CSCI-4968: ML and Optimization Spring 2023

Semidefinite Programming

For approximating Max Cut and Travelling Salesman

1 Motivation

Linear programming is a common constrained optimization problems with uses in fields math, computer science, economics, and buisness. The power and ubiquity of linear programming comes from the fact that it can solve a wide host of linear constrained optimization problems. Not only can solutions can be found, but they can also be found quite quickly. Semidefinite programming (SDP) expands on the ideas used within linear programming and generalizes them so that they can be applied to even more cases.

Instead of being able to control single variables, we are able to select for an entire matrix, with whole matricies of constraints. SDPs are widely used for combinatorial optimization - a class of problems that are abundant of network science. The notorious NP-Hard problems can be approximated quite well with SDPs. The flexibility and speed of finding solutions of what can be represented as a SDP problem is what gives it its power.

2 Semidefinite Programming

2.1 Recap of Linear Programming

To give background on semi-definite programming, we begin with a brief recap of linear programming. Suppose that you have control over a set of variables and you are attempting to find a selection for each of these variables such that some linear combination is either maximized or minimized. To give background on semi-definite programming we first start with a brief recap of linear programming. To make this more complicated suppose that these variables each have a constraint that must be met.

Formally we can write this out as the following. Suppose that we have n variables that we have control over which we can represent as:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

This problem is made trivial by setting $\vec{x}$ to be arbitrarily large or small value depending on the optimization direction. The solution becomes more meaningful if the domain of each variable is constrainted.

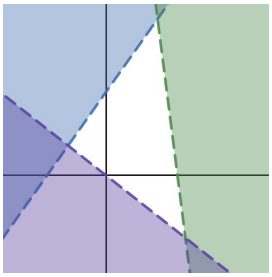
Say for each of these n variables has an associated coefficient $\vec{c} = \begin{bmatrix} c_1, c_2, ..., c_n \end{bmatrix}^\top$. We would like to find $\vec{x}$ such that the linear combinatrion of the two $(\vec{c} \cdot \vec{x})$ is minimized

$$\min \quad \vec{c} \cdot \vec{x}$$

We can extend this idea to a system of equations. We define $\vec{a}_i$ and $\vec{b}$ to vectors in $\mathbb{R}^n$. We would like to find $\vec{x}$ such that

$$\begin{aligned} \vec{a}_1 \cdot \vec{x} &= b_1 \\ \vec{a}_2 \cdot \vec{x} &= b_2 \\ &\vdots \\ \vec{a}_n \cdot \vec{x} &= b_n \end{aligned}$$

Graphically, each constraint creates a *half-space*. The intersection of the halfspaces (the white space) is our solution set. Our goal is the find the minimize value of the intersections.



We can gather our $\vec{a}_i$ vectors into a single matrix. Let $\mathbf{A} = [\vec{a_1}^\top \cdots \vec{a_n}^\top]$. so that we only need to solve:

$$\mathbf{A}\vec{x} = \vec{b}$$

In summary, a linear programming asks for an optimal solution, $\vec{x}$, given a set of linear constraints

$$\begin{array}{ll} \text{minimize} & \vec{c} \cdot \vec{x} \\ \text{subject to} & \mathbf{A}\vec{x} = \vec{b} \\ & \vec{x} \geq 0 \end{array}$$

It is needed that $\vec{x}$ is nonnegative ($\vec{x} \geq 0$). We note that our constraints form a convex set. Thus, linear programming is a subset of a convex optimization problem. Finding a solution to an instance of linear programming can be done with a variety of algorithms including the *simplex algorithm*.

Many real world problems can be modelled as a set of linear contraints. One use case arises in manufacturing. Given a set of products, how much of each kind should be produced to either minimize cost or maximize profits? Linear programming is a general approach of taking a constrained linear optimization problem into a form where a solution can easily be found.

