (31) from the previous section can be extended for multiple edge sets. The common density of a subgraph $S \subseteq V$ is defined to be its smallest density in all edge sets. Recall definition (2) of density given one graph. Now, extend this definition for a given graph $G^m \in \mathbb{G}$ in the following way $f_{G^m} = f_m(S) = \frac{|E^m(S)|}{|S|}$. Given this extended definition the common density is formulated as:

$$\delta_{\mathbb{G}}(S) = \min_{G^m \in \mathbb{G}} \ f_m(S)$$

The problem now is to find an S that maximizes the common density.

Just like in section 2.6, each edge in all edge sets is associated with a variable x_{ij}^m, that is assigned a value based on whether or not there is an edge between i and j in edgeset m. This process can be expressed, very much like in section 2.6, as:

$$x_{ij}^m = \min\{y_i,\ y_j\},\ \forall m = 1..M,\ \forall ij \in E^m$$

Introducing a new variable t and using the fact that LP (31) gives us the maximum density of a graph allows the construction of an LP that gives a solution to the densest common subgraph problem. Start with LP (31), then introduce the constraint:

$$\sum_{ij \in E^m} x_{ij}^m \geq t$$

for each graph and change the objective function to maximize over t instead. The new constraint bounds t above by the lowest density $\sum_{ij \in E^m} x_{ij}^m$ and since the objective is to maximize t the solution should give the densest common subgraph. The modified LP is referred to as DCS_LP and shown below:

$$\begin{aligned}
\text{Maximize} \quad & t && && \text{(32a)}\\
\text{subject to} \quad & \sum_{i=1}^{n} y_i \leq 1 && && \text{(32b)}\\
& \sum_{ij \in E^m} x_{ij}^m \geq t && \forall E^m,\ m = 1..M && \text{(32c)}\\
& x_{ij}^m \leq y_i,\ x_{ij}^m \leq y_j && \forall ij \in E^m && \text{(32d)}\\
& x_{ij}^m \geq 0,\ y_i \geq 0 && \forall ij \in E^m,\ \forall i = 1..n && \text{(32e)}
\end{aligned}$$

It is shown in [6] that DCS_LP will only give an optimal solution t^* if $y_i = \frac{1}{|S|}, \forall i \in S$. A proof of this is given by [6] and we present this proof with some extra explanation of each step.

Construct a feasible solution as follows: Let $S \subseteq V$, $n = |S|$ and

$$y_i = \begin{cases} \frac{1}{n}, & i \in S \\ 0, & \text{otherwise} \end{cases} \qquad x_{ij} = \begin{cases} \frac{1}{n}, & i, j \in S \\ 0, & \text{otherwise} \end{cases}$$

then,

$$\sum_{ij \in E^{(m)}} x_{ij}^{(m)} = \frac{1}{n} \sum_{ij \in E^{(m)}} 1 = \frac{|E(S)|}{|S|} = \delta_G(S)$$

this gives: max t $= \delta_{\mathbb{G}}(S)$ because of the constraint 32c.

If DCS_LP does not give the optimal solution it will give an upper bound on the optimal solution. A lower bound on the actual optimal solution of $\frac{t^*\lceil t^*+1 \rceil}{|S|}$ is given by [6]. This lower bound is tight if the subgraph is a near clique but gets progressively worse the further from a clique it is. Finding a tighter upper bound is an interesting topic for further research.

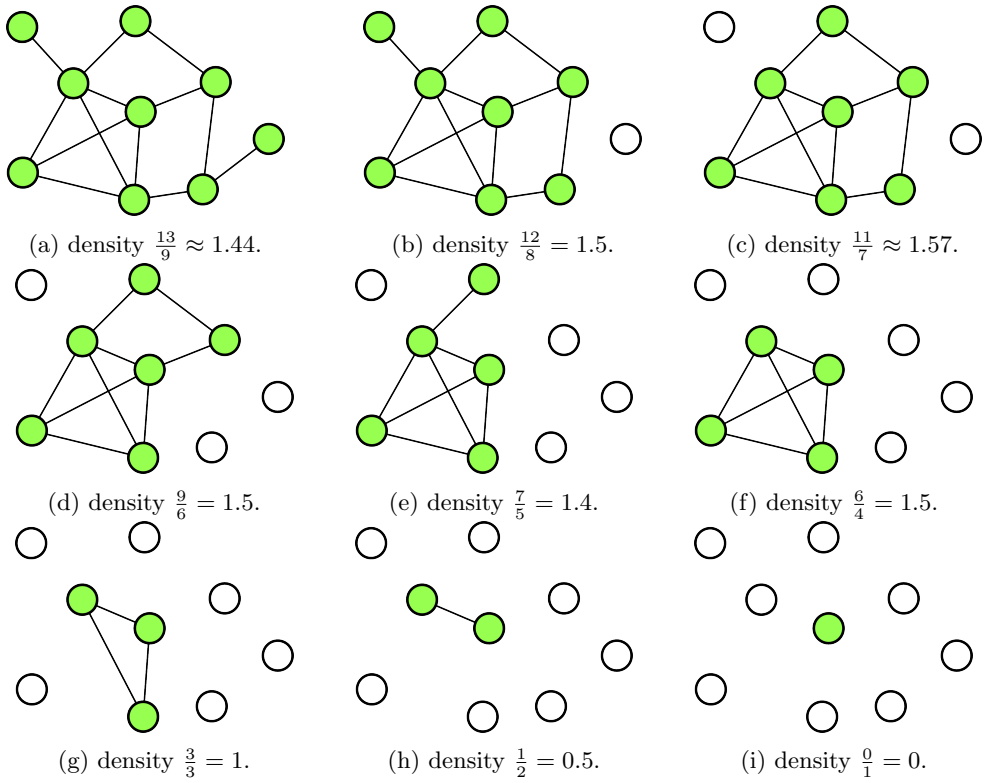
2.8 Finding the Densest Common Subgraph Using a Greedy Algorithm

