SDS - Homework1 A

Group 00: Michele Meo, Lorenzo Ceccomancini

Randomize this…

Randomized Max-Cut algorithm simulations

We use the **igraph** library to create the graph on which we will then face the Max-Cut problem. we generate the graph we use the Erdos-Renyi method and, then, we work on the adjacency matrix of the graph.

input: N number of nodes

output: the adjacency matrix associated to the graph

```
create_graph = function(N){

  graph = erdos.renyi.game(N, 0.5, type='gnp',
                           directed=FALSE, loops=FALSE)

  AM = as_adjacency_matrix(graph, type = 'both',
                           attr = NULL, edges = FALSE, names = FALSE,
                           sparse = igraph_opt("sparsematrices"))

  AM = matrix(AM, N, N)

  return(AM)
}
```

With the following simulation we implement M times the Randomized Max-Cut algorithm, for which we extract a random sub-graph U, including in U each node of the original graph with probability 0.5. We decided to work on the adjacency matrix related to the original graph, since we are able to obtain the sub-graph's adjacency matrix by filtering the matrix of U with a logical vector of values "True" or "False", respectively with probability 0.5.

input: *M* number of simulations, *N* number of nodes, *adj_matrix* adjacency matrix of original graph

output: Expected value of the cut's cardinality

```
simulate_cut = function(M, N, adj_matrix){

  sample_cut = c()

  for (m in (1:M)){

    coin = sample(c(TRUE, FALSE), N, replace=TRUE, prob = c(.5, .5))
    # Example of cut

    subgraph = adj_matrix[coin, coin]

    subgraph_comp = adj_matrix[!coin, !coin]

    E_subgraph = sum(subgraph)/2
    E_subgraph_comp = sum(subgraph_comp)/2

    card_cut = sum(adj_matrix)/2 - E_subgraph - E_subgraph_comp
    sample_cut[m] = card_cut

  }
  return(sum(sample_cut)/M)
}
```

In the following cell we have the main part of the code, where we execute the *maxcut* function of library **sdpt3r** to compute an approximation of the max cut on the input graph and check that our random algorithm verifies the following inequality: $\mathbb{E}(card(\delta U)) \geq \frac{OPT}{2}$.

```
N = 100
M = 1000

AM = create_graph(N)

expected_cut = simulate_cut(M, N, AM)
theoretical_maxcut = (-maxcut(AM)$pobj)

print(paste('The inequality is satisfied?', expected_cut >= theoretical_maxcut/2))
```

```
## [1] "The inequality is satisfied? TRUE"
```

