

# Programming Assignment 3: Routing Algorithm Assignment (MUST Use Logbook) (OSPF) (UG: Part 1: PG: Part 1 and 2)

- Due Tuesday by 17:00
- Points 200

## Assessment

<b>Weighting:</b>	15% (150 Marks for UG and 200 Marks for PG)
<b>Task description:</b>	In this assignment, you will be writing code to simulate Dijkstra's algorithm in a network of routers.
<b>Academic Integrity Checklist</b>	<p>Do</p> <ul style="list-style-type: none"><li>✓ Discuss/compare high level approaches</li><li>✓ Discuss/compare program output/errors</li><li>✓ Regularly commit your work and write in the logbook</li></ul> <p>Be careful</p> <p>🔗 Code snippets from reference pages/guides/Stack Overflow must be attributed/referenced. We understand that implementation of Dijkstra exist so it is very important that you provide attributes to any code or references you have used to build your solution. We do not mind you learning from online resources to develop your solution but the sections of any code used MUST be attributed/referenced clearly and the code must be such that it is free for use.</p> <p>🔗 Only use code snippets that do not significantly contribute to the exercise solution.</p> <p>Do NOT</p> <ul style="list-style-type: none"><li>✗ Submit code not solely authored by you.</li><li>✗ Post/share code on Piazza/online/etc.</li><li>✗ Give/show your code to others</li></ul>

## Before you begin

*This practical is marked using a battery of carefully constructed testing framework and then all marks are moderated by a marker using the following information.*

**You must complete a logbook** as you develop your code (this process should be familiar to those that have or are doing the Computer Systems course). You can find details on how to do that and what that should look like **[in this guide \(https://myuni.adelaide.edu.au/courses/95212/pages/logbook\)](https://myuni.adelaide.edu.au/courses/95212/pages/logbook)**.

- During manual marking and plagiarism checks, we will look at your development process. If we do not see a clear path to a solution (i.e. code changes and regular commits and logbook entries **reflecting** your learning to develop your implementation you **may forfeit up to 150(UG)/200(PG) marks**.
- *An example case of forfeiting all 150 marks (for UG) would be the sudden appearance of working code with no prior evidence of your development process.*
- **It is up to you to provide evidence of your development** through regular submissions and logbook entries.

This practical requires thought and planning. You need to start early to allow yourself time to think of what and how to test before modifying any code. Failing to do this is likely to make the practical take far more time than it should.

**Starting the practical in the final week or days is likely to be an unsuccessful strategy with this practical; further your logbook entries are likely to be overlapped in close succession and, depending on the quality of the entries, is likely to lead to a mark that will be scaled lower (due to possibly poor documentation in the logbook).**

## Help and Support

**Please Note:** Before attempting this practical please study the material on Reliable Data Transfer (rdt) protocols in lectures/textbook. Come and see me if you are unclear about any concepts or the finite state machines (FSM) we studied.

If you are having a lot of issues with coding, please book a time with Teaching Assistant (TA) using the **[CNA Tutor Help Request Form \(https://myuni.adelaide.edu.au/courses/95212/pages/cna-tutor-help-request-form\)](https://myuni.adelaide.edu.au/courses/95212/pages/cna-tutor-help-request-form)**. A TA will get in touch about a meeting time but feel free to suggest a few options.

---

## Aims

- Learn about routing protocols and route propagation.

- Implement a basic links state routing algorithm.

---

## Overview

In this assignment, you will be writing code to simulate a network of routers performing implementing a Link State routing protocol, such as OSPF (Open Shorted Path First) based on Dijkstra's algorithm. You will need to implement the algorithm in the basic form and advanced form. Your implementations will need to ensure that the simulated routers in the network correctly and consistently update their link state database and computes their routing tables. This activity will mimic the actions of a routing protocol such as OSPF.

You will find a more detailed description of the Dijkstra's algorithm in the course notes and in section 5.2.1 of Kurose and Ross, Computer Networking, 7th Edition.

