CSE 318 Assignment-03: Solving the Max-cut problem by GRASP

Student ID: 1905003, Section: A1

A brief report which summarizes and explains the output of the implemented solution. In this report, it is briefly explained which greedy or semi-greedy techniques or local search operators were implemented.

8/18/2023

Program/Implementation:

```
// GRASP with local search algorithm to find a Max-Cut solution
vector<int> grasp_max_cut(const vector<vector<int>>& graph, int max_iter_grasp) {
    vector<int> bestSolution;
    int bestCutValue = INT_MIN;

    for (int i = 0; i < max_iter_grasp; i++) {
        cout << "GRASP iteration no. : " << i << endl;

        // vector<int> currentSolution = randomized_construction(graph);
        // vector<int> currentSolution = greedy_construction(graph);
        vector<int> currentSolution = semi_greedy_construction(graph);

        vector<int> improvedSolution = local_search_max_cut(graph, currentSolution); // Use local search for improvement
        int currentCutValue = calculateCut(graph, improvedSolution);

        if (currentCutValue > bestCutValue) {
            bestCutValue = currentCutValue;
            bestSolution = improvedSolution;
        }
    }

    return bestSolution;
}
```

Figure 1 GRASP implementation

In the implementation for GRASP, the construction phase can be implemented with three heuristic – randomized, greedy and semi-greedy. The number of iterations is obtained via the parameter *max_iter_grasp*. After getting currentSolution from the construction phase heuristic, the solution is improved by calling the method-

```
vector<int> local_search_max_cut(const vector<vector<int>>& graph, vector<int> initialPartition)
```

This function takes the graph and an `initialSolution` as arguments from the method in Figure 1. The `initalSolution` is actually a vector describing which partition each vertex is currently included in [0 or 1]. It loops over all the vertices and decides which partition to place them depending on how much cut-weight they contribute until no further improvement of any vertex is possible. The cut-weight contribution of a vertex v is stored in local variables `sigma_0` and `sigma_1.` The integer `i` keeps count of the number of iterations of this method. `k` and `cutValue` are two global variables for counting total number of local search iterations and total cut-weight for all GRASP iterations respectively.

```
// Local search algorithm to find a Max-Cut solution
vector<int> local_search_max_cut(const vector<vector<int>>& graph, vector<int> initialPartition) {
    int n = graph.size();
    vector<int> currentPartition = initialPartition;
    bool change = true;
    int i = 0;

    while(change){
        change = false;
        i++;
        for(int v = 0; v < n; v++){
            int sigma_0 = 0, sigma_1 = 0;
            for(int u = 0; u < n; u++){
                if(graph[v][u]){
                    if(currentPartition[u] == 1)    sigma_0 += graph[v][u];
                    else if(currentPartition[u] == 0)   sigma_1 += graph[v][u];
                }
            }

            if(currentPartition[v] == 0 && sigma_1 > sigma_0){
                currentPartition[v] = 1;
                change = true;
            }
            else if(currentPartition[v] == 1 && sigma_0 > sigma_1){
                currentPartition[v] = 0;
                change = true;
            }
        }
    }

    k += i;
    cutValue += calculateCut(graph, currentPartition);

    // statsFile << "no. of iterations in local search : " << k << endl;
    // statsFile << "maxcut-value in this iteration : " << calculateCut(graph, currentPartition) << endl << endl;

    return currentPartition;
}
```

Figure 2 Local Search Implementation

Now, the three heuristic, **greedy**, **random** and **semi_greedy**, used in the program is described one by one.

```
// Greedy contstruction method for the initial solution in GRASP
vector<int> greedy_construction(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<int> currentPartition(n, -1);
    int currentCut = -1;

    int u, v, wt;
    tie(u, v, wt) = findMaxWeightEdge(graph);
    currentPartition[u] = 0;
    currentPartition[v] = 1;

    for(int i = 0; i < n; i++){
        if (currentPartition[i] == -1) {
            currentPartition[i] = 0;
            int newCut_0 = calculateCut(graph, currentPartition);
            currentPartition[i] = 1;
            int newCut_1 = calculateCut(graph, currentPartition);
            if(newCut_0 >= newCut_1){
                currentPartition[i] = 0;
            }
            else{
                currentPartition[i] = 1;
            }
        }
    }

    // statsFile << "greedy_construction: " << calculateCut(graph, currentPartition) << endl;
    // printResult(graph, currentPartition);

    return currentPartition;
}
```