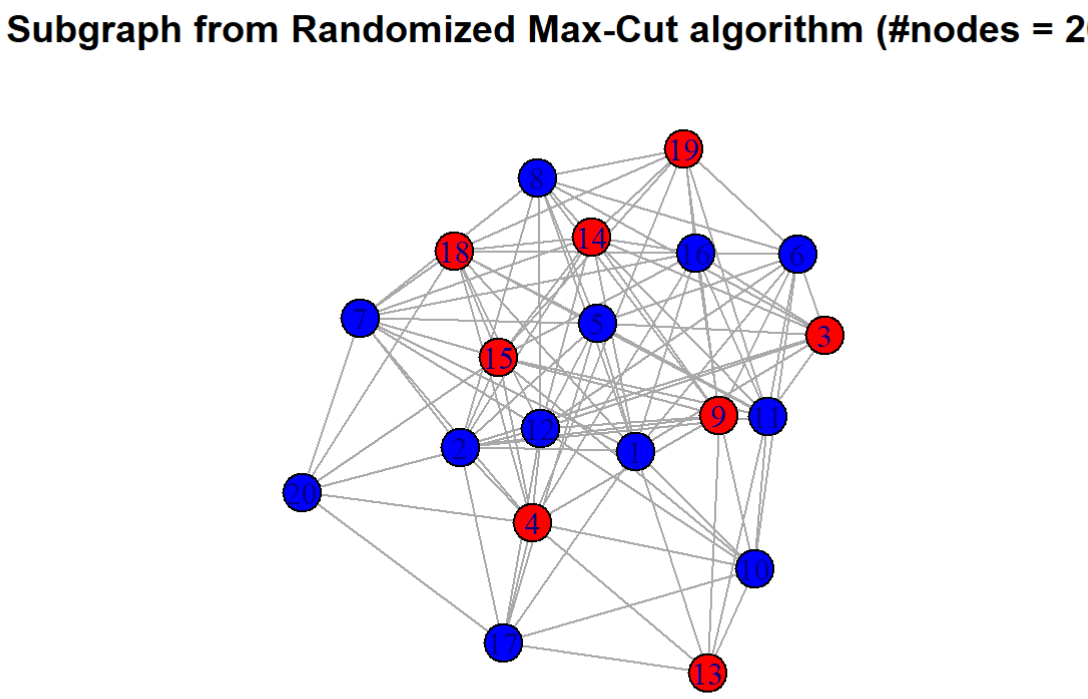
In order to visualize and get an idea of the Randomized Max-Cut algorithm action, we decide to show an iteration of the algorithm in which a specific sub-graph U is extracted and, consequently, a cut δU will be defined. The set of red nodes represents the sub-graph U selected through this single execution of the algorithm. The graph has 20 nodes just to facilitate the visualization task.

```
N = 20
g = erdos.renyi.game(N, 0.5, type='gnp',
                            directed=FALSE, loops=FALSE)

coin = sample(c(TRUE, FALSE), N, replace=TRUE, prob = c(.5, .5))

V(g)[!coin]$color = "blue"
V(g)[coin]$color = "red"

plot(g, layout=layout.kamada.kawai)
title(main = "Subgraph from Randomized Max-Cut algorithm (#nodes = 20)")
```



Computational time analysis

In the following function, we implement the function **create_graph** for a set of increasing graph size, in order to evaluate the computational time requested by Randomized Max-Cut algorithm.

input: *graph_size* vector of graph size, *M* total iteration for each graph

output: *time* vector of computational time for each simulation

```
comp_time_analysis = function(graph_size, M){

  time = c()

  counter = 1
  for (s in size){

    start = Sys.time()

    adj_matrix = create_graph(s)
    expected_cut = simulate_cut(M, s, adj_matrix)

    end = Sys.time()

    time[counter] = end-start
    counter = counter + 1
  }

  return(time)
}
```

