Assignment 3
Brunch with the Minotaur

Prof. Darrell Long
CSE 13S – Winter 2020

Due: February 2nd at 11:59 pm

1 Introduction

> *"How beautiful the world is, and how ugly labyrinths are," I said, relieved. "How beautiful the world would be if there were a procedure for moving through labyrinths," my master replied.*
>
> —Umberto Eco, *The Name of the Rose*

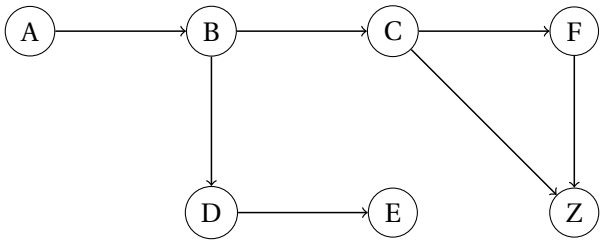
You and your comrades are hanging out at Coachella, and come across a man named Aegeus. Seeing that he is a charming and likeable man, you and your comrades decide that it is probably fine to get to know him a bit more. As the night progresses, you learn more and more interesting tidbits of information regarding his background. You learn that he is from Greece, and is incredibly tight with two blokes, Minos and Daedalus. Aegeus takes a liking to your band of friends and invites you to an island off the coast of Crete. Very suspicious. You and your comrades politely decline and turn in for the night, only to awake and find yourselves no longer at Coachella, but instead trapped within a ring of entrances to a labyrinth; labyrinthine paths as far as the eye can see. Suddenly it hits you. *Minos* is the name of the first king of Crete, who asked King *Aegeus* to send fourteen young children annually into *Daedalus*' labyrinth, the abode of the fearsome Minotaur. Your skills as an aspiring computer scientist are now put to work. You set out to code up a program that, when fed the blueprints of a labyrinth, will print out all possible paths through the labyrinth, the number of paths, and the length of the shortest path. In order to code this up however, you realize you must decide how to represent a labyrinth, how to traverse a labyrinth, and finally, how to keep track of the paths taken through a labyrinth.

2 Representing the Labyrinth

> *"Or like that other Templar, Freud," Belbo said, "who instead of probing the labyrinths of the physical underground, probed those of the psychic underground, as if everything about them hadn't already been said, and better, by the alchemists."*
>
> —Umberto Eco, *Foucault's Pendulum*

You remember that there is a data structure called a *graph*. A graph is comprised of a set of points (called *vertices*) and a set of lines (called *edges*) that connect pairs of distinct vertices. Two vertices that are

connected with an edge are *adjacent* to one another. Each junction in a labyrinth will be a vertex in a graph, and the hallways connecting junctions are edges. There are two types of edges: *undirected* and *directed*. A directed edge is much like a one-way street, in that you can only go from the starting vertex and end at the ending vertex. An undirected edge can be thought of as two one-way streets running in parallel and in opposite directions, or as just a two-way street.



For simplicity, you decide to limit your labyrinth to have at most 26 junctions, meaning the corresponding graph will have at most 26 vertices. Each vertex will have its own index i such that $0 \leq i \leq 25$. The vertices will have a one-to-one correspondence with the uppercase characters of the English alphabet. This means that index 0 corresponds to $\mathbb{A}$, index 1 corresponds to $\mathbb{B}$, and so on and so forth. By using these indices, you can then represent edges between vertices in the graph using an *adjacency matrix*. In an adjacency matrix, a directed edge starting from vertex i and ending at vertex j exists if matrix[i][j] = 1. From the graph above, we can see the following edges: $\mathbb{A}$ to $\mathbb{B}$, $\mathbb{B}$ to $\mathbb{C}$, $\mathbb{B}$ to $\mathbb{D}$, $\mathbb{C}$ to $\mathbb{F}$, $\mathbb{C}$ to $\mathbb{Z}$, to $\mathbb{D}$ to $\mathbb{E}$. These edges are represented using the following adjacency matrix:

	A	B	C	D	E	F	⋯	Z
A	0	1	0	0	0	0	0	0
B	0	0	1	1	0	0	0	0
C	0	0	0	0	0	1	0	1
D	0	0	0	0	1	0	0	0
E	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0
⋮	0	0	0	0	0	0	0	0
Z	0	0	0	0	0	0	0	0

The input to your program will be a file containing pairs of characters separated by a newline. The example below shows the contents of the file that would be supplied to your program to generate the adjacency matrix above.

```
AB
BC
BD
CF
CZ
DE
```

Example input of graph edges.

