K-Graph Partitioning Problem

This project is a collection of some algorithm for the Graph Partitioning Problem. The algorithms that we are going to suggest includes:

- Backtracking
- Integer/Constraint Programming with Google OR-Tools
- Spectral Clustering
- Greedy Graph Growing Partitioning (GGGP) Algorithm
- Local Refinement Technique (Kernighan-Lin)
- Multilevel Partitioning using Karlsruhe High Quality Graph Partitioning (KaHIP) module

Problem Statement

Given an undirected graph $G = (V,E,w)$. We need to partition V into k subsets $V_1, ..., V_k$ such that:

- The k subsets are of nearly equal size. The constraint might be either in the form $$\max_{i} |V_i| - \min_{i} |V_i| <= \alpha $$ or $$|V_i| \le (\epsilon + 1) \frac{|V|}{k} \quad \forall i $$ for some $\alpha \in \mathbf{Z_{++}}$ or $\epsilon > 0$ are imbalance factors. We denote the two versions of the problems "α-constrained" or "ϵ-constrained".
- The total weight of edges that connects two different subsets is minimized (the cut size), i.e. $$\text{minimize} \sum_{\{u,v\}\in C} w(\{u,v\}) \\\text{ for } C = \{\{u,v\}\in E : u \in V_i, v\in V_j, 1 \le i < j \le k \}$$

Input Description

For data generation and in backtracking, spectral clustering and local refinement algorithms, we use a weight-adjacency matrix. The weight of edges range from 1 to 100.

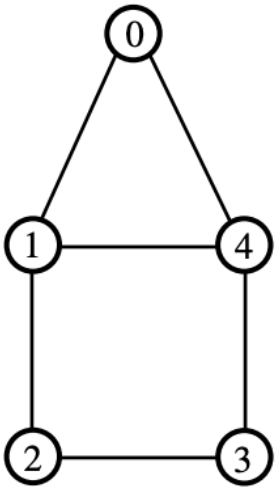
We also convert our matrix to a compressed sparsed row format to work with IP/CP and the module KaHIP. The data structure consists of four arrays:

- vwgt: the vertex weight array of size n. In our problem, the vertices are of equal weight, hence all elements are unity (1).
- xadj, adjncy: adjacency arrays of size (n+1) and (2m) to describe the edges.
- adjcwgt: the edge weight array of size 2m.

For example, the following arrays:

```
xadj = [0,2,5,7,9,12]
adjncy = [1,4,0,2,4,1,3,2,4,0,1,3]
```

would describe the following graph:



The vertices are 0-indexed. The ith vertex is adjacent to vertices in `adjncy[xadj[i]:xadj[i+1]]`. For example, the vertex 1 in the graph is adjacent to the edges with index `adjncy[2:5]`, which are $0,2,4$. Since one edge has to be described from two vertices, the `adjncy` and `adjcwgt` arrays are of size (2m).

In [ ]:
```
import numpy as np
import math
# Input for the problem in the above format
input_xadj       = np.array([0,2,5,7,9,12])
input_adjncy     = np.array([1,4,0,2,4,1,3,2,4,0,1,3])
input_vwgt       = np.array([1,1,1,1,1])
input_adjcwgt    = np.array([1,1,1,1,1,1,1,1,1,1,1,1])
epsilon          = 0.03
alpha            = 1
k                = 2
n = np.shape(input_vwgt)[0] # number of vertices
m = np.shape(input_adjcwgt)[0] // 2 # number of edges
```

In [ ]:
```
# Generate adjacency matrix, since for some of these algorithms to work, n s

adjacency = [[0 for i in range(n)] for j in range(n)]
for i in range(n):
    adjacency[i][i] = 0
for i in range(n):
    s,t = input_xadj[i], input_xadj[i+1]
    for j in range(s, t):
        adjacency[i][input_adjncy[j]] = input_adjcwgt[j]
```

If you want to generate a random graph, activate this cell.

