Assignment Two – 25%

Algorithms and Data Structures – COMP3506/7505 – Semester 2, 2024

Due: 3pm on Friday October 18th (week 12)

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 working on graph and compression problems. In this second assignment, we will leave more of the design choices up to you (like the k-mers part in A1). This assessment will make up 25% of your total grade. **We recommend you start early.**

A Getting Started

The assignment is structured similarly to *assignment one*. The skeleton codebase, data, software dependencies, implementation rules, are described below. **Rules for success:** Think before you code. Think before you post an Ed question. Use a pen and paper. Don't be afraid to be wrong. Give yourself time to think. Start thinking about these problems early. Read the entire spec before you do anything at all.

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 we provide you with (simplified) versions of the data structures built during assignment one. You are permitted to modify any of the files listed. You may also use `structures/util.py` for any utilities that do not deserve their own file, or add your own data structures if you think they may help; store them in their own files inside the `structures` directory.

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 three specific types of graphs, each with their own subdirectory. All graph types are *undirected*. `4N` graphs are simple graphs where each vertex can be thought of as occupying a position on a square 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. The appendix in Section M contains an example of each graph type.

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.

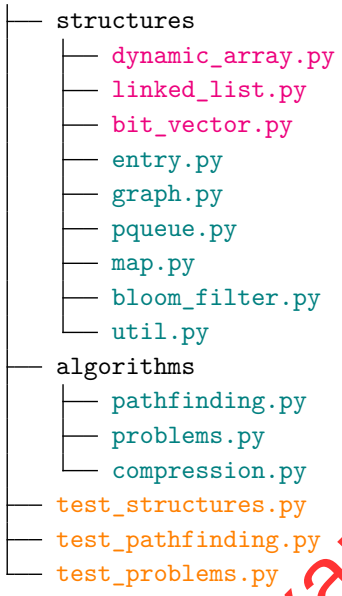


Figure 1 The directory tree organized by data structures (inside the `structures` directory), and the three executable programs (in the root directory, coloured orange).

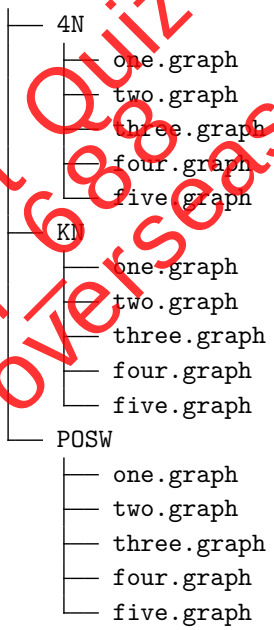


Figure 2 The data tree organized by graph types. `4N` are the most simple grid-based graphs. `KN` are graphs where each node has an arbitrary degree. `POSW` are graphs with arbitrary degree nodes and positive weights between the edges.

If you are developing locally, you may need to install `curses`. See the documentation[1] for more information. This library is already available on `moss`.

[1] `https://docs.python.org/3/howto/curses.html`

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*.

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 `clear`, `count`, `copy`, `extend`, `index`, `insert`, `remove`, `reverse`, `sort`, `min`, `max`, and so on. List functions like `sorted`, `reversed`, `zip` are also banned. Similarly, don't use any other collections or structures such as `set`. You cannot use the default `hash` function. Be sensible – if you need the functionality provided by these methods, you may implement them yourself.
- 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. You may use `len` wherever you like, and you can use list slicing if given a Python list as input. If we ask for a Python list in a function return type, you can use `append` or `pop`.

B Task 1: Data Structures

We'll start off by implementing some new data structures. 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_stuctures.py` program: `python3.11 test_structures.py`.

Task 1.1: Fix and Extend the Priority Queue (3 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 numeric and comparable (so, they may be floats or integers), with *lower* values representing a *higher* priority. In other words, we're going to be supporting a *min-heap*.

We have provided you with a semi-working priority queue in `pqueue.py`. Unfortunately, we ran out of time to get it working perfectly, so there are a few[2] bugs lurking in the implementation. First, crush these bugs so the priority queue works properly (1 mark)!

Once your heap is operating correctly, we need to handle a few more subtleties; we'd like to support *in-place construction* in *linear time* through the `ip_build` function, and *in-place sorting* via the `sort` function. Note that the *in-place* operations should operate directly on the data array without creating a copy — running in-place heap sort will yield a sorted array, but will destroy the heap ordering. As such, you may assume the user will no longer use the heap once the `sort` function has been used. Welcome to UQ(queue).

You can test via: `python3.11 test_structures.py --pq`

Task 1.2: Implement a Map (3 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.

