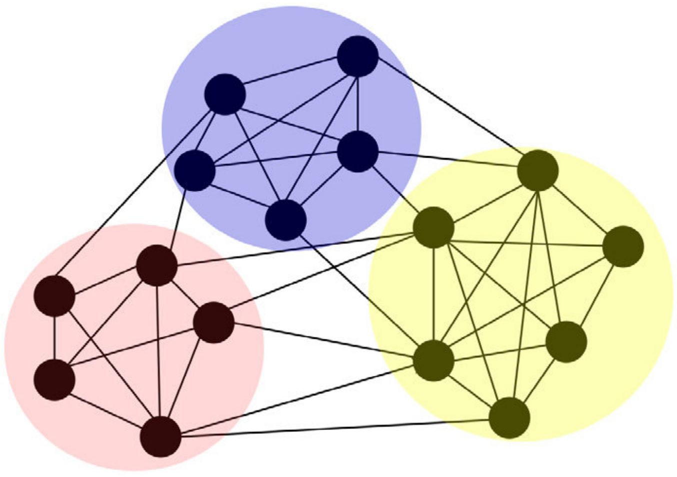
IT3052E : INTRODUCTION TO OPTIMIZATION

Graph Partitioning Problem



Team:

Huynh Sang	20214930
Nguyen The An	20210006
Vu Tung Linh	20210532

Supervisor:
Dr. Bui Quoc Trung

Semester 2022.1
Feb, 2023

Contents

Abstract

The graph partitioning problem has long been a concerned problem in optimization. It has many applications in the field of computer networks, social networking, traffic control, VLSI designs, etc. For instance, in computer networks, partitioning can be used to divide a large network into smaller sub-networks, making it easier to manage and maintain. Similarly, in VLSI design, partitioning can be used to divide a complex circuit into smaller parts, improving the design and manufacturing process.

Graph partitioning is a challenging problem that requires balancing many factors, including computational efficiency, graph structure, and application-specific requirements. As such, the development of efficient and effective graph partitioning algorithms remains an active area of research.

1 Introduction

Graph partitioning is known to be a challenging problem, and finding an optimal partition is often NP-hard. As such, various heuristics and approximation algorithms have been developed to find good quality partitions in reasonable time.

1.1 Problem Statement

The formal statement of the problem:

Given an undirected graph $G = (V, E)$ and a weight function w over the set E. Find a partition $P = (P_1, P_2, ..., P_k)$, where each partition is approximately of equal size, such that the sum of the weights of edges that connects two different partitions is minimized. That is, we have to minimize $\sum_{\{u,v\}\in E} w(\{u, v\})$.

The constraint on the size of a partition might be one of the following:

- The difference in size between any partition is less than α, or
- All partitions must be of size less than $(1 + \epsilon)\lceil|V|/k\rceil$

1.2 Input

In our source codes, all of our solvers takes input of the form (adjacency, nVertices, nEdges, k, epsilon/alpha, timelimit). A solver may utilise all of those parameters, or may ignore some of them, e.g. ignore time limit or epsilon/alpha constraint.

We are going to present several algorithms to two versions of this problem, from exact algorithms to heuristics.

2 Proposed Algorithms

2.1 Exhaustive Search

We implemented exhaustive search in the form of backtracking. First, it is crucial to note that there exists a bijection between the set of all partitions of a set and this set

$$\{(k_1, k_2, ..., k_n) \in \mathbb{Z}^n | \forall i \in \{2, ..., n\} : k_1 \leq k_i \leq \max_{0 \leq j < i} k_j + 1\}$$

Intuitively, an element would either goes into an existing set, or be the first element of a new set. For instance, the partition $\{\{1, 3\}, \{2, 4\}, \{5\}\}$ correspond to the vector $(0, 1, 0, 1, 2)$.

With that in mind, we can systematically generate all k-partitions:

```
def Generate(i=0, maxprev=0):
    if i == 0:
        partition = initialize()
        partition[0] = 0
        Generate(1, 0)
    if i == n:
        return partition
    else:
        if maxprev == k - 1:
            for j in range(k):
                partition[i] = j
                Generate(i+1, maxprev)
        elif i - maxprev + k == n + 1:
            partition[i] = maxprev + 1
            Generate(i+1, maxprev + 1)
        else:
            for j in range(max + 2):
                partition[i] = j
                Generate(i+1, max(maxprev, j))
```

The above code generates all k-partition of a set (by dictionary order). From this, we could easily add the objective, i.e. the sum of cut edges, and even implement branch and bound to further optimize the algorithm.

