

Assignment Two – 25%

Algorithms and Data Structures – COMP3506/7505 – Semester 2, 2023

Due: 3pm on Friday October 13th (week 11)

Summary

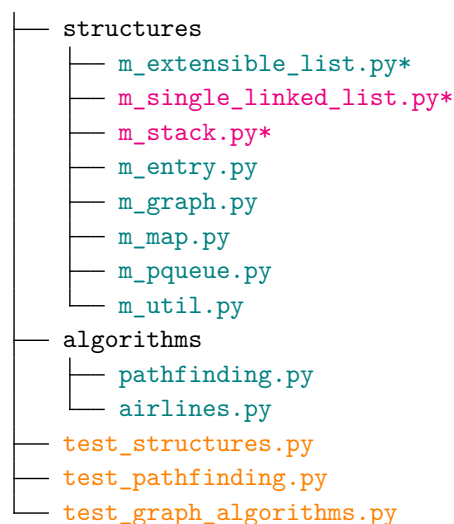
The main objective of this assignment is to extend your knowledge from assignment one to build more complex data structures and solve more complex problems. In particular, you will be efficiently solving pathfinding problems over arbitrary graphs. In this second assignment, we will leave more of the design choices up to you; you are expected to justify your choices in a short report. This assessment will make up 25% of your total grade.

A Getting Started

The assignment is structured similarly to *assignment one*. The skeleton codebase, data, software dependencies, implementation rules, and the report format are described below.

A.1 Codebase

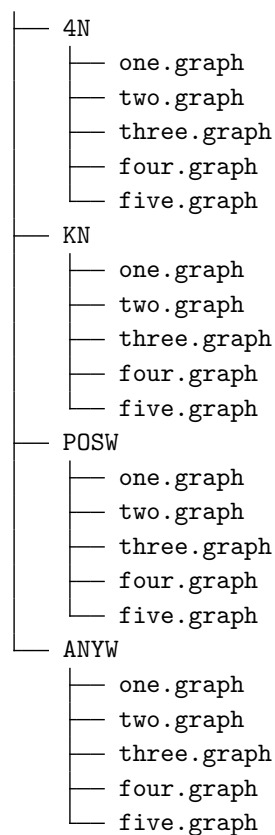
The codebase contains a number of data structures stubs that you must implement, as well as some scripts that allow your code to be tested. Figure 1 shows a snapshot of the project directory tree with the different files categorized. Note that the data structures built during assignment one (denoted with *) are provided for your reference; you may make use of these, or use your own implementation. It is possible to complete the assignment without modifying the files in pink. However, you are permitted to modify any of the files listed. You may also use `structures/m_util.py` for any utilities that do not deserve their own file, or add your own files.



■ **Figure 1** The directory tree organized by data structures (inside the `structures` directory), and the three executable programs (in the root directory, coloured orange).

A.2 Data

We also provide a number of test graphs for you to use, but you are encouraged to build further test graphs of your own; you may also share your test graphs with other students if you wish. Each graph is represented as a simple text file that stores an *adjacency list* for each vertex in the graph. There are four specific types of graphs, each with their own subdirectory. All graph types are *undirected* except for ANYW. 4N graphs are simple graphs where each vertex can be thought of as occupying a position on a grid/lattice. As such, these nodes can have *at most* 4 neighbours. KN graphs are an extension that allow an arbitrary number of neighbors. POSW graphs extend KN graphs to apply positive integer weights to edges. ANYW graphs extend POSW graphs to allow integer weights (may be positive or negative). The appendix in Section K contains an example of each graph type.



■ **Figure 2** The data tree organized by graph types. 4N are the most simple grid-based graphs. KN are graphs where each node has an arbitrary degree. POSW are graphs with arbitrary degree nodes and positive weights between the edges. ANYW are graphs with arbitrary degree nodes and arbitrary (potentially negative) weights between the (now directed) edges.

A.3 Dependencies

Our codebase is written for Python 3.10+ as we have provided type annotations; as such, you will need to use Python 3.10 at minimum. The second assignment has one special dependency – the `curses` library – that allows your algorithms to be visualized in a simple terminal window.

If you are developing locally, you may need to install `curses`. See the documentation¹ for more information. This library is already available on `moss`.

Note that you *can* do the entire assignment without using the visualizer, but it will be less fun and you won't be able to show off to your friends. The visualizer is only useful for the earlier pathfinding solutions on grids (Task 2), and it must be *executed in a terminal window*.

A.4 Implementation Rules

The following list outlines some important information regarding the skeleton code, and your implementation. If you have any doubts, please ask on Ed discussion.

- ▷ The code is written in Python and, in particular, should be executed with Python 3.10 or higher. The EAIT student server, `moss`, has Python 3.11 installed. We recommend using `moss` for the development and testing of your assignment, but you can use your own system if you wish.
- ▷ You are not allowed to use built-in methods or data structures – this is an algorithms and data structures course, after all. If you want to use a `dict` (aka `{}`), you will need to implement that yourself. Lists can be used as “dumb arrays” by manually allocating space like `myArray = [None] * 10` but you may not use built-ins like `append`, `clear`, `count`, `copy`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`, `min`, `max`, and so on. List slicing is also prohibited, as are functions like `sorted`, `reversed`, `zip`. Be sensible – if you need the functionality provided by these methods, you may implement them yourself. Similarly, don't use any other collections or structures such as `set`.
- ▷ You are not allowed to use libraries such as `numpy`, `pandas`, `scipy`, `collections`, etc.
- ▷ **Exceptions:** The only additional libraries you can use are `random`, `math`, and `functools` (but only for the `total_ordering` decorator). You are allowed to use `range` and `enumerate` to handle looping. You may use `tuples` (for example; `mytup = ("abc", 123)`) to store multiple objects of different types. If our functions return a list or tuple (or etc), you may use `len` on that structure. But in general, you should be using the `ExtList` to handle dynamic list behaviour.