2.2 Formalization of Semidefinite Programming

Instead of having just n variables that we have a matrix $\mathbf{X}$ of n^2 elements. Thus instead of finding an optimal $\vec{x}$ we are trying to find an optimal $\mathbf{X} \in \mathbb{R}^{n \times n}$ that is symmetric positive semidefinite (SPSD) such that

$$\min = \sum_{i=1}^{n}\sum_{j=1}^{n} x_{i,j}c_{i,j}$$
$$\mathbf{X} \succeq 0$$

Where $c_{i,j}$ is the coefficient corresponding to the $x_{i,j}$ variable and $\mathbf{X} \succeq 0$ ensures that $\mathbf{X}$ is SPSD. We can gather all n^2 coefficients into a matrix $\mathbf{C}$ and then rewrite our minimization term as:

$$\min \mathbf{C} \bullet \mathbf{X}$$

Where the $\bullet$ operator is simply taking the sum of all elements after doing element wise multiplication on two matricies $\mathbf{C}$ and $\mathbf{X}$ with the same shape. Formally we can write it as:

$$\mathbf{C} \bullet \mathbf{X} = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{C}_{i,j}\mathbf{X}_{i,j}$$

Of important note is the fact that $\mathbf{C} \bullet \mathbf{X} = \text{trace}(\mathbf{C}^\top \mathbf{X})$ where trace is the sum of the diagonal terms for a matrix. Calculating the trace of the matrix multiplication is not as efficient as the $\bullet$ operator but the notation comes up quite frequently within the literature so it is important to know.

Returning back to our formalization of SDP, we note that with more variables comes with more constraints. Suppose that we have n equations to satisfy. For each equation we have a matrix $\mathbf{A}_i$ and a scalar value b_i defining our constraint. Then for a given constraint we have:

$$\mathbf{A}_i \bullet \mathbf{X} = b_i$$

This is all that is needed for SDP. We can write this out formally as:

$$\begin{aligned} &\min \mathbf{C} \bullet \mathbf{X} \\ &\mathbf{A}_i \bullet \mathbf{X} = b_i \text{ for } i = 1, 2, ..., n \\ &\mathbf{X} \succeq 0 \end{aligned}$$

The parallels between semidefinite programming and linear programming should be quite obvious. In many ways, semidefinite programming is linear programming with more variables of control and more constraints.

2.3 Semidefinite Programming Running Time

The only reason why SDP would be useful at all in the first place is if it was able to be solved in polynomial time. There are a variety of algorithms that can solve SDPs, one such algorithm is Alizadeh's interior point method which runs in:

$$\tilde{O}(n^{3.5})$$

3 Travelling Salesman Relaxation

Imagine you are prospective student touring RPI. You have a list of buildings you want to visit. You have on hand the distance between each pair of buildings. Is it possible to find a path such that each buidling is visited exactly once? If so, what is the shortest possible distance you will need to travel. Since you toruing the college, it is likely you have some method of transportation to came from and need to come back to . We have the added constraint that you start at the parking lot and will need to come back to the parking lot. This is know as the travelling salesman. As our goal is to find an *efficient* route, we will be satisfied with an approximation. Or a series of buidlings to visit that is close the optimal.

3.1 Relaxation For Traveling Salesman

Let $C \in \mathbb{R}^{n \times n}$ denote the matrix of edge costs. Let J denote the all-ones matrix, and e denote the all-ones vector.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\text{trace}(CX) \\ \text{subject to} \quad & Xe = 2e \\ & X_{ii} = 0, \quad i = 1, \ldots, n \\ & 0 \leq X_{ij} \leq 1, \quad i, j = 1, \ldots, n \\ & 2I - X + (2 - 2\cos\left(\frac{2\pi}{n}\right))(J - I) \succeq 0 \\ & X \text{ is a real, symmetric } n \times n \text{ matrix.} \end{aligned}$$

X is a fractional adjacency matrix, meaning for $e = \{i, j\}$, $x_{ij} = x_{ji}$ is the proportion of edge e used.

3.1.1 Explanation For Each Constraint

1. $Xe = 2e$ means the sum of each row in the adjacency matrix should equal to 2. This is trivial as we have an origin and destination as 1 suggests.
2. $X_{ii} = 0$ is trivial as the distance from a city to itself is always 0.
3. $0 \leq X_{ij} \leq 1, \quad i, j = 1, \ldots, n$ is a relaxation that does not enforce an exact solution. i.e, we can have a fractional value for each edge.
4. $2I - X + (2 - 2\cos\left(\frac{2\pi}{n}\right))(J - I) \succeq 0$ is the tricky part of the formulation. An understanding is not required but an explanation is given below.

