Karger's randomized minimum cut algorithm - An analysis of its success frequency

Alberto Parravicini

1 Abstract

The following report will analyzes the performances, in terms of success frequency, of an R implementation of **Karger's algorithm** to compute the minimum cut of a graph by means of a randomized procedure.

The first section of the report briefly presents the algorithm, and its **R implementation**.

The second section studies the **success rate** of the algorithm, i.e. how many times Karger's algorithm gives the same result as a deterministic algorithm; the success rate of Karger's algorithm is then compared to the theoretical minimum success rate of the algorithm.
The algorithm is tested against a number of connected graphs with increasing number of nodes and edges, to get a better understanding of how the structure of a graph influences the performances of the algorithm.

2 Karger's algorithm implementation

2.1 *Theoretical background*

Karger's algorithm is employed to compute the **minimum cut** of a graph, i.e. the smallest number of edges that can be removed from a given graph without losing the edge connectivity of the graph.

Given a graph with n vertices, the algorithm works as follow:

```
for i in {1,2, ..  , n - 2}:
    pick an edge e at random from the graph
    contract the edge e (i.e.  merge its endpoints)
return the size of the set of edges that connects the two remaining vertices.
```

2.2 *Notes*

Self loops aren't allowed in the graph: if an edge contraction produces self loops, they are removed.
The following analyses assume a *connected graph*. Karger's algorithm can determine if a graph is disconnected in $O(|V|)$ time, if we assume that the edge contraction is done in constant time (with trivial modifications it would work even if $|V| > |E|$); the focus of the report is on the relation between the empirical success frequency and the theoretical success frequency of the algorithm, and using disconnected graph would positively skew the observed success frequency and make the results less useful in practice.

2.3 *R implementation*

R was chosen as the programming language of the implementation as the focus of the report isn't on the speed and computational efficiency of the implementation, but on the analysis of the success rate of the algorithm.
R offers a robust graph library called *igraph*, which in turns is a wrapper to the homonymous *C* library. That said, *igraph* doesn't provide a way to reliably create a random connected graph and to contract edges in an easy way, so it was necessary to implement these functionality by hand.

What follows are snippets of code, to explain how Karger's algorithm was implemented:

- **Generate a random connected graph**

```
make_random_connected_graph <- function(num_vertices, num_edges) {
  g <- make_empty_graph(directed = F)
  g <- g + vertex("1")
  # Add a new vertex, connect it to a random existing vertex.
  # Repeat the process num_vertices times.
  for(i in 2:num_vertices) {
    random_vertex <- V(g)[sample(1:length(V(g)), 1)]
    g <- g + vertex(as.character(i))
    g <- g + edge(i, random_vertex)
  }

  # Add the remaining num_edges - num_vertices edges.
  for(j in 1:(num_edges - num_vertices + 1)) {
    # Select two random, distinct vertices.
    random_vertices <- V(g)[sample(1:length(V(g)), 2, replace = F)]
    g <- g + edge(random_vertices[1], random_vertices[2])
  }
  return(g)
}
```

- **Karger's randomized minimum cut algorithm**

```
karger_random_min_cut <- function(g) {
  num_vertices = length(V(g))

  for(i in 1:(num_vertices - 2)) {
    # Select a random edge, and keep its endpoints.
    random_edge <- E(g)[sample(1:length(E(g)), 1)]
    from_vertex <- ends(g, random_edge)[[1]]
    to_vertex <- ends(g, random_edge)[[2]]

    # Contract the selected edge.
    edges_to_be_changed <- neighbors(g, to_vertex)
    for (j in 1:length(edges_to_be_changed)) {
      g <- g + edge(from_vertex, V(g)[edges_to_be_changed[j]]$name)
    }
    g <- delete.vertices(g, to_vertex)
   # Remove self loops from the graph.
    g <- simplify(g, remove.multiple = F, remove.loops = T)
  }
  # Return the number of edges, as our randomized min_cut
  return(length(E(g)))
}
```

3 Analysis of the success frequency

3.1 *Theoretical background*

It can be proved that, given a graph with n vertices, Karger's algorithm will return the correct minimum cut of the graph with probability greater or equal to $\frac{2}{n\cdot(n-1)}$. This is the probability of never contracting an edge belonging to the minimum cut of the graph, and assuming that the graph has the minimum possible number of edges, equal to $\frac{n\cdot k}{2}$, where k is the size of the minimum cut.

