COMP26120: Lab 7 — Graphs

Year 2019-2020, Semester 2, Week 2–5

In this lab, you will explore several graph algorithms, and a heap-based priority queue data structure.

1 Intended Learning Outcomes (ILOs)

Upon successful completion of this lab, you will be able to

1. explain the various shortest path algorithms and their advantages/disadvantages
2. use existing data structures in your algorithms
3. combine existing data structures to get new functionality
4. write C code for the above concepts
5. use a test-driven development approach for algorithms and data structures

2 Assignment

Summary: Implement the interfaces specified in `pq.h`, `sp_algorithms.h`, and the functionality described in `ap.c`.

2.1 General Notes

- Unless otherwise specified, it's fine to over-estimate required array sizes to avoid dynamic re-allocation. For example, the queue required for

BFS can be implemented as array with a size of num_nodes. This is the worst-case space required, but in most cases, significantly less nodes will be required, such that a dynamically growing queue would be more efficient (but also more complicated to implement[1])

- Try to program in a modular way. Do not duplicate functionality, but move it to own functions. Also, do not artificially inline unrelated functionality (for example, priority queue operations should not go inline into Dijkstra's algorithm!). This only makes the algorithms complicated to read, and does not even bring efficiency advantages, as modern compilers are very good at automatically inlining themselves.

- The assignment comes with a test-generator `sp.c`, which will test your data-structures and algorithms on randomly generated input data. Use this to test each module in isolation, before you build modules depending on it! It's much simpler to debug Dijkstra's algorithms if you know that your priority queue is correct!

  Also, feel free to modify the tests, e.g., to generate additional output useful for debugging your problem, or to pin down a particular bug. You can use the `msg` and `assertmsg` macros, that accept printf format strings.

  Have a look at `tests.sh` for the tests that we will run on your submission!

2.2 Min-Heaps

The elements of the heap will be nodes (`node_t` = unsigned integer). The priorities are given by an array `D` from nodes to weights (see `weights.h` for the available operations on weights)..

You have already seen a heap data structure, implemented by a tree encoded into an array. In this assignment, you will extend this idea to support an efficient operation to decrease the priority of a node that is already on the heap, as well as an efficient check whether a node is on the heap. For this, you maintain an additional mapping from nodes to heap-indexes. After decreasing a node's priority, you use its heap-index to start a sift-up operation.

[1]At least in a language like C that lacks a proper data-structure library.

The file `pq.h` specifies the interface, including the required complexities for most operations. The file `pq.c` provides some suggestions how to implement the operations. You are free to follow them, or do your own implementation from scratch.

2.3 Shortest Paths Algorithms

The file `graph.h` specifies basic graph operations. The file `shortest_path.h` provides an interface how to return results for single-source shortest path algorithms. You have to implement:

- BFS, that ignores the weights and returns paths with minimal number of edges.
- Bellman-Ford. First, don't care about negative weight cycles. Once you have got the algorithm correct for this case, you can try to correctly handle negative weight cycles: If, after $|V| - 1$ steps, you can still relax edges, set their target node's distance to $-\infty$. Continue (for at most $|V| - 1$ more steps) until all nodes reachable via negative weight cycles have distance $-\infty$. However, do not update the predecessor map in this second phase!
- Dijkstra. This will need a working priority queue implementation.
- A*. We strongly recommend to implement this after Dijkstra! You can assume a monotone heuristics, such that it will be very similar to Dijkstra.

2.4 The Airport Example

In this example you will work on a graph generated from the openflight database, that encodes connections between airports on the world.

Your task is to implement a (DFS) based algorithm to count the number of airports that can be reached from a given start airport, and to compute shortest routes between two given airports. Use your graph algorithms!

See `airports.h` for the interface, and `ap.c` for a description what to implement. Exactly stick to the output format described there!

3 Marking

Note that this may be refined to introduce extra cases reflecting special cases if required.

3.1 Algorithms

The following rubrics are per algorithm. They address ILO 2–5. ILO 3 is pretty specific to the heap implementation, and TA's should explicitly ask how the heap implementation is related to that from Lab 6, which has no decrease-key.

Algorithms: heap, BFS, BellmanFord, BellmanFord-Negative, Dijkstra, A*, Airports

Note that the duplication of BellmanFord and BellmanFord-Negative is intended. If someone implements negative cycle detection, they'll effectively get the marks twice!

3.1.1 Code Quality

Mark the coding style and consequent use of functions to avoid redundancy.

good (1) Consequent use of interface functions to avoid redundancy, i.e., no manual inlining. Well commented code. Well formatted.

fair (.5) Mostly used interface functions, but some manual inlining of simple functions, like using $pq{-}{>}H[pq{-}{>}I[i-1]]$ instead of $\_DPQ\_hprio(pq, i)$. Mostly commented. Readable, but not uniformly formatted.

poor (0) Almost no use of interface functions. The header files were amended to allow cross-module access to internal data structures, or to add new custom interfaces to the different algorithms that are used instead of the ones specified in the header files. Lots of redundancy.

3.1.2 Understanding

How good could the student explain their implementation?

good (2) Could convincingly explain their implementation, up to the details.

fair (1) Could explain their implementation, but not to every detail.

poor (0) Could not explain most parts of the implementation. Was the implementation adapted from some external source without really understanding it?

3.1.3 Passing Tests

good (1) All tests passed

fair (.5) All but one tests passed

poor (0) More than one test failed

3.1.4 Valgrind

good (1) No issues

fair (.5) Minor issues that probably don't affect functional correctness, like memory leaks

poor (0) Major issues that most likely hint at bugs, like uninitialized read, access to unallocated/freed memory

3.2 Big Picture of Shortest Paths Algorithms

The following rubric addresses the understanding of the greater picture of shortest path algorithms (ILO 1). How good could the student explain the different shortest path algorithms, in particular the advantages and disadvantages when compared to each other?

very good (5) Competent explanation of the advantages, disadvantages, complexity, and preconditions of all algorithms.

good (4) Like the above, but missing some advanced algorithms like A* or negative cycles.

fair (3) No complete overview of the big picture of shortest path algorithms, but could compare some algorithms.

poor (2) Memorized some basic facts about the algorithms, but could not explain.

none (0) Couldn't explain a single algorithm.

4 Sample Questions for TAs

Some sample questions the TA's might ask to assess this lab. Not all questions may make sense in all contexts!

4.1 Understanding

- show me your code for function xyz, and explain how you implemented it
- pointing at unusual implementation: why did you implement it like this?
- what is the idea of a predecessor map. Explain!
- for heap: show me your sift-up function. How is it related to sift-up without decrease-key (Lab 6)
- did you use assertions in your code? Show me a nice example!
- how does Bellman-Ford detect negative cycles?
- why don't you update the predecessor map in the negative-cycle phase? Give an example what might happen if you did!
- explain how Dijkstra's algorithm works
- why can you stop Dijkstra's algorithm after removing the target node from the PQ? What would happen if you stop already when adding the target node to the PQ?

4.2 Big Picture

The questions marked as advanced is what we would expect for a "very good" mark.

- What are the preconditions on the graph when using Bellman-Ford vs. Dijkstra vs. A*?
- What algorithm would you use for the airport routing application? Why?
- What are the (worst-case) complexities to find a route between two nodes, for the different algorithms.
- (advanced) In the negative-cycle round of Bellman-Ford, why don't you update the predecessor map when decreasing a node's distance to $-\infty$?
- (advanced) Compare A* to Dijkstra!