**CSCE 625**
**Programing Assignment #1**
**due: Tues, Sep 22 (by start of class)**

Objective

The goal of this assignment is to implement and compare the performance of **Breadth-first search (BFS), Depth-First Search (DFS)** and **Greedy Best-First Search (GBFS)** on a simple navigation problem, i.e. path-finding.

The focus of this assignment is on implementing the general iterative search algorithm described in the textbook (in particular, Figure 3.7). That is, the search involves maintaining a **frontier**, in which nodes are popped, goal-tested, and then the successor nodes are pushed in, and the process iterates until a goal node is found.

Although this navigation problem is relatively simple, there are a number of challenges in getting the code running. One of the practical difficulties is dealing with **visited states**. In this domain, there are many alternative paths to get to a given node. If you don't check for this, the search space will explode.

You can use any programming language you like. You may use a library implementation of data structures like a priority queue, such as provided in the Standard Template Library (STL) (you do not have to implement this from scratch).

In this project, you should see that BFS, DFS, and GBFS have **different performance characteristics**, in terms of time- and space-complexity. Using the straight-line distance heuristic with GBFS should improve the efficiency of the search. You should **evaluate and compare** these metrics for the 3 algorithms on test problems and see if the performance differences meet your expectations.

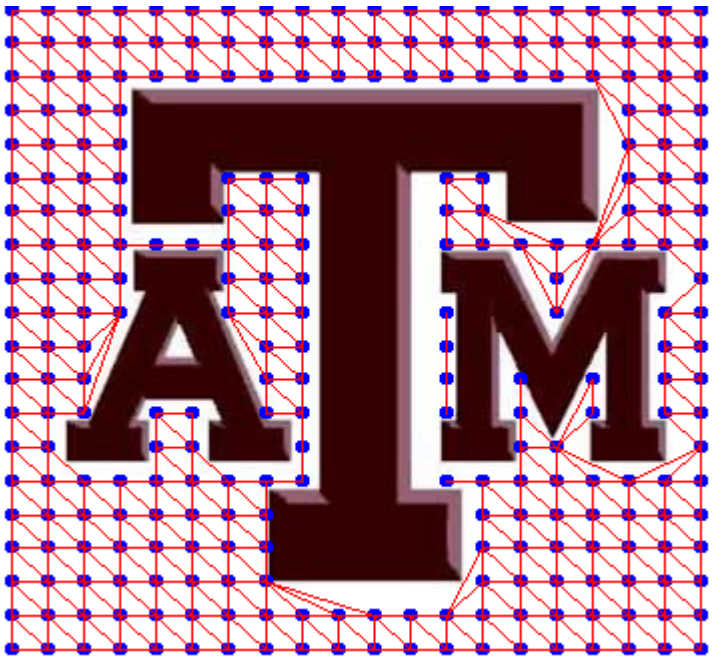
Navigation Task

Navigation is a common problem for intelligent agents (especially robots). The simple 2D version of this problem relates to everything from a robot wandering the hallways of a building, to a taxi navigating the streets of city to get to a destination. For more complicated examples, think of navigating the waterway of a complex shoreline with many islands and fjords (like in Norway or Alaska).


(from a nice survey in IEEE Robotics and Automation Magazine by Bhattacharha and Gavrilova, 2008. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4539723)

We can model navigation as a discrete search by designating way points covering the free space, e.g. laid out on a grid. The way points (or vertices) are connected by edges which may be safely travelled without causing a collision with an obstacle. Thus we can think of the way points abstractly as an undirected graph. The initial state and goal states are just given as vertices in the graph, and the objective is to find a path that connects them. For example, in the ATM graph below, we could try to go from upper-left corner to lower right (1,1) to (20,20). Or we could try to go from inside the T on one side to the other, (7,6) to (13,6).

```
coordinate system:
          X----------------------->
          1     2     3     4 ...  20
Y    1   (1,1)(1,2)(1,3)
↓    2   (2,1)(2,2)
↓    3   (3,1)
↓    :
↓    20
```



