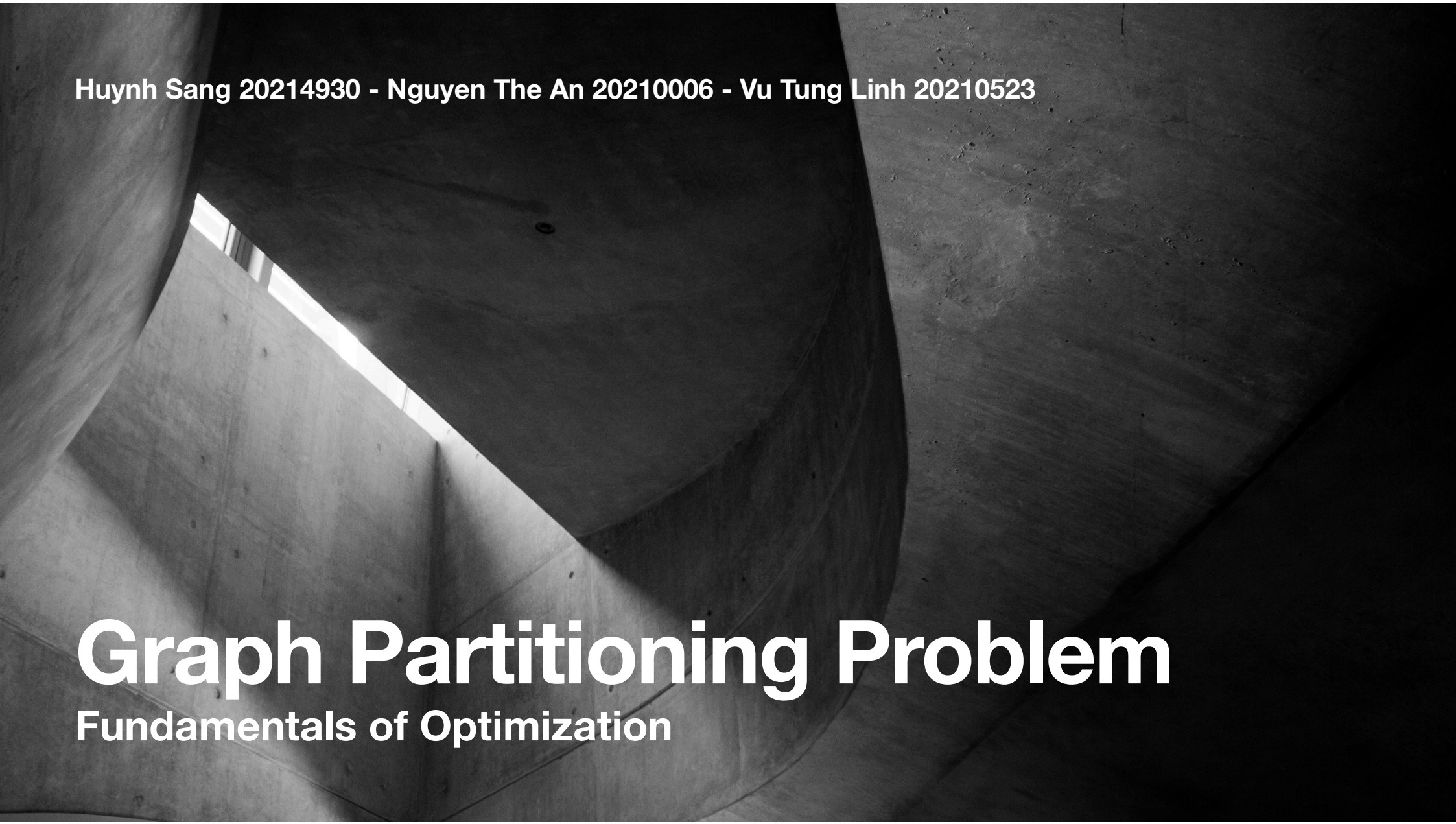
Huynh Sang 20214930 - Nguyen The An 20210006 - Vu Tung Linh 20210523



Graph Partitioning Problem

Fundamentals of Optimization

Overview

6 Algorithms

- Backtracking
- Integer/Constraint Programming
- Spectral Clustering
- Greedy Algorithm
- Kernighan-Lin Local Refinement
- Multilevel Partitioning

Problem Statement

Given: $G = (V, E, w)$. Need to partition V into k subsets, such that:

- $\max_i V_i - \min_i V_i \leq \alpha$ (alpha-constrained, original task)
- $V_i \leq V_{max} := (1 + \epsilon)\left\lceil \frac{|V|}{k} \right\rceil$ (epsilon-constrained, more practical since RHS is const)

And the total weight of cut-edges are minimized, i.e.

