CSE318 Assignment-03
Solving the Max-cut problem by GRASP

Performance Report and Implementation Details

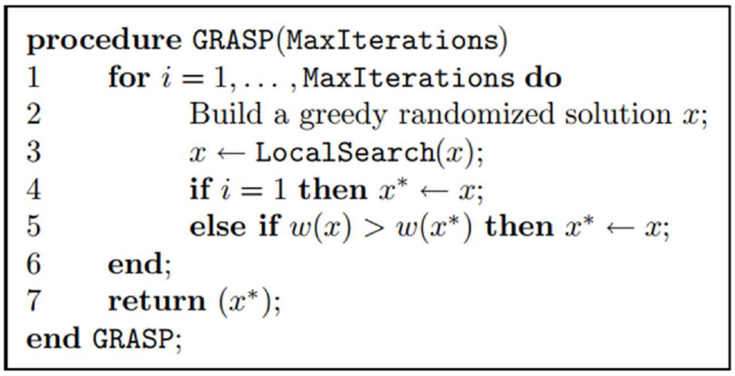
Prepared by
Wasif Jalal (1905084)

Overview

In this assignment the maximum cut of a graph was estimated using the GRASP algorithm by various constructive algorithms and local search operations on constructed approximations. For the provided dataset of 54 graphs, the implemented GRASP algorithm runs 50 iterations of each of three constructive algorithms (randomised, semi-greedy and greedy), each followed by a local search operation.

Implementation Details

The GRASP Algorithm was implemented following the generic algorithm provided in the assignment specification. For the purpose of benchmarks, a constant 50 maximum iterations were used for each graph in the dataset.



```
procedure GRASP(MaxIterations)
    for $i = 1, \ldots,$ MaxIterations do
        Build a greedy randomized solution $x$;
        $x \leftarrow$ LocalSearch($x$);
        if $i = 1$ then $x^* \leftarrow x$;
        else if $w(x) > w(x^*)$ then $x^* \leftarrow x$;
    end;
    return ($x^*$);
end GRASP;
```

Constructive Algorithm Heuristics

Three different constructive algorithms were used both individually and within the GRASP algorithm. The three algorithms were implemented deriving from a single greedy randomised approach as recommended, by just varying the α-parameter . The generic greedy randomised approach is based on the following algorithm suggested in *Optimization by GRASP* by Mauricio G.C. Resende.

```
begin SEMI-GREEDY-MAXCUT;
1  Generate at random a real-valued parameter $\alpha \in [0,1]$;
2  $w_{min} \leftarrow \min\{w_{ij} \,:\, (i,j) \in U\}$;
3  $w^{max} \leftarrow \max\{w_{ij} \,:\, (i,j) \in U\}$;
4  $\mu \leftarrow w_{min} + \alpha \cdot (w^{max} - w_{min})$;
5  $\mathrm{RCL}_e \leftarrow \{(i,j) \in U \,:\, w_{ij} \geq \mu\}$;
6  Select edge $(i^*, j^*)$ at random from $\mathrm{RCL}_e$;
7  $X \leftarrow \{i^*\}$;
8  $Y \leftarrow \{j^*\}$;
9  while $X \cup Y \neq V$ do
10     $V' \leftarrow V \setminus (X \cup Y)$;
11     forall $v \in V'$ do
12         $\sigma_X(v) \leftarrow \sum_{u \in Y} w_{vu}$;
13         $\sigma_Y(v) \leftarrow \sum_{u \in X} w_{vu}$;
14     end-forall;
15     $w_{min} \leftarrow \min\{\min_{v \in V'} \sigma_X(v),\ \min_{v \in V'} \sigma_Y(v)\}$;
16     $w^{max} \leftarrow \max\{\max_{v \in V'} \sigma_X(v),\ \max_{v \in V'} \sigma_Y(v)\}$;
17     $\mu \leftarrow w_{min} + \alpha \cdot (w^{max} - w_{min})$;
18     $\mathrm{RCL}_v \leftarrow \{v \in V' \,:\, \max\{\sigma_X(v), \sigma_Y(v)\} \geq \mu\}$;
19     Select vertex $v^*$ at random from $\mathrm{RCL}_v$;
20     if $\sigma_X(v^*) > \sigma_Y(v^*)$ then
21         $X \leftarrow X \cup \{v^*\}$;
22     else
23         $Y \leftarrow Y \cup \{v^*\}$;
24     end-if;
25 end-while;
26 $S \leftarrow X$;
27 $\bar{S} \leftarrow Y$;
28 return $(S, \bar{S})$, $w(S, \bar{S})$;
end SEMI-GREEDY-MAXCUT.
```

