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, . . . , MaxIterations do
        Build a greedy randomized solution x;
        x ← LocalSearch(x);
        if i = 1 then x* ← x;
        else if w(x) > w(x*) then x* ← 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;
Generate at random a real-valued parameter $\alpha \in [0,1]$;
$w_{min} \leftarrow \min\{w_{ij} \ : \ (i,j) \in U\}$;
$w^{max} \leftarrow \max\{w_{ij} \ : \ (i,j) \in U\}$;
$\mu \leftarrow w_{min} + \alpha \cdot (w^{max} - w_{min})$;
$\mathrm{RCL}_e \leftarrow \{(i,j) \in U \ : \ w_{ij} \geq \mu\}$;
Select edge (i^*, j^*) at random from RCL_e;
$X \leftarrow \{i^*\}$;
$Y \leftarrow \{j^*\}$;
while $X \cup Y \neq V$ **do**
  $V' \leftarrow V \setminus (X \cup Y)$;
  forall $v \in V'$ **do**
    $\sigma_X(v) \leftarrow \sum_{u \in Y} w_{vu}$;
    $\sigma_Y(v) \leftarrow \sum_{u \in X} w_{vu}$;
  end-forall;
  $w_{min} \leftarrow \min\{\min_{v \in V'} \sigma_X(v), \ \min_{v \in V'} \sigma_Y(v)\}$;
  $w^{max} \leftarrow \max\{\max_{v \in V'} \sigma_X(v), \ \max_{v \in V'} \sigma_Y(v)\}$;
  $\mu \leftarrow w_{min} + \alpha \cdot (w^{max} - w_{min})$;
  $\mathrm{RCL}_v \leftarrow \{v \in V' \ : \ \max\{\sigma_X(v), \sigma_Y(v)\} \geq \mu\}$;
  Select vertex v^* at random from RCL_v;
  if $\sigma_X(v^*) > \sigma_Y(v^*)$ **then**
    $X \leftarrow X \cup \{v^*\}$;
  else
    $Y \leftarrow Y \cup \{v^*\}$;
  end-if;
end-while;
$S \leftarrow X$;
$\bar{S} \leftarrow Y$;
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, S̄})
    change ← .TRUE.
    while change do;
        change ← .FALSE.
        for v = 1, ... , |V| while .NOT.change circularly do
            if v ∈ S and δ(v) = σ_S̄(v) − σ_S(v) > 0
            then do S ← S \ {v}; S̄ ← S̄ ∪ {v}; change ← .TRUE. end;
            if v ∈ S̄ and δ(v) = σ_S(v) − σ_S̄(v) > 0
            then do S̄ ← S̄ \ {v}; S ← S ∪ {v}; change ← .TRUE. end;
        end;
    end;
    return (x = {S, S̄});
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.

Problem			Constructive Algorithm			Local Search		GRASP			Known best solution or upper bound	GRASP Accuracy	Average accuracy with local search	Relative improvement with GRASP iterations	Construction-only Accuracy		
Input Filename	|V|	|E|	randomised	simple_greedy	semi-greedy	Iterations	Best Value	Iterations	Best Value	Constructive Algorithm					random	greedy	semi-greedy
						163	11358		11420	randomised	12078	94.55%	94.04%	0.55%			
set1/g1.rud	800	19176	11025	11297	11106	74	11394	50	11463	simple_greedy	12078	94.91%	94.34%	0.61%	91.28%	93.53%	91.95%
						124	11375		11462	semi-greedy	12078	94.90%	94.18%	0.76%			
						162	11367		11461	randomised	12084	94.84%	94.07%	0.83%			
set1/g2.rud	800	19176	11009	11236	11113	74	11405	50	11474	simple_greedy	12084	94.95%	94.38%	0.60%	91.10%	92.98%	91.96%
						118	11369		11459	semi-greedy	12084	94.83%	94.08%	0.79%			
						160	11353		11414	randomised	12077	94.51%	94.01%	0.54%			
set1/g3.rud	800	19176	11035	11276	11107	69	11395	50	11474	simple_greedy	12077	95.01%	94.35%	0.69%	91.37%	93.37%	91.97%
						123	11374		11463	semi-greedy	12077	94.92%	94.18%	0.78%			
						171	11383		11489	randomised							
set1/g4.rud	800	19176	11037	11272	11132	74	11405	50	11472	simple_greedy							
						125	11391		11522	semi-greedy							
