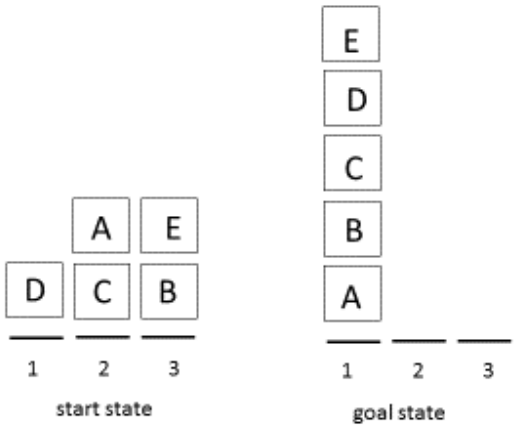
CSCE 420 - Fall 2024
Programming Assignment PA1

due: Thurs, Sep 12, 2024, 11:59 pm, pushed to your class Github account

Objective

The overall goal of this assignment is to implement Breadth-First Search (BFS) and use it to solve the classic "Blocksworld" (BW) 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 **BFS** in C++ or Python, based on the iterative algorithm (GraphSearch, also called *BestFirstSearch*) in the textbook using a **FIFO queue**. (The infrastructure you develop in this project will be used as the basis for implementing A* search in PA2.)

A BW problem instance is given as two 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 (making stack 1 empty), and then A into stack 1, and so on…)



Program Interface

The compiled program called ‘blocksworld’ should be run from the command line with an interface like this:

```
> blocksworld test1.bwp
or
> python blocksworld.py test1.bwp
```

The filename argument is a problem description in the file format described below.

```
usage: blocksworld <filename> [-MAX_ITERS <int>]
```

You might also want to add other flags on the command line. One critical aspect of the program is that it should have a built-in limit on the **maximum number of iterations** (in the main loop in *BestFirstSearch*). If that limit is reached for a given problem, the program should give up and report failure. In initial recommendation for a default for max iterations is either 100,000 or 1,000,000, depending on the speed of your computer. (Practically speaking, set it to search for a couple minutes before giving up.) Optionally, you can also include a command-line flag to adjust the limit (e.g. MAX_ITERS). Be sure to document this in your README file.

When your program finishes, it should **print out the solution path** (sequence of states between the initial state and goal), and it should print out a line of summary statistics like this (also see Transcript below):

```
statistics: prob1.bwp method BFS planlen 5 iters 9 maxq 139
```

'iters' is a count of the number of times through the main loop; 'maxq' is the maximum size of the queue. If it is not able to find a solution before reach MAX_ITERS, set *planlen* to 'FAILED'.

The size of the search space depends on the number of blocks and number of stacks. With BFS, you will only be able to solve simpler versions of BW problems, such as those requiring at most 7-10 moves (depth in the search tree). In PA2, you will develop a heuristic for A* that will be able to solve much harder BW problems, with as many as 25 moves.

File Format for Blocksworld Problems

We will use an input format for each problem, defined as follows. Line 1 indicates the number of stacks **S** and number of blocks **B** and number of moves **M** (if generated, else -1). This is followed by a separator line with 10 greater-than ('>') characters. Then the next **S** lines give the configuration of the initial state by **listing the stacks horizontally, on their side**. Blocks (assumed to be single characters, A-Z) are listed left-to-right in order from bottom to top, with no spaces. Finally, the last S lines give the goal configuration. Then, another separator line, S more lines for the goal configuration, and a final separator line. Here is an example with 3 stacks and 5 blocks (generated from 10 random moves):

```
### blocks1.bwp ###
3 5 10
>>>>>>>>>>
D
CA
BE
>>>>>>>>>>
ABCDE


>>>>>>>>>>
```

This describes the same problem as shown in the figure on page 1. Notice the two empty stacks in the goal. Do not assume the goal will not always be to have the blocks stacked in order in a single stack - any goal configuration is possible. *Your program should be able to flexibly represent states with any number of blocks and any number of states, depending on the input.*

On the course website, we will post problem set A: probA03-probA11 (with 3 stacks and 5 blocks, most of which you should be able to solve with BFS.

As part of your program, you will have to read in a problem file given as a command-line argument and use it to create objects representing the initial and goal states. You can assume that input files will always comply with this specification; you don't have to put a lot of error-checking in your function that reads these input files. Don't worry about checking for extra spaces or characters, or things like incorrect number of lines or duplicate blocks, etc.