The constraint $2I - X + (2 - 2\cos\left(\frac{2\pi}{n}\right))(J - I) \succeq 0$ is related to the idea of graph spectrum, Laplacian of Hamiltonian cycle, and algebraic connectivity. Here are some facts needed in order to understand the meaning of the constraint:

1. Let $G_C = (V, E, C)$ denote a C-edge-weighted graph with nonegative edge values, Laplacian $L(G_C)$ is defined as $L(G_C) = diag(r_1, ..., r_n) - C$, where r_i is the sum of the i-th row of C.

2. $L(G_C)$ is symmetric, positive semidefinite with the smallest eigenvalue $\lambda_1 = 0$ and the corresponding eigenvector e.

3. The algebraic connectivity $\alpha(G_C)$ is the second smallest eigenvalue of $L(G_C)$. There are two main properties: (1).$\alpha(G_C) = \min_{x \in S} x^T L(G_C) x$ (2).$\alpha(G_C) \geq 0$, and $\alpha(G_C) > 0$ iff G_C is connected.

4. Algebraic connectivity of a Hamiltonian circuit is well known in the theory of graph spectra. The Laplacian of a circuit with n vertices has the spectrum

$$2 - 2cos(2\pi j/n), j = 1, ..., n$$

and the second smallest eigenvalue is obtained for $j = 1$ and $j = n - 1$, i.e. $\lambda_2 = \lambda_3 = 2 - 2cos(2\pi/n)$. This value will be denoted by h_n, i.e. $h_n = 2 - 2cos(2\pi/n)$.

The constraint is making sure weighted graph corresponding to X (as a weighted adjacency matrix) is at least as connected as a cycle graph, with respect to algebraic connectivity. $2I - x$ gives us the Laplacian matrix corresponding to the weighted graph according to entries of x. J makes sure we ignore trivial eigenvalues of Laplacian $\lambda = 0$. $2 - 2cos(2\pi/n)$ makes sure that the non-trivial eigenvalues is greater than or equal to the second smallest eigenvalue of a connected Hamiltonian Cycle.

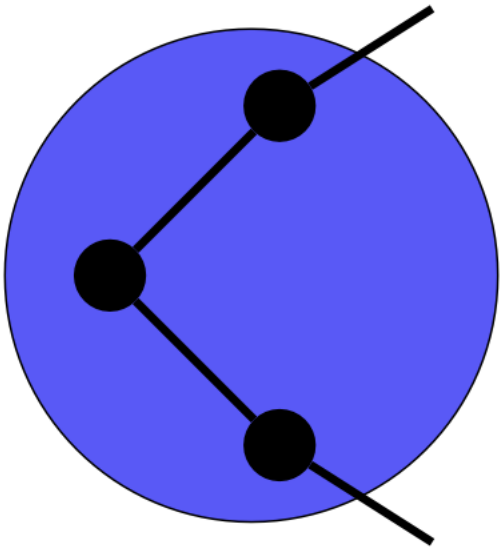


Figure 1: Caption: A node in the Hamiltonian Cycle

3.2 Performance Guarantees

The integrality gap is unbound and the time complexity is polynomial. We will prove it empirically in the Experimental Results.

4 Experimental Results

5 Integrality Gap And Running Time

Table 1: SDP/Brute Force Methods Comparison

# Of Nodes	SDP Time	BF Time	SDP Objective Value	BF Objective Value	Integrity Gap	Time Ratio
10	0.7101	0.0156	53224.4854	53228.3976	0.9999	45.519
15	0.6776	0.8224	65753.5934	67299.5625	0.9770	0.8239
20	1.2271	97.2059	69558.9865	76199.4928	0.9129	0.0126
21	1.3689	266.7778	73969.6527	77373.6362	0.9560	0.0051
22	5.4774	657.7847	66459.7265	68245.9576	0.9738	0.0083

1. BF denotes the brute force method and was implemented using dynamic programming
2. Integrity Gap is the ratio between SDP Objective Value and BF Objective Value

5.1 Visualization

To find the Hamiltonian cycle, we set the edge that has value greater than or equal to the $3^r d$ largest value in that row to 1 and else 0 to find the adjacency matrix. Then we use the adjacency matrix and coordinates of points to plot the results.