In [ ]:
```
import random

def generate_random_graph(n, m):
    graph = [[0 for i in range(n)] for j in range(n)]
    edges = set()
    for i in range(m):
        while True:
            u = random.randint(0, n-1)
            v = random.randint(0, n-1)
            w = random.randint(1, 100)
            u, v = min(u,v), max(u,v)
```

```
            if u != v and (u, v) not in edges:
                edges.add((u, v))
                graph[u][v] = w
                graph[v][u] = w
                break
    return graph

# adjacency = generate_random_graph(n,m)
```

Backtracking Approach

The backtracking algorithm works well for small datasets, and would generate k-partitions of the set {0,...,n-1}. But for larger graphs, time would rise exponentially. The total number of k-partitions would be the Stirling number of the second kind (S(n,k)).

We could combine the algorithm with branch and bound method to reduce the number of total partitions.

In [ ]:
```
partition = [0 for i in range(n)]
V_max = (1 + epsilon) * math.ceil(n/k)
ans_partition = [-1 for i in range(n)]

lower_bound = float('inf')

def Try(i, mx, ans=0):
    global lower_bound, ans_partition
    def check_valid_partition(setting=0):
        def check_alpha(count):
            mn = min(count)
            mx = max(count)
            if (mx - mn <= alpha):
                return True
            return False

        def check_epsilon(count):
            mx = max(count)
            return mx < V_max

        count = [0 for _ in range(k)]
        for i in range(n):
            count[partition[i]] += 1
        if setting == 0:
            return check_alpha(count)
        else:
            return check_epsilon(count)

    if i == n:
        if check_valid_partition() == True and ans < lower_bound:
            ans_partition = partition[:]
            lower_bound = ans
    elif i == 0:
        partition[0] = 0
        Try(1,0,0)
    else:
        if mx == k-1:
            for j in range(k):
                new_ans = ans
                partition[i] = j
                # Update the weight
                for t in range(i):
```

```
                    if partition[t] != j:
                        new_ans += adjacency[i][t]
                if new_ans <= lower_bound:
                    Try(i+1, k-1, new_ans)
                else:
                    continue

        elif (i-mx)+k == n + 1:
            new_ans = ans
            partition[i] = mx+1
            for t in range(i):
                if partition[t] != mx+1:
                    new_ans += adjacency[i][t]
            if new_ans <= lower_bound:
                Try(i+1, mx+1, new_ans)
            else:
                return
        else:
            for j in range(mx+2):
                new_ans = ans
                partition[i] = j
                for t in range(i):
                    if partition[t] != j:
                        new_ans += adjacency[i][t]
                if new_ans <= lower_bound:
                    Try(i+1, max(mx, j), new_ans)
                else:
                    continue

Try(0,0)
for i in range(n):
    print("Vertex {} belongs to partition {}".format(i, ans_partition[i]))
```

Integer Programming using OR-Tools

OR-Tools is a strong tool for operation research, developed by Google. It is used for constraint programming and integer linear programming (ILP) problems. Today we are going to suggest the mathematical model for both versions of the partitioning problem: the alpha-constrained problem and the epsilon-constrained problem.

Let start with installing the OR-Tools package in Python.

In [ ]:
```
%pip install ortools
```

Let us begin with introducing an integer programming formulation for the ϵ-constrained version. First we introduce binary decision variable for all edges and vertices of the graph. For each edge $e=\{u,v\}\in E$, let $e_{uv} \in \{0,1\}$, i.e. whether e is a cut edge. Moreover, for each $v\in V$ and subset p, let $x_{v,p} \in \{0.1\}$ to denote if v is in subset p or not. We have a total of $|E|+k|V|$ variables.

The maximum size of a subset should be: $$ V_{max} = (1+\epsilon) \lceil \frac{|V|}{k} \rceil $$

To ensure a valid partition, we have: $$ \forall \{u,v\} \in E, \forall p: e_{uv} \ge + x_{u,p} - x_{v,p} $$ $$ \forall \{u,v\} \in E, \forall p: e_{uv} \ge - x_{u,p} + x_{v,p} $$ $$ \forall p : \sum_{v\in V} x_{v,p} \le V_{max} \quad (*)$$ $$ \forall v\in V: \sum_{p} x_{v,p} = 1 $$

