Calculating Minimum-Energy Reaction Pathways

Rob Sanchez

Programming Assignment 1

CIS 677 F2017

1 Overview

The graph in Figure 1 represents a biochemical reaction system with multiple reaction pathways. Each node represents a compound, and each directed edge a specific reaction's energy requirement. To compute the minimum-energy reaction cost needed to get from node 1 (the start, S1) to node 4 (the end, S4) several approaches can be used. The Julia[1] program described in this document, *pathways.jl*, explores two of these: 1) a brute-force "edge-linking" method, and 2) Dijkstra's shortest-path algorithm.

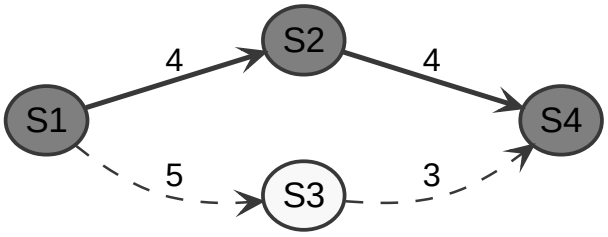

Figure 1: A biochemical reaction system (Graph 1)

It should be clear from Figure 1 that this type of directed graph representation could also be used to represent a route on a map. Finding the shortest route from one node to another might be useful to a shipping or logistics company, and is analogous to computing the minimum cost of a biochemical reaction as described above. As long as the same abstract problem is described completely by the graph, then either problem scenario can be solved by the same set of algorithms.

2 Investigation

Various ideas were sketched out initially in previous versions of the attached code without much research. An object-oriented approach using Node and Graph objects was combined with recursion to visit all nodes and neighbors and find the graph's shortest complete path. It was discovered relatively quickly that this strategy would not work, mainly because the results of the path calculations were too intertwined as a result of the recursive calls. Despite the fact that all of the proper information was available in the program output, no clear method was

found to unwind the calculations and group them properly into separate paths. As a result, the object-oriented approach was abandoned.

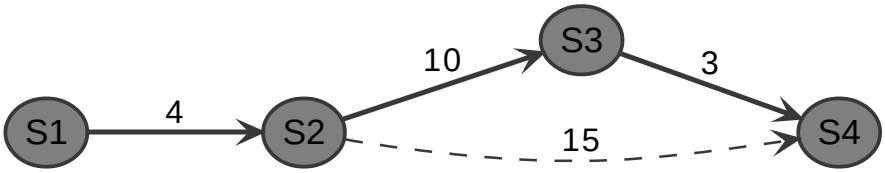

Figure 2: Graph 2 and shortest path.

After researching other possible approaches, it became clear that E.W. Dijkstra's shortest-path algorithm[2] could provide a solution. Our own class text, Grama et al's "Introduction to Parallel Computing"[3] describes the algorithm, but the pseudocode included there was not found to be very helpful. Another older text, Kruse et al's "Data Structures & Program Design in C"[4], as well as the corresponding Wikipedia page[5], were more instructive.

Seven graphs were used during development, which can be seen in the Figures presented throughout this report. A weighted adjacency matrix was used for graph representation, with unlinked nodes represented by zeroes, and linked nodes represented by positive integers. The matrix x-coordinates (rows) represent the starting node, and the y-coordinates (columns) the ending node. An example is shown in Listing 1.

```
G4 = [0 10 0  0  0  0;
      0  0 8 13 24 51;
      0  0 0 14  0  0;
      0  0 0  0  9  0;
      0  0 0  0  0 17;
      0  0 0  0  0  0;]
```

Listing 1: Graph 4 adjacency matrix.

In un-weighted adjacency matrices, zeroes represent a "false" value (no link) for unlinked nodes, and ones represent a "true" value for linked ones. This logic was extended to weighted adjacency matrices, the difference being that positive integers not only represent the existence of a link between two nodes, but also its magnitude. The matrix definitions were coded directly into the program source to avoid file I/O and parsing. These are found at the beginning of the program code as matrices named G1-G7.