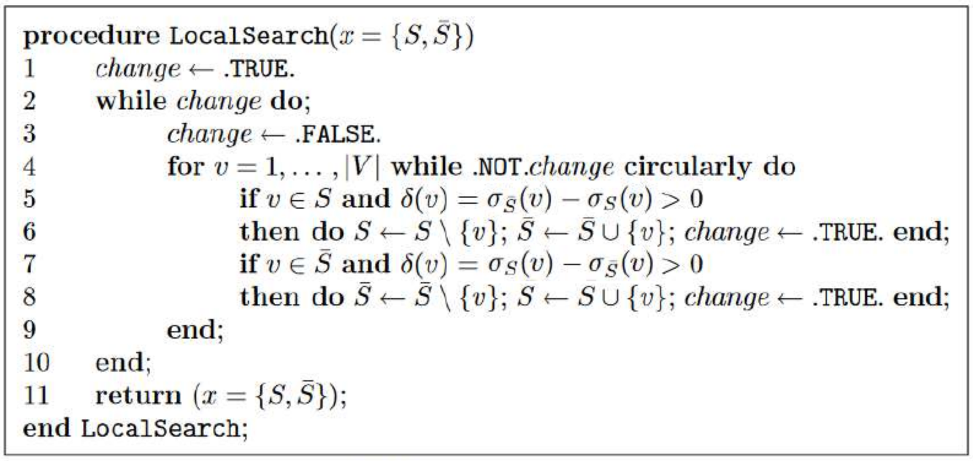
The reason why simply varying the α-parameter changes the nature of the algorithm can be seen easily from the definition of the variable μ. Setting α=0, causes mu to be the same as w_{min} which is the lowest of the sums of edge weights connected to a vertex, thus allowing any vertex to be picked for assigning to partitions, based only on the sum of edge weights of each vertex to the vertices in either partition. The greedy heuristic brought on by μ is completely eliminated. Whereas setting α=0 causes μ to be the same as w^{max} , which is the defining characteristic of a completely greedy heuristic. Varying α anywhere between, makes the heuristic half random, and half greedy. To prevent the semi-greedy heuristic from behaving similarly to the greedy and random approaches, the value of α has been kept between 0.1 and 0.9.

```
double Graph::semi_greedy_maxcut (set<int>& S, set<int>& _S) {
    double alpha = 0.01 * (10 + rand()%80);
    return greedy_random_maxcut (alpha, S, _S);
};

double Graph::simple_greedy_maxcut (set<int>& S, set<int>& _S) {
    return greedy_random_maxcut (1.0, S, _S);
};

double Graph::random_maxcut (set<int>& S, set<int>& _S) {
    return greedy_random_maxcut (0.0, S, _S);
};
```

The local search phase of GRASP was implemented following the algorithm provided in the assignment specification. This is a simple local search operator that moves vertices between partitions based on their sum of edge weights to the the opposite partition.



```
procedure LocalSearch($x = \{S, \bar{S}\}$)
1    $change \leftarrow$ .TRUE.
2    while $change$ do;
3        $change \leftarrow$ .FALSE.
4        for $v = 1, \ldots, |V|$ while .NOT.$change$ circularly do
5            if $v \in S$ and $\delta(v) = \sigma_{\bar{S}}(v) - \sigma_S(v) > 0$
6            then do $S \leftarrow S \setminus \{v\}$; $\bar{S} \leftarrow \bar{S} \cup \{v\}$; $change \leftarrow$ .TRUE. end;
7            if $v \in \bar{S}$ and $\delta(v) = \sigma_S(v) - \sigma_{\bar{S}}(v) > 0$
8            then do $\bar{S} \leftarrow \bar{S} \setminus \{v\}$; $S \leftarrow S \cup \{v\}$; $change \leftarrow$ .TRUE. end;
9        end;
10   end;
11   return ($x = \{S, \bar{S}\}$);
end LocalSearch;
```

For benchmarking, all 54 graphs in the supplied dataset have been used. At first, only the three constructive algorithms are used to estimate feasible solutions. After that GRASP is run separately using each of the three constructions with 50 iterations.

Performance Benchmarks

The GRASP max-cut program (max_cut.cpp), when compiled and run, produces several metrics in its output using all the constructive algorithms. A script (generate_table.sh) was used to generate a CSV formatted table (table.csv) for all the benchmark graphs. The .csv file was imported to a spreadsheet for further analysis of the performance of the implementation, introducing some additional metrics using the best known solutions or upper bounds supplied in the specifications. The *accuracy* of a process is just the ratio of the maximum cut estimated by it to the provided upper bound. Please refer to the spreadsheet to view the methods of calculating the metrics. A brief summary table made through querying the benchmark data table is shown below:

	Metric	Construction (only) accuracy (avg.)	Local Search accuracy (avg.)	Local Search iterations (mean)	GRASP best accuracy	Relative improvement with GRASP iterations (avg.)
Constructive algorithm						
randomised		82.10%	85.53%	137.19	86.94%	1.83%
simple_greedy		88.42%	89.74%	56.83	90.76%	1.23%
semi-greedy		86.03%	88.11%	99.72	90.30%	2.91%
Average		85.52%	87.79%	97.91	89.34%	1.99%

From the benchmarks it seems that the simple greedy constructive algorithm is the most performant in all regards, followed by the semi-greedy, and lastly the random approach. This relationship is maintained in all the metrics. The greedy algorithm is most accurate as a construction method, as well as when used within GRASP.

The ranks of accuracy are affirmed from the local search iterations column as well, where it is seen that a greedy construction requires the least number of local search moves to find a potential "better" solution. The last column indicates the relative improvement brought on by GRASP over unaided constructions. It is just the percentage of increase in the ratio of accuracy of the GRASP best solution, to the accuracy of the unaided constructive algorithm.

There it can be observed that a random construction gains the most improvement, whereas a greedy solution is already considered "good" enough by the local search operator, thus requiring less moves and gaining comparatively less improvement. On average, all the executions of 50-iteration GRASP with local search bring an improvement of almost 2% over the feasible solutions constructed by randomised, semi-greedy or greedy approach when comparing the final accuracy of the solutions. Thus in all cases it can be confirmed that GRASP iterations are indeed an improvement on the estimation process on an NP-hard problem.

A static copy of the spreadsheet, derived from the table generated by the program's outputs, is attached with this report. It contains the outputs obtained from all the approaches and the additional metrics calculated from the data.