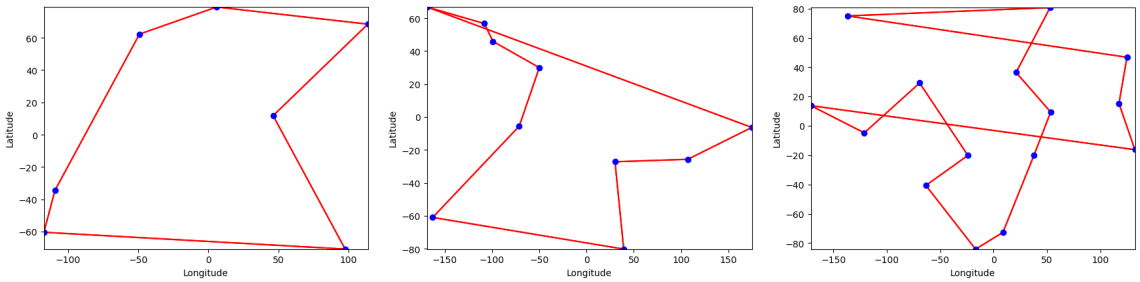


Figure 2: Sets of the Problems that SDP Relaxation is able to find a correct Solution

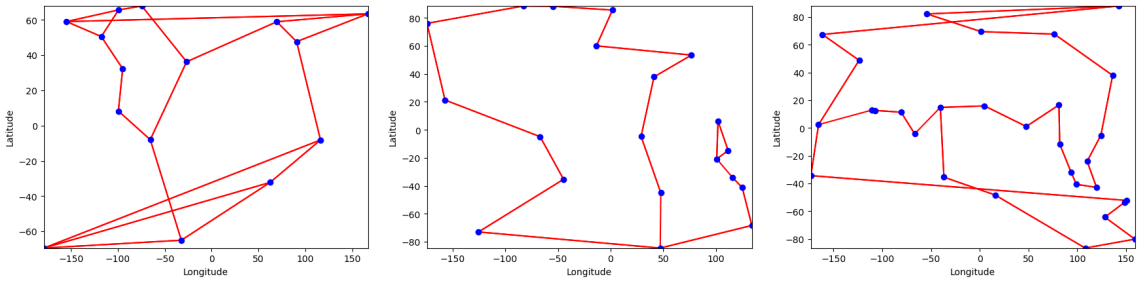


Figure 3: Sets of the Problems that SDP Relaxation is unable to find a correct Solution

6 Discussion

- can we prove the runtime - can we prove how good of a lower bound we have? - what are some easy things to prove?

6.1 Scope and limitations

7 Appendix

7.1 Problem Set

We will now give one application of semidefinite programming in approximating an NP-Hard problem. We will first formalize the SDP and then use to find the approximation.

Consider the max-cut problem where for graph G the goal is to split the vertices into two disjoint sets S and $\overline{S}$ such that the sum of the edges between the two sets is maximum. Thus we know that $S \cup \overline{S} = V(G)$ and $e(i,j)$ is the weight of the edge between vertex v_i and vertex v_j. Thus our optimization problem is:

$$\max \sum \{e(i,j) : v_i \in S \text{ and } v_j \in \overline{S}\}$$

We will collect all our edge weights into matrix $\mathbf{W}$ where $w_{i,j}$ is the edge weight between vertex v_i and v_j.

We will work our way towards a formalization of SDP.

7.1.1 Question 1.

For all of our vertices we have a corresponding variable x_i such that:

$$x_i \in \{-1, 1\}$$
$$\text{if } x_i \in S \text{ then } x_i = 1$$
$$\text{if } x_i \in \overline{S} \text{ then } x_i = -1$$

Then for two vertices $v_i, v_j \in V(G)$ what is the inuition behind the following equation:

$$w_{i,j}(1 - x_i x_j)$$

Answer: If v_i and v_j are in the same set the above equation will be zero otherwise $w_{i,j}$ will be returned. The weight will only get returned if the edge between v_i and v_j is a cut edge for the sets S and $\overline{S}$.

7.1.2 Question 2.

How can we turn $x_i x_j$ from above into a matrix given that $\vec{x} = [x_1, x_2, \ldots, x_n]$. This matrix should be square with the number of rows equal to the number of vertices in our graph. $\mathbf{X}_{i,j} = x_i x_j$: Answer:$\mathbf{X} = \vec{x}\vec{x}^\top$

7.1.3 Question 3.

What can you say about the values on the diagonal of $\mathbf{X}$? Is $\mathbf{X}$ symmetric?