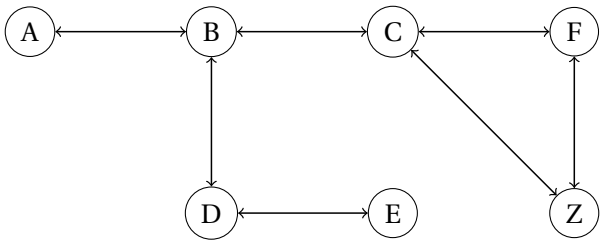
We have chosen to represent an undirected graph as a pair of edges $x \to y$ and $y \to x$. That means that every undirected edge really has two entries in the adjacency matrix, rather like a two way road having a

lane going in each direction. If the edges were really undirected, where $x \leftrightarrow y$, then we only would have matrix$[x][y] = 1$ and that would yield an upper-diagonal matrix. This would be like a road with no lanes (but without any increased probability of head-on collisions). It's merely a question of how we represent the graph, and we have chosen directed edges since it allows us to have both kinds of graph using the same matrix.

Note that the above examples showcase a *directed* graph. Let's see what the adjacency matrix and corresponding graph look like when we make each connection undirected. We know that if matrix$[i][j] = 1$, then it means that there is a connection starting from i and ending at j. Thus, to make things undirected, we simply have to set matrix$[j][i] = 1$. Using the same input as before, the resulting adjacency matrix would be:

	A	B	C	D	E	F	⋯	Z
A	0	1	0	0	0	0	0	0
B	1	0	1	1	0	0	0	0
C	0	1	0	0	0	1	0	1
D	0	1	0	0	1	0	0	0
E	0	0	0	1	0	0	0	0
F	0	0	1	0	0	0	0	0
⋮	0	0	0	0	0	0	0	0
Z	0	0	1	0	0	0	0	0

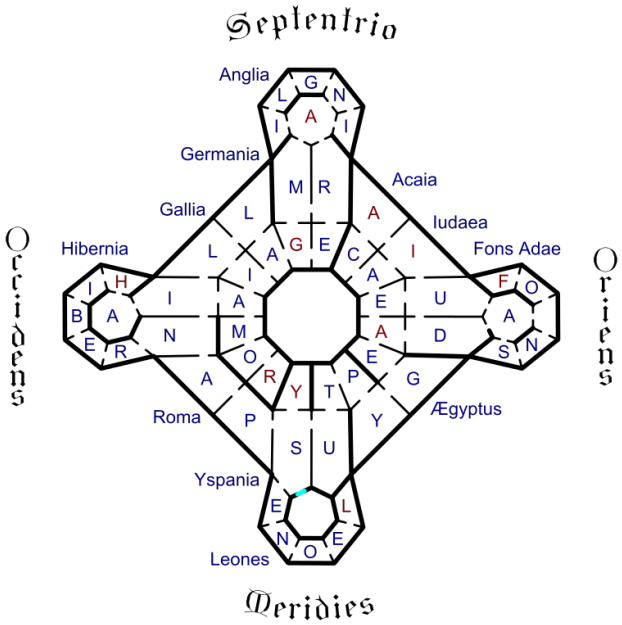
We can see that the adjacency matrix for the undirected graph is the same as the adjacency matrix for the directed graph, but *reflected across the diagonal*, yielding a symmetric square matrix. The undirected graph would then be:



3 Traversing the Labyrinth

> *"To find the way out of a labyrinth," William recited, "there is only one means. At every new junction, never seen before, the path we have taken will be marked with three signs. If, because of previous signs on some of the paths of the junction, you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures have already been marked, then you must retrace your steps . . . "*
>
> —Umberto Eco, *The Name of the Rose*

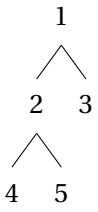
You decide to use a simple recursive backtracking algorithm to traverse through the labyrinth. Recursive backtracking is an algorithm that is used to build a solution incrementally, one step at a time, by



Secretum finis Africae manus supra idolum age primum et septimum de quatuor.

removing those solutions which fail to satisfy the constraints of the problem at any given condition or time. The idea behind backtracking to find paths through a labyrinth is quite straightforward: starting from the entrance of a labyrinth, try going down hallway connected to each junction you come across. If a dead-end is reached after going down a hallway, simply backtrack to the previous junction you were in and try a different hallway. In the event that you find the exit of the labyrinth, record the current path taken and backtrack to try to find other paths. *Important:* a path through a labyrinth in our example means that, starting from node $\mathbb{A}$, it is possible to reach node $\mathbb{Z}$.

An example of a backtracking traversal algorithm is *Depth First Search (DFS)*. For the most recently discovered vertex v, which still has unexplored edges, DFS explores edges out of v (not including v). Once the algorithm has explored all edges of v, it backtracks to explore other edges leaving the vertex from which v was originally discovered. In case of undiscovered vertices, DFS selects a new source vertex and repeats the search from the source. This process continues until every vertex has been explored.