						165	11355		11462	randomised							
set1/g5.rud	800	19176	11007	11285	11088	72	11414	50	11490	simple_greedy							
						124	11382		11480	semi-greedy							
						175	1906		1975	randomised							
set1/g6.rud	800	19176	1532	1738	1577	84	1936	50	2009	simple_greedy							
						149	1913		2000	semi-greedy							
						173	1743		1807	randomised							
set1/g7.rud	800	19176	1376	1573	1415	94	1781	50	1856	simple_greedy							
						145	1740		1823	semi-greedy							
						180	1756		1846	randomised							
set1/g8.rud	800	19176	1379	1580	1456	84	1771	50	1859	simple_greedy							
						146	1751		1858	semi-greedy							
						169	1780		1877	randomised							
set1/g9.rud	800	19176	1420	1634	1497	88	1823	50	1901	simple_greedy							
						154	1799		1886	semi-greedy							
						176	1739		1810	randomised							
set1/g10.rud	800	19176	1353	1630	1425	86	1765	50	1834	simple_greedy							
						145	1738		1819	semi-greedy							
						16	441		462	randomised	627	73.68%	70.33%	4.76%			
set1/g11.rud	800	1600	409	464	423	6	484	50	502	simple_greedy	627	80.06%	77.19%	3.72%	65.23%	74.00%	67.46%
						13	455		494	semi-greedy	627	78.79%	72.57%	8.57%			
						16	428		454	randomised	621	73.11%	68.92%	6.07%			
set1/g12.rud	800	1600	397	448	413	5	473	50	492	simple_greedy	621	79.23%	76.17%	4.02%	63.93%	72.14%	66.51%
						14	439		476	semi-greedy	621	76.65%	70.69%	8.43%			
						19	456		476	randomised	645	73.80%	70.70%	4.39%			
set1/g13.rud	800	1600	418	474	428	6	498	50	518	simple_greedy	645	80.31%	77.21%	4.02%	64.81%	73.49%	66.36%
						15	467		508	semi-greedy	645	78.76%	72.40%	8.78%			
						49	2936		2957	randomised	3187	92.78%	92.12%	0.72%			
set1/g14.rud	800	4694	2865	2943	2933	21	2971	50	2992	simple_greedy	3187	93.88%	93.22%	0.71%	89.90%	92.34%	92.03%
						25	2965		2984	semi-greedy	3187	93.63%	93.03%	0.64%			
						50	2919		2942	randomised	3169	92.84%	92.11%	0.79%			
set1/g15.rud	800	4661	2845	2915	2913	21	2947	50	2962	simple_greedy	3169	93.47%	92.99%	0.51%	89.78%	91.98%	91.92%
						25	2946		2969	semi-greedy	3169	93.69%	92.96%	0.78%			
						49	2922		2956	randomised	3172	93.19%	92.12%	1.16%			
set1/g16.rud	800	4672	2854	2940	2917	23	2954	50	2980	simple_greedy	3172	93.95%	93.13%	0.88%	89.97%	92.69%	91.96%
						26	2949		2977	semi-greedy	3172	93.85%	92.97%	0.95%			
						51	2919		2953	randomised							
set1/g17.rud	800	4667	2854	2932	2907	23	2949	50	2963	simple_greedy							
						26	2949		2971	semi-greedy							
						78	831		866	randomised							
set1/g18.rud	800	4694	694	839	721	31	881	50	919	simple_greedy							
						65	845		901	semi-greedy							
						77	746		798	randomised							
set1/g19.rud	800	4661	611	795	642	32	799	50	836	simple_greedy							
						70	757		805	semi-greedy							
						73	770		816	randomised							
set1/g20.rud	800	4672	652	785	673	32	830	50	878	simple_greedy							
						65	787		845	semi-greedy							
						82	771		813	randomised							
set1/g21.rud	800	4667	634	755	669	34	820	50	862	simple_greedy							
						69	782		838	semi-greedy							
						260	12805		12880	randomised	14123	91.20%	90.67%	0.59%			
set1/g22.rud	2000	19990	12365	12775	12620	98	12958	50	13044	simple_greedy	14123	92.36%	91.75%	0.66%	87.55%	90.46%	89.36%
						169	12882		13011	semi-greedy	14123	92.13%	91.21%	1.00%			