A.5 Report

Please keep in mind that you will be justifying your implementation choices in a short report. The report *must* use the supplied template (see the LMS for the template). Your responses must be succinct and must remain within the provided space; the space is indicative of the maximum expected answer length; you are free to keep it shorter if you can. You may fill in the report by hand and scan it in if you wish, but your submission must adhere to the same formatting/specifications as the original digital template (that is, A4, PDF, and so on). Your handwriting must be legible or you risk losing marks.

¹ <https://docs.python.org/3/howto/curses.html>

B Task 1: Data Structures

We'll start off by implementing some important data structures that will be used in your pathfinding algorithms. All we specify is the interface; the choice of design is yours, as long as the interface behaves correctly and efficiently. You may test these with the `test_structures.py` program: `python3.11 test_structures.py`.

Task 1.1: Implement a (Priority) Queue (2 marks)

A queue is a data structure that can handle efficient access to elements on a first-in-first-out basis. Recall that a *priority* queue is an extension of a simple queue that supports efficient access to the element with the *highest priority*; new elements can be inserted with a given (arbitrary) priority, and the priority queue must be able to support efficient dequeue operations based on the priority order. For this assignment, we will assume priority values are integers in the range $[0, n]$ with 0 representing the *highest* priority.

Your first task is to examine the `m_pqueue.py` file to see the basic priority queue interface, and implement the interface efficiently using a concrete data structure of your choice. Note that your queue should also support “simple” first-in-first-out behaviour as well as handling arbitrary priorities. Hint: A simple FIFO queue can be implemented by always using the same priority value — you should not need a special function to enqueue or dequeue in FIFO order. Remember that you need to justify your implementation in the report.

Test via: `python3.11 test_structures.py --pq`

Task 1.2: Implement a Map (2 marks)

Your next job is to implement a concrete data structure to support the map interface. Recall that a map allows items to be stored via *unique keys*. In particular, given a key/value pair (k, v) (otherwise known as an entry), a map M can support efficient insertions (associate k with v in M), accesses (return the value v associated with k if $k \in M$), updates (update the value stored by key k from v to v') and deletes (remove (k, v) from M). In other words, you will be supporting operations like a Python `dict` (aka `{}`) class.

You can implement your map by filling in the interface provided inside the `m_map.py` file. Again, you must pay attention to the efficiency of your implementation and justify it in your report.

Test via: `python3.11 test_structures.py --map`

Task 1.3: Implement a Sorting Algorithm (1 mark)

Later in the assignment, you will need to be able to sort data. Your next task is to implement the `sort(self)` function inside the `ExtensibleList` class to allow this sorting to occur. Your sorting algorithm must run in (at most) $\mathcal{O}(n \log n)$ in *expectation*. There are many candidate sorting algorithms that conform to this complexity.

To sort a list containing specific objects, you will need to implement a comparator such as `__lt__(self, other)` within the class definition of that object. See the Week 5 lecture slides (slides 8 to 11) and related recording for an example of this. You may assume, for simplicity, that the sort will always use $<$ as the comparator (that is, you may *hard code* the sort comparison to use $a < b$ for arbitrary elements a and b , and that is achieved via the previously mentioned `__lt__` override).

In our testing, we will always assume the `ExtensibleList` contains only one type of object (with the exception of `NoneType` since your `ExtensibleList` may have empty elements at the end), so you do not need to handle comparisons between *different* objects. We will also assume there are not `None` objects interleaved with the other object type in the list.

Test via: `python3.11 test_structures.py --sort`

C Preliminaries: The Graph Class

The remainder of this assignment will focus on pathfinding problems over a *graph* data structure. We have provided the graph structure for you, and you will need to get familiar with it in order to make progress on the remaining tasks. The graph types are defined in `structures/m_graph.py`. There are two key types of graphs. The `Graph` class is the base class which stores nodes and edges. Each node in the graph (`Node`) stores an *id* which is the *index* of the node in the graph's adjacency list. For example, if you have a node with an id 22, this means that the node will be stored at the `Graph`'s `self._nodes[22]` and can be accessed via the `Graph`'s `get_node()` function. The `Graph` also provides a function to return a list of neighbours given an index/node identifier.

There is a special `LatticeGraph` type that extends the `Graph` class (and a `LatticeNode` that extends `Node`). This specialized graph is used only for graphs that are placed on a lattice. In other words, these graphs can be thought of as simple grids, where each vertex has between zero and four neighbors. As such, some additional properties including the number of logical rows and columns in the (4N) graph are stored. For your purposes, the only real difference you need to know about with this special type is that you can ask for the (x, y) coordinates of a given `LatticeNode` using the `get_coordinates()` function. You can also directly return the nodes to the north/south/east/west using the appropriate `get_north()` (etc) functions.

Note that graphs may be *disconnected* — that is, there may exist two vertices that do not have a path between them — and this needs to be handled by your algorithms using the `TraversalFailure` enum located inside `structures/m_util.py`. This enum provides `DISCONNECTED` (for the aforementioned situation) as well as `NEGATIVE_CYCLE` which will only be used in Task 3.5.

All of the following tasks have pre-made function stubs. You should pay close attention to the type hints so you know what is expected to be taken as parameters, and what should be returned.

