COSC 2123/1285 Algorithms and Analysis
Semester 2, 2015

Project 1 Description

Due date: 5:00pm Friday, 28th August 2015 **Weight:** 15%

1 Objectives

There are two key objectives for this project:

- Use a number of fundamental data structures to implement the *bipartite graph* abstract data type.
- Evaluate and contrast the performance of the data structures with respect to different usage scenarios and different bipartite graphs.

2 Background

Pairwise relationships are prevalent in real life. For example, friendships between people, communication links between computers and pairwise similarity of trajectories. A way to represent a group of relationships is using *graphs*, which consists of a set of vertices and edges. The entities in question are represented as the nodes and the pairwise relations as the edges.

A variant of graphs is *bipartite graphs*. In bipartite graphs $G(V, E)$, we have a set of vertices V and edges $E : V \times V$, representing relationships between pairs of vertices. V consists of two disjoint sets of vertices, V_1 and V_2 (where $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$). Each set of vertex is called a *partite*. The differences between "normal" and bipartite graphs is that there are no edges between vertices in the same partite. As an example, we can represent the products (e.g., movies) that users have bought online as a bipartite graph (see Figure 1). One partite are the users, the other are movies and an (undirected) edge represent a user bought a certain product. This bipartite graph can be analysed to determine user interests.

Data with bipartite graph characteristics are very common. They can represent associations between different entities, e.g., the user-movie example. Another example is representing job assignments of people, so one partite could be people, the other can be jobs, and there is an edge if a person is assigned to a job. In logistics, a company may want to move a number of products between warehouses in different locations, with each warehouse having different capacity. The warehouses that currently have the products form one partite, while the warehouses that are receiving the products form the other partite, and weighted edges represent the cost to transport one unit of product from one warehouse to the other. The problem is to find an assignment that minimises the total cost of moving all the products (this is known as the transportation problem).

In this project, you will implement the bipartite graph abstract data type using a number of basic data structures, then evaluating their characteristics.

3 Tasks

The project is broken up into a number of tasks, to help you progressively complete the project.

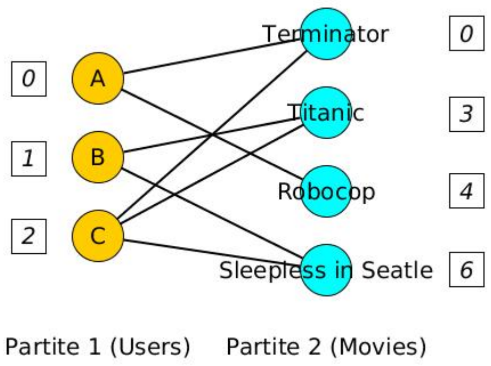

Figure 1: User-movie bipartite graph example. The yellow coloured circles represent the users, and the blue coloured represent movies. The clear squares next to each circle represent the integer based identifier used in an implementation example later, and uniquely identifies each vertex. Note that the vertices in different partites may have the same identifiers but within each partite the identifiers should be unique. Also the identifier labels do not have to be sequential, the only requirement is they form a set. This graph is in its current state after a number of vertex insertions and deletions.

Task A: Implement Bipartite Graph (8 marks)

In this task, you will implement the (directed) bipartite graph abstract data type using a number of graph structures, namely *adjacency list*, *adjacency matrix* and *edge list* (bonus) representations. Your implementation should support the following bipartite graph operations:

- Create a bipartite graph, either an empty graph or with a fixed number of vertices.
- Delete a bipartite graph.
- Insert a vertex.
- Insert an edge.
- Search for a vertex.
- Search for an edge.
- Delete a vertex.
- Delete an edge.

Notes:

- Your implementations should give exactly the same output as the provided implementation.
- The implementation details are up to you, but you must implement your own data structures and algorithms.
- All submissions should be ANSI C (C99 standard) compliant, and should compile with no warnings with "-Wall -ansi -pedantic" flags. If you develop on your own machines, remember to periodically compile and run your code on university machines (e.g., `titan.csit.rmit.edu.au`, `jupiter.csit.rmit.edu.au`, `saturn.csit.rmit.edu.au`), as you don't want to find out last minute that your code doesn't compile on these machines.
- You may alter the provided code, but remember to generate the same output format as the provided `main.c`.