Moreover, it can be shown that executing the algorithm $n \cdot (n-1) \cdot \ln n$ times, and keeping the smallest cut found, gives an upper bound on the error probability equal to $\frac{1}{n^2}$.

The performances of the previous implementation of Karger's algortihm have been analyzed against graphs with increasing number of vertices and edges, and against graphs with fixed number of vertices and increasing number of edges. As mentioned previously, all graphs are connected, as using Karger's algorithm isn't an efficient tool to determine if a graph is edge connected.

The *igraph* library provides a function to deterministically determine the minimum cut of a graph. This function was used to check the correctness of each run of Karger's algorith.

3.2 ***First test:*** *increasing number of vertices and edges*

The first test consisted in running Karger's algorithm against connected graphs with *increasing number of vertices and edges*. The number of vertices goes from 20 to 100 (with a step size of 10 vertices), and the number of edges of each graph is equal to $4 \cdot \texttt{num\_vertices}$. For each graph, Karger's algorithm was run 50 times. For each graph size, 5 different graphs were built (thus, $9 \cdot 50 \cdot 5$ iterations of the algorithm have been performed in this test).

The success rate of the algorithm with respect to each graph size has been compared to the theoretical success rate baseline.

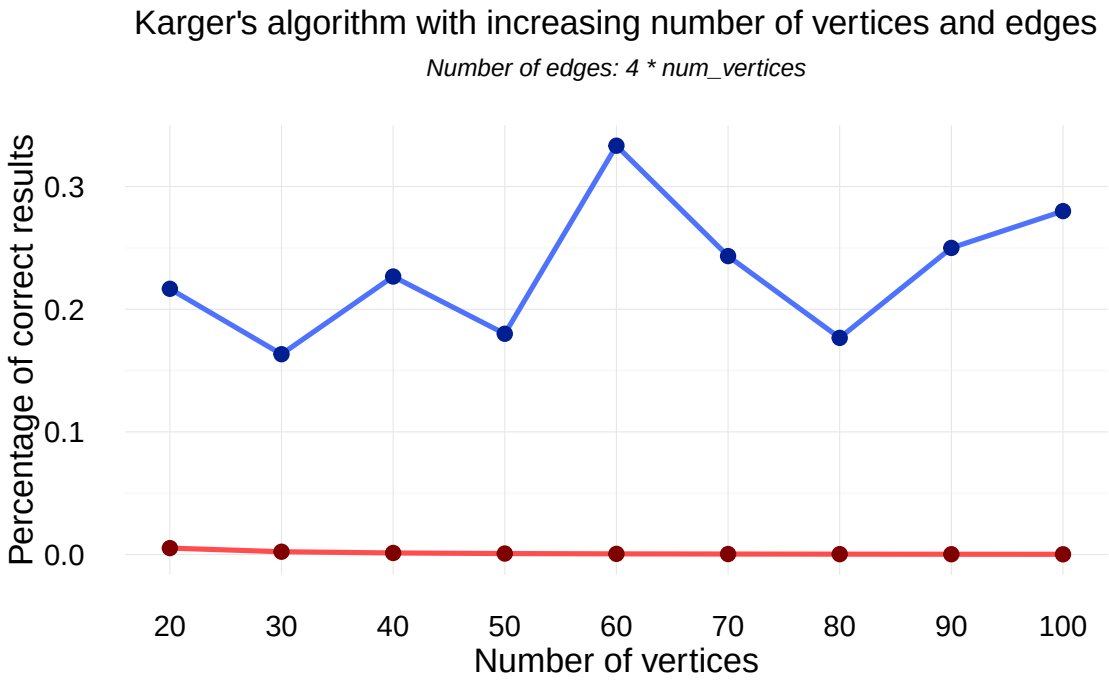


Figure 1: *In blue, the success frequency. In red, the baseline success frequency.*

It can be seen that if the ratio between number of edges and vertices is kept constant, increasing the number of vertices doesn't hinder the success rate of Karger's algorithm. In all cases, the verified success rate is much higher than the theoretical lower bound, which could be related to the low value of the ratio between number of edges and vertices.

