AMMM Final Project

Ali Arabyarmohammadi December 2021

1 Problem statement

In order to develop a shape recognition program, a surveillance company uses a surveillance system consists of a camera, which can take pictures, and a distance sensor which provides auxiliary information. The goal of this system is to determine if a specific shape appears in it. In other words, this system matches a particular pattern in an Image.

- We have an image is modelled with an undirected graph G = *(V;E)*, where each vertex $v \in V$ is a point (pixel) in the image and $E \subseteq V \times V$.
- An edge $\{u, v\} \in E$ is placed between points u and v according to an edge detection algorithm.
- Arcs E are weighted using the $\omega : E \rightarrow (0, 1)$ such that $\omega(u, v)$ is the Euclidean distance between points u and v as measured with the distance sensor and distances are scaled so that they are always less than 1.
- The shape is also modelled with a weighted undirected graph H = *(W;F)*, where $F \subseteq W \times W$ with weight function $\rho : F \rightarrow (0, 1)$.
- A shape H = *(W;F)* occurs in an image G = *(V;E)* when there is an injective function $f : W \rightarrow V$ (called the embedding of H in G) that is edge-preserving: We consider $\{x, y\}$ is an edge in H if and only if $\{(f(x), f(x))\}$ is an edge in G.
- In order to eliminate spurious cases, 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)*.

The goal of this project is, given an image and a shape, to find an optimal embedding according to criterion above.

2 Integer Linear Model

2.1 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.
(G is a symmetric matrix which represents the undirected weighted graph of the Image. Values in the matrix are distances between two vertices (two pixels in the image). If $G_{ut} = 0$ then there is no connection between two vertices.)

H_{xy} = Weight of the edge (x, y) in the Shape graph.
(H is a symmetric matrix which represents the undirected weighted graph of the Shape. Values in the matrix are distances between two vertices (two pixels in the shape). If $H_{xy} = 0$ then there is no connection between two vertices.)

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 (with the values 0 and 1). $\qquad (u,t) \in E$

BH_{xy} : Adjacency matrix of H (with the values 0 and 1). $\qquad (x,y) \in F$

2.2 Decision variables

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

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

2.3 Objective function

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

2.4 Constraints

- 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.

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

3 Meta-heuristics

It is assured that if there is an optimal solution, we will find it while implementing Integer Linear Programming. The issue is that this method takes a long time to solve problems of medium to large size. In order to find a solution in a reasonable time, we will be using heuristics to solve this optimization problem.

We 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.**
  The local search part consists of moving consecutive vertices to another location in the polygonal chain. We will improve our results combining Local-Search with Greedy algorithm together.
- **GRASP (Using Greedy and Local-Search).**
  The greedy randomized adaptive search procedure is a metaheuristic algorithm commonly applied to combinatorial optimization problems. GRASP consists of iterations made up from successive constructions of a greedy randomized solution and subsequent iterative improvements of it through a local search.

Due to the familiarity with the programming language **Python**, this has been chosen for modeling these three algorithms.

3.1 Greedy

A greedy algorithm is an algorithm that follows a heuristic for making the locally optimal choice at each stage. In many problems, a greedy strategy does not produce an optimal solution. Still, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Given a partial solution w, the greedy function $q(.)$ evaluates the cost of matching in edge in the shape graph H to an edge in the image graph G by selecting the a case by checking the absolute differences in the weight of the two edges in the edges E and F. ($x \in W$ and $v \in V$ respectively, are nodes in the shape and image graph.)

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

The function ***chechFeasByIsomorphismCond*** is explained below.

On of the feasibility condition is that the number of neighborhoods for a node in the Image graph should be equal or greater than the number of neighborhood in the Shape graph.

$$degree(v) > degree(x)$$

The function assures that if node x of the sub-graph could possibly match node v of the graph, then for every node y that is adjacent to node x in the sub-graph, it has to be possible to find a node u that is adjacent to node u in the graph. The function also checks if the edge $\{x, y\} \in W$ is an edge in *shape graph* **if and only if** $\{(f(x), f(x))\} \in E$ in the *image graph.*