A greedy algorithm for finding the densest subgraph in a *single* graph was first presented in [5]. This algorithm starts from the full graph and proceeds by removing a single node, the node with the smallest degree, at each iteration until the graph is empty (see figure 6). Which means that the complexity of the algorithm is $O(|V| + |E|)$ given a graph $G = (V,\ E)$, making it very efficient. The solution is a 2-approximation, meaning it is at worst, half as good [3] as the optimal solution as proven in [5].

In [6] the algorithm DCS_GREEDY was presented as a modified version of the greedy algorithm presented in [5]. DCS_GREEDY was modified to find the densest *common* subgraph among a set of graphs $\mathbb{G} = \{G^m = (V,\ E^m)\}_{m=1}^{M}$. This algorithm can be implemented in $O(n+m)$ time where $n = M \cdot |V|$ and $m = \sum_{i=1}^{M} |E^i|$ for the graph set $\mathbb{G}$. The key to doing this is to keep an array of lists where each list contains nodes that have the same degree. This algorithm however is not a 2-approximation but rather a good heuristic. This way the algorithm can quickly retrieve the node with least degree. Pseudo code of the algorithm, which builds a number of solutions V_t and returns the one with the highest density, is presented below:

```
Initialize V1 := V, t := 1
while degree(Vt) > 0 do
  For each node, find its minimum  degree among edge sets
  n is the node that has the smallest minimum degree
  Vt+1 := Vt \ n
  t := t+1
  density of Vt is min(|E^m|/|Vt|) among edge sets E
end while
return the Vt with the highest density
```

[3] Given the optimal density t^* and the solution s of the greedy algorithm, then $t^* \geq s \geq t^*/2$.



(a) density $\frac{13}{9} \approx 1.44$. (b) density $\frac{12}{8} = 1.5$. (c) density $\frac{11}{7} \approx 1.57$.

(d) density $\frac{9}{6} = 1.5$. (e) density $\frac{7}{5} = 1.4$. (f) density $\frac{6}{4} = 1.5$.

(g) density $\frac{3}{3} = 1$. (h) density $\frac{1}{2} = 0.5$. (i) density $\frac{0}{1} = 0$.

Figure 6: *An example of using the greedy algorithm on a single graph. Each step removes the node with the smallest degree. 6c has the highest density so that is the solution.*

3 Method and Procedure

The approach used when exploring this problem was to implement the greedy algorithm from section 2.8 and the LP from section 2.7 and compare these two with different relaxations of the LP. After these three algorithms, DCS_GREEDY, DCS_LP and a Lagrange relaxation of DCS_LP called DCS_LAG (presented in section 4.2) were implemented, we ran them on various graph sets. The data sets came from the Stanford Large Network Dataset Collection [25], and from the DIMACS collection from Rutgers University [26], as well as some synthetic data we generated. The data sets we used are presented in table 1. The LP solver we used was CPLEX [27] from IBM. The solving methods used in CPLEX were Simplex and Barrier, an interior-point method. We also experimented using the Sift method from CPLEX.

The algorithms DCS_LP, DCS_GREEDY, and DCS_LAG, were benchmarked on the quality of their solution as well as the running time of the respective program. See tables 2 and 3 for a summary of their results and properties of the solutions. We examine these qualities and their difference and discuss which method gives best results and why.

3.1 The Data Sets

Each data set is a collection of edge sets on roughly the same node set. These data sets were organized in a set of text files where each file represented an edge set. Before we could use the graphs, we had to make sure all graphs were undirected. Data sets of directed graphs were made undirected by interpreting every edge as undirected and removing any duplicate edges. We also made sure that the nodes were shared by all graphs in that graph set. This was achieved by removing any nodes that did not appear in all graphs for that set. Because as-773 contained a few smaller edge sets, any file of less than 100 Kb in size was removed. That left 694 of the original 773 graphs. See table 1 for the properties of the datasets.