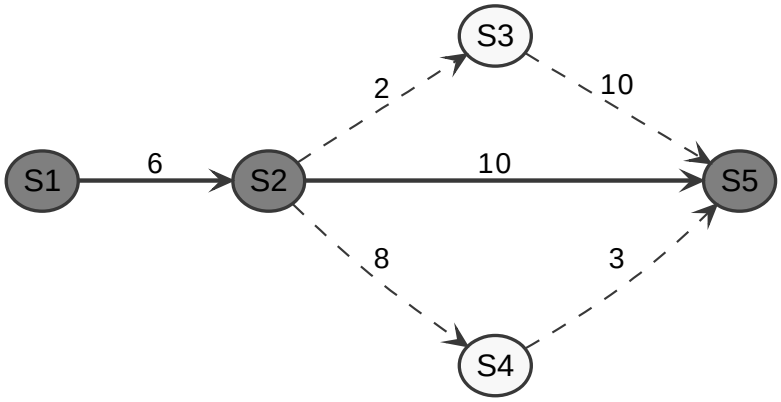

Figure 3: Graph 3 and shortest path.

The key to implementing Dijkstra's algorithm was the proper management of two sets of nodes: the visited and the unvisited. The final implementation can be found in function *dijkstra()*. Julia's built-in *setdiff()* was useful for updating the set definitions after each node and its neighbors were examined. An artificial constant "INFINITY" of value 9999 was used as an initial minimum value, and as specified in Dijkstra's algorithm, if the current distance to the node in question plus the distance between that node and its neighbor was less than the current minimum, then the minimum would be updated and the distance to that node would be tracked. Subsequently, the visited and unvisited sets of nodes would be updated. In *dijkstra()* the distance to the final target node is eventually returned from a vector of distances.

Initial versions of *dijkstra()* however, did not reverse-iterate to trace the shortest path during computation, they only computed the minimum cost. It seemed a bit useless to compute the minimum cost and not also have the resulting shortest path, but it wasn't clear at first how to generate the path from within *dijkstra()*. For this reason, a brute-force "edge-linking" method was implemented in *edgetrace()*. Simply put, *edgetrace()* extracts all of the graph edges from the adjacency matrix and links them together, until all complete and valid paths through the graph are found and traced. The cost for each path is computed and then the costs and paths are used to populate a dataframe[6] which is displayed. During development, this was sufficient to prove that the minimum-cost calculations generated by *dijkstra()* were correct.

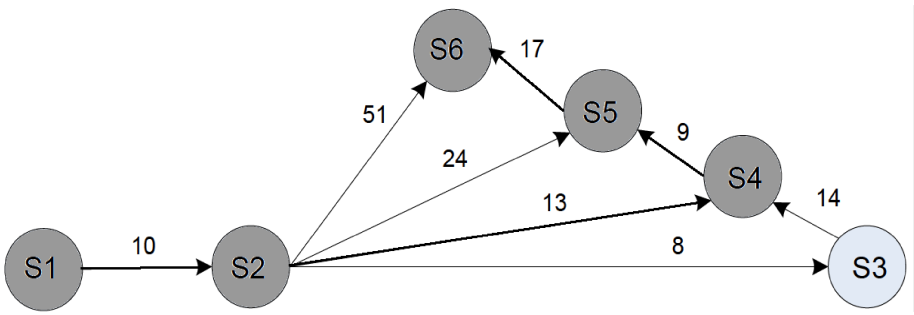

Figure 4: Graph 4 and shortest path.

After additional research, *dijkstra()* was updated to maintain a vector of previous nodes, and then to reverse-iterate and reveal the shortest path. In the program output, results for both the brute-force approach as well as the Dijkstra method are presented for each graph. Sample output for Graph 4 is presented in Listing 1. It should be noted that this particular implementation of Dijkstra's algorithm doesn't use a priority queue, only simple vectors and sets. In an effort to isolate the timing data for each algorithm, the timing measurement was conducted for the edge-tracing method before the building and presentation of the dataframe, as shown in the program output in Listing 2.

```
----------
 GRAPH 4:
----------
[1] graph traversal costs and paths,
    brute force connected edge tracing:

  0.005922 seconds (1.13 k allocations: 66.256 KiB)

4×2 DataFrames.DataFrame
| Row | cost | path                |
|─────|──────|─────────────────────|
| 1   | 61   | [1, 2, 6]           |
| 2   | 58   | [1, 2, 3, 4, 5, 6] |
| 3   | 51   | [1, 2, 5, 6]        |
| 4   | 49   | [1, 2, 4, 5, 6]     |

[2] graph minimum cost and optimal path,
    Dijkstra's algorithm:

        (49, [1, 2, 4, 5, 6])

  0.000126 seconds (136 allocations: 10.578 KiB)
```

Listing 2: pathways.jl results, Graph 4.