Algorithm 1: Greedy algorithm

```
$w \leftarrow \varnothing$
$sortedVertices \leftarrow sortedByDegreeAsc()W$
for $x$ in $sortedVertices$ do
    $candidateList \leftarrow \varnothing$
    for $v$ in $F$ do
        if $chechFeasByIsomorphismCond() = False$ then
            continue
        end
        $candidate\_cost \leftarrow q(x, v, w)$
        if $candidate\_cost \neq \infty$ then
            $candidateList \leftarrow candidateList \cup \{\{x, v\}, candidate\_cost\}$
        end
    end
    $candidateList \leftarrow sorted(candidateList)$
    if $len(candidateList) = 0$ then
        return $NO\ SOLUTION\ FOUND$
    end
    $bestCandidate \leftarrow candidateList[0]$
    $w.assign(bestCandidate)$
end
return $w$
```

The function ***sortedByDegreeAsc*** sorts nodes of the **shape** graph based on their degrees in ascending order. It is more efficient when we start iteration from a node with higher connectivity.

3.2 Local Search

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.
Typically, every candidate solution has more than one neighbor solution; the choice of which one to move to is taken using only information about the solutions in the neighborhood of the current one, hence the name local search. 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. A change in the image nodes implies the reevaluation of all nodes. Our local search Algorithm 2 has as many steps as node numbers. To compute the cost of a neighbor solution we reuse the greedy function $q()$.

Algorithm 2: Local search algorithm

```
$solution \leftarrow feasible - greedy - solution$
for $x$ in $ShapeNodes$ do
    $best\_cost \leftarrow solution.cost()$
    for $v$ in $ImageNodes$ do
        if $NodeWasMachecdWaBefore$ then
            $continue$
        end
        $move \leftarrow Move(nodeShape, matchedNodeShape, NodeImage)$
        $new\_cost \leftarrow evaluateNeighbor()$
        if $new\_cost \neq \infty \wedge new\_cost < best\_cost$ then
            $best\_cost \leftarrow new\_cost$
            $best\_matched \leftarrow new\_cost$
        end
    end
    $bestNeighbor.matched \leftarrow best\_matched$
end
$solution \leftarrow bestNeighbor.matched$
return $solution$
```

The function ***"evaluateNeighbor()"*** checks the feasibility using Isomorphism condition and return a new cost (weight).

3.3 GRASP

The greedy randomized adaptive search procedure (also known as GRASP) is a meta-heuristic algorithm commonly applied to combinatorial optimization problems.
The GRASP implementation consists of iterations made up from successive constructions of the a greedy randomized solution and subsequent iterative improvements of it through a local search. 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 limit the cost of the candidates that can be part of the RCL. For each iteration of the greedy algorithm, the RCL is defined as:

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

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

The GRASP Algorithm 3 can be executed a fixed number of times, or run during a given time.

Algorithm 3: GRASP algorithm

```
$iters \leftarrow 0$
$best\_solution \leftarrow \varnothing$
while $\neg timeout() \wedge iters \leq max\_iters$ do
    $iters \leftarrow iters + 1$
    $greedy\_solution \leftarrow solver\_GRASP.constructSolution(problem, alpha)$
    if $greedy\_solution.is\_feasible()$ then
        $local\_search\_solution \leftarrow local\_search\_solver(greedy\_solution)$
        if $best\_solution = \varnothing \vee q(local\_search\_solution) \leq q(best\_solution)$ then
            $best\_solution \leftarrow local\_search\_solution$
        end
    end
end
if $best\_solution = \varnothing$ then
    return NO SOLUTION FOUND
else
    return $best\_solution$
end
```

4 Instance Generator

Three different random graphs was generated named small, small-medium, medium-small and medium using the member of vertices in *image* and *shape* graph. for each instances, the factors *load* and *density* will be investigated to see the feasibility and quality of the solutions.
It is evident that the less *load* we have the high probability that we reach to a feasible solution is higher. In other words, when we have a lower load the image is smaller in proportion to the image and detecting it would be easier and faster. Also, the higher *density* of the graph and image is another factor to make the problem more complicated.

5 Experiments

5.1 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 1 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 1: Instance generator parameters by instance size.

5.2 GRASP alpha tuning

To tune the alpha parameter for the GRASP solver, we have generated 4 additional medium-size instances using the same parameters as the ones presented in Table 1. The GRASP algorithm has been executed 5 times per instance, using different alpha values. Each execution runs 500 times the algorithm. Each execution runs the algorithm for 150 seconds (That's more than 4,000 iterations). In Figure 1 we present the mean of the obtained objective values for each instance and alpha value.