Figure 3 Greedy Heuristic

The greedy heuristic first find the maximum-weighted edge by calling the function `findMaxWeightEdge(graph)` and puts the two endpoints `u` and `v` in two partitions, u in partition-0 and v in partition-1. `curretnPartition` was initialized to -1. Then it loops through all the vertices that doesn't have their partition assigned yet and checks to see if the cut of the graph can be increased by including the vertex in partition-0 or partition-1. Finally, it returns `resultPartition`.

```cpp
// simple random heuristic for the initial solution in GRASP
vector<int> randomized_construction(const vector<vector<int>>& graph) {
    int n = graph.size();
    vector<int> vertices(n, 0);
    vector<int> currentPartition(n, -1);
    for(int i = 0; i < n; i++) vertices[i] = i;

    // Initialize random number generator
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> uniform_dist(0, 1);

    for (int v : vertices) {
        // Randomly assign vertex to partition X (0) or partition Y (1)
        currentPartition[v] = uniform_dist(gen);
    }

    // statsFile << "randomized_construction: " << calculateCut(graph, currentPartition) << endl;
    // printResult(graph, currentPartition);

    return currentPartition;
}
```

Figure 4 Random Heuristic Implementation

`currentPartition` is initialized to -1 to indicate that none of the vertices have been assigned a partition yet. Then the sequence number of all the vertices is store in the vector `vertices`. Randomness is introduced by instantiating **Mersenne Twister 19937 Engine** using **uniform distribution**. For each vertex in vertices, its partition is randomly assigned through this object and the final `currentPartition` is returned.

The Semi-greedy heuristic is implemented identically to the pseudo code.

```cpp
// Semi-greedy randomized construction of initial solution with RCL (Restricted Candidate List)
vector<int> semi_greedy_construction(const vector<vector<int>>& graph) {
    int n = graph.size(), wmin = INT_MAX, wmax = INT_MIN;
    double alpha = rand() / RAND_MAX;
    double thres;
    vector<pair<int, int>> e_rcl;
    vector<int> v_rcl;
    unordered_set<int> X, Y, XY, R, U;
    vector<int> resultPartition(n, 0);      // X ::= 0, Y ::= 1

    for(int i = 0; i < n; i++)  { R.insert(i); U.insert(i); }

    for(int i = 0; i < n; i++){
        for(int j = i+1; j < n; j++){
            if(graph[i][j] < wmin){
                wmin = graph[i][j];
            }
        }
    }

    for(int i = 0; i < n; i++){
        for(int j = i+1; j < n; j++){
            if(graph[i][j] > wmax){
                wmax = graph[i][j];
            }
        }
    }

    thres = wmin + alpha * (wmax - wmin);
    for(int i = 0; i < n; i++){
        for(int j = i+1; j < n; j++){
            if(graph[i][j] >= thres){
                e_rcl.push_back({i,j});
            }
        }
    }
```

Figure 5 Semi-greedy Heuristic Implementation part-1

```
pair<int,int> random_edge = e_rcl[rand() % e_rcl.size()];
X.insert(random_edge.first);
Y.insert(random_edge.second);

// until all the vertices are partitioned, continue the loop
while(!are_sets_identical(XY,U)){
    // subtract XY from U to get R
    set_difference(U.begin(), U.end(), XY.begin(), XY.end(), inserter(R, R.end()));

    unordered_map<int,int> sigmaX, sigmaY;  // vertex, cut-weight-contribution
    for(int v : R){
        sigmaX[v] = 0;
        for(int u = 0; u < n; u++){
            if(graph[v][u] && Y.count(u) > 0){
                sigmaX[v] += graph[v][u];
            }
        }

        sigmaY[v] = 0;
        for(int u = 0; u < n; u++){
            if(graph[v][u] && X.count(u) > 0){
                sigmaY[v] += graph[v][u];
            }
        }
    }

    int sigmaX_min = INT_MAX, sigmaY_min = INT_MAX, sigmaX_max = INT_MIN, sigmaY_max = INT_MIN;

    for(int v : R){
        if(sigmaX[v] < sigmaX_min){
            sigmaX_min = sigmaX[v];
        }
    }
    for(int v : R){
        if(sigmaY[v] < sigmaY_min){
            sigmaY_min = sigmaY[v];
        }
    }
    wmin = min(sigmaX_min, sigmaY_min);

    for(int v : R){
        if(sigmaX[v] > sigmaX_max){
            sigmaX_max = sigmaX[v];
        }
    }
```