An example of DFS is post order traversal, where the algorithm first looks at a the left node, followed by the right node and then finally the root node. In the case of the tree provided above, the post order DFS traversal would search the nodes in the order $(4,5,2,3,1)$.

Pre-lab Part 1

1. Write pseudocode for creating the adjacency matrix from a file with contents similar to the example input file.
2. Given the graph above, draw and list the recursive traversal of the graph. This can be done by naming each edge by it's adjacent nodes where edge *AB* is the edge from *A* to *B* and listing the order in which the edges are taken.
3. Looking at the post order traversal for the tree above, find the worse case complexity of the traversal algorithm.

4 Tracking the Paths

> *Zu Mittag theilte Grethel ihr Brot mit Hänsel, weil der seins all auf den Weg gestreut hatte, aber der Mittag verging und der Abend verging, und niemand kam zu den armen Kindern. Hänsel tröstete die Grethel und sagte: "wart, wenn der Mond aufgeht, dann seh ich die Bröcklein Brot, die ich ausgestreut habe, die zeigen uns den Weg nach Haus." Der Mond ging auf, wie aber Hänsel nach den Bröcklein sah, da waren sie weg, die viel tausend Vöglein in dem Wald, die hatten sie gefunden und aufgepickt.*
>
> —Brüder Grimm

You're keenly aware that your program must keep track of the current path while traversing a labyrinth, but how best to do so? The brilliant idea of using a *stack* comes to you.

A stack is a linear data structure that follows either a LIFO (last-in, first-out) or FILO (first-in, last-out) order. The two main stack operations pushing and popping. A good example of a stack would be a stack of plates in a cupboard. Pushing a plate into this stack would mean placing the plate on the very top. Popping a plate off this stack would mean removing the plate on the very top.

The stack will be utilized in the following fashion: whenever a new junction is reached, the new junction is pushed into the stack. Whenever a dead-end is reached, we pop the current junction from the stack and backtrack. Whenever the exit of a labyrinth has been reached, the stack is printed to display a valid path through the labyrinth. Again, a path through a labyrinth means that it is possible to go from node $\mathbb{A}$ to node $\mathbb{Z}$.

Below is the header file for the stack abstract data type (ADT) that you are required to implement for this assignment. Note that there is only a `stack_empty()` function, and not a `stack_full()` function. This is because you will be implementing a dynamic stack; the capacity is doubled whenever the current stack capacity has been reached, meaning the stack can never be full. You can find a example of a dynamic stack in the lecture slides, but be sure to cite it if you decide to use it.

```
1 #ifndef __STACK_H__
2 #define __STACK_H__
3
4 #include <inttypes.h>
5 #include <stdbool.h>
6
7 #define MINIMUM 26
```

```
8
9  typedef struct Stack {
10   uint32_t *items;      // A stack holds uint32_t.
11   uint32_t top;         // Keeps track of the top index of a stack.
12   uint32_t capacity;    // Keeps track of a stack's capacity.
13 } Stack;
14
15 Stack *stack_create(void);
16
17 //
18 // Constructor for a stack.
19 // A stack is initialized to hold MINIMUM number of uint32_t.
20 //
21 // returns: A pointer to the constructed stack.
22 //
23
24 void stack_delete(Stack *s);
25
26 //
27 // Destructor for a stack.
28 //
29 // returns: Void.
30 //
31
32 bool stack_empty(Stack *s);
33
34 //
35 // Checks if a stack is empty.
36 //
37 // returns: True is the stack is empty, false otherwise.
38 //
39
40 uint32_t stack_size(Stack *s);
41
42 //
43 // returns: The number of items currently in the stack.
44 //
45
46 bool stack_push(Stack *s, uint32_t item);
47
48 //
49 // Pushes an item (a uint32_t) into the stack.
50 //
51 // returns: True if the item was pushed, false otherwise.
52 //
```

```
53
54 bool stack_pop(Stack *s, uint32_t *item);
55
56 //
57 // Pops an item (a uint32_t) off the stack.
58 // The value of the popped item is stored in the supplied pointer.
59 // Ex.: *item = popped_value
60 //
61 // returns: True if the item was popped, false otherwise.
62 //
63
64 void stack_print(Stack *s);
65
66 //
67 // Function that prints out the contents of a stack.
68 // Refer to program output for print format.
69 //
70 // returns: Void.
71 //
72
73
74 #endif
```

stack.h

Pre-lab Part 2

1. Provide pseudocode for each stack ADT function shown above.

5 Specifics

There is no idea so stupid that you can't find a professor who will believe it.

—H. L. Mencken

Your executable program must be named `pathfinder` and must support the following `getopt()` arguments:

1. `-i <input>` : specifies `<input>` as the file containing the edges of a graph (default input should be `stdin`).
2. `-u` : specifies that the graph be undirected (this is the default).
3. `-d` : specifies that the graph be directed.
4. `-m` : specifies that the adjacency matrix be printed.

