Summary of Various Construction Algorithms and the performance of GRASP with respect to each of the construction algorithms and local search optimization

Offline 3 Summary

Course: CSE 318

Sadif Ahmed
Std Id: 1905058

In this offline the task was to implement GRASP along with the construction algorithm and local search optimization algorithm in order to solve the max cut problem. After implementation, we were instructed to run these on a given benchmark data set of 54 graphs with various vertex counts and edge counts. The summary of the results of construction algorithms are recorded in the files named "construction.txt" and "construction.csv". The summary and detailed results of the GRASP algorithm are recorded in the files "best.txt" , "best.csv" , "grasp.txt" , "grasp.csv". Short description of the files and the construction algorithms of the files are given below:

Construction.txt & Construction.csv:

The results of these files show the max cut value obtained by running greedy, greedy2, greedy3 algorithms once and taking a 10 run average of algorithms semi-greedy, randomized , randomized2, randomized3 and randomized4 for each of the 54 graphs.

Grasp.txt & Grasp.csv:

These files are generated through running Grasp and local search with 8 construction algorithms one at a time for each graph. For each run I stored the input graph number, input graph vertex count, edge count, avg. local search value over number of iterations, avg. local search iterations over number of iterations, number of GRASP iterations and GRASP max cut value. This file has 54*8=432 unique results sets.

Best.txt & Best.csv:

This file is the collection of best local search and GRASP values for each input graph. Here I have stored the input graph info, best local search value and its corresponding construction and avg. iteration count over number of iterations, best GRASP value and its corresponding

construction and number of iteration and the currently known upper bounds of some input graphs.

Implemented Construction Algorithms:

Greedy Algorithm:

Here I picked the heaviest edge of the graph and initialized set X and set Y with one of the vertexes of the edge. The I ran a loop for the remaining vertexes and for each vertex v, I checked its sigma_x [the max cut contribution of v if it were part of set X] and sigma_y [the max cut contribution of v if it were part of set Y].Now if sigma_x >=sigma_y , I inserted v to set X and if sigma_x<sigma_y then I inserted v to set Y. After the loop all vertices will have been inserted into respective sets and then I calculated the weight of the cut.

Greedy2 Algorithm:

Here at first I sorted the list of edges of the graph. Then I picked the heaviest edge of the graph and initialized set X and set Y with one of the vertexes of the edge. Then I ran a loop for the sorted list of edges and for each edge vertex u, I checked if it has been inserted before. If not then I calculated sigma_x [the max cut contribution of v if it were part of set X] and sigma_y[the max cut contribution of u if it were part of set Y].Now if sigma_x >=sigma_y , I inserted u to set X and if sigma_x<sigma_y then I inserted u to set Y. At the beginning of the loop I checked whether any vertex is left to be inserted yet. If there are none then the loop will break before looping through the whole sorted list of edges. After the loop all vertices will have been inserted into respective sets and then I calculated the weight of the cut.

Semi-Greedy Algorithm:

```
begin SEMI-GREEDY-MAXCUT;
1   Generate at random a real-valued parameter α ∈ [0,1];
2   w_min ← min{w_ij : (i,j) ∈ U};
3   w^max ← max{w_ij : (i,j) ∈ U};
4   μ ← w_min + α·(w^max − w_min);
5   RCL_e ← {(i,j) ∈ U : w_ij ≥ μ};
6   Select edge (i*,j*) at random from RCL_e;
7   X ← {i*};
8   Y ← {j*};
9   while X ∪ Y ≠ V do
10      V' ← V \ (X ∪ Y);
11      forall v ∈ V' do
12          σ_X(v) ← Σ_{u∈Y} w_vu;
13          σ_Y(v) ← Σ_{u∈X} w_vu;
14      end-forall;
15      w_min ← min{min_{v∈V'} σ_X(v), min_{v∈V'} σ_Y(v)};
16      w^max ← max{max_{v∈V'} σ_X(v), max_{v∈V'} σ_Y(v)};
17      μ ← w_min + α·(w^max − w_min);
18      RCL_v ← {v ∈ V' : max{σ_X(v), σ_Y(v)} ≥ μ};
19      Select vertex v* at random from RCL_v;
20      if σ_X(v*) > σ_Y(v*) then
21          X ← X ∪ {v*};
22      else
23          Y ← Y ∪ {v*};
24      end-if;
25  end-while;
26  S ← X;
27  S̄ ← Y;
28  return (S,S̄), w(S,S̄);
end SEMI-GREEDY-MAXCUT.
```

This is the algorithm from the book “Optimization by GRASP” by Mauricio G.C. Resende. This algorithm has integrated randomness with a generic greedy approach to calculate maxcut. So here we have maintained two RCL sets one of edges and one of vertexes. At first we constructed a RCL of edges that satisfy a semi-greedy randomized weight criterion. Then we choose an edge from the constructed RCL and initialize set X and set Y with it. Then we run a while loop for the remaining vertexes. At each iteration of the while loop we calculate RCL of the vertices and randomly choose one vertex from the RCL and insert its corresponding set depending on a greedy logic checking.

Greedy3 Algorithm:

This algorithm is the randomness stripped version of the previously mentioned Semi-Greedy Algorithm. If we alter the vertex and edge selection criterion accordingly (setting alpha = 1), the RCL sets will only have the greedy candidates. If we choose from such RCL, then our result will always be maximized using greedy logic.

Randomized Algorithm:

This algorithm is similar to the greedy2 approach where I stripped the greediness (such as sorting and sigma calculation and checks) and included randomness in choosing edges to insert into the cut.

Randomized2 Algorithm:

This algorithm traverses the list of vertexes and randomly adds a vertex to a cut with 50 percent probability.

Randomised3 Algorithm:

This algorithm traverses a sorted list of edges and add each edge's vertex to set X or set Y through a random selection. The loop ends when all of the vertexes have been included in the cut.

Randomised4 Algorithm:

This algorithm is similar to the Semi-Greedy algorithm by stripping its greediness and adding randomness in it. If we alter the vertex and edge selection criterion accordingly (setting alpha = 0), the RCL sets will have random candidates. If we choose from such RCL, then our result will always be randomized.

Local Search Algorithm:

```
procedure LocalSearch(x = {S, S̄})
1     change ← .TRUE.
2     while change do;
3           change ← .FALSE.
4           for v = 1, ..., |V| while .NOT.change circularly do
5                 if v ∈ S and δ(v) = σ_S(v) − σ_S̄(v) > 0
6                 then do S ← S \ {v}; S̄ ← S̄ ∪ {v}; change ← .TRUE. end;
7                 if v ∈ S̄ and δ(v) = σ_S̄(v) − σ_S(v) > 0
8                 then do S̄ ← S̄ \ {v}; S ← S ∪ {v}; change ← .TRUE. end;
9           end;
10    end;
11    return (x = {S, S̄});
end LocalSearch;
```

GRASP Algorithm:

```
procedure GRASP(MaxIterations)
1     for i = 1, ..., MaxIterations do
2           Build a greedy randomized solution x;
3           x ← LocalSearch(x);
4           if i = 1 then x* ← x;
5           else if w(x) > w(x*) then x* ← x;
6     end;
7     return (x*);
end GRASP;
```