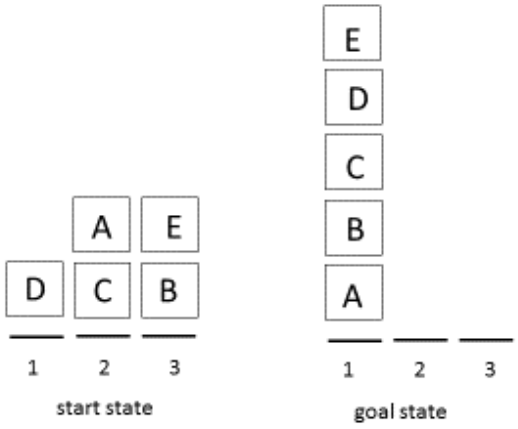
CSCE 625
Programing Assignment #1
due: Thurs Oct 5, 2017 (by start of class)

Objective

The overall goal of this assignment is to implement A* search and use it to solve a classic AI search problem: the "Blocksworld". This problem involves stacking blocks into a target configuration. You are to write a program to solve random instances of this problem using your own implementation of A* search. The focus of the project, however, is to use your intuition to develop a *heuristic* that will make solving various instances of this problem more efficient (solve them in fewer iterations).

Follow the description of A* in the textbook, i.e. GraphSearch with the frontier represented as a PriorityQueue. (Your code for A* should be written from scratch by you; however, you may use an existing implementation of PriorityQueue from a library, such as in the STL.) Specifically, you will keep the queue sorted based on f(n)=g(n)+h(n). g(n) is just the path length from the root to the current node. **The main focus of this project will be on developing a heuristic function, h(n),** for this domain that estimates the distance to the goal (number of moves to solve the problem) and testing how it affects the efficiency of the search - something more sophisticated than just "number of blocks out of place".



Description of the problem

This task involves stacking blocks. The blocks are labeled by integers. Here is an example random state and the goal state for a problem with 5 blocks on 3 stacks. (**written sideways, so the bottom of each stack is on left**)

```
examples random state:
1 |  D
2 |  C  A
3 |  B  E

goal state:
1 |  A  B  C  D  E
2 |
3 |
```

The state can be represented as a list of lists, an array, etc. However, you should make your code flexible so that you can test it on different numbers of blocks and stacks (as parameters). The initial state can be generated by just randomly assigning blocks to stacks. The goal state is always defined to have all the blocks in alphabetical order on stack 1. *The operator in this problem can pick up only the top block on each stack and move it to the top of any other stack.*

This problem becomes harder as the number of blocks scales up, and cannot be solved effectively with BFS (though you can try BFS for small numbers of blocks). You will need A* search coupled with a heuristic to find solutions efficiently. The default heuristic (call it h_0) can be taken to be the "number of blocks out of place". Your goal is to define a better heuristic that will enable your algorithm to find solutions faster (with fewer goal tests), and to be able to solve larger problems (with more blocks).

Experiments

By experimenting with your program on random problems with different numbers of blocks and stacks, you should generate a table with some measurements showing how your heuristic performs on different problems. Keep track of both time (number of iterations or goal tests) and space (max size of frontier), along with length of solutions paths (number of moves). You should try several different parameter settings, which control the difficulty of the problems. For example, start with 5 blocks and 3 stacks, then try 6 to 10 blocks to see how scalable the performance is. Then try expanding the number of stacks from 3 to 7, to see if this makes the problem easier or harder. See if you can solve "hard" problems, like with 10 blocks on 5 stacks, as shown below (the optimal solution has 18 steps):

```
1 |  D
2 |  E   F   I   J
3 |  B   G
4 |  C   H
5 |  A
```

Your heuristic does not have to be provably admissible, though it will be more efficient the closer to admissible it is. However, if it is non-admissible, it might discover sub-optimal solutions with longer paths than necessary (such as with redundant moves).

Implementation

You can use C++, Java, or Python, but you will have to make sure the TA can compile and run it for grading.

You will need to write a **Node class** for representing states in the Blockworld, and a *successor*() function for generating all legal moves from a given state.

You will have to write a "problem generator" that generates random initial states for testing. This can be done by starting from the goal state and performing a sequence of random moves to "scramble" it. You can assume the goal will always be the same, e.g. to get all the blocks in order on stack 1.

You will need to implement a method for keeping track of *visited states.* As part of applying search algorithms to real-world problems, you have to check for 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. Thus you need to develop a data structure for storing previously visited states that can be searched efficiently. You also have to find the right place to check and/or update this in your search algorithm. If you get it wrong, you risk running out of memory (because of queue expansion), or eliminating some legitimate paths to the goal. 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, whichever has least depth between the new node and the node from when it was previously visited. This is important for maintaining the *optimality* of algorithms like A*.

Finally, you will need to implement a *traceback*() method to generate and/or print out the solution path, once a goal node has been found. The solution path is the sequence of moves and intermediate states that transforms the initial state to the goal state. 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.

Make your implementation flexible so it can solve problems with an arbitrary number of blocks and an arbitrary number of stacks (e.g. provided as command-line arguments).

You might want to set an *upper-limit* on the number of iterations for convenience. (If you do so, be sure to report the number of 'failures' with your table of results.)

What to Turn in

- You will submit your 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**
- You should include a Word document with instructions on how to compile and run your program, along with a transcript that shows **example program traces** (transcripts) for your A* search.
- Include a written **description** of how your heuristic works, or what it is based on, or the logic behind it. Briefly address whether it is **admissible** or not (an informal argument will suffice).
- You should also include a **table of performance metric**s for things such as number of iterations (goal tests), maximum queue size, mean solution path length, etc., for at least 10 randomly chosen start states for each combination of parameters (number of blocks, number of stacks).
- Include a **discussion of your results**. Give some observations about the performance of your heuristic. Did it work as well as you expected? What were the largest problems you could solve? How large did the queue grow as a function of the number of iterations? Did increasing the number of stacks make the problems harder or easier? Did your program generally find optimal or sub-optimal solutions? Do you have ideas of how your heuristic could be improved?

```
> blocksworld 5 3

initial state:
1 |  B
2 |  C   E
3 |  A   D
iter=0, queue=0, f=g+h=5, depth=0
iter=1, queue=15, f=g+h=6, depth=1
iter=2, queue=27, f=g+h=6, depth=1
iter=3, queue=34, f=g+h=6, depth=1
...
success! depth=10, total_goal_tests=2339, max_queue_size=644
solution path:
1 |  B
2 |  C   E
3 |  A   D

1 |
2 |  C   E   B
3 |  A   D

1 |
2 |  C   E   B   D
3 |  A

1 |  A
2 |  C   E   B   D
3 |

1 |  A
2 |  C   E   B
3 |  D

1 |  A   B
2 |  C   E
3 |  D

1 |  A   B
2 |  C
3 |  D   E

1 |  A   B   C
2 |
3 |  D   E

1 |  A   B   C
2 |  E
3 |  D

1 |  A   B   C   D
2 |  E
3 |

1 |  A   B   C   D   E
2 |
3 |
```