- minimize $\sum_{\{u,v\} \in C} w(\{u, v\})$
- where $C = \{\{u, v\} \in E : u \in V_i, v \in V_j, 1 \leq i < j \leq k\}$

Backtracking Approach

For the backtracking problem, the hardest thing is to come up with an algorithm that can generate all k-partitions efficiently.

Note that, there exists a bijection between the set of all partitions and this set:

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

Which means, a number has to be in an existing partition or has to be in a new partition.

For instance, the 3-partition {{1,3},{2,4},{5}} correspond to the array [1,2,1,2,3].

```
n,k = 5,3
partition = [0 for i in range(n)]

def Try(i, mx, ans=0):
    global partition
    if i == n:
        print(*partition)
    elif i == 0:
        partition[0] = 0
        Try(1,0,0)
    else:
        if mx == k-1:
            for j in range(k):
                partition[i] = j
                Try(i+1, k-1, new_ans)

        elif (i-mx)+k == n + 1:
            partition[i] = mx+1
            Try(i+1, mx+1, new_ans)
        else:
            for j in range(mx+2):
                partition[i] = j
                Try(i+1, max(mx, j), new_ans)
Try(0,0)
```

Backtracking Approach

To further enhance the algorithm, we augment Branch and Bound:

- Generate an initial partition, call the cost *lower_bound*
- When building a solution, if our cost is already larger than *lower_bound*, then cut that branch

And to even reduce the size of our search tree, we turn to our epsilon-constraint:

- When building a partition, if there is a subset whose size is already larger than V_{max}, then we can cut that branch.

Integer/Constraint Programming

IP formulation:

$$\forall \{u,v\} \in E, \forall p : e_{uv} \geq +x_{u,p} - x_{v,p} \qquad \forall \{u,v\} \in E, \forall p : e_{uv} \geq -x_{u,p} + x_{v,p}$$

$$\forall p : \sum_{v \in V} x_{v,p} \leq V_{max} \text{ (*)} \qquad \forall v \in V : \sum_{p} x_{v,p} = 1$$

- $e_{uv} \in \{0,1\}$: if $\{u,v\}$ is a cut-edge or not,
- $x_{v,p} \in \{0,1\}$: if $v \in V_p$ or not.
- The first two constraint allows for definition of cut-edge
- The second two constraint allows for valid partition and valid subset size.

Integer/Constraint Programming

If our problem is alpha-constrained, then we can replace constraint (*) with:

- $\forall p : \sum_{v \in V} x_{v,p} \leq M$
- $\forall p : \sum_{v \in V} x_{v,p} \geq m$
- $M - m \leq \alpha$

Ultimately, the objective function:

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

We implement this model into both OR-Tools' CP solver and SCIP solver.

Spectral Clustering

- Powerful algorithm for non-convex clusters
- Make use of the spectrum (eigenvector, eigenvalues) to reduce dimension (similar concept as Principal Components Analysis)
- Input:
  - W: our adjacency weight matrix (symmetric)
  - D: degree matrix ($d_i = \deg$ vertex i) (diagonal)
  - $L = D - W$: the Laplacian (symmetric)

Spectral Clustering

Properties of the Laplacian:

- Positive-definite
- Smallest eigenvalue is $\lambda_1 = 0$, corresponding eigenvector is $\mathbf{1}$.
- Has n non-negative real eigenvalues $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.
- Multiplicity of $\lambda = 0$ is the number of connected components in the graph.

Proof: refer to https://people.csail.mit.edu/dsontag/courses/ml14/notes/Luxburg07_tutorial_spectral_clustering.pdf

Unnormalized Spectral Algorithm

Input: adjacency weight matrix W

- Compute the (unnormalized) Laplacian L
- Compute the first k eigenvectors of L: $v_1, v_2, \ldots, v_k$
- Let $U \in \mathbb{R}^{n \times k}$: matrix with columns $v_1, v_2, \ldots, v_k$
- For $i = 1,...,n$, let $y_i \in R^k$ be the vector corresponding to the i-th row of U.
- Cluster the points (y_i), $i = 1,...,n$ in R^k with the k-means algorithm into clusters $C_1, \ldots, C_k$.