2.2 Integer Programming

We formulate the Integer Programming model for this problem:

$$\begin{aligned} \forall \{u, v\} \in E : & e_{uv} \geq x_{u,p} - x_{v,p} \\ \forall \{u, v\} \in E : & e_{uv} \geq x_{v,p} - x_{u,p} \\ \forall p : & \sum_{v \in V} x_{v,p} \leq V_{max} \\ \forall v \in V : & \sum_{p} x_{v,p} = 1 \end{aligned}$$

Where,

- $e_{uv} \in \{0, 1\}$ denotes whether the edge $\{u, v\} \in E$ is in the cut
- $x_{v,p} \in \{0, 1\}$ denotes whether the vertex v is in the subset p

- The first two constraints ensure the relationship between the cut edges and the partition
- The next constraint ensures the size of each partition does not exceed the limit
- The last constraint ensures each vertex is assigned into exactly one partition.

We plugged the model into Google's OR-Tools Constraint Programming Solver and SCIP Integer Programming Solver.

2.3 Greedy Algorithm

Our greedy algorithm based on a simple motive, which is for each partition P_i, we find a vertex to insert into it. The base of such algorithm is below:

```
def GenericGreedy(graph, V, k):
    Initialize P = (P_0, P_1, ..., P_{k-1})
    V' = V
    for i in range(k):
        u = random vertex from V'
        P[i].add(u)
        V'.remove(u)
    i = 0
    while V is not empty:
        u = GreedyFunction(V', P, i, graph)
        P[i].add(u)
        V'.remove(u)
        p = (p + 1) % k
    return P
```

Now it is only the matter of defining a good greedy function. Ours is quite simple:

```
def GreedyFunction(V', P, i, graph):
    gain = list with n elements
    for v in V':
        gain[v] = TotalGain(v, P, i)
        # TotalGain is defined as the increase in cut-size for a partially formed pa
    u = argmax(gain)
    return u
```

While the algorithm might not perform very good, if we run it for a few iterations, the result would improve.

2.4 Local Search

Our local search approach is based on the Kernighan-Lin Algorithm in refining a bipartition $P = (P_0, P_1)$. The algorithm based on the following steps:

For each $v \in P_0$, let $I(v)$ be the internal cost of v, i.e. the sum of the costs of edges between v and other nodes in P_0; $E(v)$ be the external cost of v, i.e. the sum of the costs of edges between v and nodes in P_1. Similarly, define I(u) and E(u) for each $u \in P_1$. Hence, if we exchange $v \in P_0$ and $u \in P_1$, the reduction of cost is:

$$C_{old} - C_{new} = E(u) - I(u) + E(v) - I(v) - 2w(\{u, v\})$$

The algorithm attempts to find an optimal series of interchanges between P_0 and P_1.

```
def KernighanLin(V, E, P_0, P_1):
    do:
        Computes E(v), I(v) for all v in V
        g_list = []
        u_list = []
        v_list = []
        for i in range(n // 2):
            Find u from P_0, v from P_1, such that
                g = E[u] - I[u] + E[v] - I[v] - 2 * w[u,v]
            is maximal.
            Remove u and v from consideration
            g_list.append(g)
            u_list.append(u)
            v_list.append(v)
            Update E() and I()
        k = argmax(g_cummulative) # g_cummulative[j] = sum(g_list[:j+1])
        g_max = g_cummulative[k]
        if g_max > 0:
            for i in range(0,k+1):
                exchange(u_list[i], v_list[i])
    while g_max > 0
```

2.5 Recursive Bisection

Provided that we have a bipartitioning algorithm (in this case, the Kernighan-Lin algorithm), a natural extension to that would be to recursively bipartition the graph. Here we also implemented this method, using Kernighan-Lin algorithm with the initial partition being random.

```
def RecursiveBisection(graph, k):
    partition = initialize() # initialize to all 1
    iteration = 1
    while k != 2**(iteration - 1):
        # After each iteration, each number j would become 2 * j or 2 * j + 1
        for j in range(2 ** (it - 1), 2 ** (it)):
            Pj = set()
            for v in range(n):
                if partition[v] == j:
                    Pj.add(v)
            n’ = len(Pj)
            Establish a bijection "FN" between Pj and 0..n’
```

```
                graph' = subgraph(graph, Pj) # Clear rows, columns with indices not in P
                P0, P1 = bipartitioner(graph', n')
                P0', P1' = FN^-1(P0, P1)
                for v in P0':
                    partition[v] = partition[v] * 2
                for v in P1':
                    partition[v] = partition[v] * 2 + 1
            iteration += 1
        return [x - k for x in range(n)]
```

A drawback of this algorithm is that it would only work on $k = 2^p$ (k being a power of 2). We would see the performance of this algorithm in the result section.