$\mathbb{G}$	graphs	nodes	edges	Source
as-773	694	1 024	2 746	Stanford
as-Caida	122	3 533	22 364	Stanford
Oregon-1	9	10 225	21 630	Stanford
Oregon-2	9	10 524	30 544	Stanford
brock800	4	800	207 661	DIMACS
p_hat700	3	700	121 912	DIMACS
p_hat1000	3	1 000	246 266	DIMACS
p_hat1500	3	1 500	567 042	DIMACS
slashdot	2	77 360	559 592	Stanford
Amazon	4	262 111	2 812 669	Stanford

Table 1: *A brief description of the test data listing the number of graphs in the data set, the number of nodes in each graph and the average number of edges in the graphs.*

3.2 Comparing Resulting Subgraphs

We have compared the resulting DCSs from the different algorithms by different qualities. These are: how close the resulting DCS are to being cliques, their diameter, their triangle density, and their clustering coefficient. We have also compared run time of the various algorithms by

measuring the time they take. We deem these properties sufficient to draw conclusions about the efficiency of the algorithms.

3.3 Tool Kit Used

DCS_GREEDY and DCS_LP are from the paper [6] and DCS_LAG is a Lagrangian Relaxation derived from DCS_LP. We implemented DCS_GREEDY and the other scripts used in the Python programming language. We used the snap interface [28] for graph analysis and the python interface to CPLEX for solving LPs. DCS_GREEDY is single-threaded, and the interior-point method used to solve DCS_LP and DCS_LAG is multi-threaded. The algorithms were run on a laptop with an Intel Core i5-4210U CPU @ 1.70GHz and 8GB RAM.

4 Results and Discussion

In this section, the results from the test runs are presented with an analysis of the data. Arguments for the methods used are also presented and discussed together with other interesting results about the problem.

4.1 Interior-Point vs Simplex

There is no method for solving an LP that is best in every situation [23]. Which method is superior depends largely on the structure of the LP. When testing the simplex method and the interior-point method on our LP it was clear that the interior-point method was much faster. In this section we will give some mathematical explanation for why Simplex is slower, and why interior-point is faster for our LP.

4.1.1 Why Simplex is Ill Suited

As mentioned in section 2.5.1, the simplex method works by traversing the extreme points and in each step finding a new value for the objective function that is at least as good as the previous value. If many of the basic variables are zero in the feasible solution, the new value will often not increase at all; this is because the objective function does not change value when pivoting these rows. This problem is most easily explained using a small graph as an example.

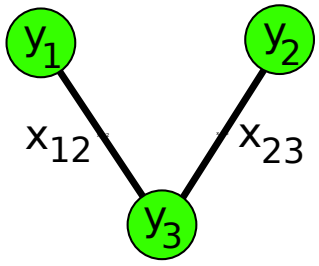


Figure 7: *A very small graph and the corresponding variables for the nodes and the edges that is created by DCS_LP.*

Given the graph shown in Figure 7 we construct the matrix for the densest subgraph problem using DCS_LP formulated in (31). That matrix is given below with the first row representing the names for added clarity and $s_1 - s_5$ are slack variables:

$$\begin{array}{c} \begin{array}{cccccccccccc} z & x_{12} & x_{23} & y_1 & y_2 & y_3 & s_1 & s_2 & s_3 & s_4 & s_5 & b \end{array} \\ \begin{pmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{33}$$

The values of the variables are the standard basic solution with:

$$\begin{aligned} x_{12} = x_{23} = y_1 = y_2 = y_3 = 0 \\ s_1 = s_2 = s_3 = s_4 = 0 \\ s_5 = 1 \end{aligned}$$

The problem is that the basic feasible solution is degenerate. Given an LP with constraints $A\mathbf{x} = \mathbf{b}$ where $\mathbf{b} \in \mathbb{R}^n$, it is degenerate if there exists $b_i = 0$, $0 \leq i \leq n$. That means that at least one of the constraints has a value of 0. When a solution is degenerate, then a pivot will not increase the value of the objective function, because adding a multiple of 0 does not change the value. This means that the algorithm will perform multiple pivots without finding a better result. Due to the structure of constraints in DCS_LP, the simplex method will often get "stuck" on degenerate solutions making the algorithm slow. Therefore, it is clear that the simplex method does not work well for this problem.

4.1.2 Why Interior-Point is Well Suited

As stated in [22] the most computationally heavy segment of the interior-point method is solving for the Newton steps given by equation (24). Since our LP has a very sparse constraint matrix, this multiplication is significantly faster than the worst case $O(n^3)$. For example, when solving oregon-1 the LP has 204 897 rows, 389 353 columns, and out of the approximately $8e10$ elements only 983 592 or roughly 0.001% are nonzero. For our Lagrange relaxation of this LP, there are 788 911 nonzero elements.

A problem with interior-point, however, is that it does not give exact solutions. One way to remedy this is by doing a crossover to simplex and initialize simplex to a basis close to the value obtained from the interior-point solver. The obvious downside to using crossover is that it is more time consuming, especially in this case since the LP is ill-suited for simplex.

A problem-specific approach to recovering an exact solution given a solution from the interior-point solver is rounding. Because the problem is originally an integer problem, it can only take on discrete values. As given by the problem formulation, a node is either picked or not picked. The picked nodes will be given the value $\frac{1}{|S|}$ and all other the value 0. Because of this, we may round any nodes close to $\frac{1}{|S|}$ to $\frac{1}{|S|}$ and round all other nodes down to 0. We implemented this by picking the largest variable value and setting it as max_value. For all variable values greater than $\frac{max\_value}{100}$ we add one to a counter n. These variables are then given the value $\frac{1}{n}$ and the rest of them the value 0.