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

Task 1.3: Implement a Bloom Filter (3 marks)

Bloom filters are an interesting probabilistic data structure that can support extremely efficient and compact set *membership* operations. In particular, they use hashing in combination with bitvectors to toggle on sets of bits; when looking up a given key k, the key is hashed via a series of unique hash functions and mapped to various indexes of the bitvector. Next, these bits are observed; if they are all *on*, then we return `True` which means "yes, this key *might* be in the set." Otherwise, we return `False` which means "No, this key is *definitely not* in the set." Your Bloom filter does not need to double check that the `True` values are definitely in the set; that job is for another data structure.

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

---

[2] $3 \leq$ few ≤ 7 – Logic errors

[3] Sorting will be done via comparators like $<$ or $>$ — we will only ever pass objects that are comparable.

C Preliminaries: The Graph Class

Many of the following problems (all of Task 2, and some aspects of Task 3) will require the use of a *graph* data structure. We have provided a concrete implementation of a graph data structure for you, and you will need to get familiar with it in order to progress. The graph types are defined in `structures/graph.py`.

Graph Types

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.

Your Implementations

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.

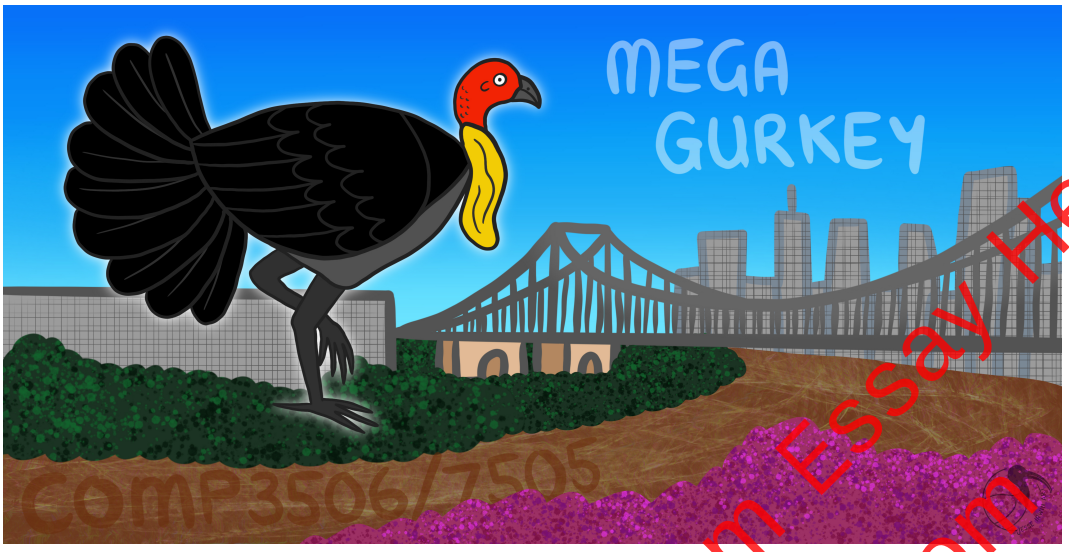


Figure 3 The Mega Gurkey – Artwork by Jesse Irwin.

D (Optional) Backstory for the Remainder of Assignment Two

Last year, the COMP3506/7505 cohort helped Barry Malloc capture an enterprising Australian Brush Turkey[4] (named Gurkey Tobbler) that was ruining his garden. Afterwards, the chief scientist at `MallocLabs` (Dr. Amongus) transported Gurkey to the lab to conduct some genomic sequencing. Thanks to your great work on DNA compatibility, Dr. Amongus has since discovered that Gurkey DNA is compatible with that of the *Loxodonta Africana*, the African Bush Elephant![5]

While this is a crowning scientific discovery, there is one (big) problem; Dr. Amongus has created a giant hybrid mega Gurkey through the irresponsible use of genetic modification tools. Our goal in this section is to find the mega Gurkey before it's too late, and to help Barry conduct further analysis on the Gurkey genome.

Meta comment: Why do we make up these crazy backstories and bury details inside them? Well, because you need to practice looking at a problem you have and extracting the important details. It is highly unlikely you will ever be given an extremely well specified problem. It is also a lot more fun this way :-)

[4] `https://en.wikipedia.org/wiki/Australian_brushturkey`
[5] `https://en.wikipedia.org/wiki/African_bush_elephant`

E Task 2: Pathfinding Algorithms

Getting Started

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, your solutions here must also work on the `data/KN` and `data/POSW` graphs (note that `KN` and `POSW` are the same types of graph if an algorithm does not use edge weights).