You must be able to support any combination of these flags, except for the "`-u`" and "`-d`" flags, which if both are specified should result in an error. Again, *undirected edges* should be the default of your program.

There will be some example input files made available to you, as well as corresponding output files, for you to base your program output off of. Other specifications:

- The file that contains the `main()` function for your program must be named `pathfinder.c`. Do not name it anything else. You will lose points.
- The executable generated by your `Makefile` must be named `pathfinder`. It cannot be named anything else. You will lose a *lot* of points.
- Typing `make` must build your program. This means no errors and no warnings when using the compiler flags supplied in the coding standards and in previous assignments.
- All source code must be formatted `clang-format`.
- Your program *must* run on the time share. If it does not, your program will receive a 0. To avoid this, test your program on the time share.
- Your program must pass `infer` with no errors. If there are any, fix them; for those that you cannot fix, make sure to document them in your `README`.

6 Hints

> *It is no use trying to sum people up. One must follow hints, not exactly what is said, nor yet entirely what is done.*
>
> —Virginia Woolf

1. Start first by implementing your stack ADT. Code up a test harness for it and test each stack operation.
2. Familiarize yourself with opening files in **C**. You will most likely want to use `fopen()` to open files for reading, and `fscanf()` to read in the pairs from the supplied input file. Read the manpage on both for more details.
3. You will likely want to look into the functions `isalpha()` and `toupper()` when reading in the pairs and converting them into integers between 0 — 25 inclusive.
4. Write your recursive function to traverse through a labyrinth. To help you get started, here's some pseudocode:

```
Initialize path stack.

void dfs(uint32_t curr_node):
  if curr_node is the exit:
    print current path
```

```
    return

  for all possible next_node:
    path.push(curr_node)
    dfs(next_node)
    path.pop()

  return
```

5. Think carefully about traversing the labyrinth. Should you go back into a junction you've already visited? How should you keep track of which junctions you've visited? An array, perhaps?

7 Deliverables

> *The design process is about designing and prototyping and making. When you separate those, I think the final result suffers.*
>
> —Jonathan Ive

You will need to turn in:

1. `pathfinder.c`: The source file containing your program's `main()` function.
2. `stack.c` and `stack.h`: The files containing your implemented stack ADT.
3. `Makefile`: Allows the grader to type `make` to build your program.
   - The compiler flags: `-Wall -Wextra -Werror -Wpedantic` must be included.
   - The compiler used must be `clang`.
   - `make clean` must remove all compiler generated files.
   - `make infer` must build and run `infer` on your program, passing without errors. Again, any errors that you cannot fix should be documented in your `README`.
   - `make` should build your program, as should `make all`.
   - Your program executable must be named `pathfinder`.
4. `README.md`: This must be in markdown. This must describe how to use your program and `Makefile`.
5. `DESIGN.pdf`: This *must* be a PDF. The design document should contain answers to the pre-lab questions in the beginning and describe your design for your program with enough detail that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

   You *must* push the `DESIGN.pdf` before you push *any* code.
6. `WRITEUP.pdf`: This PDF will contain an explanation of your thoughts on the following:

(a) Contemplate the number of possible paths that you can take to achieve the expected outcome.

(b) Is this an inherently hard computational problem?

All of these files must be in the directory `asgn3`.

8 Submission

Better three hours too soon than a minute too late.

—William Shakespeare

To submit your assignment, follow the steps on how to submit your assignment through `git`. Remember: *add, commit,* and *push*!

1. Add it!

```
git add file_name.c
```

   As mentioned before, you will need to first add the files to your repository using the `git add <filenames>` command. You will be submitting these files into the `asgn` directory.

2. Commit it!

```
git commit -m "Your commit message here"
```

   Changes to these files will be committed to the repository with `git commit`. The command should also include a commit message describing what changes are included in the commit. For your final and last commit for submission, your commit message should be "final submission".

3. Push it!

```
git push
```

   The committed changes are then synch'd up with the remote server using the `git push` command. You must be sure to push your changes to the remote server or else they will not be received by the graders.

Your assignment is turned in *only* after you have pushed. If you forget to push, you have not turned in your assignment and you will get a *zero.* "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often.*

9 Supplemental Readings

The more you read, the more things you will know. The more that you learn, the more places you'll go.

—Dr. Seuss

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 4 §4.5–4.11
  - Chapter 5 §5.1–5.5, 5.7
  - Chapter 6 §6.1–6.2, 6.7
- *The Art of Computer Programming* by Donald E.Knuth
  - Chapter 2 §2.3.4.1
- *Wikipedia*
  - Graph Theory
  - DFS

Partying with Ariadne—What could possibly go wrong?