D Task 2: Basic Pathfinding Problems

Optional Backstory: After refusing to buy avocados for the last 23 years, your long time friend and collaborator Barry Malloc has recently purchased his first house. His backyard is a beautiful wooded forest containing various walking trails weaving in and out of trees and through the bush. What the real estate agent neglected to tell him, however, was that the backyard is heavily “curated” by a local Australian Brush Turkey² named Gurkey Tobbler.

Being a very smart and analytically minded fellow, Barry has modelled the current state of his backyard as a *graph* data structure, and has built a small visualization tool to see the trails. His goal is to better understand how he might best traverse the maze of trails in order to catch Gurkey (while Gurkey is sleeping) so he can release Gurkey elsewhere.

Since you owe him one from his help on the DNA-Refgrid project at Frankenstein Labs, you agree to complete the pathfinding simulations to help Barry catch Gurkey Tobbler.

Getting Started

After completing the earlier tasks, along with the solutions from *assignment one*, you should have all of the data structures you need in order to solve some interesting pathfinding problems. However, you are free to implement more structures if you think they may help; store them in their own files inside the `structures` directory.

In this section, to get started, we will focus on *lattice graphs*. Note that we have provided some graphs for you already, and the ones we are interested (for now) are those in the `data/4N` directory. However, we will extend to more complex graphs in the next sections, so keep this in mind when you implement your solutions here.

We have provided a program called `test_pathfinding.py` to help you get started. This program allows different pathfinding algorithms through **two dimensional** mazes to be tested (mandatory) and visualized (optional). Note that in order to make life easier, we’re randomly generating the origin and goal vertices, so you will need to supply a seed to the random number generator (via `--seed`) to yield different origins/goals each time you run the program. All implementations for Task 2 must be inside the `algorithms/pathfinding.py` file, where appropriate stubs are provided for you.

Task 2.1: Implement Depth-First and Breadth-First Search (2 marks)³

Two fundamental algorithms for pathfinding are *depth-first search* (DFS) and *breadth-first search* (BFS). Given some arbitrary start vertex A , and some goal vertex B , both algorithms systematically walk across the graph until they either find B or run out of vertices to explore. The main difference between them is the way in which the *unvisited* set of vertices is maintained; DFS uses a stack, and BFS uses a queue. Figure 3 and Figure 4 provide sketches of this process. Implement your algorithms inside the `dfs_traversal()` and `bfs_traversal()` stubs. Please see the type annotations for the specific details about what these functions should return.

To make your results reproducible, you must enqueue/push the unvisited neighbours *in the order they are given to you* from the `get_neighbours()` function.

Finally, while we will be visualizing our DFS and BFS on the lattice graphs, **you must** ensure that your algorithms translate to graphs with arbitrary degree. This should be

² https://en.wikipedia.org/wiki/Australian_brushturkey

³ Be careful about self-plagiarism, especially if you’ve done this in other courses...

trivial to implement. For the avoidance of doubt, your DFS and BFS algorithms **will** be tested on the KN graphs. Hint: You have implemented something quite similar to DFS before...

Test via: `python3.11 test_pathfinding.py --graph data/4N/one.graph --dfs --seed <number> [--viz]`

Note that the `--viz` flag is optional (and triggers the visualizer to run) and `<number>` should be substituted with an integer. You can test BFS by changing `--dfs` to `--bfs`.

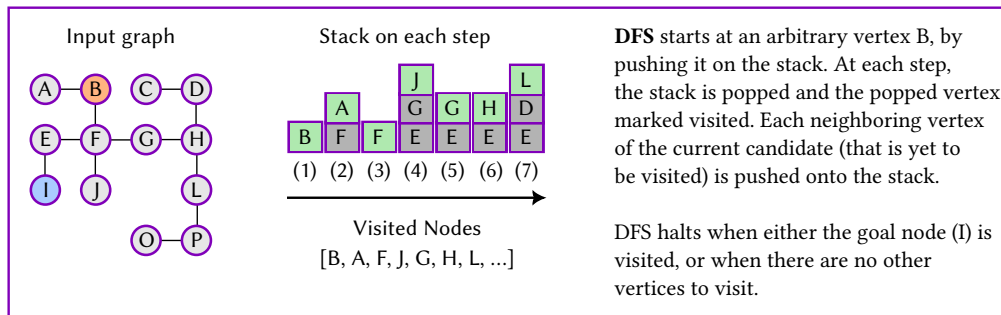


Figure 3 A sketch of depth-first search starting at vertex *B* and searching for vertex *I*. A stack keeps track of the next vertices to visit, and they are visited as they are popped from the stack. A list can be used to track the order in which nodes are visited.

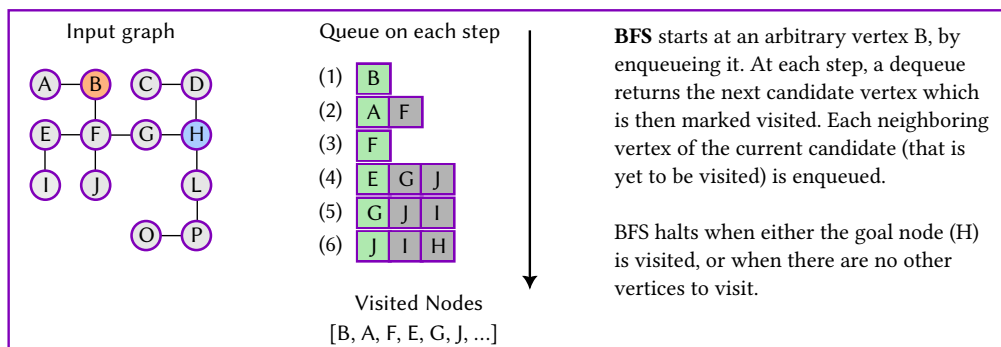


Figure 4 A sketch of breadth-first search starting at vertex *B* and searching for node *H*. A queue keeps track of the next vertices to visit, and they are visited as they are dequeued. A list can be used to track the order in which nodes are visited.

