Design Assignment 3

Pre-lab pt1:

1)

Variables:

Alphabet_array = {A,B,C,D...Z} //Array made to correlate alphabets with indices for matrix.

Adjacency_matrix[][]//Stores 1 for any specified combination of two letters where a path exists. Stores 0 if no path exists.(Letters represented by array indices)

```
function indexLetter(letter) {
  If letter in Alphabet_array:
     Return index of letter in array
}
//Read from file
function TakeFileInput(File) {
Variables:  char1, char2
  if file exists:
    while (end of file isn't reached, read two characters every line and store to char1 and
char2 respectively) {
       if char1 and char2 are alphabets:
          char1, char2 = uppercase()
          If graph is undirected:
            Adjacency_matrix[indexLetter(char1)][indexLetter(char2)] = 1
             Adjacency_matrix[indexLetter(char2)][indexLetter(char1)] = 1
         else if graph is directed:
             Adjacency_matrix[indexLetter(char1)][indexLetter(char2)] = 1
  }

     If there is a 'A' and 'Z 'in the characters read  {
          Path from node A to Z exists = true
     }
}
```

```
//Read from user input stream
function TakeConsoleInput(File) {
Variables:  char1, char2
 if file exists:
   while (user input isn't terminated by Ctrl+D, read two characters entered every line
and store to char1 and char2 respectively) {
      if char1 and char2 are alphabets:
         char1, char2 = uppercase()
         If graph is undirected:
           Adjacency_matrix[indexLetter(char1)][indexLetter(char2)] = 1
            Adjacency_matrix[indexLetter(char2)][indexLetter(char1)] = 1
        else if graph is directed:
            Adjacency_matrix[indexLetter(char1)][indexLetter(char2)] = 1
 }

    If there is a 'A' and 'Z 'in the characters read  {
        Path from node A to Z exists = true
    }
}
```

2) For the given graph:

Travel AB
Travel BC at junction B (because C comes before D in alphabetical order)
Travel CF at junction C(because F comes before Z in alphabetical order)
Travel CZ
if Z reached, stack is popped to last junction with unvisted edges
    (Path: A - B - C - F - Z)
    Backtrack from Z to F (FZ)
    Backtrack from F to C (FC) since undiscovered edges at junction C
Unvisited paths at junction C? -> Yes, 1 remaining
Travel undiscovered edge CZ
If Z reached, stack is popped to last junction with unvisted edges(Path printed)
    (Path: A - B - C - Z)
    Backtrack from Z to C (ZC): no new edge at Z
    Backtrack from C to B (CB): since undiscovered edges at junction B
Unvisited paths at junction B? -> Yes, 1 remaining
Travel undiscovered edge BD
Travel next unvisited edge DE
Unvisited paths at Junction E -> No, Dead end(E != Z). Back track

Backtrack from E to D (ED): no new edge at E
Backtrack from D to B (DB): no new edge at D
Backtrack from B to A (BA): no new edge at B
Undiscovered edges at A?
No-> All paths found.

3)

For the given tree, the worst case scenario, given DFS, would be reaching node 3 since it would first travel from 1->2->4. When it doesn't find any edges to 3 from 4, it will go back to 2, then go to 5. When it doesn't find any edges out of 5, it will pop back to 2. Then, since there will be no new edges out at node 2, it will pop back to 1. It will then explore the one unexplored edge to 3. As as a result, it would have to travel through 7 nodes before getting to node 3.

Pre-lab pt. 2;

***Note: Used code from Darrell Long's lecture slides for pop push and create function.**

```
Stack *stack_create(void) {
  Create new stack objected and allocate memory for it the hold its members
  Initialize top of stack to 0
  Set capacity to minimum of 26
  Allocate memory to stack array to hold at at least 26 integers to begin with.
}

bool stack_empty(Stack *s) {
  if the top of s is at index 0:
    return true to indicate empty stack.
}

Void stack_delete(Stack *s) {
  Free memory allocated to stack array
  Set array to null
  Free memory allocated to stack object
  Ser object pointer to null
}
```

```
uint32_t  stack_size(Stack *s) {
  Return top(index of next possible element on next)
}

bool stack_push(Stack *s, uint32_t item) {
  If stack size is at capacity:
     Multiply capacity by 2
  Move top index of stack by 1
  Add item to top of stack
  Return true
}

Bool stack_pop(Stack *s, uint32_t *item) {
  If stack s is empty:
    Return false
  Else:
    Decrease top index of stack by 1
    Make ‘item’ point to value in array at top index
    Return true;
}

Void stack_print(Stack *s) [
  For loop iterating over stack array s(from 0 to top index):
    Print array values at each index
}
```

Search Algorithm Implementation:

```
void searchpath(uint32_t current, Stack *path) {
  Variable: *item_popped, visited_junction_array = [0,0...]
   Push item to stack path
   Visited_junction_array = true(visited)
   if (alphabet[current] == Z) {
      If stack size is less than max possible stack_length:
        stack_length = current stack size
      Increment Number of paths found by 1
      Print path taken
   }
```

```
  Else {
    For loop iterating over of number of alphabets {
        If adjacency_matrix[current][iterator] has path and visited_array_junction[iterator] =
false and current != iterators:
        Run dfs(iterator, stack path) on recursion
}
Pop from stack(stack path, item_popped)
Set last visited junction with unexplored edges to false(unvisited)
}
```

Printing function:

```
Function path_print() {
  For loop iterating over stack:
    Print "Found path: " + Alphabet_array[value in stack corresponding to letter + "->" ]
}
```

Main program flow: In the main function, the command line flags and arguments are parsed with getopt() and boolean variables are used to store the requirements for the desired type of output as indicated by the flags and arguments. This boolean variables are then used later in the main function to determine factors like if the adjacency matrix is directed or undirected, if the matrix should be printed, and if the path inputted through command line or text file.