We have provided a program called `test_pathfinding.py` to help you test your algorithms. 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 and 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: Breadth-First Search (2 marks)[6]

Given some arbitrary start vertex u, and some goal vertex v, Breadth-First Search (BFS) systematically walks across the graph until either v is found, or there are no remaining vertices to explore. Figure 4 provides a sketch of this process. You must implement the `bfs_traversal()` stub; note that both the visited list, and the path, are expected to be returned. Please see the type annotations for the specific details about what should be returned.

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 BFS on the lattice graphs, **you must** ensure that your algorithms translate to graphs with arbitrary degree. This should be trivial to implement. For the avoidance of doubt, your BFS algorithm **will** be tested on the `KN` graphs.

Test via: `python3.11 test_pathfinding.py --graph data/4N/one.graph --bfs --seed <number> [--viz]`
Note that the `--viz` flag is optional (and triggers the visualizer to run) and `<number>` should be substituted with an integer.

Task 2.2: Dijkstra's Algorithm (3 marks)

BFS is nice; it is quite simple and it works well at finding the Gurkey when the graph is unweighted. However, Brisbane is a hilly city, and some paths are more expensive than others; we'll need to take this into account to find the true shortest path to the Gurkey. We also don't necessarily know where the Gurkey will be, so it would be good to find the shortest path from our current location to *all* possible locations.

Your goal is as follows. Given a weighted graph and a source node, return the *cost of the lowest-cost path to all reachable nodes.* If a node is not reachable (for instance, if the Gurkey has destroyed all of the bridges) then you should not return it in your list. Please see the type annotations for the specific details about what this function should return.

Test via: `python3.11 test_pathfinding.py --graph data/POSW/one.graph --dijkstra --seed <number>`

[6] Be careful about self-plagiarism, especially if you've done this in other courses...

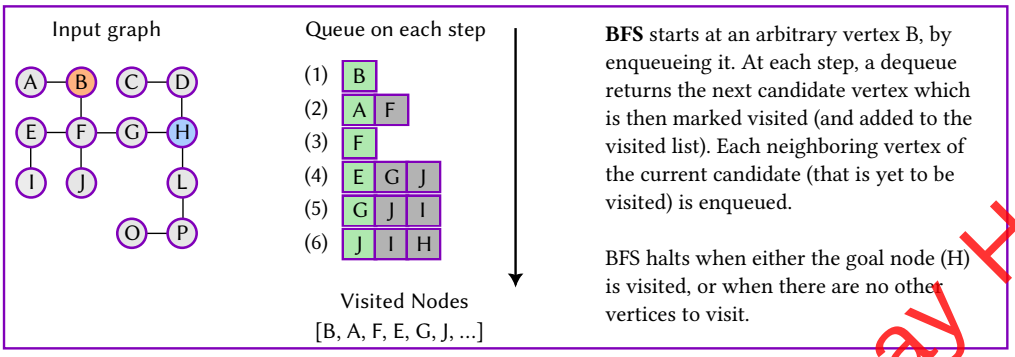


Figure 4 A sketch of breadth-first search starting at vertex B and searching for vertex H. A queue keeps track of the next vertices to visit, and they are visited as they are dequeued. A list can be used to track the order in which nodes are visited.

Note that it does not really make sense to use the `--viz` flag with Dijkstra, because the `4N` graphs do not have edge weights (and the viz tool needs to use `4N` graphs).

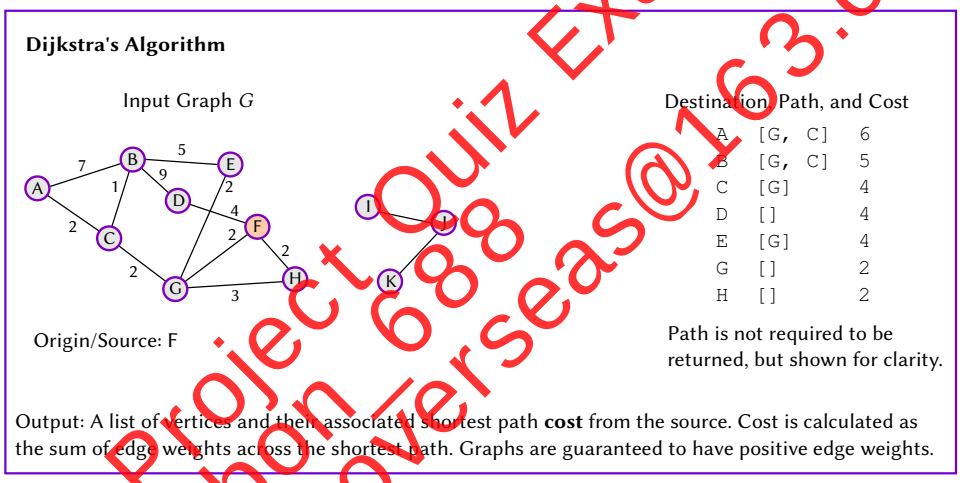


Figure 5 A sketch of Dijkstra's algorithm. See the code for the expected output format and structure.