If we want the partition size to be constrained by alpha instead of epsilon, then $(*)$ should be replaced with

$$ \forall p : \sum_{v \in V} x_{v,p} \le M $$$$ \forall p : \sum_{v \in V} x_{v,p} \ge m $$$$ M - m \le \alpha $$
The objective function is:

$$ \text{minimize } \sum_{\{u,v\}\in E}e_{uv} w(\{u,v\}) $$

In [ ]:
```
from ortools.linear_solver import pywraplp
solver = pywraplp.Solver.CreateSolver('SCIP')
solver.set_time_limit(5)
infinity = solver.infinity()
X = dict()
E = dict()
for v in range(n):
    for p in range(k):
        X[(v,p)] = solver.IntVar(0,1,'X[{},{}]'.format(v,p))

for u in range(n):
    for j in range(input_xadj[u],input_xadj[u+1]):
        v = input_adjncy[j]
        if u < v:
            E[(u,v)] = solver.IntVar(0,1,'E[{},{}]'.format(u,v))

print("Number of Variables: {}".format(solver.NumVariables()))

for u,v in E:
    for p in range(k):
        solver.Add(E[u,v] >= X[u,p] - X[v,p])
        solver.Add(E[u,v] >= -X[u,p] + X[v,p])

mx = solver.IntVar(0,n, 'mx')
mn = solver.IntVar(0,n, 'mn')


for p in range(k):
    constraint = solver.RowConstraint(0,infinity,'')
    for v in range(n):
        constraint.SetCoefficient(X[v,p], 1)
    constraint.SetCoefficient(mn, -1)
for p in range(k):
    constraint = solver.RowConstraint(0, infinity,'')
    for v in range(n):
        constraint.SetCoefficient(X[v,p], -1)
    constraint.SetCoefficient(mx, 1)

solver.Add(mx - mn <= alpha)

for p in range(k):
    constraint = solver.RowConstraint(0,infinity,'')
    for v in range(n):
        constraint.SetCoefficient(X[v,p], 1)

for v in range(n):
    constraint = solver.RowConstraint(1,1,'')
    for p in range(k):
        constraint.SetCoefficient(X[v,p], 1)

objective = solver.Objective()
for u,v in E:
    objective.SetCoefficient(E[(u,v)], int(adjacency[u][v]))
```

```
objective.SetMinimization()

status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL or status == pywraplp.Solver.FEASIBLE:
    print('Objective Value =', solver.Objective().Value())
    for v in range(n):
        for p in range(k):
            if X[(v,p)].solution_value() == 1.:
                print("Vertex {} belongs to Partition {}".format(v,p))
    print()
    print('Problem solved in %f milliseconds' % solver.wall_time())
    print('Problem solved in %d iterations' % solver.iterations())
    print('Problem solved in %d branch-and-bound nodes' % solver.nodes())
else:
    print('The problem does not have an optimal solution.')
```

Constraint Programming using Google Ortools

We also implemented the above model using a constraint programming solver to benchmark the performance of two solvers on the same modelling. The below implementation is α-constrained

In [ ]:
```
from ortools.sat.python import cp_model

model = cp_model.CpModel()

X = dict()
E = dict()

for v in range(n):
    for p in range(k):
        X[(v,p)] = model.NewIntVar(0,1,'X[{},{}]'.format(v,p))

for u in range(n):
    for v in range (n):
        if adjacency[u][v] !=0 and u < v:
            E[(u,v)] = model.NewIntVar(0, 1,'E[{},{}]'.format(u,v))

for u,v in E:
    for p in range(k):
        model.Add(E[u,v] >= X[u,p] - X[v,p])
        model.Add(E[u,v] >= -X[u,p] + X[v,p])

model.Add(sum(X.values())==n)
for v in range (n):
    model.Add(sum(X[(v,p)] for p in range (k)) == 1)

for p in range (k):
    for q in range (k):
        model.Add(sum(X[(u,p)] for u in range (n)) - sum(X[(u,q)] for u in r

model.Minimize(sum(E[u,v] * adjacency[u][v] for u, v in E))
solver = cp_model.CpSolver()
status = solver.Solve(model)
if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
    print('Objective Value =', solver.ObjectiveValue())
    for v in range(n):
```

```
        for p in range(k):
            if solver.Value(X[(v,p)]) == 1:
                print("Vertex {} belongs to Partition {}".format(v,p))
    print()
    print('Problem solved in %f milliseconds' % solver.WallTime())

else:
    print('The problem does not have an optimal solution.')
```

Spectral Clustering

In multivariate statistics, spectral clustering techniques make 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, we need to know about some special matrices.