---

## Your Task

### Part 1: Link State Algorithm Basic Form (150 marks)

1. Reads information about a topology from the standard input (typed using a keyboard).
2. Uses link state updates and Dijkstra's algorithm, as appropriate and:
  - Output the information in various tables in the required format during intermediate steps.
3. Read updates to the topology from standard input and repeat step 2 above, until no further input is provided.

The Dijkstra's algorithm program you are to provide should be named `Dijkstra`.

### Part 2 (**PG only**): Link State Algorithm Advanced Form (50 marks)

The second part of the assignment is to write a second version of the program that is the same as the first, except that network is stored in a data structure with  **$\log(n)$**  performance for all relevant operations. This will require small modifications to your `Dijkstra` program.

The modified Dijkstra's algorithm program you are to provide should be named `DijkstraNlogN`. **This part is for postgraduate students only.**

## Report & Stress Test

You should produce: 1) a report, no longer than **1 page** in length as a PDF document named,

**Comparison.PDF**, 2) test network configurations and 3) script (automating the running of the test cases to generate results). Add these to your SVN repository.

- In the report, present clear evidence that your two solutions are producing results in the appropriate complexity bound. You should provide in your report: **1) a graph, and 2) textual explanations** that support this argument.
- The report should describe how to run your code to generate the results in your report using your script.
- Test cases should allow you to demonstrate the key benefits of the advanced implementation.

Marks are allocated based on the following:

**1)** Content of the report **graph and brief textual explanations**, justifying the complexity of the two implementations.

**2)** The combination of script/program/commands in the report describing how to run the programs **Dijkstra** and **DijkstraNlogN** along with the test network configurations submitted to SVN to generate the data used for the graphs. These scripts should print out the summary run-time information in a table format with the following headings:

For Dijkstra with  $N * N$  complexity

<b>Number of Nodes</b>	<b>Number of Links</b>	<b>Execution Time (<math>N*N</math>)</b>
...	...	....

For Dijkstra with  $N \log N$  complexity

<b>Number of Nodes</b>	<b>Number of Links</b>	<b>Execution Time (<math>N \log N</math>)</b>
...	...	....

- Use at least **4 data points** for your graph, i.e. four network typologies with increasing numbers of nodes.

## In Your Task


You will need to craft any internal data structures and design your program in such a way that it will reliably and correctly generate the correct tables. We have deliberately not provided you with code templates, and this means that you will have more freedom in your design but that you will have to think about the problem and come up with a design.

**For postgraduate students:** you will generate four data points to compare the runtime of **Dijkstra** and **DijkstraNlogN**.



You will need to record your progress and development cycle in a logbook as described in the 'Before you Begin' section above.

## Programming Language/Software Requirements

You may complete this assignment using the programming language of your choice, **with the following restrictions:**

- For **compiled** languages (Java, C, C++ etc.) you must provide a Makefile.
  - Your software will be compiled with `make` (**Please look at this resource on how to use Makefile build tool:** <https://makefiletutorial.com/>  [\(https://makefiletutorial.com/\)](https://makefiletutorial.com/))
  - Pre-compiled programs will **not** be accepted.
- Your implementation must work with the versions of programming languages installed on the Web Submission system, these are the same as those found in the labs and on the **uss.cs.adelaide.edu.au** server and include (but are not limited to):
  - **C/C++:** g++ (GCC) 4.8.5
  - **Java:** java version "1.8.0\_201"
  - **Python:** python 3.6.8
- Your implementation may use any libraries/classes available on the Web Submission system, but **no external libraries/classes/modules**.
- Your programs will be executed with the command examples below:
  - For C/C++

```
make
./Dijkstra
./DijkstraNlogN
```



You can find a **simple** example makefile for C++ [HERE \(https://myuni.adelaide.edu.au/courses/95212/files/15083484?wrap=1\)](https://myuni.adelaide.edu.au/courses/95212/files/15083484?wrap=1)   [\(https://myuni.adelaide.edu.au/courses/95212/files/15083484/download?download\\_frd=1\)](https://myuni.adelaide.edu.au/courses/95212/files/15083484/download?download_frd=1) . **A good resource is here:** <https://makefiletutorial.com/>  [\(https://makefiletutorial.com/\)](https://makefiletutorial.com/)