Task 2.3: Depth-First Search (2 marks – COMP7505 only)

Depth-First Search (DFS) operates very similarly to Breadth-First Search. However, instead of using a FIFO queue, it uses a LIFO stack. You must implement the `dfs_traversal()` stub (plus any additional data structures you may require); note that both the visited set, and the path, are expected to be returned. Please see the type annotations for the specific details about what these functions should return.

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

Finally, while you can visualize DFS on the lattice graphs, **you must** ensure that your algorithms translate to graphs with arbitrary degree. This should be trivial to implement. For the avoidance of doubt, your DFS algorithm **will** be tested on the `KN` graphs.

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.

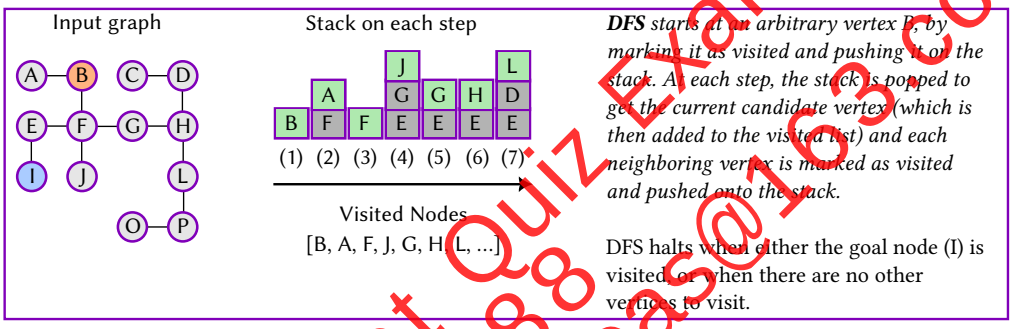


Figure 6 A sketch of depth-first search starting at vertex B and searching for node I. A stack keeps track of the next vertices to visit, and they are visited as they are popped from the stack. A list can be used to track the order in which nodes are visited.

F Task 3: Problem Solving

Now that the mega Gurkey is back in the lab, we will need to conduct additional testing to ensure such an event never happens again (well, at least until COMP3506/7505 2025). Unfortunately, Dr. Amongus has already been fired, so it is our job to help Barry Malloc determine how this all happened in the first place.

Task 3.1: Maybe Maybe Maybe (3 marks)

`MallocLabs` has a huge database of k-mers that have been sequenced throughout their many years of operation. To determine which genomes may have been involved in the genetic modification of the Gurkey, we can simply compare the Gurkey genome to all genomes in the database to find out which ones match, and return those for further analysis. The problem, however, is that the database contains trillions of k-mers.

Our job is to create a fast and compact *filtering* algorithm that, given a list of database k-mers, D, and another list of *query* k-mers Q, returns a list of k-mers that from Q that are *likely to be* in D. We award more marks for having lower false positive rates; the maximum allowed false positive rate is 10%, and then we will measure at 5% and 1%. Note that lower false positive rates might come at higher time and space costs.

Test via: `python3.11 test_problems.py --maybe --seed <number>`

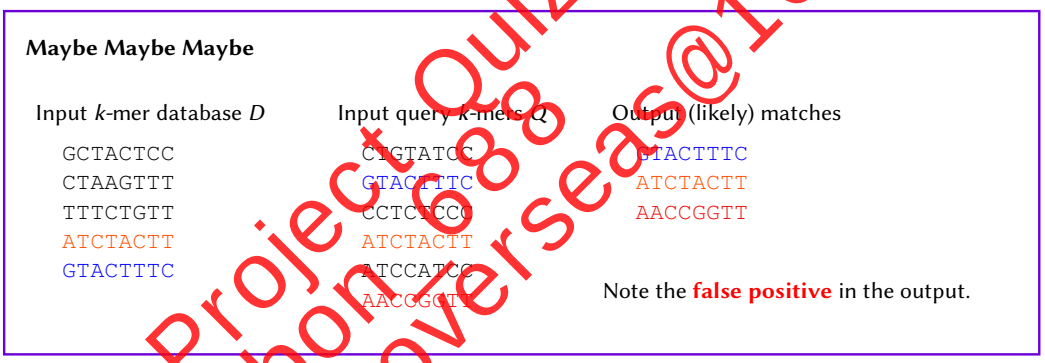


Figure 7 A sketch of Maybe[3] – See the code for the expected output format and structure.