First, the weighted adjacency matrix of the graph is the matrix $W = (w_{ij}), i,j=1,...,n$. w_{ij} equals the weight of the edge that connects 2 vertices i and j. If w_{ij} = 0 this means that the vertices vi and vj are not connected by an edge.

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 ∈ V is defined as $d_i = \sum_{j=1}^n wij $

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 one vector 1
- L has n non-negative, real-valued eigenvalues 0 = $λ_1$ ≤ $λ_2$ ≤ . . . ≤ $λ_n$.
- (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.

Now we would like to state the most common unnormalized spectral clustering algorithms.

Input: The weighted adjacency matrix W represents the undirected graph $G = (V,E,w)$

- Step 1: Compute the unnormalized Laplacian L.
- Step 2: Compute the first k eigenvectors u_1, . . . , u_k of L.
- Step 3: Let U ∈ $R^{n×k}$ be the matrix containing the vectors u_1, . . . , u_k as columns.

- Step 4: For $i = 1, . . . , n$, let y_i ∈ R^k be the vector corresponding to the i-th row of U.
- Step 5: Cluster the points (y_i), $i=1,...,n$ in R^k with the k-means algorithm into clusters C_1, . . . , C_k.

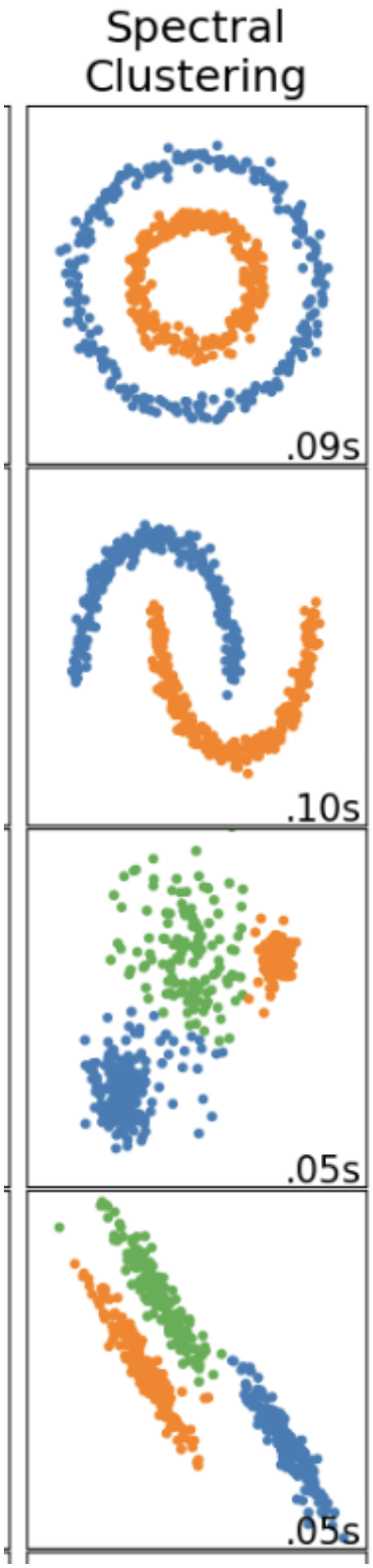
Output: Clusters A_1, . . . , A_k with A_i = {$j : y_j ∈ C_i$}.

In [ ]:
```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans

W = np.array(adjacency)

D = [[np.sum(W, axis=0)[i] if i == j else 0 for i in range (n)]
     for j in range (n)]
L = np.array(D-W)

eigenvals, eigenvcts = np.linalg.eig(L)
def project_and_transpose(eigenvals, eigenvcts, num_ev):
    """Select the eigenvectors corresponding to the first
    (sorted) num_ev eigenvalues as columns in a data frame.
    """
    eigenvals_sorted_indices = np.argsort(eigenvals)
    indices = eigenvals_sorted_indices[: num_ev]

    proj_df = pd.DataFrame(eigenvcts[:, indices.squeeze()])
    proj_df.columns = ['v_' + str(c) for c in proj_df.columns]
    return proj_df

proj_df = project_and_transpose(eigenvals, eigenvcts, num_ev=2)

def run_k_means(df, n_clusters):
    """K-means clustering."""
    k_means = KMeans(random_state=25, n_clusters=n_clusters)
    k_means.fit(df)
    cluster = k_means.predict(df)
    return cluster

cluster = run_k_means(proj_df, n_clusters=2)
print(cluster)
```

Besides this implementation, there are also several approaches to spectral clustering, some does not even require a full similarity matrix to be constructed to reduce memory. One notable references is from Ng, Jordan:

In testing, we use the more robust implementation from `scikit-learn` library. Since this is a clustering algorithm, it would work best if the data points form some sort of clusters already, and a huge drawback of this technique is that the cluster sizes might not be the same. However, we think it is still very useful to propose and test this algorithm.