From practical experience, we observed that the difference between the values of variables of used and unused nodes was very large. We also note that if the LP decides to split nodes, which is explained in 4.7, they receive values much greater than $\frac{1}{|S| \cdot 100}$.

4.2 Lagrangian Relaxation of DCS

In order to formulate an LP that scales well for large graphs we made a Lagrangian relaxation of DCS_LP. The objective function when the constraint (32c):

$$\sum_{ij \in E^m} x_{ij}^m \geq t$$

is moved up can be written as:

$$\begin{aligned} &t + \sum_{m=1}^{M} \lambda_m \left(t - \sum_{ij \in E} x_{ij}^m \right) = \\ &t \left(1 + \sum_{m=1}^{M} \lambda_m \right) - \sum_{m=1}^{M} \sum_{ij \in E} \lambda_m x_{ij}^m \end{aligned} \tag{34}$$

The choice to relax constraint (32c) is motivated by the use of interior-point methods in solving DCS_LP, since relaxing this constraint gives us fewer entries in the constraint matrix which makes it easier to solve the problem using interior-point methods. The complete LP with Lagrangian Relaxation, which we will call DCS_LAG is:

$$\begin{aligned} &\text{DCS\_LAG}(\boldsymbol{\lambda}) = \\ &\text{Maximize} \qquad t \left(1 + \sum_{m=1}^{M} \lambda_m \right) - \sum_{m=1}^{M} \sum_{ij \in E} \lambda_m x_{ij}^m \\ &\text{subject to} \qquad \sum_{i=1}^{n} y_i \leq 1 \\ &\qquad\qquad x_{ij}^m \leq y_i, \; x_{ij}^m \leq y_j \qquad\qquad \forall ij \in E^m \\ &\qquad\qquad x_{ij}^m \geq 0, \; y_i \geq 0 \qquad\qquad \forall ij \in E^m, \; \forall i = 1..n \end{aligned} \tag{35}$$

We did implement a subgradient, but we chose not to use it in the end because it was too slow. If DCS_LAG were allowed to iterate until it found the same solution as DCS_LP, it would take a longer time to find it. With only five iterations the time would be comparable to DCS_LP, but the quality of the solution would be worse than DCS_LP. With only one iteration the time it took would be roughly twice as fast compared to DCS_LP, but the result would be comparable to DCS_GREEDY.

4.3 Presentation of Collected Data

We ran DCS_LP, DCS_LAG and DCS_GREEDY on the data sets shown in table 1. The resulting subgraphs and the running time obtained from the different methods are presented in table 2. All values shown are lower bounds. Properties of the subgraphs found by the algorithms are presented in table 3. An upper limit in running time was chosen at two hours for the tests of the algorithms. This choice was inspired by [6] who chose the same time limit.

	DCS_LP		DCS_LAG		DCS_GREEDY	
$\mathbb{G}$	$\|S\|$	time (s)	$\|S\|$	time (s)	$\|S\|$	time (s)
as-773	40	430	42	120	42	13
as-Caida	56	610	95	50	33	7.6
Oregon-1	76	26	60	6.9	80	1.4
Oregon-2	131	30	160	12.9	160	1.5
brock800	800	56	800	32.8	800	0.8
p_hat700	679	24	679	12.6	679	0.7
p_hat1000	973	58	973	35.2	973	0.7
p_hat1500	1 478	168	1 478	88.2	1 478	1.6
slashdot	3 440	1 700	4 892	1 700	4 892	3.7
Amazon	X	X	X	X	262 111	15.4

Table 2: *Results from running the algorithms DCS_LP, DCS_GREEDY, and DCS_LAG. $|S|$ is the size of the subgraphs found by the respective algorithms. Time is the number of seconds the algorithm takes to complete. X indicates that the algorithm was unable to complete in less than two hours.*

DCS_LP	$\delta_{\mathbb{G}}(S)$	%-clique	τ	d	cc
as-773	6.634*	0.331	0.066	2	0.737
as-Caida	7.714*	0.281	0.040	3	0.509
Oregon-1	12.03	0.320	0.0728	3	0.585
Oregon-2	22.98*	0.353	0.0906	2	0.656
brock800	259.166	0.649	0.273	2	0.649
p_hat700	87.274	0.257	0.021	2	0.301
p_hat1000	122.412	0.252	0.020	2	0.294
p_hat1500	190.053	0.257	0.021	2	0.302
slashdot	27.648	0.017	0.000	5	0.125
Amazon	X	X	X	X	X
DCS_LAG					
as-773	6.547	0.319	0.062	2	0.754
as-Caida	6.116	0.130	0.009	3	0.544
Oregon-1	11.86	0.402	0.105	2	0.649
Oregon-2	22.449	0.308	0.059	3	0.614
brock800	259.166	0.649	0.273	2	0.649
p_hat700	87.151	0.249	0.020	2	0.298
p_hat1000	122.412	0.252	0.020	2	0.294
p_hat1500	190.053	0.257	0.021	2	0.301
slashdot	27.082	0.016	0.000	5	0.108
Amazon	X	X	X	X	X
DCS_GREEDY					
as-773	6.196	0.275	0.045	3	0.744
as-Caida	6.879	0.430	0.3559	2	0.645
Oregon-1	11.85	0.300	0.0716	3	0.605
Oregon-2	22.11	0.306	0.0781	3	0.665
brock800	259.166	0.649	0.273	2	0.649
p_hat700	87.274	0.257	0.021	2	0.301
p_hat1000	122.412	0.252	0.020	2	0.294
p_hat1500	190.053	0.257	0.021	2	0.302
slashdot	15.838	0.006	0.000	4	0.069
Amazon	3.433	0.000	0.000	29	0.420