Implementation

Reading in the graph: The graph is represented as a text file (provided on the website) that lists vertices and edges. You will have to write some simple initialization code to read this file into your program and set up some simple data structures for vertices and edges. The first line gives the number of vertices V. It is followed by V lines where the format is**: ID X Y**. ID is the vertex identifier, and X and Y are coordinates of the vertex (in the coordinate system mentioned above, integers). Then comes a line indicating the number of edges E, followed by E lines of the format: **ID V1 V2**, where ID is the edge identifier and V1 and V2 and identifiers of vertices connected by the edge. Note that they are only listed in one direction, i.e. for V1<V2. But the edge from V2 to V1 is implicit.

```
vertices: 275
0 1 1
1 1 2
2 1 3
...
edges: 641
0 17 18
1 37 38
2 56 57
...
```

An important data structure to define is a **Node** class. A Node is like a Vertex, but it is more than just a simple pair of coordinates - it is represents a state in the search space, including the path to it. As such, a Node has a pointer to the parent from which it was generated (except the start state, which has no parent). A node also records the depth in the search tree (i.e. path length from the start). Furthermore, nodes may have additional information, such as heuristic value (estimated distance to goal, for GBFS).

In addition to storing this information about the search state, several important methods need to be defined for Nodes. A critical method is **successor()**, which, when called, returns a list of successors (children Nodes) to be used in the queue-based search. In the context of this assignment, the successors will be neighbors in the navigation graph. However, the concept of a successor() function is very general, and in other problems, successor() could generate alternative states of an arbitrary nature, such as different configurations of blocks, or chess boards, or transportation schedules...

Finally, you will want to implement a **traceback()** method to generate and/or print out the solution path, once a goal node has been found. This can be achieved by calling the parent of the node, the parent's parent, and so on back up to the root (start state), recursively. This is why it is necessary to keep track of the parent state from which a node was generated.

Thus I suggest you start with a class definition that looks something like this:

```
class Node
{
public:
  int v; // index into vertices (vector of Points with coords)
  Node* parent;
  int depth;
  float heur;
  Node(int I) {...} // constructor, takes vertex id
  Node(int I,Node* p) {...} // this constructor takes parent too
  vector<Node*> successors() {...} // returns neighbors
  vector<Node*> traceback() {...} // returns path to root
};
```

Search Algorithms

The main focus of this project is implement a general **Search()** function. The input arguments should be an initial state and goal state (or goal_test() function, more generally). For this domain, these states could be indexes or coordinates of vertices in the waypoint graph.

The Search() function should be implemented as an iterative procedure that processes nodes in a generalized queue-based data structure called a frontier (follow the *GraphSearch* algorithm in Fig 3.7 in the textbook). The frontier is initialized with the start node. On each iteration, the "front" node is popped off the frontier, goal-tested, and expanded to generate successors (using the class method described above), which are then pushed onto the queue. If the frontier is implemented as a standard LIFO queue, the Search() function will produce a breadth-first strategy or behavior. If the frontier is represented as a FIFO stack, it should simulate DFS. For GBFS, the frontier should be a **priority queue** sorted on the heuristic score of each node. The natural heuristic to use in this domain is $h_{SLD}$, the Straight-Line Distance (i.e. Euclidean distance). Here is some pseudo-code suggesting what your Search() function might look like in C++.

```
Node* Search(Node* initial_state,Vertex* goal)
{
  MyQueue* frontier;
  frontier.push(initial_state);
  while frontier is not empty
  {
    // pop front node
    // if it satisfies the goal, return it
    // call node.successors() to generate children
    // push them on the queue
  }
}
```

You can change "MyQueue" to be a Queue, Stack, or PriorityQueue depending on which type of search you want. (You could pass in an argument to indicate which data structure/search type you want. Or you could use a compiler flag to change the data type of the frontier. Or you could make 3 separate but nearly identical functions, BFS_Search(), DFS_Search, GBFS_Search(), and select the one you want to use with a command-line argument).