Some examples on toy datasets (generator available on documentation page):



In [ ]:
```
from sklearn.cluster import SpectralClustering

sc = SpectralClustering(n_clusters=k, affinity='precomputed')
partitions = sc.fit_predict(adjacency)

print(partitions)
```

Greedy Graph Growing Partitioning Algorithm

For the graph partitioning problem, we propose a very simple greedy framework:

```
P = (P_0, P_1, ..., P_{k-1})
V' = V
Assign 1 random vertex for each P_i
i = 0
while |V'| > 0:
    choose vertex u using a greedy_function(V', P, i, graph)
    P_i.add_vertex(u)
    V'.remove(u)
    p = (p + 1) % k
return P
```

Many greedy algorithms have been proposed and most of them have similar structure. Our greedy algorithm is quite simple: we choose a vertex with minimum total gain, where the gain of a vertex for a partially-formed partition is the total increase in cut-size of that vertex. If many vertices have the same minimun total gain, then tie is broken randomly.

```
greedy_function(V, P, i, graph):
    return argmin_v{total_gain(v, P, i) for v ∈ V}
    where total_gain(v, P, i): gain in cutsize if add v into
subset i for a partially formed partition P
```

In [1]:

```
def greedy_partition(graph, n, k):
    part = [-1] * n
    vert = list(range(n))
    random.shuffle(vert)

    for p in range(k):
        u = vert[-1]
        part[u] = p
        vert.pop()

    p = 0

    def total_gain(P,p,v,G):
        ans = 0
        for u in P:
            if P[u] != p and P[u] != -1:
                ans += G[v][u]
                # Gain
        return ans

    def greedy(V, P, p, G):
        m = float('inf')
        C = []
        gain = [0] * len(V)
        for i, v in enumerate(V):
            g = total_gain(P,p,v,G)
            m = min(m,g)
            gain[i] = g
        for i in range(len(V)):
            if gain[i] == m:
                C.append(V[i])
        return random.choice(C)

    while len(vert) > 0:
        b = greedy(vert, part, p, graph)
```

```
            part[b] = p
            vert.remove(b)
            p = (p + 1) % k
        return part
```

Using this algorithm, the partition is guaranteed to distribute vertices as equally as possible, hence we would not have to worry about alpha nor epsilon. However, its performance is quite poor, hence we would usually run it for several (hundreds) iterations and choose the best result.

Local Refinement Technique (Kernighan-Lin)

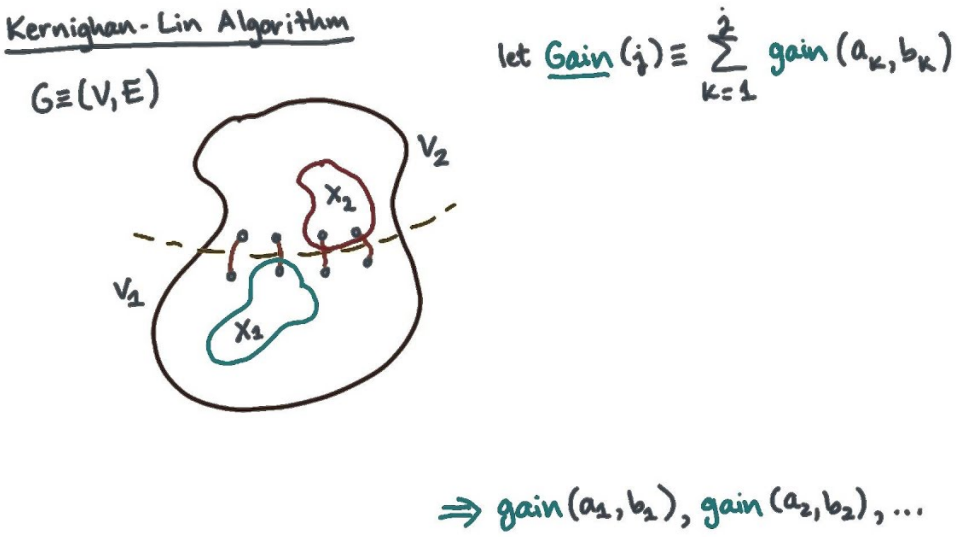
The Kernighan-Lin algorithm aims to improve the initial bipartition of a weighted graph. The idea is to iteratively swap vertices between the two partitions such that the total cost is minimized.

