Algorithmic Methods for Mathematical Models Course Project

Ali A.Yarmohammadi

Master students in Innovation and Research in Informatics
Polytechnic University of Catalonia (UPC)
Faculty of Computer Science - FIB

Project Presentation, December 2021

Table of Contents

Table of Contents

Introduction

A surveillance company is developing a shape recognition program. The goal of this system is to determine if a specific **shape** appears in the **pictures**. In other words, this system **matches** a particular **pattern** in an Image.

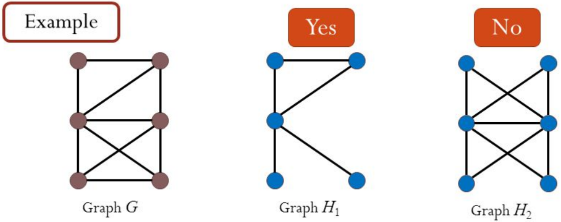
The goal is ...

The goal of this project is, given an image and a shape, to find an optimal embedding according to the specific criterion.
We are particularly interested in the embeddings that **minimize** the **sum of absolute differences between the weight** of an edge ***e*** and the weight of ***f(e)*** .

Introduction

Subgraph Isomorphism Problem

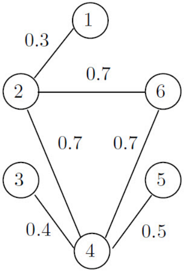
- **Input:** Two graphs $G=(V_G, E_G)$ and $H=(V_H, E_H)$
  - $|V_H| \leqq |V_G|$ and $|E_H| \leqq |E_G|$
- **Question:** Is H a subgraph isomorphic to G?
  - Is there an injective map f from V_H to V_G
    - $\{f(u), f(v)\} \in E_G$ holds for any $\{u, v\} \in E_H$



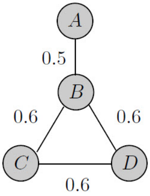
Introduction

We have some rules and restrictions

- Arcs E are weighted using the $\omega : E \to (0, 1)$.
- The shape is a weighted undirected graph $H = (W;F)$, where $F \subseteq W \times W$ with weight function $\rho : F \to (0, 1)$.
- A shape $H = (W;F)$ occurs in an image $G = (V;E)$ when there is an injective function $f : W \to V$. We consider $\{x, y\}$ is an edge in H if and only if $\{(f(x), f(x))\}$ is an edge in G.



Image



Shape

Table of Contents

Integer Linear Model

Input Data

- n = Number of vertices in the Image graph.
- m = Number of vertices in the Shape graph.
- V = Set of vertices in the Image graph. $(V = \{1, .., n\})$
- W = Set of vertices in the Shape graph. $(W = \{1, .., m\})$
- G_{ut} = Weight of the edge (u, t) in the Image graph.
- H_{xy} = Weight of the edge (x, y) in the Shape graph.

Auxiliary data

- $Cost_{ut,xy}$: Cost of matching each edge $(u, t) \in E$ and, $(x, y) \in F$ which are absolute differences of their weights. $Cost_{ut,xy} = |H_{xy} - G_{ut}| \qquad 0 \leq Cost_{ut,xy} \leq 1$
- BG_{ut} : Adjacency matrix of $G \in \mathbb{B}$. $\qquad (u, t) \in E$
- BH_{xy} : Adjacency matrix of $H \in \mathbb{H}$. $\qquad (x, y) \in F$

Integer Linear Model

Decision variables

$$a_{x,u} \in \mathbb{B} : \begin{cases} \textit{True (1)} & \textit{If there is a matching between the node } x \in W \textit{ and } u \in Y. \\ \textit{False (0)} & \textit{Otherwise.} \end{cases}$$

$$b_{xy,ut} \in \mathbb{B} : \begin{cases} \textit{True (1)} & \textit{If there is a matching between the edge } (x,y) \in F \textit{ and } (u,t) \in E. \\ \textit{False (0)} & \textit{Otherwise.} \end{cases}$$

Objective function

$$\textbf{minimize} \quad 0.5 \cdot \sum_{u \in V} \sum_{t \in V} \sum_{x \in W} \sum_{y \in W} Cost_{ut,xy} \cdot b_{xy,ut}$$

Integer Linear Model

Decision Variables

- Every vertex of W should be matched to a unique vertex of V.