Task 2.2: Implement Greedy Search (2 marks)

While BFS and DFS are useful, they are still not quite good enough. Barry is having limited success, as the pathfinding is taking too long... What we really need is a way to inform the *direction* of our algorithm so that we always move towards the goal. Given that we're operating on a 4N-grid, we can implement some sort of *distance function* that, given the coordinates of two nodes (x_1, y_1) and (x_2, y_2) , can estimate the distance between those nodes. We can then use this distance to *prioritize* the next element to be examined. We call this *greedy* or *best-first* search, as we always dequeue what we think is the best node to examine next. Your task is to implement an appropriate distance metric (there are many acceptable choices such as Manhattan, Chebyshev, and Euclidean distance) and extend your BFS implementation to use a *priority queue* to ensure the frontier is explored in a strategic

way. You can access the coordinates of any given node in the 4N graph by calling the `get_coordinates()` method belonging to the `LatticeNode` type. Implement your algorithm inside the `greedy_traversal()` stub. There is also a `distance()` stub provided for you. Note that the algorithm is exactly the same as BFS, except the queue is now a priority queue, and the enqueue/dequeue operations need to deal with elements in terms of their distance to the goal node. Simple! Note that we *do not* expect your solution to work on any other graph types, as the distance computation relies on having points in a 2-dimensional space.

Task 2.3: Run Away!!!! (1 Mark for 7505; 1 Bonus Mark for 3506)

Barry has discreetly let you know that while your fantastic work on the DFS/BFS/Greedy algorithms has helped him track down Gurkey, he's actually too frightened to catch him; it turns out Turkeys have large talons. However, Barry's wife Sally Malloc (née Free) is not thrilled at Barry's reluctance. As such, Barry needs a scapegoat — yes, it's you! — and he's asked you to design an algorithm that *maximizes* the total number of vertex visits across the graph *before* discovering the location of Gurkey. This will make him look busy but avoid conflict with the turkey. Implement your algorithm inside the `max_traversal()` stub. You should explain how your algorithm works in the report. As in 2.2, this algorithm is only expected to work on the lattice-based graphs.

If you are a COMP3506 student, you may implement this algorithm for a bonus mark; if you implement this algorithm, you must also respond in the corresponding section of the report (or you will not receive the bonus mark). To be clear, this is 1 bonus mark in total. You do not get 1 bonus mark for this algorithm and another bonus mark for the corresponding report question.

E Task 3: Extended Graph Problems

After your great success with Task 2, Barry has decided that your algorithmic knowledge is good enough to solve many other interesting real-world problems. However, we've only explored graphs where each vertex has at most degree 4 (neighbours could be left, right, above or below on the grid) and edges are unweighted. The next set of tasks extend your previous solutions/experience to design and build algorithms that can tackle more difficult problems by casting them into more complex graph types. These graphs will contain cycles among other complexities outlined in each task. It is also now time to say goodbye to our visualization tool as it can only cope with 4N graphs; **it will not work for the remaining tasks**. The remaining tasks can be tested using `test_graph_algorithms.py` with the appropriate flags.

The following problems will be solved on behalf of *Bogan Airlines* (BA), a next-generation flight company.

Task 3.1: Cycle Detection (2 marks)

BA is trying to generate new flight paths to improve the efficiency of their operations. Each path is represented by waypoints corresponding to some GPS coordinates. However, it is important that the paths generated by BA do not contain cycles, as this could result in unsafe separability between aircraft on the same path.

Given a candidate graph G where vertices represent waypoints and the edges between them correspond to viable flight paths, your task is to determine whether G contains any cycles (that is, a non-empty path from any vertex A that returns to itself via a non-zero amount of other vertices and only traversing unique edges). Your algorithm simply has to return `True` if a cycle is detected, or `False` otherwise.