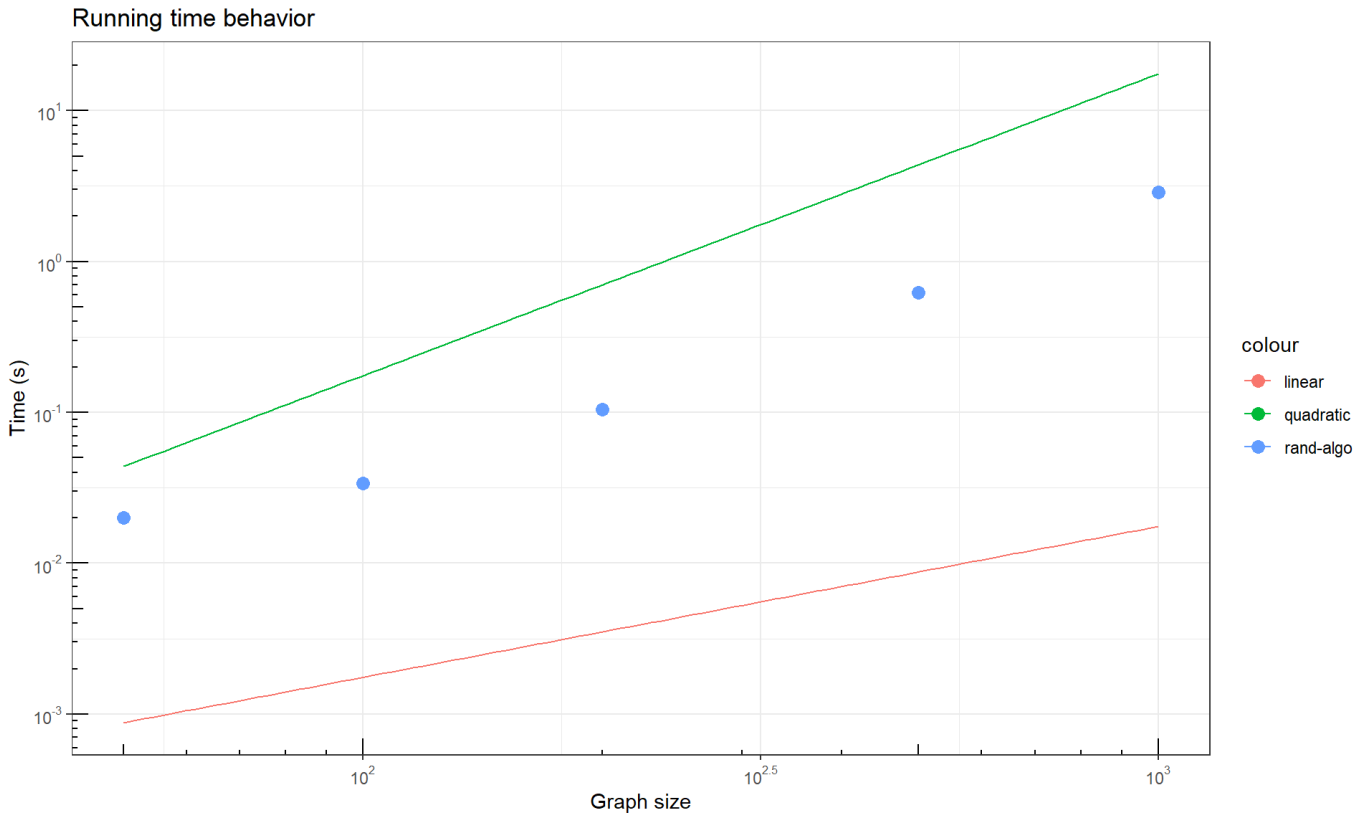
```
size = c(50, 100, 200, 500, 1000)

comp_time = comp_time_analysis(size, M)
time_df = data.frame(size,comp_time)

linearFit = lm(log10(time_df$comp_time) ~ log10(time_df$size), data=time_df)
intercept = linearFit$coefficients[1]
slope = linearFit$coefficients[2]
```

```
## [1] "The Randomized Max-Cut algorithm has a polynomial complexity of power: 1.7"
```

Assuming that the algorithm has a polynomial computational complexity, we plot the results obtained by the time analysis in a log-log plot to better appreciate the polynomial power that, in this kind of plot, is the line's slope.



As we can see from the plot, the running time of the randomized algorithm, RT_{rand}, seems to be between a quadratic and a linear behaviors. In particular, although with only 5 points we were able to obtain a rough estimate of the algorithm's polynomial power with a linear fit on the logarithm of the time and graph size: $O(s) < RT_{rand} \approx O(s^{1.7}) < O(s^2)$, where s is the graph size.

Inequality Check

In the end, we decided to check the inequality robustness, testing it on 100 trials. Given that the algorithm is random, we would like to ensure that the inequality is verified with probability close to 1 for a large number of trials, so in the following cell we run 100 simulation of the Randomized Max-Cut algorithm on the same initial graph of 100 nodes and then we evaluate the ratio, N_+ / N_{tot}, between the number of times the inequality is verified and the number of total trials. As we can see, the results allows us to asses that the inequality is verified with probability close to 1.

```
trials = 100
N = 100
freq = 0

tot_expected_cut = c()

for(c in (1:trials)){

  expected_cut = simulate_cut(M, N, AM)
  tot_expected_cut[c] = expected_cut

  if(expected_cut >= theoretical_maxcut/2){
    freq = freq + 1
  }
}

print( paste("The inequality is verified with frequency:", freq/trials))
```

```
## [1] "The inequality is verified with frequency: 1"
```

Instead, in the following we print some statistics, obtained by the 100 trials, about the variable $\mathbb{E}(card(\delta U))$.

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1233    1234    1235    1235    1236    1237
```

```
## [1] "The variance is: 0.76"
```