Details

Each of the graph structures can be implemented using a number of data structures. We provide an array-linked list implementation of the adjacency list to help you get started. You are to implement the following graph structures using the specified data structures:

- The adjacency list representation, using a linked list-linked list implementation.
- The adjacency list representation, using a binary search tree (partially implemented)-linked list implementation.
- The adjacency matrix representation, using an array-array implementation.

Operations to the bipartite graph ADT are provided in the input files, which are text files providing lists of operation commands to execute. They are in the following format:

```
<command> <sub-command> <arguments>
```

where command is one of {C, D, I, R, S, P} and sub-command is one of {V, E, X} and arguments take the following form:

- `C X 0 0` – create an empty graph.
- `C X <vertNum1> <vertNum2>` – create a graph, with 'vertNum1' number of vertices in the first partite, and 'vertNum2' number of vertices in the second partite. If vertNum1 or vertNum2 are greater than 0, then the created vertices should have ids starting from 0. For example, if vertNum1 = 3, then the created vertex should have ids '0', '1', and '2'.
- `D X` – destroy the graph.
- `I V <vid> <part>` – insert vertex 'vid' into the partite 'part' of the graph.
- `I E <sid> <tid> <srcPart>` – insert edge '(sid, tid)' into the graph, with the partite 'srcPart' to be the source one.
- `S V <vid> <part>` – search for vertex 'vid' in partite 'part' of graph.
- `S E <sid> <tid> <srcPart>` – search for edge '(sid, tid)' in the graph.
- `R V <vid> <part>` – delete vertex 'vid' from the partite 'part' of the graph.
- `R E <sid> <tid> <srcPart>` – delete edge '(sid, tid)' from the graph.
- `P X` – prints the current bipartite graph.

For the desired output of print, it should be in the following format (see the linked list implementation in `bpGraphAdjList_AL.c` for an example):

```
Vertices:
Part 1:
<list of vertices for partite 1>
Part 2:
<list of vertices for partite 2>
Edges:
Part 1 to 2:
<list of edges for partite 1 to 2>
Part 2 to 1:
<list of edges for partite 1 to 2>
```

As an example, if `P X` was executed for the graph of Figure 1, the following output should appear (remember the graph in the example is an undirected graph, which can be represented as edges going in both directions in a directed graph representation):

```
Vertices:
Part 1:
0 1 2
Part 2:
0 3 4 6
Edges:
Part 1 to 2:
0 0
0 4
1 3
1 6
2 0
2 3
2 6
Part 2 to 1:
0 0
0 2
3 1
3 2
4 0
6 1
6 2
```

Task B: Evaluate your Data Structures (7 marks)

Write a report on your analysis and evaluation of the different implementations. In particular, focus on the differences in time among the implementations when executing the basic operations, and also how much space each data structure uses. Consider in which scenarios each type of implementation (adjacency list, adjacency matrix and edge list if implemented) would be most appropriate, and subsequently which data structures you recommend to implement those in. The report should be 4 pages or less. See the assessment rubric (Appendix A) for the criteria we are seeking in the report.

Bonus Task: Implement and compare the triplet-sparse implementation for edge list (bonus 2 marks)

Do not attempt this unless you have completed all the other tasks. It requires much more effort to get a bonus marks than a conventional mark.

The edge list representation stores the bipartite graph as a list of edges. One well known approach to reducing the space of edge lists is the *triplet* representation (see `http://www.coin-or.org/Ipopt/documentation/node37.html`). In this task, you will implement and additionally include this data structure in your comparisons and report.

