Sapienza
Università di Roma

Data Science Master's Degree

Homework 1

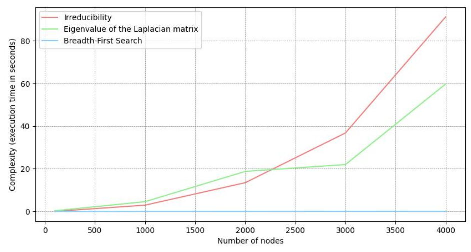
Networking for big data

Kleinrock Group:
Laura Concari, 1890490
Laura Lòpez Sànchez, 2125723
Giuliana Prinzi, 1952137
Erika Ioana Zetu, 1888659

Academic Year 2023/2024

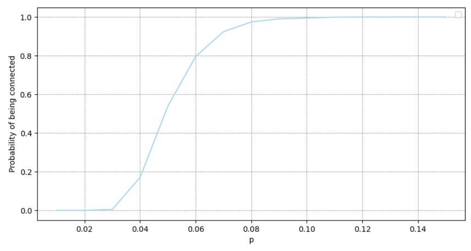
1 Part 1

Complexity checking algorithms



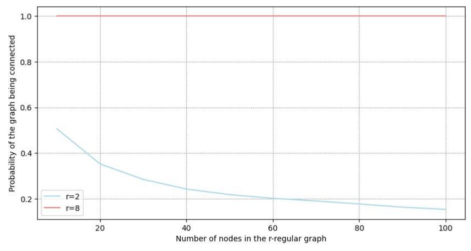
The plot demonstrates runtime comparisons of graph connectivity methods. Examining adjacency matrix irreducibility is the slowest ($O(n^3)$). Evaluating the second smallest Laplacian eigenvalue falls between $O(n^2)$ and $O(n^3)$, also relatively slow. Conversely, BFS traversal, with $O(n+m)$ complexity, proves notably faster. Thus, BFS emerges as the most efficient approach for connectivity checks, outperforming other methods.

Probability of a connected ER random graph



In a Monte Carlo simulation, 10,000 Erdos-Renyi graphs with 100 nodes were generated for each p. p denotes the probability of edge creation, ranging from 0.01 to 0.16. The plot illustrates the probability of connectivity (y-axis) against p (x-axis). Surprisingly, even at low p values like 0.05, the graph's likelihood of connectivity exceeds 0.5. At $p = 0.1$, 99% of the graphs exhibit connectivity, and at $p = 0.15$, connectivity is nearly certain. This suggests that despite low edge creation probability, the graph maintains a high likelihood of being connected.

Probability of a connected r-regular random graph



In a series of Monte Carlo simulations, we generated 1,000 r-random regular graphs with $r = 2$ and $r = 8$. Connectivity was assessed across 10 simulations, incrementing K from 10 to 100 nodes. $r = 2$ graphs, with every node having degree 2, displayed consistently low connectivity probabilities, nearing zero as K reached 100. This aligns with expectations, as achieving connectivity relies solely on ring topology, challenging with random edges. $r = 2$ proves impractical for real-world scenarios with larger networks. Conversely, for $r = 8$, graphs were almost always connected even with 100 nodes. This aligns with expectations, given the significant surplus of edges over the minimum required for connectivity.