Task 3.2: Dora and the Chin Bicken (3 marks)

`MallocLabs`' spies have recently discovered that their main competitor `CallocLabs`[7] has hired Dr. Amongus, and are planning to release a giant Ibis named Chin Bicken to wreak havoc on `MallocLabs` HQ! Barry and the team need to get prepared. The head honchos at MallocLabs have decided on the following strategy:

- Chin Bicken will, at some point, attack the `MallocLabs` HQ;
- Since Chin Bicken is enormous, it may attack different parts of the building simultaneously;
- At this time, `MallocLabs` will release a robot – `Dora` – which does the following:
  1. It will receive as input an undirected graph G where vertices represent rooms in `MallocLabs` HQ, and edges represent undamaged connections between rooms.
  2. From its starting location, it will explore all reachable rooms of the building to collect genomic data left by the Chin Bicken.
  3. This genomic data comes in the form of special gene symbols, s, represented by a single character; there is one at each vertex of G.
  4. Next, the robot builds a gene symbol frequency table T which maps each gene symbol s to its total frequency in G, denoted f_s.
  5. Once T is computed, the robot builds a minimum redundancy code via Huffman's algorithm, resulting in a codebook C_G mapping each s to a codeword c_s.
  6. Finally, the robot receives a sequence $L = \langle s_0, \ldots s_{n-1} \rangle$ of n symbols, drawn from all symbols appearing in G. This sequence represents the specific body part of Chin Bicken that `MallocLabs` believes is its weak point. The robot will use C_G to encode all $s \in L$ into one long bitvector B. That is, B will hold the concatenation of the encoding of each symbol in L: $c_{s_0} c_{s_1} \cdots c_{s_{n-1}}$.
- Once the robot produces B, Barry can feed it into the `GeneCoder6000` to develop a weapon to fend off the Chin Bicken. Of course, `Dora` will need to be fast. It has to visit all vertices in the graph before Chin Bicken causes any further chaos, after all. You have been tasked to write the logic for `Dora`. Get to it!

```
python3.11 test_problems.py --dora data/KN/one.graph --seed <number>
```

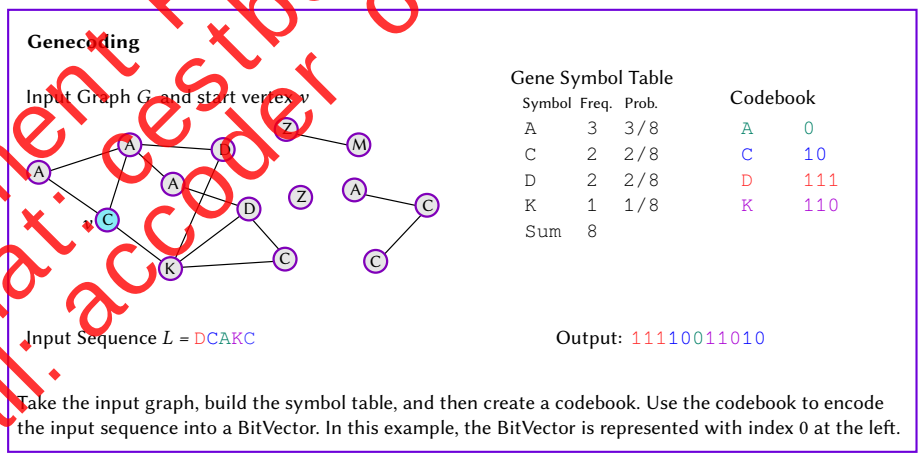


Figure 8 A sketch of *Dora*. See the code for the expected output format and structure.

[7] `CallocLabs`: *from* 0 *to hero.*

Task 3.3: Chain Reaction (3 marks)

To progress further with the reconstruction of Dr. Amongus' cloning programme, Barry now needs to find what is called the optimal *reaction* compound. We are given n candidate compounds, each of which is represented by a unique $\langle x, y \rangle$ coordinate based on their reactivity in two specific dimensions of interest. Each compound also holds a floating point value known as the *spike radius* r.