Hints

- The first thing you will have to do is write a simple parser to read the input file (.bwp) with the problem specification, and construct the initial and goal state objects.
- You will probably want to implement a *State* class, which stores basic information about the state (stacks of blocks), and operations on them.
- You will probably also want to implement a *Node* class. A Node is like a wrapper around State; it also contains other things like a pointer to the parent from which it the state was generated (which is useful for printing out the solution path at the end when you find the goal node), the depth in the tree, etc.
- Printing out the solution path (once you find a goal node), can be done by calling a recursive function that traces the parent pointers from the goal node up to the root of the search tree. Something like this pseudocode:

```
node::print()
        if node is a root node: state->print()
        else:
                parent->print()
                state->print()
```

- An important function to write is *successors*() which returns a list of all the successors of a given state by any legal move. This generates the branches in the search tree. Don't forget to keep a pointer to the parent Node.
- You will have to implement the **GraphSearch version of BestFirstSearch** for efficiency, to keep track of visited (or reached) states. One way to do this is to create a unique string representing any given state as a hash key, and then store them in an *unordered_map* or *dict*.
- The numbers/statistics in your output do not have to exactly match those shown in my example transcripts. Number of iterations and queue size will be slightly different for everybody, because it depends on a number of implementation details. So don't worry if you can't exactly reproduce my output. But the trends should be similar (e.g. more iterations for harder problems with longer solution paths). Also pay attention to the path length of the solutions; they should be at least close to optimal (as defined by BFS).

What to Turn In

Create a subdirectory in your (private) Github project for the class called 'PA1'. All programming assignments and homeworks for this course will be checked into the same github project, using different folders.

In PA1/, you should have the following files checked-in:

- **README** – should explain how your program works (e.g. command-line arguments), and any important parameters or constraints
  - If your program takes optional flags on the command line, document them.
  - If your program has any known limitations, such as that it only works with a certain number of stacks, or number of maximum blocks, or can only solve problems of a certain type or difficulty, document all that.
- **blocksworld.cpp** – your main program should be called Blocksworld.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
- **makefile** – *we must be able to compile your C++ code on compute.cs.tamu.edu by simply calling 'make'*
- **if you are using python, then the program will be called blocksworld.py**
- **RESULTS.txt** (or RESULTS.docx) - this is a text file
  - **Give a table summarizing the performance statistics (number of iterations, plan lengths, failures, max queue size) on the test cases provided** (probA03-A11). See the example 'statistics' shown below.
- **Transcripts** – A text file showing the output for at least 3 test cases, including the solution path. If the tracing information is too large, you can just truncate it. (name these files like this, for example: probA03.transcript.txt, etc.)

The date you commit your files and push them to the **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:

- 20% - Does it compile on *compute.cs.tamu.edu*?
- 20% - Does it run (on simple test cases)?
- 20% - Does the implementation of BFS (uses a queue, successor function, checking for visited states, etc) look correct?
- 20% - Performance on test problems (probA03-probA11), as reported in the RESULTS.txt file you submit. Are solution path lengths, number of iterations, max queue size, and other metrics reasonable?
- 20% - Performance on other test cases (we will run it on a few additional test cases of our choice).

Final reminders:

- Your program MUST have an iterative function that looks like BestFirstSearch.
- You must include a written document (RESULTS.txt or .docx or .pdf) that includes a table that summarizes the performance of your program on the test problems (probA set).
- Your program does not have to exactly reproduce the output shown in my examples.

Example Transcript

```
> Blocksworld probs/probA05.bwp -MAX_ITERS 100000
success! iter=256, depth=5, max queue size=260
move 0

BD
ACE
>>>>>>>>>>
move 1
E
BD
AC
>>>>>>>>>>
move 2
EC
BD
A
>>>>>>>>>>
move 3
EC
B
AD
>>>>>>>>>>
move 4
ECB

AD
>>>>>>>>>>
move 5
ECB
D
A
>>>>>>>>>>
statistics: probs/probA05.bwp method BFS planlen 5 iter 256 maxq 260
```

RESULTS

using BFS…

```
statistics: probA03.bwp method BFS planlen 3 iter 74 maxq 91
statistics: probA04.bwp method BFS planlen 4 iter 87 maxq 113
statistics: probA05.bwp method BFS planlen 5 iter 256 maxq 260
statistics: probA06.bwp method BFS planlen 6 iter 415 maxq 350
statistics: probA07.bwp method BFS planlen 7 iter 981 maxq 540
```

```
statistics: probA08.bwp method BFS planlen 8 iter 1304 maxq 595
statistics: probA09.bwp method BFS planlen 9 iter 1801 maxq 646
statistics: probA10.bwp method BFS planlen 10 iter 2462 maxq 646
statistics: probA11.bwp method BFS planlen 11 iter 2348 maxq 602
```