The algorithm takes an adjacency matrix and an initial bipartition of the vertices. The initial bipartition can be chosen arbitrarily, or in our implementation we improve directly on the partition made by the greedy algorithm above.

The algorithm then proceeds in the following steps:

- Calculate the degree of each vertex in the graph.
- Initialize two empty sets for vertices swapped between partitions.
- Create a dictionary to keep track of the gain of each vertex. The gain of a vertex is defined as the reduction in cut weight that would result from swapping the vertex between the two partitions.
- Sort the vertices in descending order of their gain.
- Swap vertices between the two partitions in order of their gain, until no further improvement is possible. To do this, add the vertex with the highest gain to the set of vertices that will be moved to the other partition. Then recalculate the gains for all remaining vertices and add the vertex with the highest gain to the set of vertices that will be moved to the other partition. Continue this process until no further improvement is possible, i.e. until no more vertices can be swapped to further reduce the cut weight.
- Return the final bipartition and the cut weight.

The Kernighan-Lin algorithm has a time complexity of $O(n^3)$, where n is the number of vertices in the graph. However, it has been shown to be effective in practice and can produce high-quality partitions for many real-world graphs.



In [ ]:

```
def KL(adj, P0, P1):
    # Algorithm takes an adjacency matrix, partition P = (P0, P1)
    improvement = True
    iteration = 50000

    def Ex(u, P):
        # external cost
        cost = 0
        for v in P:
            cost += adj[u][v]
        return cost

    def In(u, P):
        # internal cost
        cost = 0
        for v in P:
            cost += adj[u][v]
        return cost

    def gP_u(u, P, P_):
        return Ex(u, P_) - In(u, P)
    n = len(adj)
    while improvement and iteration > 0:
        g = dict()
        for u in P0:
            g[u] = gP_u(u, P0, P1)
        for v in P1:
            g[v] = gP_u(v, P1, P0)

        L0 = set()
        L1 = set()

        gains = []

        for _ in range(n // 2):
            max_gain = -1 * float('inf')
            pair = []

            for u in P0:
                if u not in L0:
                    for v in P1:
                        if v not in L1:
```

```
                        gain = g[u] + g[v] - 2 * adj[u][v]
                        if gain > max_gain:
                            max_gain = gain
                            pair = (u,v)
            a = pair[0]
            b = pair[1]
            L0.add(a)
            L1.add(b)
            gains.append(((a,b), max_gain))

            for u in P0:
                if u not in L0:
                    g[u] += 2 * adj[u][a] - 2 * adj[u][b]

            for v in P1:
                if v not in L1:
                    g[v] += 2 * adj[v][b] - 2 * adj[v][a]

        gmax = -1 * float('inf')
        jmax = 0
        for j in range(1, len(gains) + 1):
            gsum = 0
            for i in range(j):
                gsum += gains[i][1]
            if gsum > gmax :
                gmax = gsum
                jmax = j


        if gmax > 0:
            for i in range(jmax):
                P0.remove(gains[i][0][0])
                P0.add(gains[i][0][1])
                P1.remove(gains[i][0][1])
                P1.add(gains[i][0][0])
        else:
            improvement = False
        iteration -= 1
    return P0, P1
```