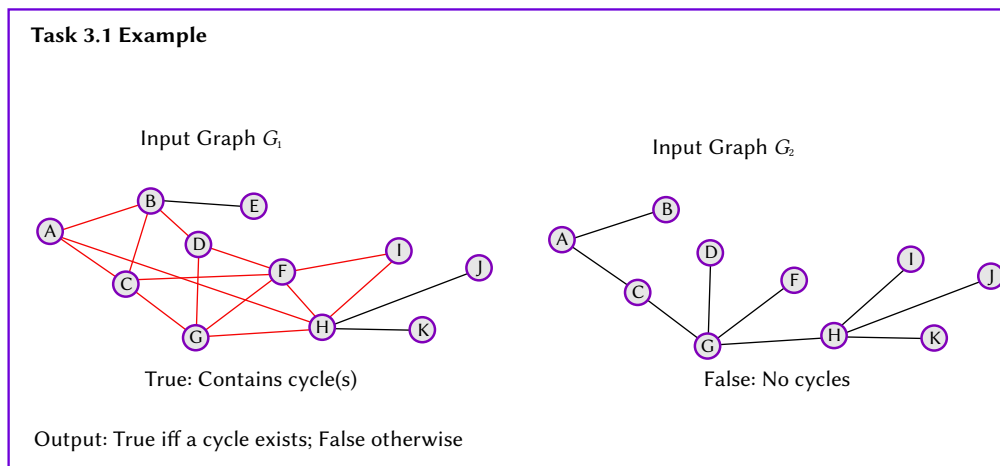
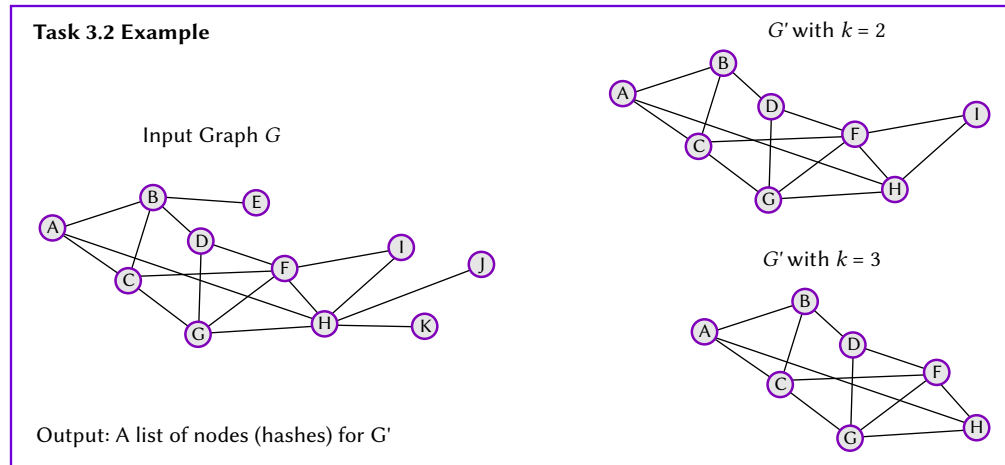


Figure 5 A sketch of the cycle detection problem. The left graph has many cycles; $[A, B, C, A]$, $[B, D, F, H, G, C, A, B]$, and etc. The right has none.

Task 3.2: Hub Enumeration (2 marks)

BA, being a new airline, has plans for expansion. As part of this expansion, there are numerous candidate destination cities that are yet to be serviced by BA. However, before committing to expansion, BA needs to find the current set of *hub* airports, as they do not wish to expand to airports that are outside their critical network.

Given a candidate graph G , where vertices represent cities, and edges represent flights between those cities, design an algorithm that can return a list of vertices corresponding to the *largest subgraph* G' of G where each vertex $g \in G'$ has degree of at least k .



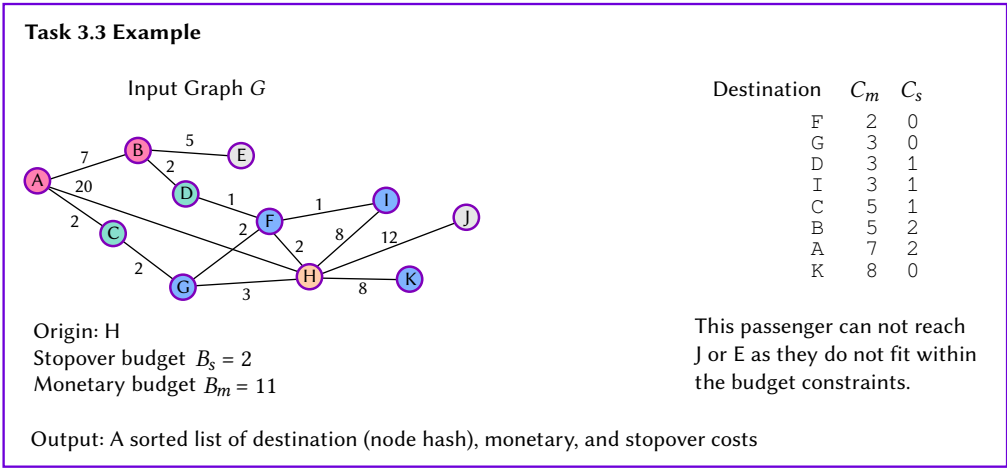
■ **Figure 6** A sketch of the hub enumeration problem. The top right graph contains the largest subgraph G' of G (left) with nodes of degree $k \geq 2$. The lower right is the same, but with $k \geq 3$.

Task 3.3: Big Bogan Budget Bonanza (4B problem) (2 marks)

BA is having a big budget bonanza sale. The sale consists of discounted airline fares across many of their hubs, allowing people to jet around the world at highly competitive prices! However, it turns out that some of BA’s clientele are not willing to make a large number of stopovers to get to their destination, and are preferring to fly with other companies. In order to help filter their flight searches, BA want to implement a flight budget calculator.

Given an *origin* A (some vertex in the input graph G), a *total stopover budget* B_s (representing the total number of stopovers the given passenger is willing to take to get to the given destination), and a monetary budget B_m (the maximum total cost of the outbound trip), we must return a sorted ExtensibleList of *viable destinations* with their corresponding stopover and monetary cost (C_s and C_m). The list should be sorted ascending on the monetary cost C_m , with ties broken by prioritizing destinations with a lower stopover cost C_s . If both C_m and C_s are tied for any destinations, break ties on the node identifiers (ascending). Both costs should correspond to only the outbound trip. Furthermore, if there is more than one viable way to a given destination, the one with the *lowest* C_m should be the one returned (we prefer to minimize C_m as our primary objective so long as the trip still satisfies the stopover budget B_s).

Note that we’re in a situation where our graph contains positively weighted edges between vertices (where the vertices represent cities, and the edge weights represent the monetary cost of a flight in either direction between those cities). These are the POSW graphs provided in the data.