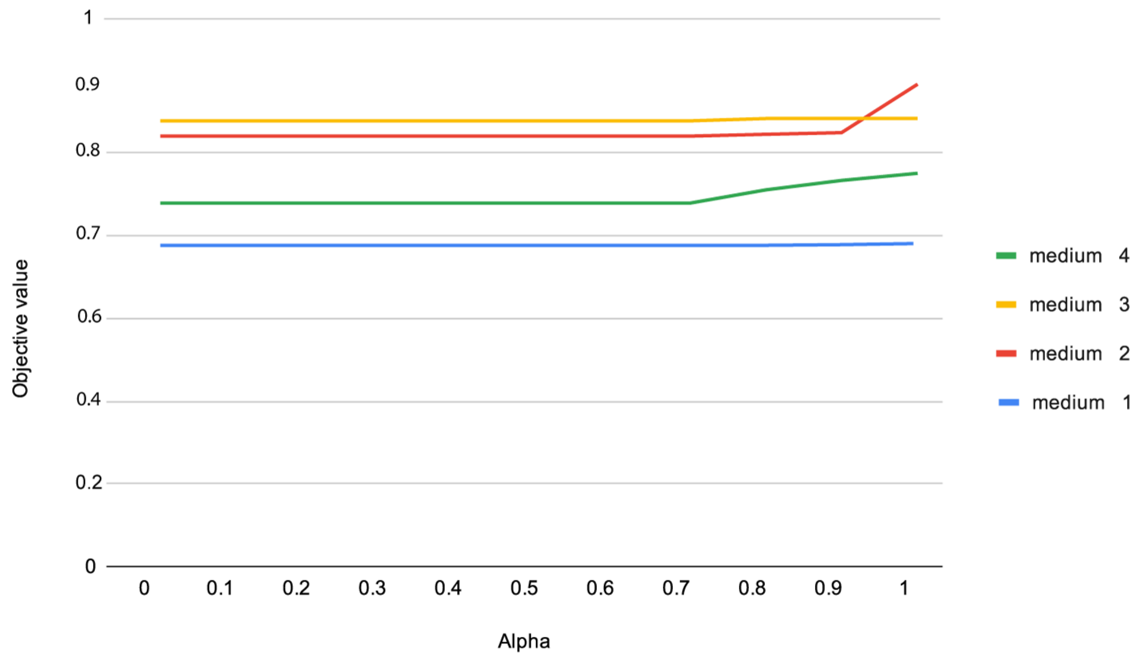

Figure 1: GRASP: Objective value obtained in the execution of each instance.

It seems that the best alpha value is between 0 and 0.7. To keep some randomness without compromising the algorithm's results, we will use 0.4 as the alpha value for the experiments.

5.3 Results

We have run each of the generated instances in CPLEX, and all three implemented meta-heuristic algorithms. For CPLEX we have set the maximum execution time to 30 minutes.. The GRASP algorithm runs 500 times without a time limit, with an alpha value of 0.4.

Objective value for each instance and algorithm. As expected, the local-search solver improves or maintains the results of the greedy solver, the GRASP solver improves or maintains the results of the local-search solver, and CPLEX improves or maintains the results of GRASP. The difference in the objective between the algorithms increases with the instance size.

While CPLEX could have the best results, it also has the worst performance. CPLEX takes more than 30 minutes to solve the huge instance, while the greedy solver needs less than 100 ms.

As the instances are fairly small for the meta-heuristic solvers, we can not appreciate the speedup of the greedy solver over the local search one since almost all the execution time is dedicated to reading the input data. We have executed the greedy and local-search solver with much bigger instances, and we have seen that the local-search doubles the execution time of the greedy solver.We can not claim the same for the GRASP solver in the medium to big instances and the chosen load and sizes of the configuration.

6 Conclusions

We have modeled the given problem and implemented it in an OPL language. We have also developed three meta-heuristic solvers for the problem; greedy, local-search, and GRASP.

To tune the alpha parameter for the GRASP solver, we created four instances.

We have generated increasing size instances, and we have executed them using CPLEX and all three meta-heuristic algorithms.

CPLEX takes minutes to solve fairly small instances that can be solved in milliseconds using the meta-heuristic algorithms. For the generated instances, the local-search solver has the best performance-results ratio, taking roughly two times more than the greedy algorithm to execute while giving results close to GRASP. For the smallest instances, GRASP gave the same results as CPLEX while taking 5-10 less time to execute.

It would be interesting to compare the meta-heuristic algorithms with much larger instances that CPLEX could never solve.