Compound A is said to cause a *reaction* with compound B if the circle centered on $\langle x_A, y_A \rangle$ with radius r overlaps with the compound[8] at $\langle x_B, y_B \rangle$; however, reactions do not occur naturally — they must be triggered by some other reaction. When a compound reacts, any compound that it is reactive with it will also be triggered (and so on).

You are given one *charged* molecule to set off a *chain reaction*, and you must select the given compound $i \in [0, n-1]$ that will *maximize* the total number of compounds in the chain reaction. If there are ties, return the one with the smallest identifier.

Test via: `python3.11 test_problems.py --chain --seed <number>`

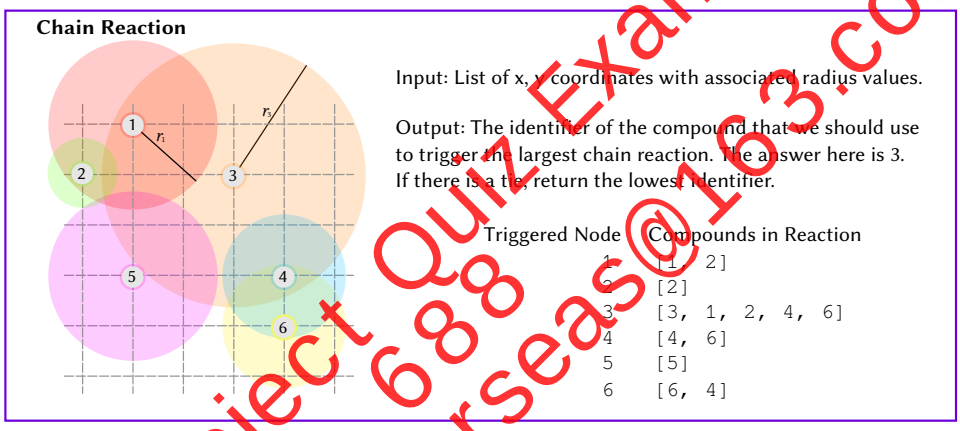


Figure 9 A sketch of the Chain Reaction problem. See the code for the expected output format and structure.

[8] Be careful: A reaction will not occur unless the *center of the cirle* is covered!

Task 3.4: Lost in the Labyrinth (aka $n$otably $m$ore k-$c$ool) (2 marks)

The attack from `CallocLabs` compromised some of the building structure at `MallocLabs` HQ, and the team is concerned that the Gurkey might break free. Barry would like to build a labyrinth to contain the Gurkey, and has offers from various construction companies. However, he is concerned that some of these companies are trying to scam him, so we need to help him to come up with an algorithm to determine whether a labyrinth can even be constructed from each offer.

Each company treats the design of a labyrinth as a graph problem. They provide us with four integers: n, m, k, and c, where n is the number of vertices ($|V|$), m is the number of edges ($|E|$), k is the *diameter* of the graph, and c is the cost to produce it. From these four integers, we must determine if their offer is valid or not. A labyrinth is considered valid if it conforms to the following rules:

- It is a connected graph.
- It has no double edges or self loops.
- The largest *shortest simple path* between any two vertices v_1 and v_2 (the diameter) is at most k. (In other words, if you found the shortest simple path between every pair of vertices, the diameter of the graph is the length of longest one of these.)

Given a list of offers, you must return the cheapest offer that can be constructed. If there are ties, return the one with the smallest identifier.

```
python3.11 test_problems.py --labyrinth --seed <number>
```

G Task 4: Txt Cmprsn (up to 3 bonus marks)

Keen for more punishment? We have just the thing... We'll be running a simple compression challenge. You will be given an arbitrary file, and you need to provide a compression/decompression algorithm. The stubs are provided for you in the `compression.py` file. There will be no marks given for incorrect/lossy algorithms; the output (after decompression) must exactly match the provided input. The marking scheme is as follows.

- One mark: Your algorithm can compress our file to at least half of its original size.
- One mark: Your algorithm is in the top 50% of those submitted.
- One mark: Your algorithm is in the top 10 of those submitted.

We will have a public compression leaderboard available that can be observed. There will be a separate submission area on Gradescope for this part. Please just submit your single file `compress.py` without any zipping. All of your references need to be placed in this file.

Recommendations

Data compression is an extremely large field and there are many rabbitholes for you to venture into. Since we will learn *Huffman coding* and a simple *Lempel-Ziv* compression scheme, this would be a very logical place to focus. What *I* would be doing, for example, is first exploring a simple Huffman coder where I count the frequency of each input byte (after all, bytes are just integers in the range $[0, 255]$) and use this to build a codebook based on the static frequency of each byte. You will of course run into a few challenges that will need to be solved now:

- How do we encode the text back into a byte stream?
- Where and how do we write/represent the codebook?