$$\sum_{u\in V} a_{x,u} = 1 \qquad \forall i \in W$$

- Every vertex of V should be matched to at most a vertex of W.

$$\sum_{x\in W} a_{x,u} \leq 1 \qquad \forall u \in V$$

- Every arc of $(x, y) \in F$ should be matched to to a unique arc of $(u, t) \in E$.

$$\sum_{u\in V}\sum_{t\in V} b_{xy,ut} \cdot BG_{ut} = BH_{xy} \qquad \forall x \in W,\ \forall y \in W$$

- Two different vertices of V should not be matched to a common vertex of W at the same time.

$$\sum_{u\in V} b_{xy,ut} \cdot BG_{ut} = (a_{x,t} + a_{y,t}) \cdot BH_{xy} \qquad \forall x \in W,\ \forall y \in W,\ \forall t \in V$$

$$\sum_{t\in V} b_{xy,ut} \cdot BG_{ut} = (a_{x,u} + a_{y,u}) \cdot BH_{xy} \qquad \forall x \in W,\ \forall y \in W,\ \forall u \in V$$

- A constraint relates the decision variables a and b together. (if there is no any edge between the nodes u ant t in the image graph, this constraints will be neutralized.

$$\sum_{x\in W} (a_{x,u} + a_{x,t}) - \sum_{x\in W, y\in W} b_{xy,ut} \cdot BH_{xy} \leq 2 - BG_{ut} \qquad \forall t \in V,\ \forall u \in V$$

Table of Contents

Meta-heuristics

I have implemented three different algorithms:

- **Greedy.**
  Greedy algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem
- **Greedy + Local-Search.**
  We will improve our results combining Local-Search with Greedy algorithm together.
- **GRASP (Using Greedy and Local-Search).**
  GRASP (Greedy Randomized Adaptive Search) consists of iterations made up from successive constructions of a greedy randomized solution and subsequent iterative improvements of it through a local search.

Programming language

In the ILP part we have used **CPLEX** and in the Meta-heuristic part to achieve the reasonable execution time, I have chosen **Python** as the programming language. I have used the package **Networkx**, as some basic functions for working with the graphs was needed e.g. ”Node neighborhood” or sorting nodes based on the degree and etc, I have used the package Networkx.

Greedy Algorithm

Greedy algorithm idea

Given a partial solution w, the greedy function $q()$ evaluates the cost of matching a node i to a node in the shape graph. This function evaluate mapping an edge in the the shape graph to the image graph.

Greedy Function

$$\text{q(x,v, w)} = \begin{cases} |cost(x) - cost(v)| & x \in W, v \in V, \quad chechFeasByIsomorphismCond(x, v) \\ \infty, & \text{otherwise} \end{cases}$$

Greedy Pseudo-code

Algorithm 1: Greedy algorithm

```
w ← ∅
sortedVertices ← sortedByDegreeAsc()W
for x in sortedVertices do
    candidateList ← ∅
    for v in F do
        if chechFeasByIsomorphismCond() = False then
            continue
        end
        candidate_cost ← q(x, v, w)
        if candidate_cost ≠ ∞ then
            candidateList ← candidateList ∪ {{x, v}, candidate_cost}
        end
    end
    candidateList ← sorted(candidateList)
    if len(candidateList) = 0 then
        return NO SOLUTION FOUND
    end
    bestCandidate ← candidateList[0]
    w.assign(bestCandidate)
end
return w
```

Local Search

Local search idea

A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution. This is only possible if a neighborhood relation is defined on the search space.

In our case, the local search algorithm starts with a feasible solution given by the greedy method. For each step of the algorithm, the neighborhood is defined as all the possible matching of the node of the shape to a node of Image.

Local Search Pseudo-code

Algorithm 2: Local search algorithm

```
solution ← feasible − greedy − solution
for x in ShapeNodes do
    best_cost ← solution.cost()
    for v in ImageNodes do
        if NodeWasMachecdWaBefore then
            continue
        end
        move ← Move(nodeShape, matchedNodeShape, NodeImage)
        new_cost ← evaluateNeighbor()
        if new_cost ≠ ∞ ∧ new_cost < best_cost then
            best_cost ← new_cost
            best_matched ← new_cost
        end
    end
    bestNeighbor.matched ← best_matched
end
solution ← bestNeighbor.matched
return solution
```

GRASP

GRASP idea

- Solve the problem using a randomized version of the greedy algorithm.
- Improve the solution using local-search.
- Repeat.