Checking for Visited States

One of the significant challenges in this assignment is dealing with visited states. As part of applying search algorithms to real-world problems, you have to detect when multiple paths lead to the same state. This frequently happens in many domains, including navigation (where a high degree of connectivity and bidirectional movement along edges creates the possibility for many alternative paths between vertices). If you allow multiple paths to the same state to be generated and placed into the frontier, the queue will quickly expand, and the search will explode exponentially. Keeping track of visited states is a little tricky. You might set up an array parallel to the list of vertices with an entry to keep track of whether a path has previously been found to it. Without this, you risk of running out of memory (because of exponential queue expansion). Also, if you come across a state that has been previously visited, don't just discard it - you have to keep track of the *shortest* path to the node. This is important for maintaining *optimality* of algorithms like GBFS.

Testing

Initially, you will want to test your program by running it on very simple start/goal combinations on the Block ATM navigation problem above. For example a simple test is getting from (1,1) to (4,4), which involves a 3-step plan.

```
> BFS_nav ATM.graph 1 1 4 4
```

Other test cases could be:

```
> BFS_nav ATM.graph 1 20 20 20 // across the bottom
> BFS_nav ATM.graph 1 1 20 20 // corner to corner
> BFS_nav ATM.graph 1 20 20 1 // crossing other diagonal
> BFS_nav ATM.graph 13 6 7 6 // from under T on one side the the other
```

It is sufficient to print out the solution path once it is found - you do not need to display it graphically. For example,

```
> BFS_nav ATM.graph 1 1 4 4
solution path:
 vertex 0 (1,1)
 vertex 21 (2,2)
 vertex 42 (3,3)
 vertex 62 (4,4)
```

You should also print out some summary statistics at the end each run that report things like how many iterations were required, the maximum length of the queue, total number of vertices visited, and solution path length.

```
search algorithm = BFS
total iterations = 16
max frontier size= 9
vertices visited = 23/275
path length      = 3
```

For debugging purposes, you will want to print out diagnostic information during the search. For example, in each pass of the main loop, print out the iteration count, the size of the queue, the identity and depth of the top node that is popped, whether it is discarded because it has been visited, and the successor nodes that are pushed onto the frontier. An example of a full transcript is shown below.

```
> BFS_nav ATM.graph 1 1 4 4
vertices=275, edges=641
start=(1,1), goal=(4,4), vertices: 0 and 62
iter=1, frontier=0, popped=0 (1,1), depth=0, dist2goal=4.2
 pushed 1 (1,2)
 pushed 20 (2,1)
 pushed 21 (2,2)
iter=2, frontier=2, popped=1 (1,2), depth=1, dist2goal=3.6
 pushed 2 (1,3)
 pushed 22 (2,3)
iter=3, frontier=3, popped=20 (2,1), depth=1, dist2goal=3.6
```

```
 pushed 40 (3,1)
 pushed 41 (3,2)
iter=4, frontier=4, popped=21 (2,2), depth=1, dist2goal=2.8
 pushed 42 (3,3)
iter=5, frontier=4, popped=2 (1,3), depth=2, dist2goal=3.2
 pushed 3 (1,4)
 pushed 23 (2,4)
iter=6, frontier=5, popped=22 (2,3), depth=2, dist2goal=2.2
 pushed 43 (3,4)
iter=7, frontier=5, popped=40 (3,1), depth=2, dist2goal=3.2
 pushed 59 (4,1)
 pushed 60 (4,2)
iter=8, frontier=6, popped=41 (3,2), depth=2, dist2goal=2.2
 pushed 61 (4,3)
iter=9, frontier=6, popped=42 (3,3), depth=2, dist2goal=1.4
 pushed 62 (4,4)
iter=10, frontier=6, popped=3 (1,4), depth=3, dist2goal=3.0
 pushed 4 (1,5)
 pushed 24 (2,5)
iter=11, frontier=7, popped=23 (2,4), depth=3, dist2goal=2.0
 pushed 44 (3,5)
iter=12, frontier=7, popped=43 (3,4), depth=3, dist2goal=1.0
 pushed 63 (4,5)
iter=13, frontier=7, popped=59 (4,1), depth=3, dist2goal=3.0
 pushed 75 (5,1)
 pushed 76 (5,2)
iter=14, frontier=8, popped=60 (4,2), depth=3, dist2goal=2.0
 pushed 77 (5,3)
iter=15, frontier=8, popped=61 (4,3), depth=3, dist2goal=1.0
iter=16, frontier=7, popped=62 (4,4), depth=3, dist2goal=0.0
==============
solution path:
 vertex 0 (1,1)
 vertex 21 (2,2)
 vertex 42 (3,3)
 vertex 62 (4,4)
search algorithm = BFS
total iterations = 16
max frontier size= 9
vertices visited = 23/275
path length      = 3
```