This will **need to be customised** for your implementation. Make sure you use tabs (**actual tab characters**) on the indented parts

- For java:

```
make
java Dijkstra
java DijkstraNlogN
```

You can find a **simple** example makefile for Java [HERE \(https://myuni.adelaide.edu.au/\)](https://myuni.adelaide.edu.au/)

[courses/95212/files/15083475?wrap=1](https://myuni.adelaide.edu.au/courses/95212/files/15083475?wrap=1)  (https://myuni.adelaide.edu.au/courses/95212/files/15083475/download?download\_frd=1) . A good resource is here: <https://makefiletutorial.com/>  (https://makefiletutorial.com/)

This will **need to be customised** for your implementation. Make sure you use tabs (**actual tab characters**) on the indented parts

<https://myuni.adelaide.edu.au/courses/95212/files/14654456/download?wrap=1>

- For Python (**no make file** is needed), so we will type:

```
./Dijkstra
./DijkstraNlogN
```

Programs written using an interpreted language such as Python:

- Will need to use UNIX line endings (always test on a uni system such as the **uss.cs.adelaide.edu.au** cloud instance).
- Will not be built with make (as shown above, because they are not compiled)
- Will require a 'shebang line' at the start of your file to run as above.  
e.g. `#!/usr/bin/env python3` (Python 3).

---

# Algorithm

## Key Assumptions

In a real routing environment, messages are not synchronised and routers send out their initial messages with their neighbours as needed or configured (i.e. after a certain time interval has elapsed).

In this environment, to simplify your programs:

- Link state routing messages are not part of the simulation and you can assume that all of the routers will have their Link-State Databases synchronized the moment a new link is established/added/removed by reading a link state from the standard input. So the Link-State Databases (LSDBs) are always synchronised and no intermediate steps need to be modelled.
- All neighbour tables are instantly updated at every router, the moment a new link is established/added/removed by reading a link state from the standard input.
- When Dijkstra algorithm is run, and multiple least cost paths exist to select from in an iteration, always select the first one in alphabetical order (in alphabetical order, by router name, so if you have a choice of A, B and C, then select A and add it to the list of nodes for which the shorted path is now known).
- Where multiple best routes exist, always select the first one in an alphabetical ordering of the next hop router name.

# Dijkstra's Algorithm

At each node, :

$$D(v) = \min(D(v), D(w) + c(w,v))$$

The cost to node  $v$  is the minimum cost among the current cost to node  $v$  and the cost to the via node  $w$  plus the cost to node  $v$  from the via node  $w$ .

At each node  $x$ :

INITIALISATION:

$N' = \{u\}$

for all nodes  $v$

if  $v$  is a neighbor of  $u$   
then  $D(v) = c(u,v)$

else  $D(v) = \infty$

LOOP

find  $w$  not in  $N'$  such that  $D(w)$  is a minimum

add  $w$  to  $N'$

update  $D(v)$  for each neighbor  $v$  of  $w$  and not in  $N'$ :

$D(v) = \min(D(v), D(w) + c(w,v))$

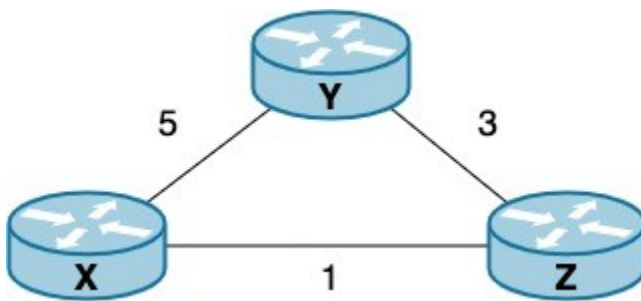
/\* new cost to  $v$  is either old cost to  $v$  or known least path cost to  $w$  plus cost from  $w$  to  $v$  \*/