The final three more complex graphs are presented in the figures that follow. They were generated with PlantUML[7] and Graphviz[8]. Complete program output for all graphs is attached.

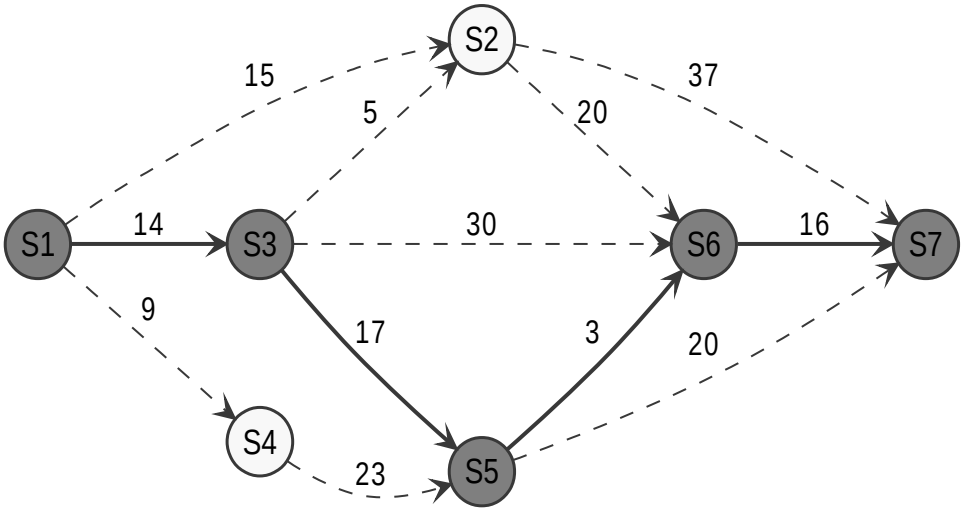

Figure 5: Graph 5 and shortest path.

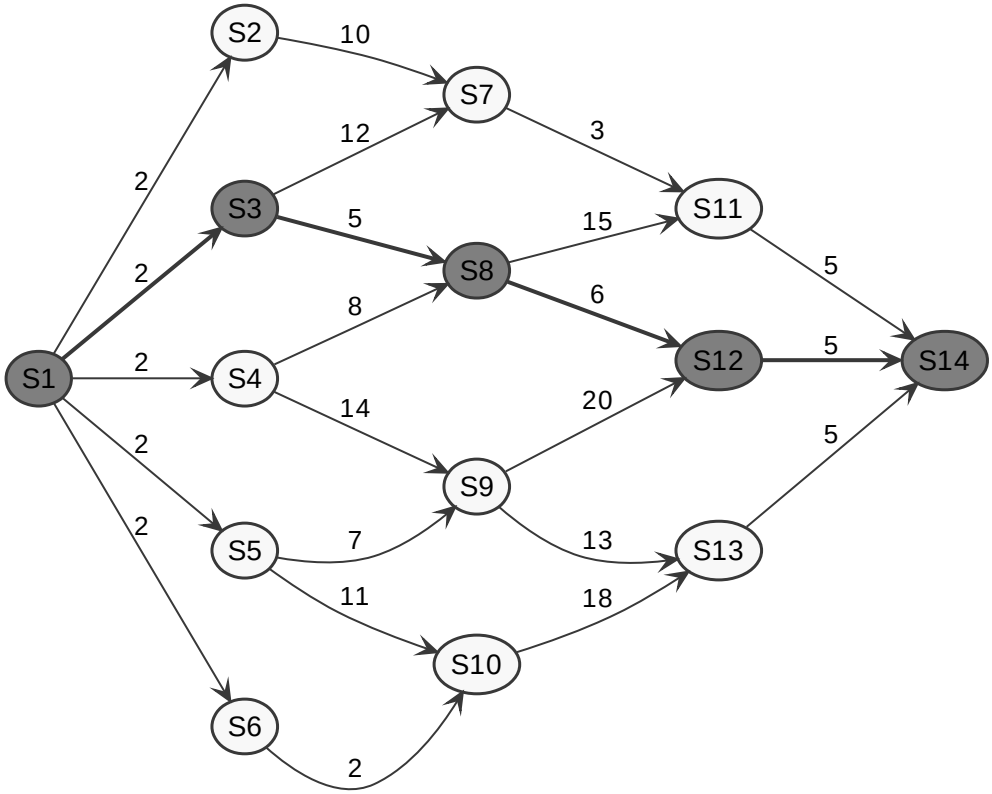

Figure 6: Graph 6 and shortest path.