3.3 ***Second test:*** *increasing number of edges, fixed number of nodes*

From the previous test, it was empirically shown that as long as the ratio between number of edges and vertices is constant, Karger's algorithm success rate isn't affected by the size of the graph. On the other hand, increasing the number of edges while keeping the number of vertices fixed could have an impact on the algorithm success percentage.

To verify this hypothesis, Karger's algorithm was run on connected graphs with 50 vertices and a number of edges ranging from 50 to 500 (with a step size of 50 edges). For each number of edges, 5 different graphs were built. On each of them, the algorithm was run 50 times, for a total of $10 \cdot 5 \cdot 50$ iterations of the algorithm. Once again, the results are compared to the theoretical success rate baseline.

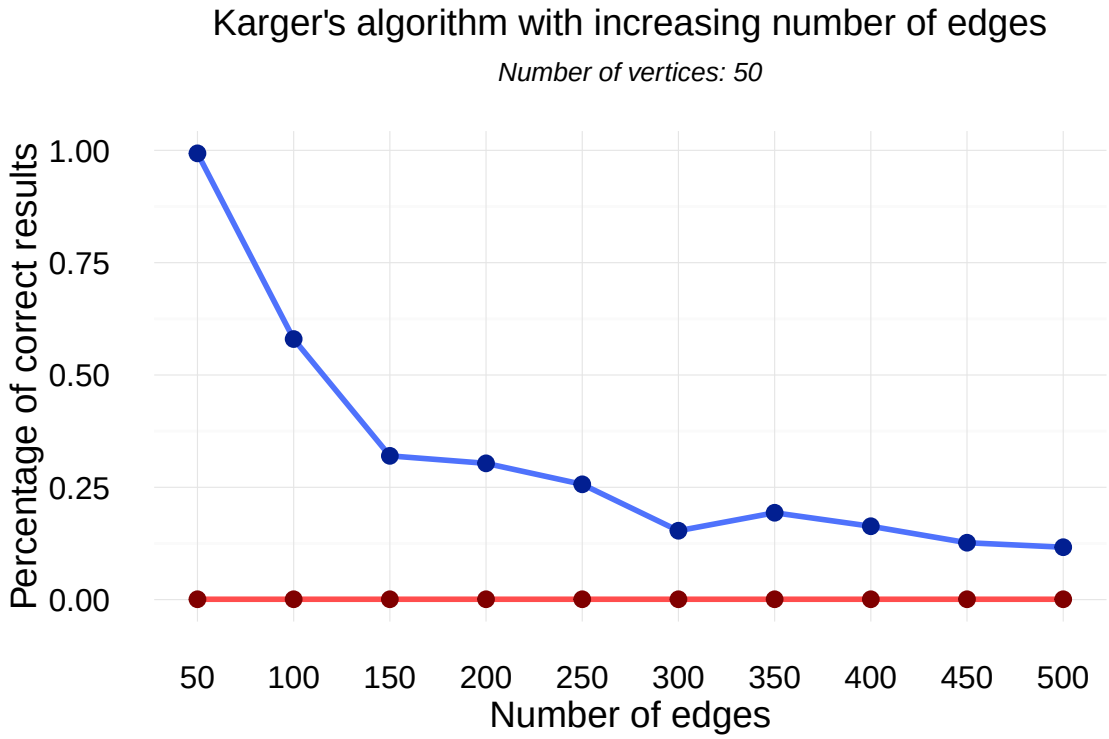


Figure 2: *In blue, the success frequency. In red, the baseline success frequency.*

It can be seen that increasing the number of edges while keeping the number of vertices fixed dramatically reduces the algorithm success rate, to the point where the theoretical lower bound of the success rate is almost reached.

This can be intuitively explained as follows: Karger's algorithm will output the correct minimum cut size as long as no edge belonging to the minimum cut is contracted.