Here is the solution for going corner to corner using BFS. The solution is depicted as the yellow path below. Note that BFS finds the path with the fewest edges (definition of optimal), though it is not necessarily the shortest path.

```
> BFS_nav ATM.graph 1 20 20 1
...
solution path:
 vertex 19 (1,20)
 vertex 18 (1,19)
 vertex 17 (1,18)
 vertex 37 (2,18)
 vertex 56 (3,18)
 vertex 72 (4,18)
 vertex 84 (5,18)
```

```
 vertex 96 (6,18)
 vertex 110 (7,18)
 vertex 127 (8,18)
 vertex 153 (11,19)
 vertex 158 (12,19)
 vertex 171 (13,19)
 vertex 181 (14,17)
 vertex 180 (14,16)
 vertex 179 (14,15)
 vertex 192 (15,15)
 vertex 191 (15,14)
 vertex 204 (16,14)
 vertex 231 (18,15)
 vertex 268 (20,14)
 vertex 267 (20,13)
 vertex 266 (20,12)
 vertex 265 (20,11)
 vertex 264 (20,10)
 vertex 263 (20,9)
 vertex 262 (20,8)
 vertex 261 (20,7)
 vertex 260 (20,6)
 vertex 259 (20,5)
 vertex 258 (20,4)
 vertex 257 (20,3)
 vertex 256 (20,2)
 vertex 255 (20,1)
search algorithm = BFS
total iterations = 270
max frontier size= 22
vertices visited = 271/275
path length      = 33
```

The green path is a solution for (13,6) to (7,6). The yellow path is for (20,1) to (1,20).

Evaluation

It will be interesting to compare the performance of the three algorithms on example search problems. For time-efficiency, count the number of goal-tests performed (not necessarily the wall-clock time). For memory-efficiency, keep track of the maximum size of (number of nodes in) the frontier. Run each of the algorithms on various instances of the navigation problem on the graph above and **answer the following questions**:

- Which algorithm is fastest (finds goal in fewest iterations)?
- Which is most memory efficient (smallest max frontier size)?
- Which visits the fewest vertices?
- Which generates the shortest path length? (Is any of them optimal?)
- Are the performance differences what you expected based on the theoretical complexity analysis?
- Does BFS always find the shortest path? Does GBFS always go "straight" to the goal, or are there cases where it gets side-tracked?

What to Turn in

- You will submit your **source code** for testing using the web-based CSCE *turnin* facility, which is described here: https://wiki.cse.tamu.edu/index.php/Turning_in_Assignments_on_CSNet
- Include a **README** file that provides any instructions necessary for compiling and running your program. We will run it on some other test problems (navigation tasks on a different graph). It is your responsibility to make sure the TA can compile and test your program.
- You should include a Word document that shows **example program traces** (transcripts) for all 3 algorithms.
- You should also include a **table of performance metric**s for the 3 algorithms, for comparing things including: number of iterations (goal tests), maximum queue size, mean solution path length, etc., either for at least 5 specific problems, or averaged over at least 5 randomly chosen start/goal combinations. **Use the data to answer the Evaluation questions listed above**.