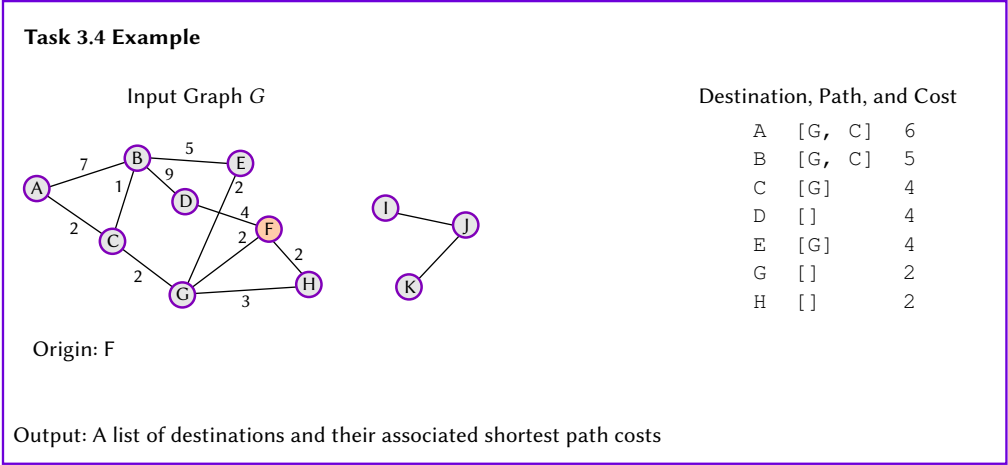


■ **Figure 7** A sketch of the 4B problem. Note that D and I are tied in both dimensions, so D comes before I. In practice, you will have integer node identifiers (not letters like A, B, ...) but the same idea applies.

Task 3.4: BA Field Maintenance Optimisation (3 marks)

Another problem that BA faces is the rising cost of *field maintenance*, where parts and engineers must be ferried to an aircraft at some location in order to allow the airframe to be maintained for a safe return flight. The problem arises when an airframe at some *origin* is requiring maintenance, but either the suitable engineering team or parts are not available.

In order to decide which engineering team to send (or where the parts should come from), we need an algorithm to return the *minimum cost path* from the *origin* to *all other destinations*. That is, given origin *A*, you must return a list containing the *lowest cost path* to *every other reachable destination*. This cost is, again, based on minimizing the *sum of the edge weights*. If a destination is not reachable, you should not return it in your list.



■ **Figure 8** A sketch of the field maintenance optimisation problem.

Task 3.5: All City Logistics (2 marks)

Warning: This is a difficult problem. Please be aware of this, and consider how best to spend your assignment time. I expect less than 5% of students will correctly solve this problem.

After doing some data analytics on BA’s company logistics, a new cost-benefit forecasting model has been generated. This model can be represented as a graph where *negative* edge weights represent *savings* that BA can benefit from, and *positive* edge weights represent *costs* that BA must incur. Intuitively, an edge with a negative weight may represent a flight from city *A* to *B* that is completely sold out on a newer, efficient aircraft, whereas a positive weight may represent ferrying an empty cargo plane. However, these are simply examples; the actual weights represent averages taken across various logistic measurements that represent typical airport-to-airport flight movements.

Given a weighted and directed graph *G* (with potentially negative weights), your task is to find the *lowest cost path* between *every pair of vertices* *A, B* ∈ *G*. However, there is a twist. You must compute your solution once in a *pre-processing* step, and return a **Map** that can return the cost from an arbitrary origin node *A* to a goal node *B* in (expected) $\mathcal{O}(1)$ time.

If a negative cycle⁴ is found, the value returned for the given *A, B* pair should be the special **NEGATIVE_CYCLE** marker located inside the **TraversalFailure** enum (**structures/m_util.py**). You can also use **DISCONNECTED** when *B* cannot be reached from *A*.

Our graph framework will ensure that the provided neighbours for a given node obey the directed nature of the graph — you do not need to check this yourself.

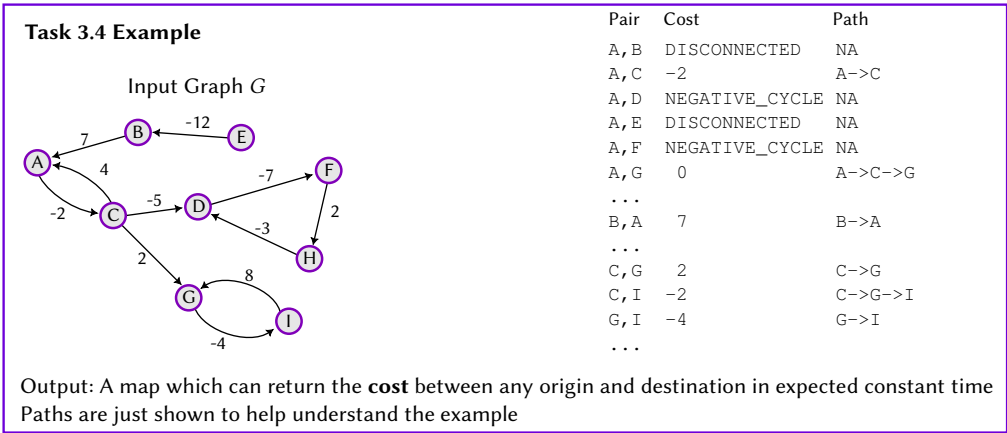


Figure 9 A sketch of the all city logistics problem.