Table 3: *Properties of the subgraphs given as results from running DCS_LP, DCS_GREEDY, and DCS_LAG.* $\delta_{\mathbb{G}}(S)$ *is the density of the subgraph* S *given by the solution of the respective method. A* $*$ *indicates that the LP does not find an optimal solution.* τ *is the triangle density,* d *is the diameter, and cc is the clustering coefficient of the subgraph. X means no results since the algorithm did not complete.*

The greedy algorithm has, as mentioned in section 2.8, a complexity of $O(n+m)$, i.e. it is linear in the total amount of nodes and edges in the graph set. The LP is polynomial in the total amount of edges in the graph [6]. The greedy algorithm is therefore expected to be much faster. We can see in table 2 that it holds true. The running time becomes a bit slower when the number of graphs becomes large, see as-773 and as-Caida in table 2.

Although the LP is significantly slower than the greedy algorithm, the use of interior-point methods has significantly sped up the LP. It is only for the Amazon data set, which is much bigger than the other sets, that our implementation fails to converge. It fails because our computer

Name	Lower bound	Upper bound	Ratio	Reduced nodes
as-773	6.634	6.646	0.998	2
as-Caida	7.714	7.718	0.999	1
oregon-2	22.98	23.022	0.998	3

Table 4: *Comparison between different graphs that gave non-optimal solutions. The upper bound is given by the objective function of the linear program run with a simplex crossover to yield an optimal solution. The lower bound is from setting all nodes to* $\frac{1}{|S|}$ *by the process described in section* 4.1.2. *Ratio is the ratio* $\frac{Lowerbound}{Upperbound}$. *Reduced nodes is the number of non-zero nodes that received a value below* $\frac{1}{|S|}$.

does not have enough memory to run DCS_LP or DCS_LAG. DCS_LP required around 11 GB of memory to run one of the four graphs that make up the Amazon set.

4.4 Some Interesting Observations

As can be seen in table 3, when run on slashdot, DCS_GREEDY gets a density of 15.838 while DCS_LP finds the optimal value 27.648, or close to it in DCS_LAG's case. This is a notable difference between the solutions compared to the other data sets. This difference could be because of the structure of the graph set as a whole. This is a motivation to using the LP instead of the greedy heuristic algorithm.

An observant reader might have noticed that the subgraph found for the Amazon graphs by DCS_GREEDY is the whole node set. This led us to investigate further, so we ran algorithms to find cliques on the Amazon graphs and the cliques we found had a size in the range of five nodes, which is very small in comparison to the size of the whole graph. We concluded that the graph is very "even" i.e. it does not have dense or sparse areas.

The DCS_LP results we get from oregon-1 in table 3 matches the results presented in [6]. Switching to an interior-point method makes it possible for DCS_LP to solve oregon-2 in a matter of seconds instead of hours as in [6].

Running the algorithms on As-Caida produce some unique results. DCS_LP produces a larger graph with higher $\delta_{\mathbb{G}}(S)$ than DCS_GREEDY. However, DCS_GREEDY produces a tighter graph when looking at all other metrics. This is the only graph where DCS_GREEDY produces a smaller and tighter graph than DCS_LP.

4.5 Quality of LP Solutions

When an algorithm is found to solve a problem efficiently, it is also important to analyze the quality of that solution. To analyze the quality of the solution given by DCS_LP, we look at how close it is to an optimal solution, the running time, and density. These results are also compared to those of DCS_GREEDY and DCS_LAG.

4.5.1 LP and Greedy as Bounds for the Optimal Solution

The objective function of DCS_LP will return an upper bound of the density of the tested graph. Recall however that a solution with all selected nodes — nodes with non-zero values — equal to $\frac{1}{|S|}$ is optimal as shown in section 2.7. Because not all solutions are optimal, it is interesting to see how close to an optimal solution this upper bound is. Of the ten tested graphs only three gave non-optimal solutions, these were as-773, as-Caida and Oregon-2. They are shown in table

Type of Graphs	Optimal	DCS_LP	DCS_GREEDY
Sparse	1.866	1.867	1.859
Sparse with clique	4.000	4.000	4.000
Dense	7.140	7.140	7.140
Dense with clique	7.708	7.708	7.708

Table 5: *LP and greedy solutions compared to an optimal solution. The table shows mean density over 100 runs.*

4. This table clearly shows how close to the optimal solution DCS_LP is. From our results, it is either optimal or within 99.8% of the optimal value.

This differs from the theoretical lower bound of $\frac{t^*\lceil 2t^*+1 \rceil}{|S|}$ by quite a bit. For As-773, as-Caida and oregon-2 the theoretical lower bound is 2.5, 2.3, 8.2 respectively which is around $\frac{1}{3}$ of the actual optimal density. This correlates clearly to the clique % of each of these graphs which the theoretical lower bound is based on.