2 Part 2

The algorithm

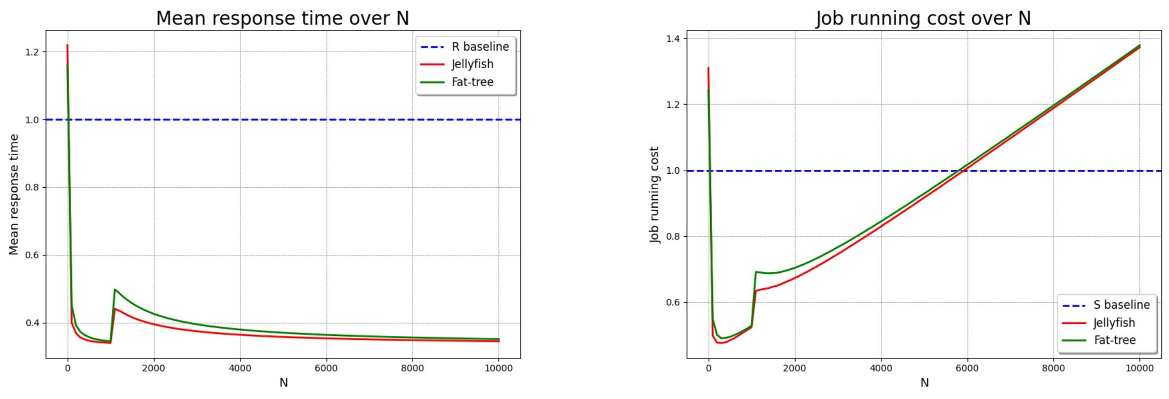
In this section of the assignment, our goal is to evaluate how the mean response time and job running cost evolve in two different network topologies: `Fat-tree` and `Jellyfish`. We aim to analyze these parameters under scenarios where a task is performed by a single server or multiple servers. Furthermore, we want to observe the variation of these metrics for different numbers of servers performing the tasks, ranging from 1 to 10,000.

To begin, we outline the statistical procedure used to compute the mean response time. The formal statement of the algorithm for this evaluation considers parameters such as the network topology graph (`topo`), the number of participating servers (`N`), a dictionary containing attributes of each node (`t`), and a list containing all the servers (`servers`).

The algorithm selects a random server (`Server A`) from the list of available servers and calculates the number of TCP connections based on given parameters. It then finds the shortest paths from `Server A` to all other nodes in the network, filters out the nearest host nodes, and selects the nearest `N` hosts for task assignment. The number of hops from `Server A` to the selected nearest servers is determined, and transmission times for each hop are computed.

The algorithm also generates random exponential data for processing time (`X_i`) at each server and uniform data for output representing the amount of output data. Response times for each server are calculated, and total time for each server is computed, considering transmission, processing, and response times. Finally, the maximum total time divided by the time taken by `Server A` and the sum of all `X_i` plus `N` times `T_0` divided by `S` are returned for evaluation.

Plot of mean response time and job running cost



(a) *Normalized mean response time*

(b) *Normalized Job running cost*

Figure 1: Plot of mean response time and job running cost

Optimal value, analysis of results and takeaways

The figure 1a depicts the mean response time (normalized with respect to R_{baseline}) as a function of the number of servers. The response time for all three configurations increases as the number of jobs (N) increases. This is expected because the servers have to handle more requests. However, the increase is much smaller for the multi-server configurations (`Jellyfish` and `Fat-tree`) compared to the single server (`Server A`). This shows that load balancing across multiple servers helps to distribute the workload and improve response time as the number of jobs increases. Both `Jellyfish` and `Fat-tree` topologies exhibit similar behavior. Looking accurately the response time for the `Fat-tree` configuration is quite lower than the `Jellyfish` configuration for all values of N. This suggests that the `Fat-tree` topology is more efficient for load balancing in this case. Interestingly, there's a peak in response time around $N = 1000$, likely due to increased communication hops between `server A` and the many servers. This peak suggests limitations in network scalability at high server counts.

The figure 1b depicts the job running cost $E[S]$ (normalized with respect to S_{baseline}). Initially, the curve shows a similar trend to the mean response time, with a peak around $N = 1000$. However, beyond $N = 1000$, the behavior changes due to the influence of $E[\Theta]$, the average server time used to run the job. After $N = 1000$, the job running cost increases with N, and for N over 6000, it exceeds S_{baseline}, representing the cost of running the job on a single server. This is because the overhead of managing the job across multiple servers starts to outweigh the benefits of parallelization. This suggests that beyond a certain point, splitting the job among multiple servers may not be beneficial. In conclusion, intuitively, the value of N that minimizes the job running cost is **250**, as it corresponds to the first local minimum before the initial peak. In con clusion, the takeaways from this analysis are:

- Splitting the job among a small number of servers (up to around $N = 1000$) can improve the job running cost.
- Splitting the job among too many servers can actually increase the job running cost due to the overhead of managing the job across multiple servers.