						257	12822		12936	randomised	14129	91.56%	90.75%	0.89%			
set1/g23.rud	2000	19990	12352	12784	12631	94	12955	50	13043	simple_greedy	14129	92.31%	91.69%	0.68%	87.42%	90.48%	89.40%
						174	12881		12984	semi-greedy	14129	91.90%	91.17%	0.80%			
						256	12818		12925	randomised	14131	91.47%	90.71%	0.83%			
set1/g24.rud	2000	19990	12354	12819	12642	97	12952	50	13018	simple_greedy	14131	92.12%	91.66%	0.51%	87.42%	90.72%	89.46%
						173	12897		12991	semi-greedy	14131	91.93%	91.27%	0.73%			
						254	12820		12915	randomised							
set1/g25.rud	2000	19990	12368	12807	12629	93	12950	50	13033	simple_greedy							
						175	12885		12991	semi-greedy							
						255	12811		12930	randomised							
set1/g26.rud	2000	19990	12347	12790	12609	97	12945	50	13030	simple_greedy							
						175	12883		13006	semi-greedy							
						270	2811		2912	randomised							
set1/g27.rud	2000	19990	2302	2594	2405	126	2907	50	2975	simple_greedy							
						236	2834		2958	semi-greedy							
						276	2784		2897	randomised							
set1/g28.rud	2000	19990	2298	2673	2394	127	2871	50	2985	simple_greedy							
						231	2801		2893	semi-greedy							

Problem			Constructive Algorithm			Local Search		GRASP			Known best solution or upper bound	GRASP Accuracy	Average accuracy with local search	Relative improvement with GRASP iterations	Construction-only Accuracy		
Input Filename	|V|	|E|	randomised	simple_greedy	semi-greedy	Iterations	Best Value	Iterations	Best Value	Constructive Algorithm					random	greedy	semi-greedy
set1/g29.rud	2000	19990	2352	2769	2499	270	2868	50	2994	randomised							
						134	2963		3032	simple_greedy							
						236	2898		3026	semi-greedy							
set1/g30.rud	2000	19990	2384	2785	2489	265	2864	50	2981	randomised							
						125	2967		3053	simple_greedy							
						234	2888		3003	semi-greedy							
set1/g31.rud	2000	19990	2280	2606	2384	274	2777	50	2889	randomised							
						133	2885		2961	simple_greedy							
						230	2818		2948	semi-greedy							
set1/g32.rud	2000	4000	1021	1192	1058	41	1099	50	1152	randomised	1560	73.85%	70.45%	4.82%	65.45%	76.41%	67.82%
						13	1209		1242	simple_greedy	1560	79.62%	77.50%	2.73%			
						33	1120		1216	semi-greedy	1560	77.95%	71.79%	8.57%			
set1/g33.rud	2000	4000	977	1152	1021	43	1068	50	1102	randomised	1537	71.70%	69.49%	3.18%	63.57%	74.95%	66.43%
						12	1196		1226	simple_greedy	1537	79.77%	77.81%	2.51%			
						36	1092		1192	semi-greedy	1537	77.55%	71.05%	9.16%			
set1/g34.rud	2000	4000	974	1148	1032	45	1066	50	1106	randomised	1541	71.77%	69.18%	3.75%	63.21%	74.50%	66.97%
						12	1194		1218	simple_greedy	1541	79.04%	77.48%	2.01%			
						36	1083		1226	semi-greedy	1541	79.56%	70.28%	13.20%			
set1/g35.rud	2000	11778	7185	7377	7355	126	7358	50	7392	randomised	8000	92.40%	91.98%	0.46%	89.81%	92.21%	91.94%
						53	7446		7478	simple_greedy	8000	93.48%	93.08%	0.43%			
						64	7434		7471	semi-greedy	8000	93.39%	92.93%	0.50%			
set1/g36.rud	2000	11766	7166	7359	7355	124	7354	50	7400	randomised	7996	92.55%	91.97%	0.63%	89.62%	92.03%	91.98%
						54	7438		7476	simple_greedy	7996	93.50%	93.02%	0.51%			
						62	7429		7486	semi-greedy	7996	93.62%	92.91%	0.77%			
set1/g37.rud	2000	11785	7190	7378	7365	127	7364	50	7409	randomised	8009	92.51%	91.95%	0.61%	89.77%	92.12%	91.96%