The randomized greedy solver uses a restricted candidate list (RCL) to select a feasible candidate among the best ones on each step of the algorithm. To create the RCL we use a threshold value α that limits the cost of the candidates that can be part of the RCL.

RCL in each step of the algorithm:

$$RCL = \{candidate \in feas\_candidates | q(candidate) \leq q^{min} + \alpha(q^{max} - q^{min})\}$$

where q^{max} and q^{min} are respectively the costs of the worst and best candidate in the list of feasible candidates.

GRASP Pseudo-code

Algorithm 3: GRASP algorithm

```
iters ← 0
best_solution ← ∅
while ¬timeout() ∧ iters ≤ max_iters do
    iters ← iters + 1
    greedy_solution ← solver_GRASP.constructSolution(problem, alpha)
    if greedy_solution.is_feasible() then
        local_search_solution ← local_search_solver(greedy_solution)
        if best_solution = ∅ ∨ q(local_search_solution) ≤ q(best_solution) then
            best_solution ← local_search_solution
        end
    end
end
if best_solution = ∅ then
    return NO SOLUTION FOUND
else
    return best_solution
end
```

Table of Contents

Methodology

We have used CPLEX 20.1.0 and all the experiments have been executed on a Laptop with an Intel Core 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz, and 16 GiB RAM, using Windows 10 to compile the meta-heuristic algorithms.
Table **??** shows the parameters passed to the instance generator to generate each of the instances.

	Instances			
	small	**small-medium**	**medium-small**	**medium**
Number of nodes in Image	10	16	20	30
Number of edges in Image	15	17	29	75
Density of Image graph %	33%	14%	15%	17%
Number of nodes in Shape	4	10	15	24
Number of edges in Shape	7	19	18	19
Density of Shape graph %	100%	42%	17%	6%
Load %	40%	63%	75%	80%

Table: Instance generator parameters by instance size.

GRASP alpha tuning

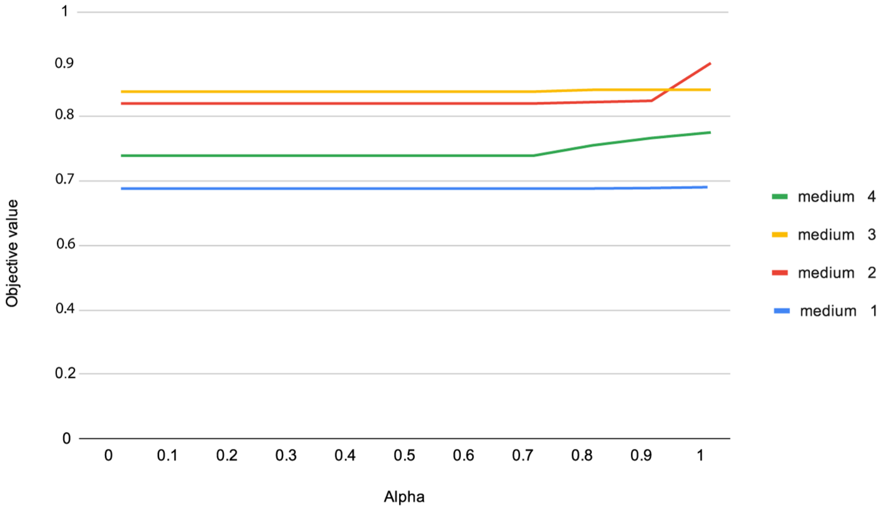


Figure: GRASP: Objective value obtained in the execution of each instance.

Conclusion

OPL performance

In general, CPLEX takes much time to find the optimal solutions for relatively big Instances.
When using Heuristic algorithms the time needed to find a good solution for the same instances is much smaller in comparison. Nevertheless, we've seen that for small instances it could be not the best option to choose the heuristics due to the fact that the same or better results can be obtained relatively fast with CPLEX.

Meta-heuristics performance

When comparing GRASP and GRASP+LocalSearch, for the generated instances, the localSearch solver has the best performance, taking not much time to find clearly better results than the GRASP algorithm.

Thank you