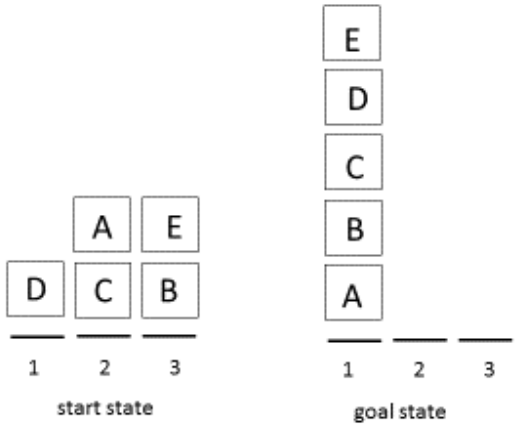
CSCE 420 - Spring 2021
Programing Assignment #2
due: Fri, Mar 5, 2021, 12:00pm (noon)

Objective

The overall goal of this assignment is to implement A* Search and use it to solve the classic "Blocksworld" (BW) AI search problem. This problem involves stacking blocks from a random initial configuration into a target configuration. You are to write a program to solve random instances of this problem using your own implementation of A* in C++, based on the iterative algorithm in the textbook using a **priority queue**. This project builds on the code you wrote for Project 1; the function for BFS() (including checking for visited states, i.e. GraphSearch) should only require slight modification to adapt it for A*. The focus of this assignment is to develop a **heuristic, h(n),** for the Blocksworld that can be used by A* to improve the efficiency of the search, enabling it to solve harder problems with longer solution paths.

A BW problem instance is given as 2 states: an initial state and a goal state. An example is shown below. The operator for generating moves in this problem can only pick up the top block on any stack and move it to the top of any other stack. Each action has equal (unit) cost. The objective is to find a sequence of actions that will transform the initial state into the goal state (preferably with the fewest moves, hence the shortest path to the goal.) Your program should be able to handle problems with different numbers of stacks and blocks, and the goal state will not always have blocks stacked in order on the first stack – it could be an arbitrary configuration. (This instance requires 10 moves to solve, starting with moving D onto E, and then A into (empty) stack 1…)



Recall that a heuristic h(n) is an estimate of the distance from a node in the search space to the goal. A* uses the heuristic score, in combination with the pathcost to n, g(n), to keep the nodes sorted in the frontier by f(n)=g(n)+h(n) using a priority queue. In the Blocksworld, the pathcost is just the depth of a node, which you should already be tracking. The heuristic function is best implemented as a member function of the State class. It also depends on the goal state, which could be different for each problem instance:

```
float State::heuristic(State& goal);
```

Your Node constructor will have to call this state->heuristic(goal) and add the value to the depth to get the overall f(n) score for the node. This will then be used to keep the nodes sorted in the priority queue, for which you should use *priority_queue* in the STL.

Your objective is to come up with the best heuristic possible, that will enable you to solve the most BW problems in the least time (measured as number of goal tests). It is not easy to come up with a quantitative function that accurately approximates the number of moves remaining; everybody will come up with different ideas; use your intuition/reasoning about the problem. You will probably end up developing a series of heuristic with better and better performance. There is always the trivial heuristic, $h_0(n)=0$, which returns a constant 0 for all nodes, and the default heuristic, $h_1(n)$=number of blocks out of place. Your job is to develop additional heuristics $h_2(n)$, ,… that perform even better. Note: your heuristic does not have to be admissible, but it should still approximate the distance to the goal (in number of moves).

Blocksworld Problem Challenge Set

I used the blockgen.py script (see Project 1) to generate a series of 45 blocksworld problems, with difficulty ranging from easy to hard (based on solution path length, and hence depth of the goal in the search tree). This is checked into the course Github project as a subdirectory (Proj2/BWCHP/) containing the 45 blocksworld problems (in the same .bwp file format we used for Project 1). These problems represent: Nstacks=3 or 5 or 10, Nblocks=5 or 10 or 20, with 5 instances for each of the 9 combinations.

What to Turn In

Create a subdirectory in your (private) Github project for the class (on the TAMU Github server) called 'Proj2'. In Proj2/, you should have at least 3 files:

- **README** – should explain how your program works (e.g. command-line arguments), and any important parameters or constraints
- **STATISTICS.txt or STATISTICS.xlsx** – this is a table (or spreadsheet) containing results for solving each of the 45 problems in the challenge set, giving the following statistics for each: solved or failed? number of goal tests, max queue size, solution path length. Note: set MAX_ITERS=100000.
- **RESULTS.txt or RESULTS.docx -** this is a text file
    - **Explain your heuristic**. Show 1 or 2 examples of states with their h(n) score to illustrate your heuristic.
    - **Give a descriptive summary of how your best heuristic performs** – how many problems could you solve? were there any trends in terms of which problems could be solved, i.e. maximum number of blocks or stacks? did it ever find sub-optimal solutions (e.g. where there exists an obvious solution with a shorter path length)? how did the number of goal tests and max queue size scale up with solution path length?
- **makefile** – we must be able to compile your C++ code on compute.cs.tamu.edu by simply call 'make'
- **BlocksworldAstar.cpp** – your main program should be called BlocksworldAstar.cpp, but you might also want to have other .cpp or .hpp files, e.g. if you want to split your code into multiple source files based on classes for modularity

The date you commit your files and push them to the TAMU Github server will be used to determine whether it is turned in on time, or whether any late penalties will be applied.

Grading

The materials you turn in will be graded according to the following criteria:

- 25% - does it compile and run without problems?
- 25% - does the implementation (of A*, successor function, heuristic, etc) look correct?
- 25% - is the heuristic described in RESULTS.txt designed well? does the summary/explanation of performance make sense?
- 25% - performance on challenge problems: number of challenge problems solved as reported in STATISTICS.txt (relative to the best in the class); are solution path lengths and other metrics reasonable?