until  $N' = N$

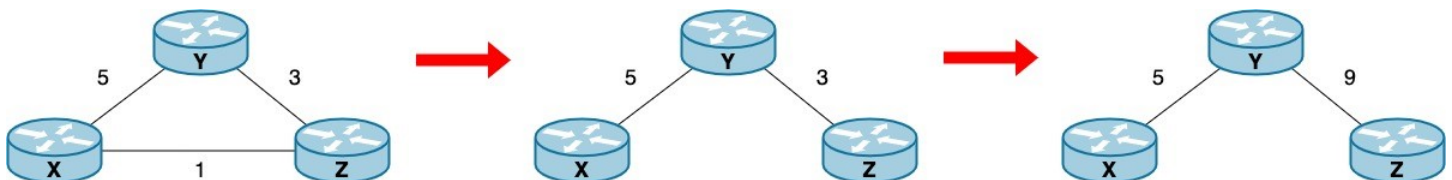
## Expected Input Format

### Sample Topology

Consider the following network sample topology in our description of the required input format:



Your program will need to read input from the standard input (terminal/command line) to construct such a given topology. Then, the perform updates, as illustrated below, and also given via the standard input (terminal/command line).



Please refer to this sample topology with the following updates above when looking into the expected input examples below.

The expected input format, taking the sample topology as an example, is shown below:

```
X
Y
Z
LINKSTATE
X-Z 1 X,Y
X-Y 5
Y-Z 3 X,Z
UPDATE
X-Z -1 X,Y
Y-Z 9 Y,Z
END
```

The input begins with name of each router/node in the topology.

- Each **name** is on a new line.
- Router **names** are case-sensitive, use **alphanumeric characters** only.
- Router **names** may not contain spaces.
- This section ends with the keyword "**LINKSTATE**".

The input continues with the details of each link/edge in the topology followed by a list of chosen router(s).

- Written as the names of two routers/nodes with a **-** in between, followed by the weight of that link/edge, then an **optional list** of routers (names separated by comma **,**) and the list of chosen routers can be empty).
- Your algorithm should run after each line of "**LINKSTATE**" input and build/update the neighbour table, link-state database (LSDB), and routing tables for all routers with links to other routers, according to the details of the link/edge given in the input. Then, show the **Expected Output** for each router in the list of chosen routers.
- Weight values should always be integers.
- A weight value of **-1** indicates a link/edge to remove from the topology if present.
- This section ends with the keyword "**UPDATE**".

The input continues with the link state update details of each link/edge in the topology given a link and cost followed by a list of chosen routers.

- The values in each line of input in this section should be used to update the current topology.
- As above, a weight value of **-1** indicates a link/edge to remove from the topology if present.
- Your implementation should update the neighbour tables, link-state databases, and routing tables as a result of receiving the link state update given in the input. Then, show the



**Expected Output** for each router according to the list of chosen routers (if any).

- If an unseen new router/node name has been inputted in this section, your program should be able to **add** this new router into the topology.
  - `Y-A 10 X,Y`
  - From the example input given above, your program should add **A** as a new router into the topology where it has a link with a cost of 10 to **Y**. Then show the **Expected Output** for routers X and then Y. Again, the *list of chosen routers* (X,Y in the example) is optional and none may be given.
- A user may input 0 or more lines of link-state updates in this section.
- Continues until the keyword, **"END"** is inputted, at which point the program exits normally.

## Expected Output Format

Each router should maintain a **Neighbour Table**, **Link-state Database (LSDB)** and **Routing Table**. We will ask you to print to standard out (screen/terminal) the

- **Neighbour Table**
- **Link-state Database (LSDB), and**
- **Routing Table**

of the chosen routers in **alphabetical order**.