4 Report Structure

As a guide, the report could contain the following sections:

- Description of the scenarios that each data structure performs well and provide complexity analysis to back this up (think of the input size/variable as both the number of vertices and edges).
- Empirically evaluate your data structures using either manually constructed or generated data (or real data, if you can find some). Describe your data and experimental setup. Show your results, and explain and discuss them, paying attention to comparing them with your analysis in the previous section and also between the different data models and data structures.
- Summarise your findings as recommendations.

5 Submission

The final submission will consist of three parts:

- Your C source code of your implementations. A Makefile is provided and you can modify it to build the executables. However, the commands `make bpGraphAL_AL`, `make bpGraphAL_LL`, `make bpGraphAL_BL` and `make bpGraphAdjMat` should compile your four implementations as four different executes, with the same names as currently produced by the provided Makefile. Your source code should be in a flat structure, i.e., all the files should be in the same directory, and that directory should be named as `Assign1-<your student number>`. Specifically, if your student number is s12345, when `unzip Assign1-s12345.zip` is executed then all the source code files should be in directory Assign1-s12345.
- A written research report in PDF format. Submit this separate to the zip file of your source code.

Note: submission of the report and code will be done via WebLearn. We will provide details closer to the submission deadline.

6 Assessment

The project will be marked out of 15. Note that there is a hurdle requirement on your combined continuous assessment mark for the course, of which Project 1 will contribute 15 marks. Late submissions will incur a deduction of 3 marks per day, and no submissions will be accepted 5 days beyond the due date.

The assessment in this project will be broken down into two components. The following criteria will be considered when allocating marks.

Implementation (8/15):
You implementation will be assessed based on the number of tests it passes in our automated testing. In addition, commenting and coding style will make up a portion of your marks.

Report (7/15):
The marking sheet in Appendix A outlines the criteria that will be used to mark your evaluation report.

7 Plagiarism Policy

University Policy on Academic Honesty and Plagiarism: You are reminded that all submitted project work in this subject is to be your own individual work. It is not a group project. Multiple automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student(s) concerned. Plagiarism of any form will result in zero marks being given for this assessment, and can result in disciplinary action.

For more details, please see the policy at `http://rmit.info/browse;ID=sg4yfqzod48g1`.

8 Getting Help

There are multiple venues to get help. There are weekly consultation hours (see Blackboard for time and location details). In addition, you are encouraged to discuss any issues you have with your Tutor or Lab Demonstrator. We will also be posting common questions on Blackboard and we encourage you to check and participate in the discussion forum on Blackboard. Although we encourage participation in the forums, please refrain from posting solutions.

A Marking scheme for the Report

Critical Analysis (Maximum = 5 marks)	**Report Clarity and Structure** (Maximum = 2 marks)
5 marks Analysis is thorough and demonstrates understanding and critical analysis. Insightful comparisons are made and illustrated with well designed and typical scenarios. All analysis and comparisons are supported by empirical evidence and theoretical analysis.	2 marks Very clear, well structured and accessible report, an undergraduate student can pick up the report and read with no difficulty.
4 marks Analysis is reasonable and demonstrates reasonably good understanding and critical analysis. Adequate comparisons are made and illustrated with most typical scenarios. Most analysis and comparisons are supported by empirical evidence and theoretical analysis.	1.6 marks Clear and structured for the most part, with some minor deficiencies/loose ends.
3 marks Analysis is adequate and demonstrates some understanding and critical analysis. Some comparisons are made and illustrated with one or two scenarios. A portion of analysis and comparisons are supported by empirical evidence and theoretical analysis.	1.2 marks Generally clear and well structured, but there are notable gaps and/or unclear sections.
2 marks Analysis is below what is expected of a 2nd year undergraduate or postgraduate student, and demonstrates minimal understanding and critical analysis. Few comparisons are made and illustrated with one or two scenarios. Little analysis and comparisons are supported by empirical evidence and theoretical analysis.	0.8 mark The report is unclear on the whole and the reader has to work hard to understand.
1 mark Analysis is almost non-existent and demonstrates no understanding and critical analysis. No comparisons are made. No analysis and comparisons are supported by empirical evidence and theoretical analysis.	0.4 marks The report completely lacks structure and is barely understandable.