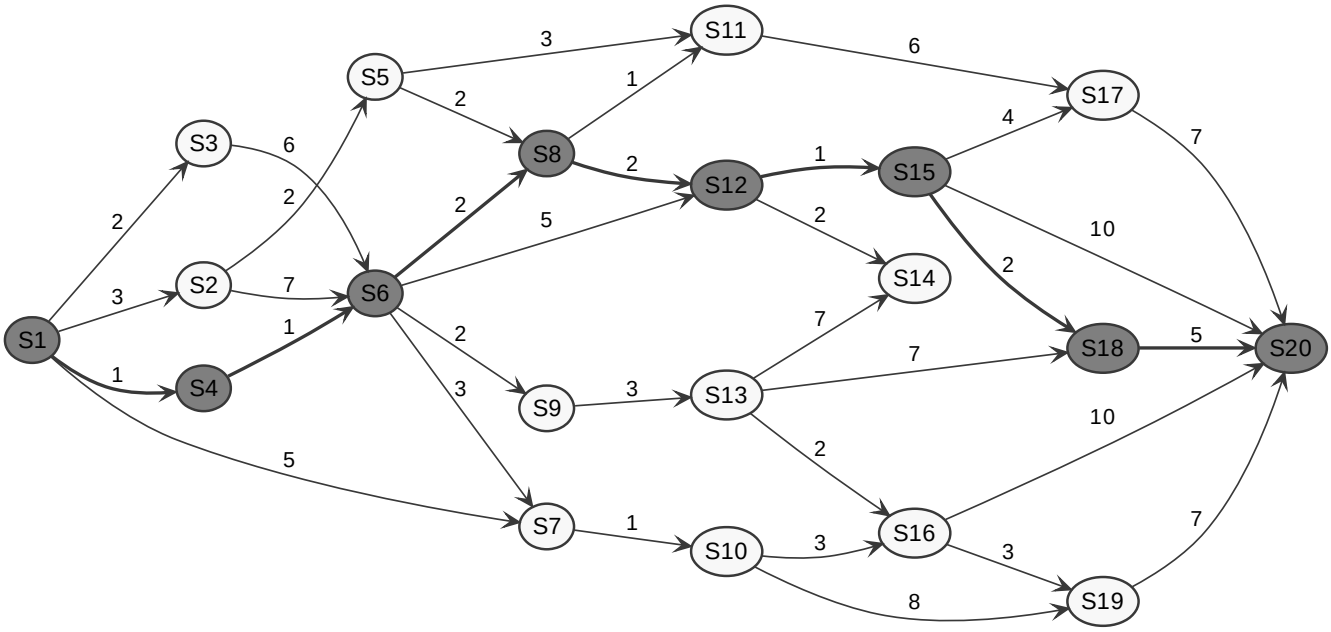

Figure 7: Graph 7 and shortest path.

References

[1] Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, *59*(1), 65–98. http://doi.org/10.1137/141000671

[2] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. http://doi.org/10.1007/BF01386390

[3] Grama, A. et al. (2003). Introduction to Parallel Computing, Second Edition. Addison-Wesley.

[4] Kruse, R. et al. (1997). Data Structures & Program Design in C, Second Edition. Prentice-Hall.

[5] Dijkstra's algorithm (n.d.). In Wikipedia. Retrieved September 10, 2017, from https://en.wikipedia.org/wiki/Dijkstra's_algorithm

[6] *DataFrames.jl* (version 0.7) [Computer software]. (2017). Available from https://github.com/JuliaData/DataFrames.jl

[7] *PlantUML* (version 8047) [Computer software]. (2017). Available from http://plantuml.com

[8] *Graphviz* (version 2.40.1) [Computer software]. (2017). Available from http://graphviz.org