However, increasing the number of edges while keeping fixed the number of vertices implies that the graph will have many multi-edges. Clearly, if an edge between two vertices belongs to the minimum cut, all edges between these two edges will also belong to the minimum cut.
If we have a multi-edge between two nodes, and this multi-edge belongs to the minimum cut, the probability of contracting it (which would lead to a fail of Karger's algorithm) will be higher that the probability of selecting a single edge.

It can also be noted that if $|V| = |E|$, Karger's algorithm always give the correct minimum cut value (i.e. 1): this happens because the graph is a tree, and contracting the edges will never form a multi-edge. Hence, after $n - 2$ iterations, the graph will always have just 1 edge left.

4 Addendum

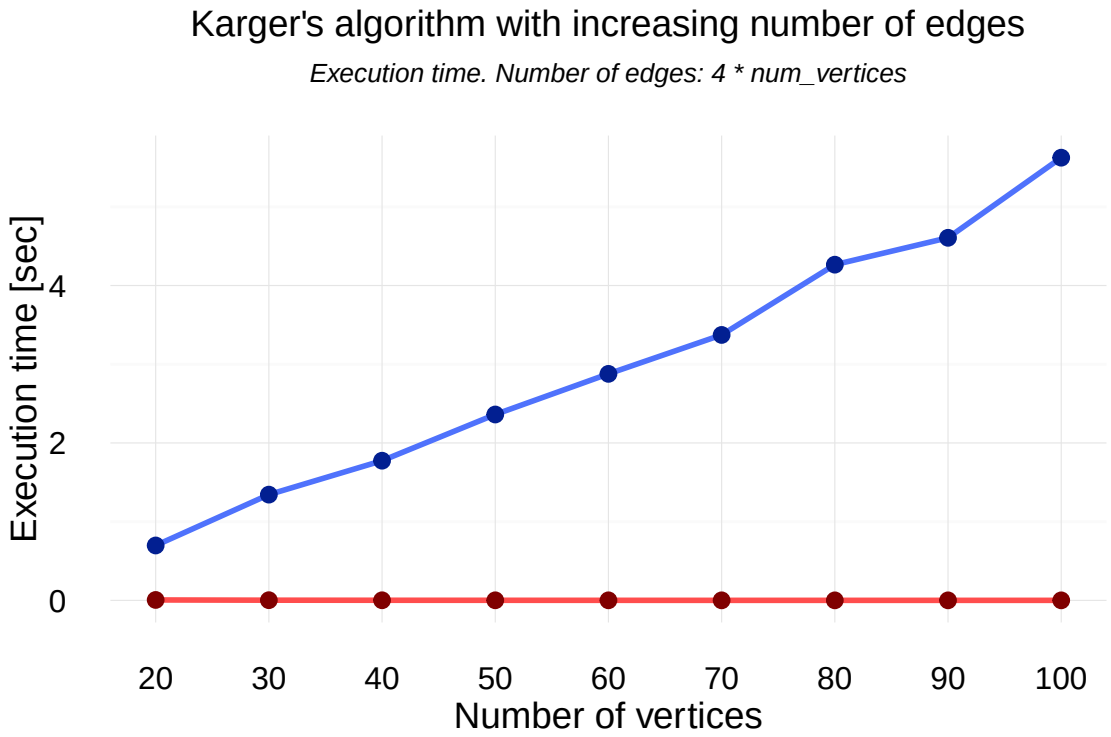
4.1 *Execution time analysis*

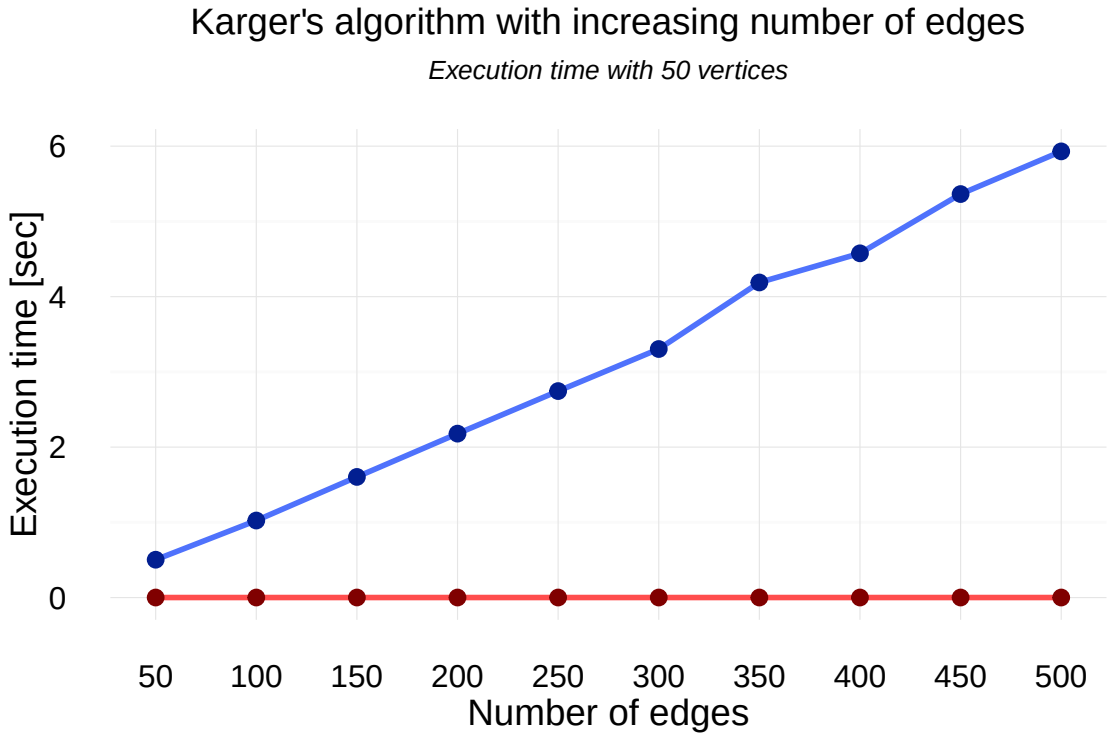
By making use of the *microbenchmark* library, it is possible to monitor the execution time of the implementation of Karger's algorithm.
Even though the implementation is not optimized for fast execution times, it is still interesting to study how the execution time scales with respect to the number of nodes and edges of a graph.

For both the previous tests, execution times were recorded and summarized in the following plots.

It emerges a linear scaling with respect to the number of vertices and edges.
This empirical results are coherent with the theoretical ones if we assume that the contraction operation happens in constant time.
Even without knowing the details of *igraph*, it is clear that contracting an edge requires a loop whose number of steps is equal to the size of the neighbourhood of a given vertex (i.e. at most $|V|$).