When analyzing the solution, we also find that very few of the selected nodes have reduced values, i.e. values below the expected $\frac{1}{|S|}$. This may be a contributing factor to the results being so close to the optimal.

Any upper bound found by DCS_LP may be reduced to a lower bound by adding all selected nodes to the set S, setting the value of all nodes in S to $\frac{1}{|S|}$. And finally letting all edges in the induced subgraph of S take the value $\frac{1}{|S|}$. This lower bound is the value used in table 3 as $\delta_{\mathbb{G}}(S)$.

4.5.2 Comparisons Between LP, Greedy and an Optimal Solution

Rather than placing the optimal solution somewhere in an interval, it can be calculated and compared directly to both LP and the greedy algorithm. To make this comparison the optimal solution must be found by exhaustive search. To perform such a search all the possible subgraphs must be tested, which for n nodes is equal to the power set $\mathcal{P}(V)$, which contains 2^n different sets. The number of subgraphs grows exponentially, so this test can only be done for very small problems. In this experiment problems with two graphs of 26 nodes were used.

The graphs were fully connected and contained either a small number (sparse) or a large number (dense) of random edges. The tests were done both with and without a clique of nine nodes common to all graphs. The clique was added to ensure there is actually a densest common subgraph to be found. In the tests with such a clique the LP performed better.

Many graph sets were generated and for each, a solution was found using Linear Programming, the greedy method, and exhaustive search. The results can be seen in table 5. The only test in which a difference between the solutions could be seen was for sparse graphs without clique, in all other tests the difference was negligible.

4.6 The Sifting Algorithm in CPLEX

Sifting exploits LPs where the constraints matrix has a large aspect ratio, in other words the ratio of the number of columns to the number of rows is high. This method works particularly well when most of the variables of the optimal solution are expected to be at its lower bound [29]. In our case most of the variables will be at their lower bound, i.e. 0.

We used the sifting method in CPLEX to see if it was successful on our LP. However, there was a large difference between different running times on similar data sets. For some graphs,

the optimal value was retained within seconds and faster than using interior-point methods. For others, even if the amount of data was lower, it did not converge in two hours. One explanation for this is the squareness of the constraint matrix. We hoped that the relatively low number of non-zeros in the variables would contribute to good running times but even for graphs such as oregon-1 and oregon-2 the running time was sometimes comparable to using simplex. We could not identify a structural difference between the data sets that would cause this massive difference in running time. These factors made us discard this method.

4.7 Example of a non Optimal LP Solution

Jethava and Beerenwinkel proved that DCS_LP presented in section 2.7 is only optimal when $y_i = \frac{1}{|S|} \forall i \in S$ [6]. However, for some graphs, DCS_LP produces a non-optimal solution. To find the kind of graphs that cause non-optimal results we used delta debugging (ddmin). The debugger uses a technique to decrease the size of the input to the LP. When we find a smaller input that still fails the algorithm discards the rest of the input. Therefore, we only find a local minimal example [30]. There exists a graph of eight nodes originating from oregon-2 that gives a non-optimal solution and is shown in figure 8. With such a small example it is clear that the solution provided by DCS_LP is not optimal.

Let $S_1 = \{a,\ b,\ c,\ d\}$ and $S_2 = \{e,\ f,\ g,\ h\}$. The LP's solution to the graphs in figure 8 has assigned $y_i = 1/12$ for $i \in S_1$ and $y_i = 1/6$ for $i \in S_2$. Consequently the subgraph of S_2 will account for most of the density in the solution.

Below is a high-level description of why the LP exhibits this behavior of setting higher values for one set of nodes and lower for others. Consider the bottom graph in figure 8. If this was the only graph S_2 would be the densest subgraph. However the top graph in figure 8 is less dense in this area and S_2 is not the densest subgraph of the top graph. The overall density $\delta_{\mathbb{G}}(S)$ benefits from including parts of S_1. The densest common subgraph of the graphs in figure 8 is $(S_1 \cup S_2) \setminus a$ or $(S_1 \cup S_2) \setminus c$. The LP, however, includes all of S_1 but at half value. This gives the nodes in S_2 value $\frac{1}{6}$ which is higher than the $\frac{1}{7}$ of the optimal solution. The LP, therefore, gives the final result of $\frac{8}{6}$ while the optimal is $\frac{9}{7}$ as it is impossible to choose nodes at half value.

This result led us to believe that the subset of vertices found by DCS_LP will be a superset to the vertices of the optimal subgraph. However, it is possible to construct examples where this is not the case. Let $\delta_{\mathbb{G}}(S)$ denote the common density of the subgraph S. Given a set of graphs $\mathbb{G} = \{G^m = (V, E^m)\}$ and let $S \subset V$ be the subset of vertices found by DCS_LP and t be the value of the objective function. Then given $Z \subset V$ so that $S \cap Z = \emptyset$ it is possible that $t > \delta_{\mathbb{G}}(Z) > \delta_{\mathbb{G}}(S)$. That means that the LP does not find the optimal subgraph nor does it find any node of the optimal subgraph.