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

Spectral Clustering

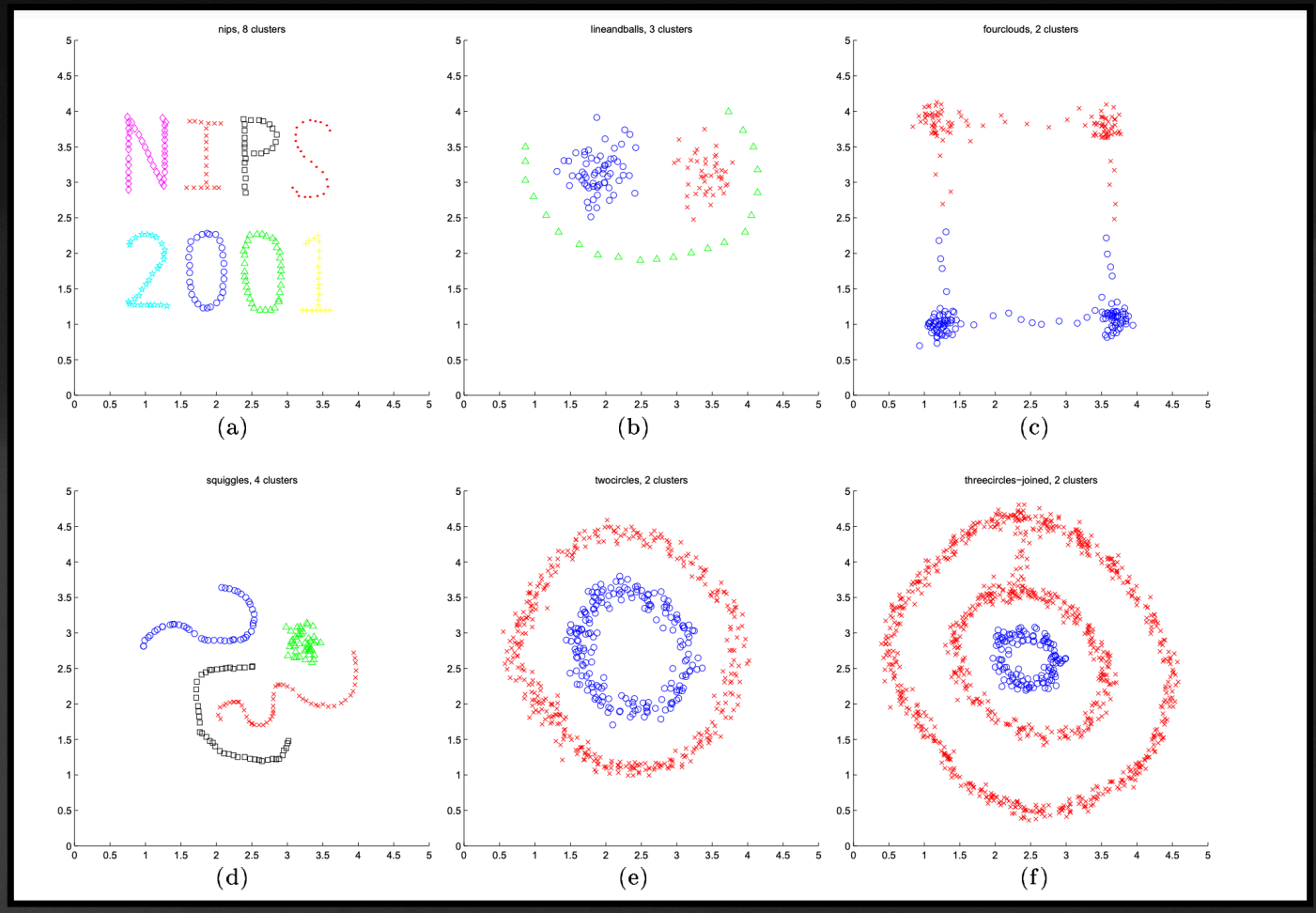
Implementation:

Note that, this is a very basic spectral algorithm; there are variations with normalized Laplacian, etc.