This would give a theoretical complexity of $O(|V|^2)$; however, if the graph is rather sparse, it might happen that the size of the neighbourhood of a given vertex is quite small, and thus the contraction operation happens quite fast in practice.

In the second plot, where the number of vertices is kept constant, this behaviour emerges even more: having a constant number of vertices would imply a constant execution time, but in practice it can be noticed once again a linear scaling; this happens because the graph becomes progressively less sparse, and thus the contraction operation becomes more and more time consuming.


Karger's algorithm with increasing number of edges
Execution time. Number of edges: 4 * num_vertices
Execution time [sec]
0
2
4
20
30
40
50
60
70
80
90
100
Number of vertices


Karger's algorithm with increasing number of edges
Execution time with 50 vertices
Execution time [sec]
0
2
4
6
50
100
150
200
250
300
350
400
450
500
Number of edges

As it was possible to notice from the reported execution times, the *R* implementation of Karger's algorithm is very time consuming, even for graphs of modest size.
To overcome this limitation, the same algorithm was re-implemented by making use of *Rust* and its *petgraph* library. The *Rust* implementation is orders of magnitude faster than the *R* one:

Vertices	Edges	R Exec. Time [sec]	Rust Exec. Time [sec]
50	500	5.93075	0.00042
100	400	5.62426	0.00029
1000	4000	72.18256	0.621770866

Still, the algorithm implemented in *Rust* is fundamentally the same as the *R* implementation. As such, *Rust* implementation doesn't affect the results that emerged from the analysis of the *R* version if the algorithm.

• **Full code (R and Rust implementations + R profiling) available at https://github.com/AlbertoParravicini/data_structures_and_algorithms**