7.1.4 Question 4.

Before we found that our optimization problem was:

$$\max \sum \{e(i,j) : v_i \in S \text{ and } v_j \in \overline{S}\}$$

And in question 1. we found that for a single vertex pair we have:

$$w_{i,j}(1 - x_i x_j)$$

How can we rewrite our optimization problem to account for all vertex pairs using $\mathbf{X}$ and $\mathbf{W}$ and the $\bullet$ operator. Ensure that in your solution that no edge is counted twice:

Answer:

$$
\begin{aligned}
&= \max \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}[w_{i,j}(1 - x_i x_j)]\right) \\
&= \max \sum_{i=1}^{n}\sum_{j=1}^{n}[w_{i,j} - w_{i,j}x_i x_j] \\
&= \max \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}[w_{i,j}] - \mathbf{W} \bullet \mathbf{X}\right)
\end{aligned}
$$

The $\frac{1}{2}$ comes from the fact that $\mathbf{X}$ and $\mathbf{W}$ are symmetric and thus each cut edge is counted twice.

7.1.5 Question 5.

We have formalized the value that we want to maximize in the previous question. One of our constraints for SDP is that $\mathbf{X}$ has to SPSD. Thus we know that $\mathbf{X} \succeq 0$. Currently our constraints are that for $x_{i,j} \in \mathbf{X}$ that $x_{i,j} \in \{-1, 1\}$. Such a hard constraint is more specific than SDP can solve. We can relax our problem such that $\mathbf{X} \succeq 0$ and $x_{i,i} = 1$ and dropping our previous $x_{i,j} \in \{-1, 1\}$. What is the inuition behind having $x_{i,i} = 1$?

Answer: $x_{i,i} = 1$ ensures that vertex v_i is only in one set. This is represented by $x_i x_i = 1$.

Formally write out the SDP problem:

$$
\begin{aligned}
&\max \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}[w_{i,j}] - \mathbf{W} \bullet \mathbf{X}\right) \\
&\mathbf{X}_{i,i} = 1 \\
&\mathbf{X} \succeq 0
\end{aligned}
$$

7.1.6 Question 6.

We have now formalized an SDP problem for an approximation of the max-cut problem. Solving this will give us a lower bound for the max-cut problem. However we can get an approximate solution. In python generate n random 2-dimensional points. Find the distance between each pair of points. Each point will be a vertex in our graph and the pairwise distances between points will be our edges. scipy.spatial.distance.cdist is useful for this

Answer:

```
n = 20
points = np.random.rand(n, 2)
adj_matrix = cdist(points, points)
W = adj_matrix
```

Using the cvxpy example as a reference (https://www.cvxpy.org/examples/basic/sdp.html) setup the SDP problem and solve it. Your solution should look something like the following (here we set $n = 5$):

```
>>> print(X.value)
[[ 1.          1.00000039 -1.00000052 -1.00000054  1.00000065]
 [ 1.00000039  1.         -1.00000055 -1.00000057  1.00000072]
 [-1.00000052 -1.00000055  1.          1.00000033 -1.00000081]
 [-1.00000054 -1.00000057  1.00000033  1.         -1.00000083]
 [ 1.00000065  1.00000072 -1.00000081 -1.00000083  1.        ]]
```

What is the lower bound for the max-cut problem?

7.1.7 Question 7.

For a value $x_{i,j}$ if $x_{i,j} < 0$ then v_i and v_j are in different sets. Determine these two sets, plot the points of these two sets using two different colors and show the result:

7.1.8 Question 8.

What is the value of the max-cut for the two sets that you found?

7.1.9 Extra Credit

- The SDP problem that we formulated was an approximation for the max-cut problem - not the optimal solution. If we were able to set $x_{i,j} \in \{-1, 1\}$ then we would have an optimal solution. We will try and see how well the SDP solver did:
- Create a brute force max-cut solver and compare the results to what the SDP found.
- Note that such a brute force solver will have a $O(2^n)$ runtime so be careful for large values of n.
- Using a simulation approach, how well of a bound does the SDP problem set?
- In question 7. we got a potential max-cut. How well does this compare to the actual optimal max-cut?

7.2 References

- https://users.math.msu.edu/users/iwenmark/Teaching/MTH995/Papers/SDP_notes_Marina_Epelman_UM.pdf
- https://www.uit.edu.mm/storage/2020/09/WWM-2.pdf
- https://www.cs.princeton.edu/ arora/pubs/mw-focs.pdf
- https://dl.acm.org/doi/pdf/10.1145/2184319.2184343
- https://www.cs.princeton.edu/ arora/pubs/mw-focs.pdf