F Task 4: Report (5 marks for 3506; 7 marks for 7505)

As mentioned in Section A.5, you need to justify your implementations in a report. A template with further instructions and specific questions is provided to you on the LMS. Your answers should be clear and concise. You must answer each question in entirety. Please see the Section A.5 and the assessment guide below for additional information.

⁴ See the appendix

G Submission

You need to submit your programming solution to *Gradescope* under the *Assignment 2: Autograder* link in your dashboard. You also need to submit your report, separately, to the *Assignment 2: Report* link. Once you submit your solution, a series of tests will be conducted and the results will be provided to you. However, the assessment will also include a number of additional *hidden* tests, so you should make sure you test your solutions extensively. You may resubmit as often as you like before the deadline.

Structure

The easiest way to submit your solution is to submit a `.zip` file. The autograder expects a specific directory structure for your solution, and the tests will fail if you do not use this structure. In particular, you should use the same structure as the skeleton codebase that was provided. Failure to do so will result in Gradescope failing all of the tests. Please check that your submission has been submitted correctly, ensuring that the files are not nested within an additional directory. **Submissions without the declaration of originality (`statement.txt`) will be given zero marks with no exceptions.** See Section H.3 for more information.

H Assessment

This section briefly describes how your assignment will be assessed.

H.1 Mark Allocation (Code)

Marks will be provided based on an extensive (hidden) set of unit tests. These tests will do their best to break your data structure in terms of time and/or correctness, so you need to pay careful attention to the efficiency and the validity of your code. Each test passed will carry some weight, and your autograder score will be computed based on the outcome of the test suite. If you did not rigorously test your programs/code, you should go back and do so!

The marks provided in each task above are indicative of the total score available for each part, but marks may be taken off for poor coding style including lack of commenting, inefficient solutions, and incorrect solutions. Our code quality checks are not as strict as PEP8, but we assume typical best practices such as informative variable and function names, commenting, and breaking long lines. While the overall grade/score will be calculated mathematically, an indicative rubric is provided as follows:

- ▷ **Excellent:** Passes at least 90% of test cases, failing only sophisticated or tricky tests; well structured and commented code; appropriate design choices; appropriate selection and application of data structures.
- ▷ **Good:** Passes at least 80% of test cases, failing one or two simple tests; well structured and commented code; good design choices with some minor improvements possible; good application of data structures with some minor improvements possible.
- ▷ **Satisfactory:** Passes at least 70% of test cases; code is reasonably well structured with some comments; most design choices are reasonable but significant room for improvement; reasonable application of data structures but significant improvements possible.
- ▷ **Poor:** Passes less than 70% of test cases; code is difficult to read, not well structured, or lacks comments; design choices do not demonstrate a sound understanding of the desired functionality; little or no suitable application of data structures.

H.2 Mark Allocation (Report)

Your report will be assessed based on its accuracy with respect to your code. You should keep your answers succinct, and ensure you answer each (sub) question being asked. The amount of space provided for each response can be used as a rough guide on the expected level of detail. An indicative rubric is provided as follows:

- ▷ **Excellent:** Clearly and succinctly answers all questions with accurate and correct responses. Justifications are clear and relevant, and support the choices made.
- ▷ **Good:** Answers most questions accurately and correctly with some minor mistakes or unanswered components. Some choices are not entirely justified.
- ▷ **Satisfactory:** Answers at least half of the questions accurately and correctly with some mistakes or unanswered components. Design choices are not entirely justified or correct. Clarity of responses could be improved.
- ▷ **Poor:** Does not answer the majority of questions. Answers or justifications do not align with the submitted code. Answers are difficult to understand or follow.

H.3 Plagiarism and Generative AI

It's 2023; we know all about generative AI technology like ChatGPT or Github Copilot. The course ECP actually has a statement about the use of such technology, repeated here:

It is the position of UQ that the use of AI outputs without attribution, and contrary to any direction by teaching staff, is a form of plagiarism and constitutes academic misconduct. The assessment tasks evaluate students' abilities, skills and knowledge without the aid of Generative Artificial Intelligence (AI). Students are advised that the use of AI technologies to develop responses without attribution is strictly prohibited and may constitute student misconduct under the Student Code of Conduct.

If you want to actually learn something in this course, our recommendation is that you avoid using such tools: You need to think about what you are doing, and why, in order to put the theory (what we talk about in the lectures and tutorials) into practical knowledge that you can use. This is often what makes things “click” when learning. Mindlessly lifting code from an AI engine won't teach you how to solve algorithms problems.

If you are still tempted, note that we will be running your assignments through sophisticated software similarity checking systems against a number of samples including including your classmates' and our own solutions (including 30 AI generated). Note also that the exam may contain questions or scenarios derived from those presented in the assignment work, so cheating could weaken your chances of successfully passing the exam.

As part of your submission, you should create a file called `statement.txt` containing the following text:

I, [firstname lastname (studentid)], hereby declare that this is my own original work, and that no part of this assignment has been copied from any other source or person except where due acknowledgement is made; no part of the work has been previously submitted for assessment in this or any other institution except where explicitly acknowledged; I/We have read PPL 3.60.04, UQ's Student Integrity and Misconduct Policy and understand its implications.

In the same file, you should then provide attribution to any sources or tools used to help you with your assignment, including any prompts provided to AI tooling. If you did not use any references or resources, you **must explicitly state that**. This includes any previous coursework that you have referred to in order to implement your solutions to this assessment.

I Resources

We provide a number of useful git and/or unix resources to help you get started. Please go onto the Blackboard LMS and see the **Learning Resources > Resources** directory for more information. Please also check and search Ed regularly. There is also a FAQ/Assignment Specification “reading” on the pinned Ed post. Keep an eye out for a video accompanying this specification, and assignment help sessions — these are in the works.