4.8 Efficiency of LP when Solving DCS for Many Graphs

The DCS_GREEDY algorithm is faster than all of the LP methods, but there are circumstances where the difference in speed is reduced.

Consider figure 9 which shows the time it takes LP to solve a range of densest common subgraph problems. The first problem in this range has a single graph, and each subsequent problem doubles the number of graphs but halves the size of the graphs, so that the total number of nodes and edges stay the same.

DCS_GREEDY solves all problems in about the same time but the LPs gain speed quickly until the number of graphs is above ten, after which the gains start to slow down. It does not seem that an LP could catch up to the run time of the greedy algorithm if the tests were to

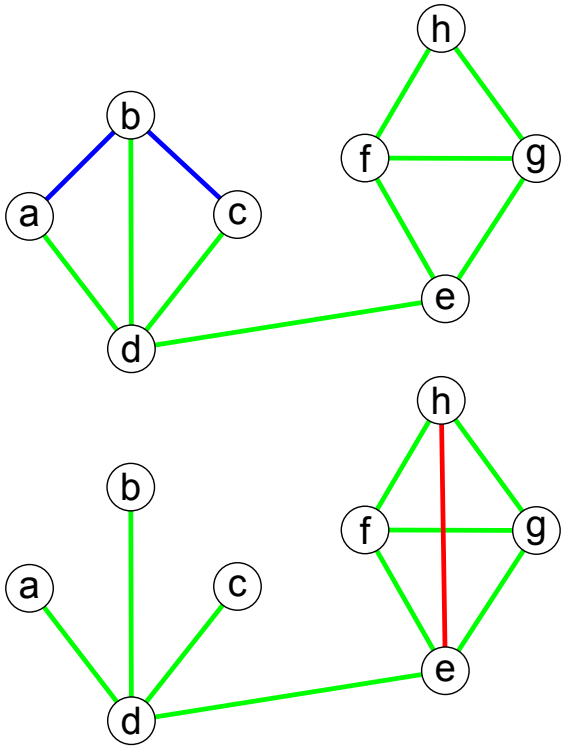


Figure 8: *A small example of when the LP produces sub-optimal solutions. The green edges exists in both graphs. Red and blue edges are unique to their respective graph. The DCS among these graphs is either all nodes except a or all nodes except c with density* $\frac{9}{7}$*. The LP's solution is* $\{a,\ b,\ c,\ d\}$ *as half-nodes, i.e.* $y_i = 1/12,\ i \in \{a,\ b,\ c,\ d\}$ *and* $y_i = 1/6,\ i \in \{e,\ f,\ g,\ h\}$ *the edges in contact with these nodes as half-edges producing a faulty density of* $\frac{8}{6}$*.*

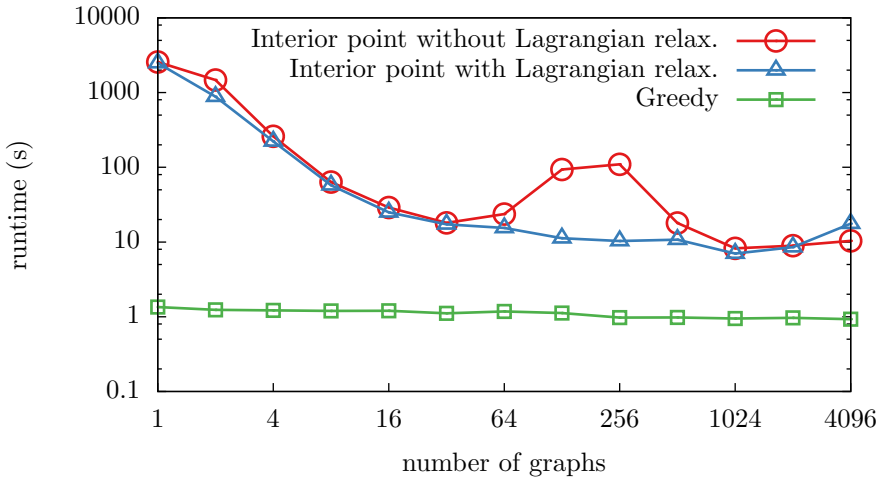


Figure 9: *The runtime of the LP decreases as the number of graphs increase. As the number of graphs increase, the sizes of those graphs are reduced so that the total number of nodes and edges is the same in each run.*

continue with even greater numbers of graphs, but it does show that the more graphs a problem has, the more viable LP becomes as a method for finding the solution.

A peculiarity in the data is the "hump" on the curve representing the LP without Lagrangian relaxation. The hump, which occurs where the number of graphs is around 64–512, was reproduced in several test runs. A possible explanation for it could be that the LPs for these problems have dimensions that are hard to optimize. This is reflected in the constraint matrix which has more non-zero elements in this area.

4.9 Implications of Developing DCS Algorithms

Since the internet is a huge directed graph, sites with similar target groups will probably have links to each other and thus certainly be dense subgraphs. It is clear that algorithms, although for directed graphs, can be used to find these dense subgraphs on the web as well. This could be used for mapping people with certain interests or opinions on forums and social networks. A fact that is very useful for advertisement companies.