Figure 6 Semi-greedy Heuristic Imeplementation part-2

```
        for(int v : R){
            if(sigmaY[v] > sigmaY_max){
                sigmaY_max = sigmaY[v];
            }
        }
        wmax = max(sigmaX_max, sigmaY_max);

        thres = wmin + alpha * (wmax - wmin);

        for(int v : R){
            if(max(sigmaX[v], sigmaY[v]) >= thres){
                v_rcl.push_back(v);
            }
        }

        int random_vertex = v_rcl[rand() % v_rcl.size()];

        if(sigmaX[random_vertex] > sigmaY[random_vertex]){
            X.insert(random_vertex);
        }
        else{
            Y.insert(random_vertex);
        }

        // testing
        // showElements(X, outFile);
        // outFile << endl;
        // showElements(Y, outFile);
        // outFile << endl << endl;

        // store union of X and Y in XY
        XY = X;
        XY.insert(Y.begin(), Y.end());
    }

    for(int v : Y){
        resultPartition[v] = 1;
    }

    // statsFile << "semi_greedy_construction: " << calculateCut(graph, resultPartition) << endl;

    return resultPartition;
}
```

Figure 7 Semi-greedy Heuristic Imeplementation part-3

Local variables:

alpha => calculated randomly by rand()/RAND_MAX, RAND_MAX being the maximum random number that can be generated by rand().

thres => corresponds to μ in the pseudo code.

e_rcl => RCL for edges. Each element of e_rcl is a pair<int,int> of vertices.

v_rcl => RCL for vertices.

X, Y => the set of vertices included in partition-0 and partition-1 respectively. These two sets are updated at each iteration of the while loop.

XY => union of X and Y.

U => universal set of all the vertices. There is no change to this set throughout the code.

R => set of remaining vertices, that is, the vertices that haven't been assigned a partition yet. This set is obtained by subtracting XY from U at the start of each iteration of the while loop.

resultPartition => final partition of the vertices, initially, all the vertices are placed in partion-0. At the end of the while loop, all the vertices of the graph will be processed. So, resultpartition of the vertices that belongs to set Y is changed to 1.

```cpp
int main(int argc, char* argv[]) {
    srand(static_cast<unsigned>(time(0)));  // Seed the random number generator

    int n, e;                               // number of vertices -> n, number of edges -> e
    inFile >> n >> e;

    int u, v, wt;                           // u -> starting vertex, v -> ending vertex, wt -> weight of the edge (u---v)
    vector<vector<int>> graph(n, vector<int>(n,0));
    for(int i = 0 ; i < e; i++){
        inFile >> u >> v >> wt;
        graph[u-1][v-1] = wt;
        graph[v-1][u-1] = wt;
    }

    int max_iter_grasp = 10;                // Number of GRASP iterations

    // Check if the argument are provided
    if (argc >= 2) {
        max_iter_grasp = std::atoi(argv[1]);    // overwrite default value
    }

    vector<int> maxCutPartition = grasp_max_cut(graph, max_iter_grasp);

    // Output the results
    statsFile << "semi_greedy_construction:\n";
    statsFile << "average no. of iterations of local search : " << (k * 1.0/max_iter_grasp) << ", average cut-value of max_cut_local_search() : "
    << (cutValue * 1.0/max_iter_grasp) << endl;
    statsFile << "no. of iterations of GRASP : " << max_iter_grasp << ", best cut-value of grasp_max_cut() : " << calculateCut(graph,
    maxCutPartition) << endl << endl;

    printResult(graph, maxCutPartition);

    return 0;
}
```

