18.337 Project - Parallelized Kolter Mixing

Jonathan Edelman

May 17, 2023

1 Introduction

The maximum cut (MAXCUT) problem consists of finding a partition of the nodes of a graph $G = (V, E)$ into two disjoint sets V_1 and V_2 ($V_1 \cap V_2 =, V_1 \cup V_2 = V$), in such a way to maximize the number of edges that have one endpoint in V_1 and the other in V_2 [3]. The MAXCUT problem is useful in VLSI (Very large scale integration) design, so it is useful to be able to solve fast. The decision version of MAXCUT is known to be NP-complete. We can rewrite the MAXCUT problem as the following optimization problem:

$$\max_{y_i \in \{-1,1\}} \frac{1}{4} \sum_{i,j} w_{i,j}(1 - y_i y_j)$$

where $w_{i,j}$ is the weight of the edge between nodes i and j (and 0 if there is no such edge). While this problem is NP-complete, we can relax the constraint that the matrix Y be rank 1, and instead consider the well-known SDP relaxation of MAXCUT:

$$\min_{Y \succeq 0 \in \mathbb{R}^{n \times n}} < W, Y > \text{ s.t. } Y_{ii} = 1, i = 1, ..., n \tag{1}$$

SDP solves can solve problems like this in polynomial time, but they do not scale well with the size of the problem. [1]

The inefficiency of these solvers stems from the fact that Lagrangian convex solvers rely on a finely tuned step size and penalty parameter. A method was presented that removes the need for this slow gradient descent known as Kolter Mixing. This low-rank approximation to the SDP problem relaxation in order to speed up the algorithm is orders of magnitude faster than more common convex solvers [4]. This algorithm was found to outperform semi definite programming solvers and gradient descent solvers on randomized MAXCUT instances [2]. In this paper, we compare the speeds and accuracy of a standard SDP solver on the MAXCUT relaxation, the a randomized mixing method, as well as the mixing method implemented using parallelism. All are implemented in Julia.

2 Background

Here we present the Kolter Mixing method as described in [4]. Specifically, this is a method to solve problems of the form:

$$\min_{X \succeq 0 \in \mathbb{R}^{n \times n}} < C, X > \text{ s.t. } X_{ii} = 1, i = 1, ..., n \tag{2}$$

i.e. an SDP with n equality constraints. This is the MAXCUT SDP, which corresponds to the SDP relaxation of maxcut where C is the adjacency matrix of a graph. In factorized form, $X = V^T V$, we have the following non-convex problem:

$$\min_{V \in \mathbb{R}^{k \times n}} < C, V^T V > \text{ s.t. } ||V_i||_2 = 1, i = 1, ..., n \tag{3}$$

where $v_i \in \mathbb{R}^k$ is the ith column of V.

If we hold all but one v_i fixed, there is a closed form expression for the remaining v_i given by:

$$v_i := \frac{-\sum_{j \neq i} c_i j v_j}{|| - \sum_{j \neq i} c_i j v_j ||_2} \tag{4}$$

The Kolter Mixing method provides a simple method solution to this type of non-convex problem. Iterate through the vectors and apply (3) until convergence. This does not rely on a step size or a well-tuned parameter, and is thus much faster than Lagrangian solvers. As long as $k > \sqrt{2n}$, this method will converge to the global optimum of the original SDP.

While the original literature iterates over the the vectors until convergence, we found that picking a random vector at each "step" of mixing parallelized better, so that method was chosen instead.

3 Experiment

All experiments were run on JuliaHub with the following settings: V100 GPU, 8 vCPUs, 61GB Memory.

A random dense 200 x 200 adjacency matrix with all edges of weight 1 was used. This matrix was forced to have 0s on the diagonal.

The SDP solver was implemented with JuMP and solved with Julia's SCS package.

See appendeix for implementations.

See 4 for results.

4 Results

We present the results of the different experiments as follows:

Name	Timing (s)	Optimal Cost
SCS	8.677958	-2598.494864065229
Serial Mixing	3.351775	-2598.5266944198256
Parallel Mixing	1.464107	-2598.2425151176885

5 Interpretation of results

randserial and randparallel were designed to run for a fixed number of iterations, not until convergence is achieved. If they had been, they would both take longer and be more accurate. Speed came at the expense of accuracy, and as both are randomized algorithms, they both have the chance of failing (albeit this is unlikely). Running them multiple times leads to both different timings and difference optimum values. Cost vs iteration is plotted in 7 (generated from the serial mixing), and it shows that the cost asymptotically approaches the optimal cost given from the SCS method. Further experimentation could be used to determine how many iterations are truly necessary, as this would also speed up the algorithm.