This is not something people necessarily want their government, or anyone else to be able to do. This is, however, a problem shared along many academic disciplines. New techniques can be used for malicious purposes by bad people.

5 Conclusions

Interior-point methods are a great way to speed up the running time of the LP. Interior-point methods are very effective when each constraint only considers a few variables, meaning the gradient can be easily calculated. By using interior-point methods we have shown it is possible to find the densest common subgraph in graphs which failed to converge in two hours for [7]. To further improve the running time, a specialized solver could be implemented that uses the special structure of the DCS problem to more efficiently solve the problem. A better relaxation may also be formulated to yield faster convergence.

The solutions of DCS_LP is an upper bound to the optimal density, but it can easily be converted lower bound solutions. Our limited tests on real data show that DCS_LP is within 99.8% of the optimal density. Consequently, the LP produced a very tight upper bound.

Comparing the algorithms for finding the densest common subgraph, it is evident that DCS-_GREEDY is by far the fastest. However using DCS_LP yields a much better solution in terms density measures. DCS_LAG produces worse results than DCS_LP, but it is at least twice as fast. In comparison to DCS_GREEDY the quality of their solutions is roughly equal, but since DCS_GREEDY is so much faster the usefulness of DCS_LAG is limited. If the goal is an optimal solution, DCS_LP is well suited and not unfeasibly slow. If a quick estimate is wanted, DCS-_GREEDY is practically instant. We have not been able to find a use for DCS_LAG in light of DCS_GREEDY's speed and solution quality, but if you are limited to LP for solving the DCS problem, DCS_LAG is of some use.

Owing to the definition of density used, we see that the algorithms prefer larger subgraphs over smaller cliques which are something we expected. An interesting problem would be to use another definition of density in the objective function to find smaller, more clique-like subgraphs.

As mentioned previously, an interesting topic of further research is to construct an algorithm that is guaranteed to find an optimal solution to the DCS problem or to construct a proof of hardness for the problem.

References

[1] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView, Sponsoreed by EMC Corporation*, 2012.

[2] The zettabyte era: Trends and analysis, May 2015.

[3] Aristides Gionis and Charalampos E Tsourakakis. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2313–2314. ACM, 2015.

[4] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[5] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. *Proceedings of APPROX*, 2000.

[6] Vinay Jethava and Niko Beerenwinkel. Finding dense subgraphs in relational graphs. In *Machine Learning and Knowledge Discovery in Databases*, pages 641–654. Springer, 2015.

[7] Samir Khuller and Barna Saha. On finding dense subgraphs. In *Automata, Languages and Programming*, pages 597–608. Springer, 2009.

[8] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. In *Algorithms and Models for the Web-Graph*, pages 25–37. Springer, 2009.

[9] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl 1):i213–i221, 2005.

[10] Mohammed J. Zaki Arlei Silva, Wagner Meira Jr. Mining attribute-structure correlated patterns in large attributed graphs. *Arxiv*, 2012.

[11] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[12] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[13] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 104–112. ACM, 2013.

[14] Leon S. Lasdon. *Optimization theory for large scale systems*. Dover Publications, INC, 1970.

[15] Jiří Matoušek and Bernd Gärtner. *Understanding and Using Linear Programming*. Springer, 2007.

[16] Marshall L Fisher. An applications oriented guide to lagrangian relaxation. *Interfaces*, 15(2):10–21, 1985.

[17] Michael T. Heath. *Scientific computing, An introductory survey, Second edition*. McGraw-Hill, 2005.

[18] AMPL. Ampl streamlined modeling for real optimization. http://ampl.com, May 2016.

[19] Robert Robere. Interior point methods and linear programming. *University of Toronto*, 2012.

[20] https://commons.wikimedia.org/wiki/User:Sdo. Polyhedron of simplex algorithm in 3d.

[21] Anders Forsgren, Philip E Gill, and Margaret H Wright. Interior methods for nonlinear optimization. *SIAM review*, 44(4):525–597, 2002.

[22] Masakazu Kojima, Shinji Mizuno, and Akiko Yoshise. *A primal-dual interior point algorithm for linear programming*. Springer, 1989.

[23] Margaret Wright. The interior-point revolution in optimization: history, recent developments, and lasting consequences. *Bulletin of the American mathematical society*, 42(1):39–56, 2005.

[24] Abraham Charnes and William W Cooper. Programming with linear fractional functionals. *Naval Research logistics quarterly*, 9(3-4):181–186, 1962.

[25] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, February 2016.

[26] M. Brockington, P. Soriano, and M. Gendreau. Dimacs clique benchmarks. ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/, 1993.

[27] IBM Corp. Ibm ilog cplex optimization studio, version 12.6.3. http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud, February 2016.

[28] Jure Leskovec and Rok Sosič. Snap.py: SNAP for Python, a general purpose network analysis and graph mining tool in Python. http://snap.stanford.edu/snappy, June 2014.

[29] IBM Corp. Sifting optimizer. http://www.ibm.com/support/knowledgecenter/api/content/nl/en-us/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/UsrMan/topics/cont_optim/simplex/10_sifting.html, May 2016.

[30] Andreas Zeller, Member, IEEE Computer Society and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 28, NO. 2, FEBRUARY*, 2002.