### Neighbour Table

The Neighbour Table includes all neighbours of the router and the cost to the neighbours.

```
X Neighbour Table:
Y|2
Z|7
```

1. The name of the router/node.
2. The neighbours of the router/node.
3. The cost/distance to the neighbours.
4. The neighbour and the cost/distance should be separated by a pipe/vertical bar `|`.
5. The neighbours need to be **printed in alphabetical order**.
6. A blank line at the end of the table.
7. If the table is empty, the expected output is:

```
X Neighbour Table:
```

## Link-state Database (LSDB)

The Link-state Database includes all the unique links for the network topology reachable by the router along with the cost of each link in the topology.

X LSDB:

X|Y|2

X|Z|7

Y|Z|1

1. The name of the router/node.
2. Each unique link is represented by 2 routers/nodes that are directly connected to each other. The 2 routers/nodes are separated by a pipe/vertical bar |.
3. The cost/distance of the unique link.
4. The unique link and the cost/distance should be separated by a pipe/vertical bar |.
5. The unique links need to be **printed in alphabetical order** (in the example, X|Y first and then X|Z).
6. If a link is registered as down/removed, **do not** include this in the LSDB.
7. A blank line at the end of the table.
8. If the table is empty, the expected output is:

Y LSDB:

## Routing Table

The Routing Table includes the destination, next hop and total minimum cost/distance to the destination in the network topology reachable by the router.

X Routing Table:

Y|Y|2

Z|Y|3

where

1. The name of the router/node.
2. The name of the destination router/node.
3. The next hop. **If there are two paths with the same cost to the destination, pick the path with the least hops.** For example:
  - If there are **two paths** (Path 1 and Path 2) from router A to B with the **same cost=5** as shown below:
 

Path 1 (cost=5): A -> D -> B

- Path 2 (cost=5): A -> C -> E -> B

- The router must use Path 1 which requires less hop to the destination. Therefore, the routing table of A to destination B should look like:

- A Routing Table:  
B|D|5

4. The total minimum cost/distance to the destination.
5. Each column is separated using a pipe/vertical bar |.
6. The destinations need to be **printed in alphabetical order**.
7. A blank line at the end of the table.
8. If the table is empty, the expected output is:

Y Routing Table:

Below is an example of what the output should look like for the *provided topology and link state updates*.

X Neighbour Table:  
Z|1

X LSDB:  
X|Z|1

X Routing Table:  
Z|Z|1

Y Neighbour Table:

Y LSDB:

Y Routing Table:

X Neighbour Table:  
Y|5  
Z|1

X LSDB:  
X|Y|5  
X|Z|1  
Y|Z|3

X Routing Table:  
Y|Z|4  
Z|Z|1

Z Neighbour Table:  
X|1  
Y|3

Z LSDB:  
X|Y|5  
X|Z|1  
Y|Z|3

Z Routing Table:

```
X|X|1
Y|Y|3
```

```
X Neighbour Table:
Y|5
```

```
X LSDB:
X|Y|5
Y|Z|3
```

```
X Routing Table:
Y|Y|5
Z|Y|8
```

```
Y Neighbour Table:
X|5
Z|3
```

```
Y LSDB:
X|Y|5
Y|Z|3
```

```
Y Routing Table:
X|X|5
Z|Z|3
```

```
Y Neighbour Table:
X|5
Z|9
```

```
Y LSDB:
X|Y|5
Y|Z|9
```


```
Y Routing Table:
X|X|5
Z|Z|9
```


```
Z Neighbour Table:
Y|9
```

```
Z LSDB:
X|Y|5
Y|Z|9
```

```
Z Routing Table:
X|Y|14
Y|Y|9
```

## Recommended Approach

1. Start by ensuring you're familiar with Dijkstra's algorithm.
  - Review the course notes, sections 5.2.1 Kurose & Ross (7th Ed.), and the [Wikipedia entry](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)  ([https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)).
  - Be sure to add logbook entries as you go.
2. Manually determine the expected distance and routing tables at each step for the sample topology