6 Conclusion

Being able to solve these SDP relaxations of MAXCUT is useful as any solution to the relaxed version of MAXCUT is also an upper bound on the true solution of MAXCUT (since any solution to the SDP relaxation is also a solution to MAXCUT itself). The Branch and Bound method is an algorithm that can be used for solving MAXCUT, by breaking down each problem into smaller sub-problems and then using a bounding function to eliminate the sub-problems that cannot contain the optimal solution. The MAXCUT SDP relaxation acts as one such bounding function. Because solving the SDP needs to be run many times, its speed is often the limiting step for solving MAXCUT problems. Using the mixing method sacrifices accuracy, but the parallelized version is more than five times faster than the conventional solver. Also, if the MAXCUT problem is trying to be solved with integer edge weights, the difference between the optimal values is not large enough (since they have the same floor and ceiling) to affect the branch and bound algorithm.

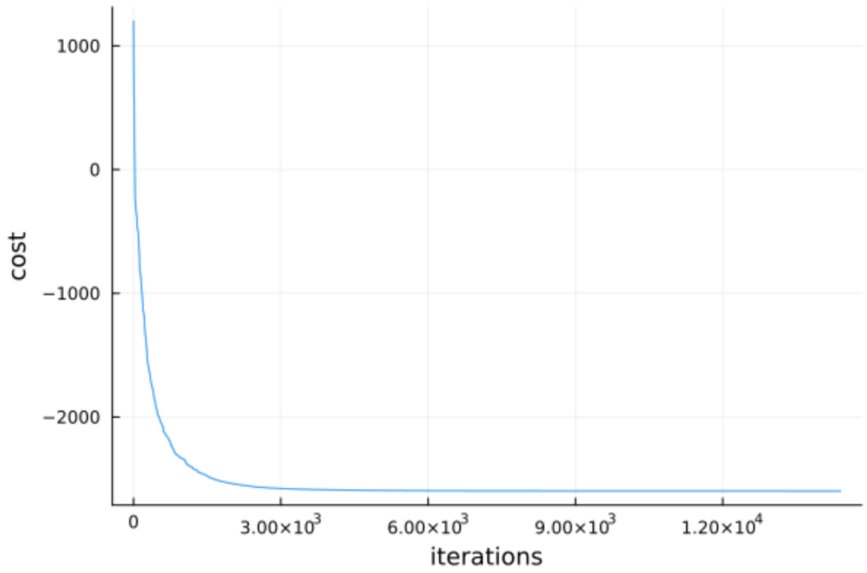
Further work should be done to implement these methods into a branch and bound solver of a general MAXCUT problem. We hypothesize parallelization can also be used to speed up the branch and bound method, by considering multiple branches at once.

7 Appendix

```
function JuMPSolve()
        model = Model(SCS.Optimizer)
@variable(model, X[1:n,1:n], PSD)
for i in 1:n
@constraint(model, X[i,i] == 1)
end
@constraint(model, X >= 0, PSDCone())
@objective(model, Min, sum(C.*X))
optimize!(model)
return value(X)
 end

begin
    function randserial(n,k,C)
    Costs = []
    V=rand(k,n)
    counter=0
    Vs = []
    for i in 1:n
        V[:,i] = mix(i,V,C)
    end
    for j=1:n*k*8
            i=rand(1:n)
            V[:,i] = mix(i,V,C)
        push!(Costs,tr(V* C* V'))
        push!(Vs, V'*V)
    end
    Vs
    Costs
    print(Costs[end])
    plot(Costs)
    end
function randparallel(n,k,C)
    Costs = []
    V=rand(k,n)
    counter=0
    Vs = []
    for i in 1:n
        V[:,i] = mix(i,V,C)
    end
```

```
    Threads.@threads for j=1: n*k*8
                i=rand(1:n)
                V[:,i] = mix(i,V,C)
        push!(Costs,tr(V* C* V'))
        push!(Vs, V'*V)
    end
    Vs
    Costs
    print(Costs[end])
    try
    finally
    end
    end
    function mix(i,V,C)
ans=-(V*C[i,:])
ans/=norm(ans)
return ans
    end
 end
```



References

[1] Dominic Eelkema. Convergence of the mixing method: An iterative algorithm for solving diagonally constrained semidefinite programs.

[2] Murat A. Erdogdu. Convergence rate of block-coordinate maximization burer–monteiro method for solving large sdps. *Mathematical Programming*, 195:243–281, 2022.

[3] Pablo Parillo. Mit 6.7230 - algebraic techniques and semidefinite optimization lecture 3. 2023.

[4] J. Zico Kolter Po-Wei Wang. Low-rank semidefinite programming for the max2sat problem.