Figure 8 main() Function

Finally, the main function takes input graph using `ifstream` object `inFile`. `max_iter_grasp = 10` by default, if no command line argument is provided. Otherwise, the provided argument is used instead. This function calls `grasp_max_cut(graph, max_iter_grasp)` to obtain `maxCutPartition`. It also prints average number of local search iterations performed and average max-cut value for a given `max_iter_grasp` in an `ofstream` object `statsFile`. By calling the function `printResult(graph, maxCutPartition),` it prints the final partitions and max-cut in an output file.

Output:

Problem			Constructive Algorithm			Local-1		GRASP-1		
name	|V| or n	|E| or m	Randomized-1	Greedy-1	Semi-greedy-1	No. of Iterations	Best Value	No. of Iterations	Best Value	Known Best Solution
G11	800	1600	460	464	492	2.47	464.4	30	492	627
G13	800	1600	474	468	496	3	477.2	30	496	645
G12	800	1600	452	450	476	3	450	30	476	621
G15	800	4661	2922	2934	2958	3	2934	30	2958	3169
G16	800	4672	2937	2949	2965	3	2949	50	2965	3172
G14	800	4694	2955	2962	2971	3	2962	50	2971	3187
G1	800	19176	11454	11397	11433	15	11397	80	11454	12078
G43	1000	9990	6460	6409		8	6409	80	6460	7027

Figure 9 Results from Different Test Graphs

The output results are computed from the `statsFile` which was modified from `main()` to include various information related to the simulation as given below. These results are obtained by changing the number of GRASP iterations from the command line and/or changing the heuristic in the function,

```
vector<int> grasp_max_cut(const vector<vector<int>>& graph, int max_iter_grasp)
```

For example, in case of G16, the `statsFile` contains the following info:

```
----------g16------------
randomized_construction:
average no. of iterations of local search : 4.18, average cut-value of max_cut_local_search() : 2903.38
no. of iterations of GRASP : 50, best cut-value of grasp_max_cut() : 2931

randomized_construction:
average no. of iterations of local search : 4.1, average cut-value of max_cut_local_search() : 2899.67
no. of iterations of GRASP : 60, best cut-value of grasp_max_cut() : 2933

randomized_construction:
average no. of iterations of local search : 4.24286, average cut-value of max_cut_local_search() : 2901.86
no. of iterations of GRASP : 70, best cut-value of grasp_max_cut() : 2925

randomized_construction:
average no. of iterations of local search : 4.025, average cut-value of max_cut_local_search() : 2902.47
no. of iterations of GRASP : 80, best cut-value of grasp_max_cut() : 2937
```

randomized_construction:
average no. of iterations of local search : 4.29, average cut-value of max_cut_local_search() : 2900.8
no. of iterations of GRASP : 100, best cut-value of grasp_max_cut() : 2928

greedy_construction:
average no. of iterations of local search : 3, average cut-value of max_cut_local_search() : 2949
no. of iterations of GRASP : 20, best cut-value of grasp_max_cut() : 2949

semi_greedy_construction:
average no. of iterations of local search : 3.32, average cut-value of max_cut_local_search() : 2936.5
no. of iterations of GRASP : 50, best cut-value of grasp_max_cut() : 2965

Thus, in the row labeled G16 in the table, the columns are filled in the following ways:

- Randomized-1: max(best cut-value of grasp_max_cut() for the statistics under the heading randomized_construction) = 2937
- Greedy-1: max(best cut-value of grasp_max_cut() for the statistics under the heading greedy_construction) = 2949
- Semi-greedy-1: max(best cut-value of grasp_max_cut() for the statistics under the heading semi_greedy_construction) = 2965
- Local-1:
    - Best Value: max(average cut-value of max_cut_local_search()) = 2949
    - No. of Iterations: average no. of iterations of local corresponding to the Best Value = 3
- GRASP-1:
    - Best value: max(best cut-value of grasp_max_cut() among all headings) = 2965
    - No. of Iterations: no. of iterations of GRASP corresponding to the Best Value = 2965