						53	7447		7482	simple_greedy	8009	93.42%	92.98%	0.47%			
						63	7439		7478	semi-greedy	8009	93.37%	92.88%	0.52%			
set1/g38.rud	2000	11779	7181	7375	7366	129	7355	50	7382	randomised							
						53	7445		7479	simple_greedy							
						66	7434		7471	semi-greedy							
set1/g39.rud	2000	11778	1677	2095	1745	196	2014	50	2065	randomised							
						75	2131		2180	simple_greedy							
						177	2029		2151	semi-greedy							
set1/g40.rud	2000	11766	1640	1978	1714	201	1979	50	2063	randomised							
						76	2118		2199	simple_greedy							
						174	2005		2118	semi-greedy							
set1/g41.rud	2000	11785	1661	1996	1731	192	1994	50	2098	randomised							
						74	2119		2156	simple_greedy							
						169	2020		2146	semi-greedy							
set1/g42.rud	2000	11779	1730	2120	1799	195	2075	50	2158	randomised							
						77	2199		2255	simple_greedy							
						174	2093		2214	semi-greedy							
set1/g43.rud	1000	9990	6179	6422	6316	128	6396	50	6476	randomised	7027	92.16%	91.02%	1.25%	87.93%	91.39%	89.88%
						50	6459		6508	simple_greedy	7027	92.61%	91.92%	0.76%			
						82	6435		6498	semi-greedy	7027	92.47%	91.58%	0.98%			
set1/g44.rud	1000	9990	6174	6373	6319	123	6390	50	6450	randomised	7022	91.85%	91.00%	0.94%	87.92%	90.76%	89.99%
						50	6461		6530	simple_greedy	7022	92.99%	92.01%	1.07%			
						80	6433		6505	semi-greedy	7022	92.64%	91.61%	1.12%			
set1/g45.rud	1000	9990	6162	6416	6312	123	6395	50	6465	randomised	7020	92.09%	91.10%	1.09%	87.78%	91.40%	89.91%
						50	6463		6535	simple_greedy	7020	93.09%	92.07%	1.11%			
						81	6437		6498	semi-greedy	7020	92.56%	91.70%	0.95%			
set1/g46.rud	1000	9990	6170	6392	6336	128	6399	50	6464	randomised							
						48	6462		6508	simple_greedy							
						77	6437		6497	semi-greedy							
set1/g47.rud	1000	9990	6184	6395	6336	123	6397	50	6470	randomised							
						50	6465		6519	simple_greedy							
						79	6437		6495	semi-greedy							
set1/g48.rud	3000	6000	4913	6000	6000	74	5084	50	5178	randomised	6000	86.30%	84.73%	1.85%	81.88%	100.00%	100.00%
						1	5993		6000	simple_greedy	6000	100.00%	99.88%	0.12%			
						1	5984		6000	semi-greedy	6000	100.00%	99.73%	0.27%			
set1/g49.rud	3000	6000	4891	6000	6000	75	5068	50	5126	randomised	6000	85.43%	84.47%	1.14%	81.52%	100.00%	100.00%
						1	5994		6000	simple_greedy	6000	100.00%	99.90%	0.10%			
						1	5988		6000	semi-greedy	6000	100.00%	99.80%	0.20%			
set1/g50.rud	3000	6000	4918	5874	5843	76	5080	50	5182	randomised	5988	86.54%	84.84%	2.01%	82.13%	98.10%	97.58%
						1	5871		5878	simple_greedy	5988	98.16%	98.05%	0.12%			
						2	5846		5876	semi-greedy	5988	98.13%	97.63%	0.51%			
set1/g51.rud	1000	5909	3588	3709	3685	62	3688	50	3712	randomised							
						28	3729		3754	simple_greedy							
						33	3726		3756	semi-greedy							
set1/g52.rud	1000	5916	3610	3711	3691	63	3693	50	3722	randomised							
						27	3735		3755	simple_greedy							
						32	3731		3757	semi-greedy							
set1/g53.rud	1000	5914	3599	3704	3694	63	3689	50	3709	randomised							
						27	3732		3756	simple_greedy							
						32	3724		3755	semi-greedy							
set1/g54.rud	1000	5916	3608	3683	3688	60	3692	50	3728	randomised							
						27	3731		3752	simple_greedy							
						32	3727		3755	semi-greedy							