- Feel free to ask questions and check your tables with your peers on Piazza.
  - Be sure to add logbook entries as you go.
3. Plan your implementation
- Determine what data structures you'll need, choose a programming language, plan how you're going to parse the input and generate output, plan your algorithm's implementation.
  - Be sure to add logbook entries as you go.
4. Implementation
- Develop your implementation, testing as you go.
  - Write a makefile (not needed for Python).
  - Be **sure to add logbook entries** as you go.
5. Testing
- Ensure your code runs on the university cloud (e.g. `uss.cs.adelaide.edu.au`).
  - Develop additional scenarios and topologies to ensure your systems function as expected.
  - Be sure to add logbook entries as you go.
  - Use **input/output redirection** (you don't have to type inputs using a keyboard each time) - read the input from a file and redirect the output into a file for *easy testing and diff'ing* (see: <https://www.geeksforgeeks.org/input-output-redirection-in-linux/>  <https://www.geeksforgeeks.org/input-output-redirection-in-linux/>.)
- 

## Submission, Assessment & Marking

### Submission

Create and checkout a new folder in your SVN repository at

<https://version-control.adelaide.edu.au/svn/aXXXXXXX/yyyy/s1/cna/routing>

### Assessment

The **part 1** of the assignment is marked out of **150**. These marks are allocated as follows using automated testing:

- Acceptance Testing (available to each submission before deadline; see below for details) (**total of 70 marks**)
  - *Dijkstra* correctly calculates the routing table for sample configuration: **40 marks**
  - *Dijkstra* correctly calculates the routing table after the link weights are changed: **30 marks**
- Full Testing (***applied only after the deadline***; see below for details) (**total of 80 marks**)
  - *Dijkstra*, correctly calculate tables for arbitrary unseen networks: **80 marks**

The **part 2 (PG only)** of the assignment is marked out of **50**. These marks are allocated as follows using automated testing and manual marking:

- Full Testing (***applied only after the deadline***; see below for details) **(50 marks)**
  - *DijkstraNlogN*, correctly calculate tables for arbitrary unseen networks: **(20 marks)**
  - Report, test cases, an automated script for reproducing the results in the report: **(30 marks)**

However, your marks are **scaled and reviewed** based on the following:

1. Up to 10 marks may be deducted for poor code quality. Below is a code quality checklist to help (notably this is not an exhaustive list but describes our expectations):
  - write comments above the header of each of your methods, describing
  - what the method is doing, what are its inputs and expected outputs
  - describe in the comments any special cases
  - create modular code, following cohesion and coupling principles
  - don't use magic numbers
  - don't misspell your comments
  - don't use incomprehensible variable names
  - don't have long methods (not more than 80 lines)
  - don't have TODO blocks remaining
2. As noted earlier, up to 150/200 marks (all marks) may be deducted for poor/insufficient/missing evidence of development process.

The two above will be assessed manually. To obtain all of the marks allocated from tests, *you will need to ensure your code is of sufficient quality and document your development process using the logbook entries.*

## Marking Process

You should not be using Web Submission for debugging. As part of your design phase, you should work out what sequence of updates you expect to happen and what you expect the final neighbour tables, link-state database, and routing table will be. *There are no partial marks for implementation and all marks are assigned based on passing test cases.* As such your submission will be marked in **3 stages**:

1. All submissions before the deadline will be run against an acceptance testing script.
  - This script will compile your programs and run your Dijkstra code for the sample config.
  - Use this as a sanity check to ensure your code compiles and runs on the Web Submission system and works as expected.
  - Be sure to add a logbook **entry for each submission you** make here.
2. After the deadline **your most recent submission prior to the deadline** will be run against a set of additional tests by a marker.
  - So, it is important that you've thoroughly tested your implementation with your own routing

configurations and changes and test cases you have developed (other than what we provide you).

3. Your code will then be reviewed for quality and evidence of your development process by a marker. Marks will be adjusted if your code and/or development process are not documented in the manner we have asked for.
4. **For postgraduate students (PG)**, we will also mark your report, including running your script, following the instructions in the PDF report to obtain the results (data points) used in your report.