J Changelog

- ▷ V1.0: Initial release.
- ▷ V1.1: Added a negative cycle to the figure in the appendix.

K Additional Details and Information

The remainder of the report contains additional details or descriptions that may be helpful for your understanding.

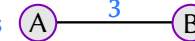
Graphs at a Glance

A vertex (or a node) 

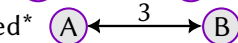
Edges connect vertices



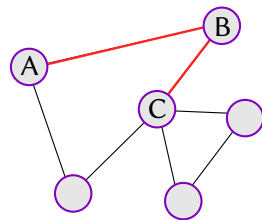
Edges may have weights



Our graphs are undirected*

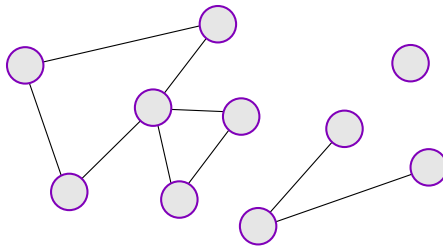


*Except for ANYW which are only used in Task 3.5

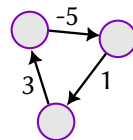
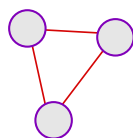


[A, B, C] is a path

A path (for our purposes) is a sequence of vertices connected by edges



Graphs may be disconnected!



Negative cycles are cycles where the sum of edge weights is negative.

Graphs may contain cycles! (Except for 4N graphs)

Figure 10 A brief look at some graph terminology. This may help you to conceptualize the problems we are solving.

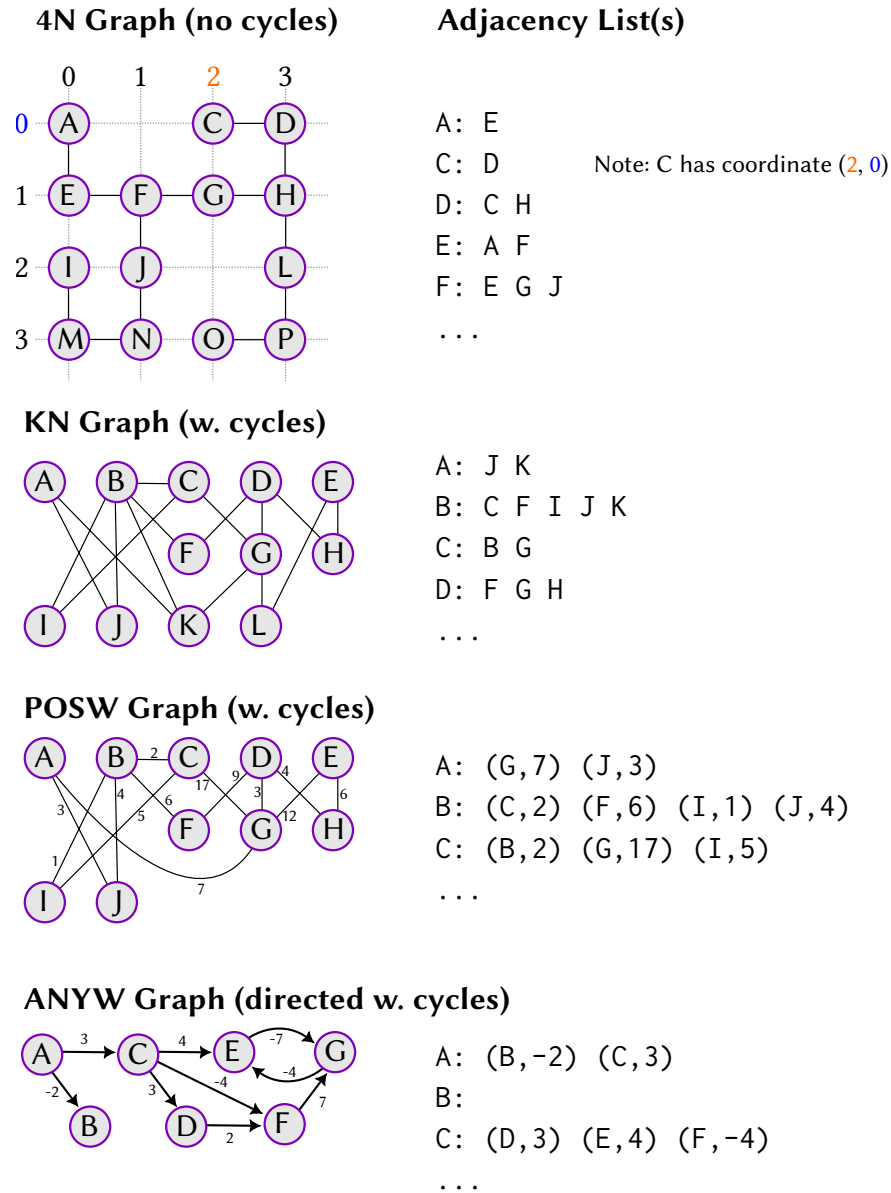


Figure 11 The four types of graphs used in this assignment. 4N graphs are based on grids and each node has an (x, y) coordinate associated. 4N graphs will not have cycles. KN graphs have an arbitrary degree for each node and may include cycles. POSW graphs are the same as KN graphs but have positively weighted edges. ANYW graphs are the same as POSW graphs but edges can have arbitrary (integer) weights including negative weights, and edges are directed. All graph types except for ANYW are undirected, and their adjacency lists are shown to the right.