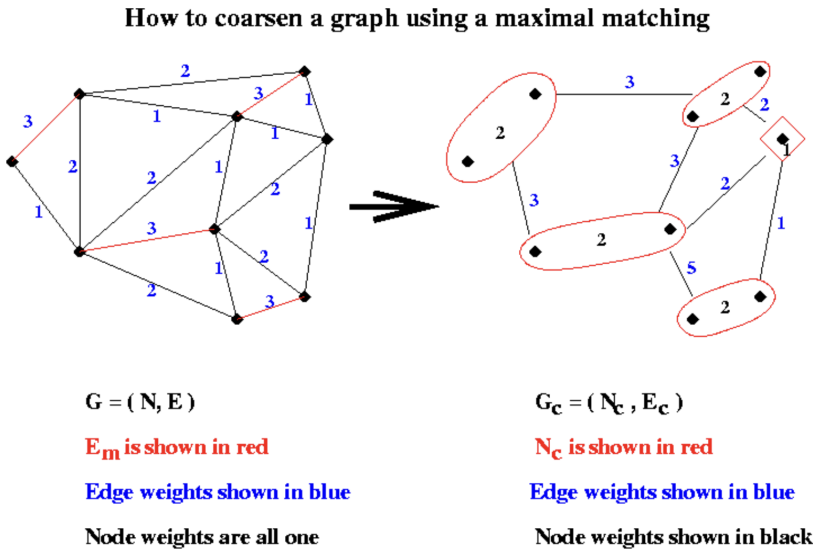
Karlsruhe High Quality Graph Partitioning Module

This library is a powerful library for multilevel graph partitioning problem. Its multilevel algorithm incorporates many valuable idea, including graph coursening, local search refinements, etc.

Multilevel graph partitioning is a very practical and powerful paradigm in solving real-life problem; hence many research were carried out. The general algorithm consists of three phase:

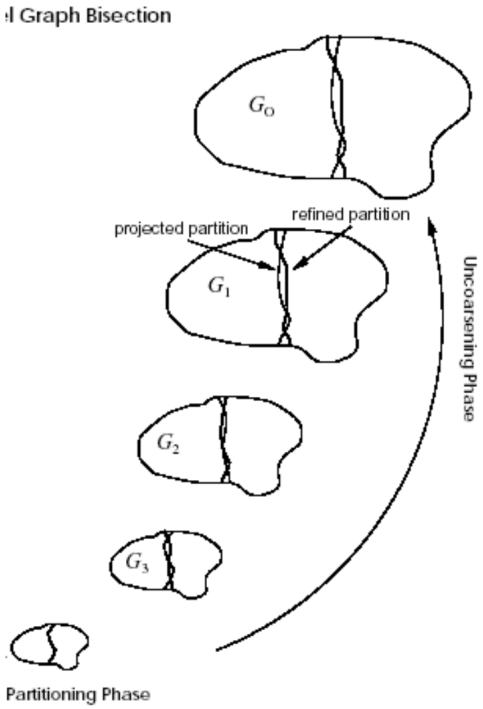
- Phase 1: Graph contraction (Coursening)
  When an edge {u,v} is collapsed to form a hypervertex, its weight is the sum of weight of vertex u and v. Initially, the weights of all vertices are 1. A rating of an edge illustrate how much it would make sense to contract that edge. A reasonable matching algorithm tries to maximize the sum of the ratings of the contracted edges. This phase stops when the size of the hypergraph is small enough to

partition it.



- Phase 2: Partition the coursened graph
  Any partitioning algorithm could be used in this phase, from Integer Programming, Spectral Clustering to Greedy approaches.

- Phase 3: Refinement (Uncoursening)
  Local search heuristics are used in this phase, when we try to move nodes between blocks to improve the cut size or balance. The 'gain' of a move is the decrease in edge cut if a node is moved to a different block. There are 2 types of refinement algorithm usually used: k-way refinement and 2-way refinement. The former is allowed to move a node to an arbitrary block while the latter is only allowed to move nodes between pair of blocks.



To install KaHIP package for Python, please refer to https://github.com/KaHIP/KaHIP where there are detailed instructions. The cell below is to download and build the

package with cmake right on Google Colab.

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

Go to directory: `KaHIP/deploy`

In [ ]:
```
%cd KaHIP/deploy
%pwd
```

Then you could import the KaHIP module in your program. There are many configurations of the module that one could use (choose solvers, etc.). To read more about the module and how it works, visit https://kahip.github.io/ and refer to the user guide and publications.