2.6 Partitioning with Karlsruhe High Quality Partitioner (KaHIP)

There are many existing high quality partitioner for these graph partitioning problem, namely: METIS, KaHyPar, etc. We use KaHIP here since it has a wrapper for Python.

To install KaHIP for Python, run the following script:

```
!python3 -m pip install pybind11
!git clone https://github.com/KaHIP/KaHIP
!KaHIP/compile_withcmake.sh BUILDPYTHONMODULE
```

Then, "import kahip" for personal use from the directory "KaHIP/deploy". There are many modes and many solvers in this module that you can explore in its documentations.

2.7 Spectral Clustering

Spectral clustering makes use of the spectrum (eigenvalues) of the similarity matrix of the data to perform dimensionality reduction before clustering in fewer dimensions. The similarity (affinity) matrix is provided as an input and consists of a quantitative assessment of the relative similarity of each pair of points in the dataset.

To implement it, let's learn about some special matrices:

First, the weighted adjacency matrix of the graph is the matrix $W = (w_{ij}), i, j = 1, ..., n$. Second, the degree matrix D is defined as the diagonal matrix with the degrees $d_1, \ . \ . \ . \ , d_n$ on the diagonal. The degree of a vertex $v_i \in V$ is defined as $d_i = \sum_{j=1}^{n} w_{ij}$.

Third, the unnormalized graph Laplacian matrix is defined as $L = D$ - W. This matrix has 4 properties:

- L is symmetric and positive semi-definite
- The smallest eigenvalue of L is 0, the corresponding eigenvector is the constant vector **1**.
- L has n non-negative, real-valued eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$.
- (Number of connected components and the spectrum of L) Let G be an undirected graph with non-negative weights. Then the multiplicity k of the eigenvalue 0 of L equals the number of connected components $A_1, \ . \ . \ . \ , A_k$ in the graph. The eigenspace of eigenvalue 0 is spanned by the indicator vectors $1_{A_1}, \ . \ . \ . \ , 1_{A_k}$ of those components.

Here is a pseudocode for a generic spectral clustering method:

```
def spectral_clustering(G, k):
    # Compute the Laplacian matrix of G
    L = compute_laplacian(G)

    # Compute the k smallest eigenvectors of L
    eigvals, eigvecs = compute_eigenvectors(L, k)

    # Construct a matrix V with the k eigenvectors as columns
    V = construct_matrix(eigvecs)

    # Normalize the rows of V to have unit length
    V = normalize_rows(V)

    # Cluster the rows of V using k-means or normalized cuts
    labels = kmeans(V)

    # Return the resulting clustering of nodes
return labels
```

Properties proofs, more insights about the method could be found here.

3 Results

4 Conclusion

4.1 Future Improvements

In the future, we would want to look into direct k-way partitioning local search techniques, and more efficient refinement techniques. We would also like to experiment with some other optimization paradigms, namely genetic algorithms.

One thing that we have not utilised from the problem statement is that since the constraint allowed for different in subset sizes, we are not restricted to only exchanging between subsets; but we can change the subset of a vertex, as long as the partition still satisfies the size constraint.

4.2 Final Remarks

To conclude, this project is both "easy" and hard for us, in the sense that the problem itself is easy enough to be able to be understood right away, but quite hard because the amount of research done on the topic is very rich and we hardly could find any room for improvement. However, the results shown above has already given us some very powerful insights about OR-Tools Constraint and Integer Programming solvers, and we had also provided valuable techniques and applications that can be applied to different problems.

4.3 Acknowledgement

We would like to thank our supervisor, Dr. Bui Quoc Trung, for being one of the most supportive lecturers to our queries, and has suggested valuable insights about the problem in our presentation. The mini-project is a valuable experience for all of us to learn and develop ourselves.