The former task is easy to solve. We have provided a function in the `BitVector` that will let you dump it back into a byte array. This way, you could do your encoding by simply appending encoded output to the bitvector. The latter is a bit more difficult. Given the encoded bitvector (mapped into a byte array), you still need to represent and store your codebook. After all, if you just wrote the encoded byte array to file, we would have no mechanism of recovering the text inside (oops)! So, you'll need to think about storing the codebook in a compact way, *inside* your compressed output file.

Next, you might observe that instead of taking single bytes, you might be able to improve your code by taking *pairs* (or *triplets*, etc) of bytes instead. This is one avenue to investigate.

Another way to go would be to take the input byte string and try to exploit redundancy within. So, applying dictionary coding to the input byte string (something like *LZ77* or *LZ88*) prior to Huffman coding may also improve your file size.

We leave the rest up to you. We're very happy for you to explore and tinker, this is part of the fun. However, we ask that you do not simply "port" an existing compression mechanism into Python — We trust you to operate in good faith. It's fine to look up LZ77 and implement it based on your understanding, but it's naughty to go looking for a Python LZ77 implementation and essentially copying it into your own program. We hope you see the difference.

Measurement

We'll provide you with a few data files to play with; these can be located on the LMS. For the final testing, we will be using hidden files. However, these files will not differ drastically in style or size from the test files we provide you.

H Assessment

This section briefly describes how your assignment will be assessed.

Mark Allocation

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! You *will* fail hidden tests if you only code to our public tests.

The marks (percentages) 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 are used such as informative variable and function names, commenting, and breaking long lines. This style marking is done automatically by the autograder. We provide the output from the linter; please pay attention. 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 application of data structures/algorithms for solving Tasks 2/3.
- **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/algorithms for solving Task 2/3 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/algorithms for solving Task 2/3, 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 or algorithms towards solving Task 2/3.

Plagiarism and Generative AI

If you want to actually learn something in this course, our recommendation is that you avoid using Generative AI 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, and 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, and if you're not caught here, you'll be caught soon enough by prospective employers.

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 a number that have been developed with AI assistance). If we believe you may have used AI extensively in your solution, you may be called in for an interview to walk through your code. Note also that the final 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 must create a file called `statement.txt`. In that file, you must 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 such tooling, you can make a statement outlining that fact. Failing to submit this file will yield you zero marks.

I Submission

You need to submit your solution to *Gradescope* under the *Assignment 2: Autograder* link in your dashboard. Please use the appropriate link as there is a separate submission for 3506 and 7505 students. Once you submit your solution, a series of tests will be conducted and the results of the public tests 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, but please try to be respectful of both the amount of computation you are offloading to Gradescope, and the fact that we expect you to write your own tests. **Please write your own tests!**

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. Submissions without the `statement.txt` will be given zero marks, and the autograder will notify you of this. No exceptions.

J 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.

K Indicative Timeline

It is impossible to estimate the time required for each task; however, we are providing an indicative guide for you to stay on schedule. You should aim to have each task done by the given date here. Falling too far behind will significantly jeopardize your ability to complete this assignment on time. You cannot complete this assignment in the last week.

- T1.1–T1.3: Week 8–9. Based on material from Week 5 and 6.
- T2.1–T2.3: Mid sem break. Based on material from Week 8 and 9.
- T3.1: Mid sem break–Week 10. *No dependency on Task 2. Start earlier if you like.*
- T3.2: Mid sem break–Week 10. Based on material from Weeks 8–10.
- T3.3: Week 10–12, but you could solve this at any time. Requires a bit of thinking...
- T3.4: Week 9–10. *No dependency on Task 2. Start earlier if you like.*
- Bonus: Any time, but you will need to do your own research/reading.

L Changelog

- V0.1: Initial release (without code skeleton).
- V1.0: Initial release (with code skeleton v1). Some minor changes to the python calls shown.
- V2.0: Clarifications/updates added to the end of the spec. **Please read carefully.**

M Additional Details and Information

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

Banned Code

Our philosophy for banning specific python functionality is to avoid the situation where some complex operations are being hidden by syntactic sugar. For example, consider:

```
my_list = [1, 6, 105, 4, 9]
x = 4
if x in my_list:
    print ("Yay!")
```

This innocent looking code is actually hiding an $\mathcal{O}(n)$ execution, where n is the length of `my_list`. That's because Python allows us to search for an item using nice syntax like `if x in my_list`. That's why we would prefer you to write something like:

```
...
for item in my_list:
  if x == item:
    print ("Yay!")
```

Even though this is longer, it clearly demonstrates that the list is being iterated over.