It is better to import from *sklearn.cluster.SpectralClustering* than to reimplement our own version in practice.

```
import pandas as pd
from sklearn.cluster import KMeans

W = np.array(affinity)

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)
```

Spectral Clustering



Clusterings from Ng, Jordan paper

Greedy Graph Growing Partition

- 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
```

Greedy Graph Growing Partition

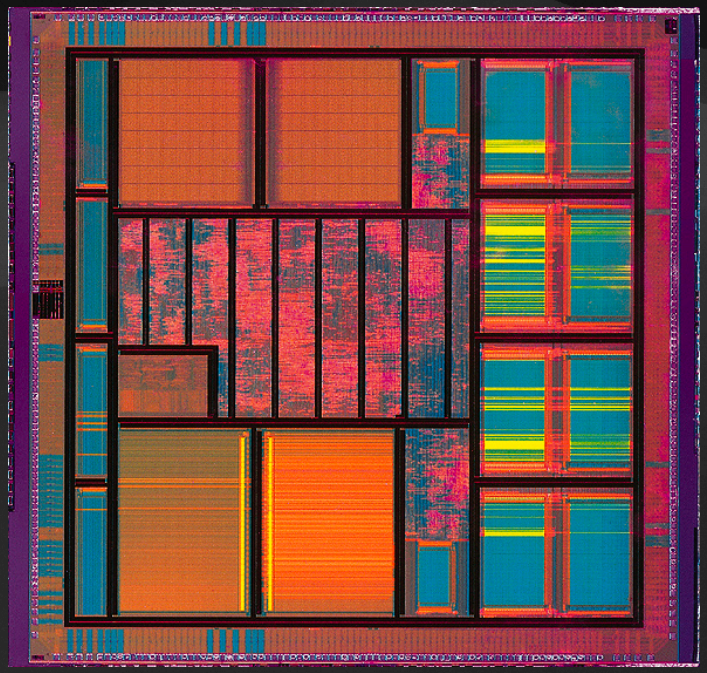
- Our greedy function:

```
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
```

- Algorithm most likely is not going to perform well, but could run for many iterations and get usable partition.

Kernighan-Lin Local Refinement

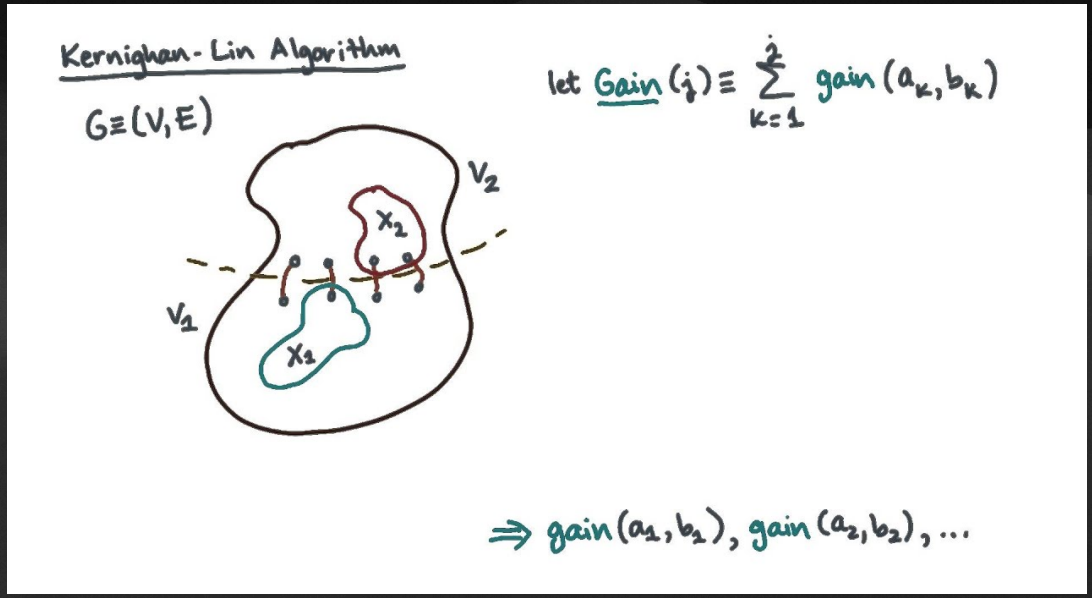
- Historically important; one of the first practical heuristics for graph partitioning
- Invented by Kernighan and Lin (1970)
- Designing electronic layout of Very Large-Scale Integration



A VLSI-Integrated Circuit die

Kernighan-Lin Local Refinement

- Perform refinement on bipartition $P = (P_1, P_2)$
- Choose $X_1 \in P_1, X_2 \in P_2$ to swap



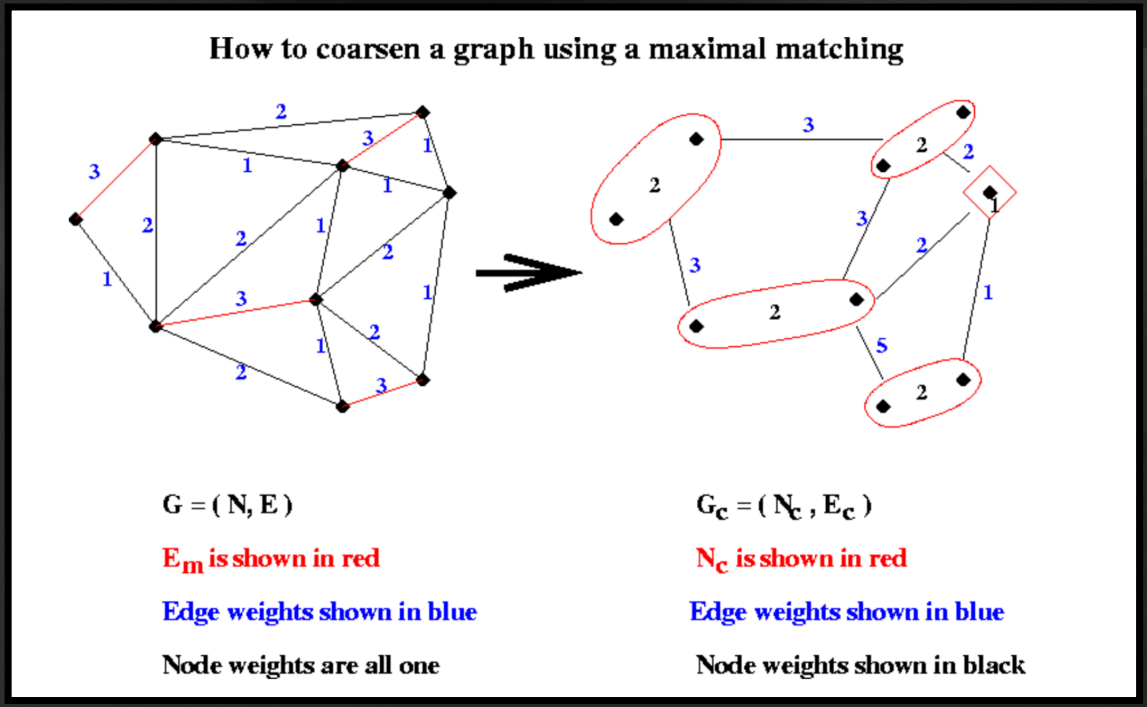
- Algorithm choose iteratively:
  - Choose 2 vertices from $P_1 \backslash X_1, P_2 \backslash X_2$ such that swapping them would yield maximum decrease in cut-size
  - Stop when no improvement can be made

Multilevel Algorithms

- A practical and useful paradigm in partitioning problems
- Phase 1: Graph contraction (coursening), many approaches:

  maximal matching,
  maximal cliques, etc.

- Phase 2: Partition
  When the contracted graph is small enough, carry out partition using IP/greedy/...



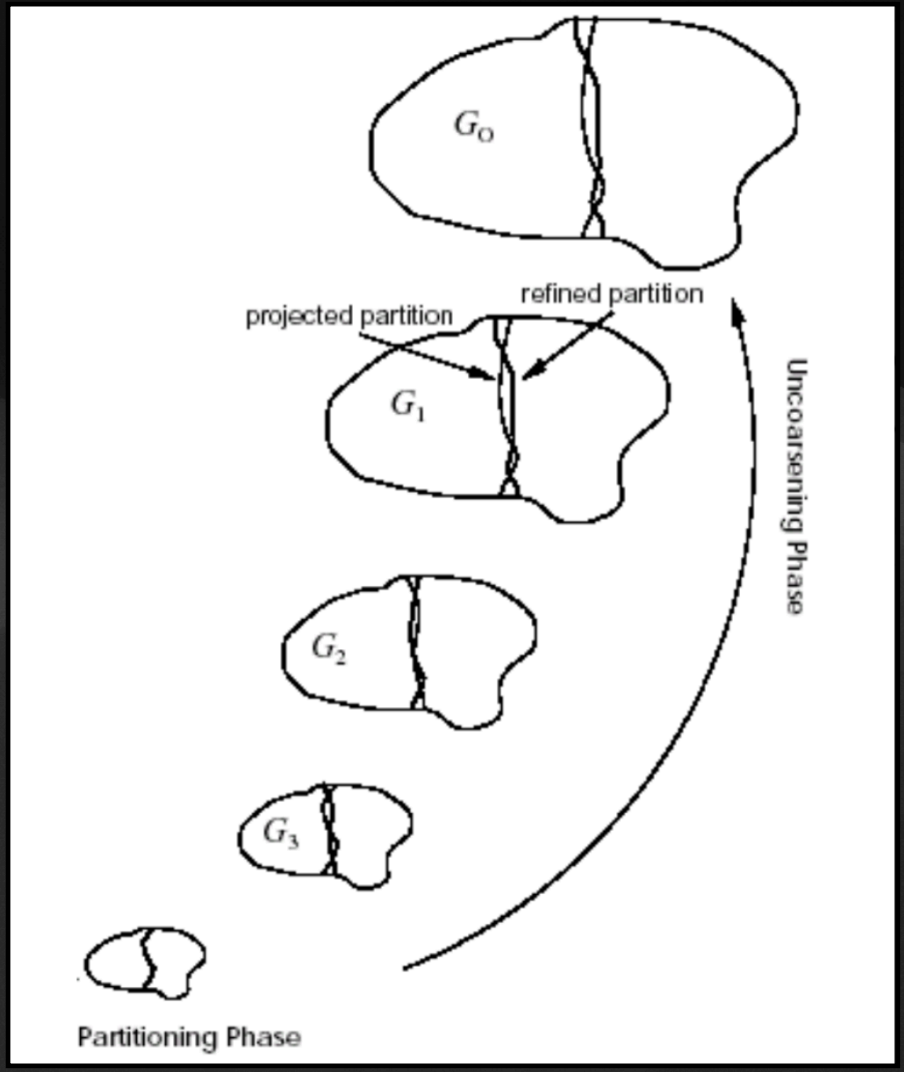
Multilevel Algorithms

- Phase 3: Uncoursening:

  Make use of local search heuristics for refining partitions.

  Refine partition = swap vertices from one subset to another to decrease cost or rebalance (such as Kernighan-Lin)

- Exists many other modules: METIS, KaHyPar, etc.



Results

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;

Results

Small sample: set2

Remarks in small test sets:

- Backtracking, IP, CP can deliver exact solutions quickly
- Greedy algorithm may get optimal results in smaller test sets; difference from exact solution is acceptable
- With KL refinement, result becomes better
- Spectral's imbalance is noticable

	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

Results

Medium sample: set3 (n=50, sparse) , set4 (n=50, dense)

Remarks in medium sample

- set3: IP/CP can terminate in <1s
- set4: cannot reach exact sol.
- epsilon formulations are slightly faster
- IP performed worse than CP
- Greedy alone can yield usable results
- Greedy+KL: very good
- Spectral: good in time & cost; balancedness questionable

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

Results

Large sample:

set6.2 (n=100, m=2250, k=4)

- IP shows much poorer performance comparing to CP

set7.1 (n=1000, m=20000, k=2)

- greedy+KL: impressive but slow
- greedy: not very competitive if run for few iterations

set7.2 (n=1000, m=20000, k=8)

- multilevel: excellent + can be fast if necessary
- spectral: good and fast

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
multilevel(strong)	9.243	379925
set7.2	**avgtime**	**avgcost**
greedy	23.534	878642
spectral	0.282	741639
multilevel(strong)	13.590	711914
multilevel(fast)	0.677	732573

Future works

- Implement Recursive Bisection
- Direct k-way partitioning
- Explore different techniques, i.e. Genetic Algorithms, etc.

Final remarks:

- In practice, we should use available tools, e.g. KaHIP module, because it will most likely outperform our own implementation in both partition quality and time taken.


THANKS FOR LISTENING