In [ ]:
```
import kahip

#build adjacency array representation of the graph
xadj            = input_xadj
adjncy          = input_adjncy
vwgt            = input_vwgt
adjcwgt         = input_adjcwgt
supress_output = 0
imbalance       = epsilon
nblocks         = k
seed            = 0

# set mode
#const int FAST           = 0;
#const int ECO            = 1;
#const int STRONG         = 2;
#const int FASTSOCIAL     = 3;
#const int ECOSOCIAL      = 4;
#const int STRONGSOCIAL   = 5;
mode = 2

edgecut, blocks = kahip.kaffpa(vwgt, xadj, adjcwgt,
                               adjncy,  nblocks, imbalance,
                               supress_output, seed, mode)

print(edgecut)
print(blocks)
```

Results

After testing with random graphs generated with the code above, we benchmarked the algorithms according to time taken and solution found on our randomly generated datasets:

```
Small dataset
10vers, 25 edges, 2 clusters, alpha = 2
20vers, 50 edges, 2 clusters, alpha = 2
Test against: backtracking (2 versions); CP (2 versions); IP
(2 versions); Greedy (500 & 1000 iterations); Spectral;
Greedy+KL
```

```
Medium dataset
50vers, 125 edges, 2 clusters, alpha = 2
50vers, 750 edges, 2 clusters, alpha = 2
Test against: CP (2 versions); IP (2 versions); Greedy (500 &
1000 iterations); Spectral; Greedy+KL

Large dataset
100vers, 250 edges, 2 clusters
100vers, 2250 edges, 2 clusters
Test against: CP (eps); IP (eps); Greedy (50 iterations);
Spectral; Greedy+KL
100vers, 250 edges, 4 clusters
100vers, 2250 edges, 4 clusters
Test against: CP (eps); IP (eps); Greedy (50 iterations);
Spectral
1000vers, 20000 edges, 2 clusters
Test against: Greedy (1 iteration); Spectral; Greedy+KL
1000vers, 20000 edges, 8 clusters
Test against: Greedy (1 iteration); Spectral;
```

After analysing the results, we gain some key insights.

First, let's look at some numbers from the small dataset (set2):

set2	avgtime	optimal?	avgcost
backtrack(alpha)	0.0839	yes	583
ip(alpha)	0.0979	yes	583
cp(alpha)	0.0321	yes	583
backtrack(eps)	0.0487	yes	523
ip(eps)	0.0791	yes	523
cp(eps)	0.0258	yes	523
greedy(500)	0.1089	no	683
greedy(1000)	0.2135	no	652
greedy+KL	0.0044	sometimes	597
spectral	0.0047	not balance	477

Here, we notice that backtracking, IP and CP all return optimal results in reasonable time. Greedy, on the other hand, does not yield optimal result; however for 500 iterations more we definitely achieve a slightly better result. In spectral clustering, since the cluster is most likely not balanced, it yields an even smaller cost than the optimal result.

Let's look at another example from the medium dataset (set3, n=50 and sparse; set4, n=50 and dense):

set4	avgtime	avgcost
ip(alpha)	limit to 10s	17549
cp(alpha)	limit to 10s	15482
ip(eps)	limit to 10s	17263

set4	avgtime	avgcost
cp(eps)	limit to 10s	15228
greedy(500)	1.4687	17114
greedy(1000)	2.9194	16890
greedy+KL	0.0062	15329
spectral	0.0061	14431

In set3, IP and CP can both get to optimal result in less than 1s. On the other hand, greedy alone can yield usable partitions, and with refinement is very good.

Last but not least, large test sets (set6.2 and set7.1)

set6.2	avgtime	avgcost
ip(eps)	limit to 3s	85042
cp(eps)	limit to 3s	78492
greedy(50)	1.1566	82878
spectral	0.0075	73201

set7.1	avgtime	avgcost
greedy(1)	21.212	504179
greedy+KL	64.318	377295
spectral	0.216	390367

In both set4 and set6.2, IP shows much poorer performance than CP. Moreover, while greedy+KL shows the best in set7.1, it is very slow.

Future Work and Remarks

For future work, we would like to implement the greedy + Kernighan-Lin algorithm into a recursive bisectioning algorithm to allow it to work for $k = 2^p$. We would also want to look into direct k-way partitioning heuristics and refinement techniques.

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 also provide valuable techniques and applications that can be applied to different fields, i.e. unsupervised learning, network routing, etc.

Source code, graph data and benchmark results can be found here: https://github.com/sanghuynh0929/GraphPartitioning

References: https://lume.ufrgs.br/bitstream/handle/10183/67181/000872783.pdf