Some examples of things that are OK or not:

- Generators: These are OK, because they hide annoying/ugly complexity, but do not mask the fact that they are used for iteration.
- Dictionaries: Clearly not OK, because you need to be able to implement a *map* data structure to get this behaviour. Same for Sets. (More in Week 5/6/7).
- Dunder methods: These are OK, as you will need to implement them anyway (and indeed, you will implement `__str__` for example).

In general: If you want to use something, you need to implement it yourself, not use an "out of the box" implementation.

Why Moss?

You do not need to use Moss. We recommend you do because it provides a shared platform for us to help if anything goes wrong. However, you are more than welcome to develop/test your work elsewhere.

Graphs/Properties of Graphs

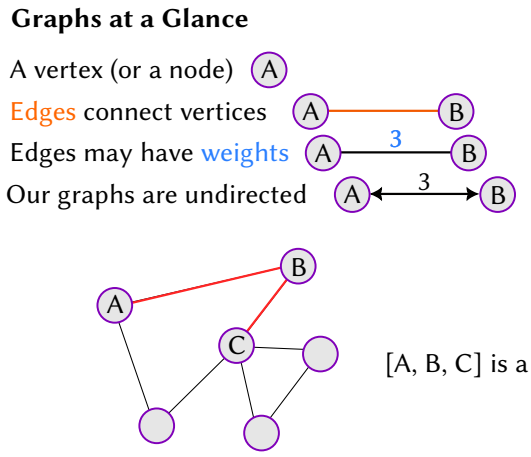
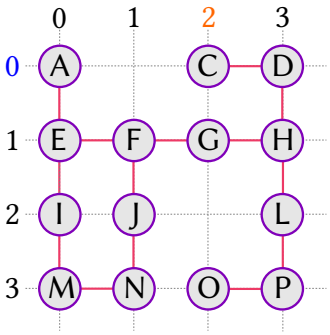


Figure 10 A brief look at some graph terminology. This may help you to conceptualize the problems we are solving.

4N Graph (w. cycles)

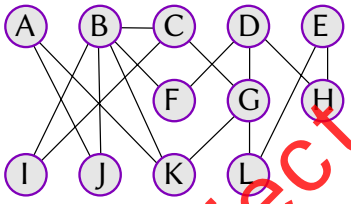


Adjacency List(s)

```
A: E
C: D
D: C H
E: A F I
F: E G J
...
```

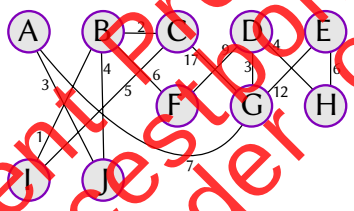
Note: C has coordinate (2, 0)

KN Graph (w. cycles)




```
A: J K
B: C F I J K
C: B G
D: F G H
...
```

POSW Graph (w. cycles)




```
A: (G,7) (J,3)
B: (C,2) (F,6) (I,1) (J,4)
C: (B,2) (G,17) (I,5)
...
```

Figure 11 The three types of graphs used in this assignment. `4N` graphs are based on grids and each node has an (x, y) coordinate associated. `KN` graphs have an abitrary degree for each node. `POSW` graphs are the same as `KN` graphs but have positively weighted edges. All graphs are undirected, and their adjacency lists are shown to the right.

V2.0 Updates

This section captures any changes we've made at v2.0 to both the **spec and the code**.

- **Priority Queue:** You cannot use the `DynamicArray sort()` function to implement your in-place build or heap-sort. That defeats the purpose...
- **Depth First Search:** If you are implementing DFS recursively, please wrap your `get_neighbour()` calls in `reverse()` so you get the same order as a stack-based implementation.
- **Dora:** The sequence L will only be drawn from symbols in the *reachable component of* G (from the origin vertex).
- **Chain Reaction:** We have nerf'd this problem to have a maximum of 10/50/100 nodes for the "it works"/"Exhaustive"/"Welcome to COMP3506" tests. Please ignore the docstrings.
- **Compression:** Please submit using the same structure as the A2 code, including any extra functionality you may need. Essentially, just implement your compression codec inside `compress.py` as usual, but this change allows you to include data structures/utilities from your `structures` or `algorithms` files. Previously, we said just to submit the single `compress.py` file.
- **Compression:** Regarding the marking scheme - we will provide the mark for the "top 50% of those submitted" by awarding the mark to all submissions that score strictly at (or greater than) the mean.