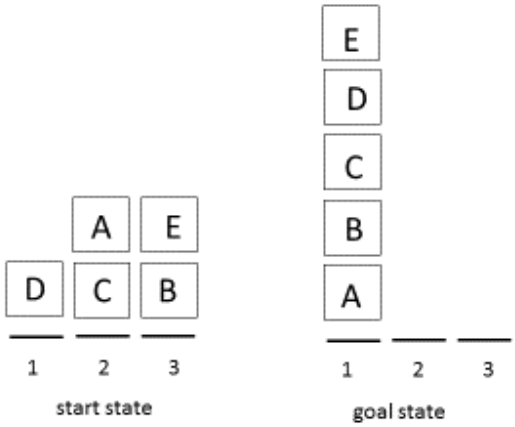
CSCE 420 - Fall 2024
Programming Assignment PA2

due: Thurs, Sep 26, 2024, 11:59 pm, pushed to your class Github account

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 will build on the code you implemented for PA1; basically, changing the algorithm from BFS to A* only requires switching from a FIFO queue to a **priority queue**, which keeps nodes sorted by f(n)=g(n)+h(n) (f(n)=estimated total path cost, g(n)=past cost from root to current node, h(n)=heuristic estimate of remaining path cost to goal). The main 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 BW problems requiring longer solution paths.

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…)



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 in the search tree. The heuristic can be thought of as a function of 2 states: the current state, and the goal state, which is constant for a given run, but could be different for each problem instance.

The simplest non-trivial heuristic function (let's call it H1) is "number of blocks out of place" (which you should implement for purposes of comparison). But H1 is simplistic and approximate (it generally underestimates the true number of moves required) and will only work

moderately well – you won't be able to find solutions to some harder problems with it. Your goal is to think of other features about the current and goal state that you could calculate that would allow you to estimate the number of moves remaining, and then let A* use this information to guide the search and find solutions more efficiently.

Program Interface

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

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

The filename argument is a problem description in the file format described below. You might also want to add other flags on the command line. For example, you might want to add a flag specifying which heuristic function to use. By default, the program should use your best heuristic; however, you can change it with the '-H' flag; this is useful for simulating BFS using H0.

```
usage: blocksworld_Astar <filename> [-H H0|H1|…] [-MAX_ITERS <int>]
```

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: probB05.bwp method Astar 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'.

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 a second set of harder problem: probB03-probB20 (with 5 stacks and 10 blocks). The 'probB' set are much harder than the 'probA' set from PA1, because the branching factor is higher. The challenge in PA2 is to design an intelligent heuristic that allows A* to efficiently discover solution for as many of these problems as possible.

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

- You should be able to re-use (or extend) the code for your BFS (i.e. GraphSearch), (with the infrastructure for checking for visiting states, printing solution paths, etc) by switching from a FIFO queue to a priority queue, which would be sorted on 'scores' you compute for each node (by adding path cost and heuristic value to get f(n); calculate these once during initialization and store the score in each node for quick access later).
- You don't have to reimplement priority queues from scratch; you can use *priority_queue* in the STL (for C++ users). (Hint: put *Node** pointers in the queue, not *Node* objects; and don't worry about memory deallocation for this project). You will have to figure out how to get it to sort Nodes on f(n) by using a comparator function. For python users, you can use any priority-queue package or implementation you want, such as *queue.PriorityQueue*. Pay attention to *direction* the queue is sorted; make sure you pop Nodes with the lowest f(n) score first.
- While you are developing your A* code, it will be helpful to *simulate* BFS at first, by setting h(n) to 0, which we can call h_0(n) (or 'H0'). Thus f(n)=g(n)=depth will be used to sort the priority queue. Later, as you develop your heuristic function, you can compare the number of iterations to BFS (A* with H0) to see if you can make it more efficient so it can solve harder problems (e.g. with solution paths of 10-20 steps). You can easily add

H1 as described above (number of blocks out of place). Then, you might create a series of better and better (more accurate) heuristics, which perform better by finding solutions in fewer iterations (or solving more problems before reaching the iteration limit and reporting failure).

- 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 'PA2'. All programming assignments and homeworks for this course will be checked into the same github project, using different folders.

In PA2/, 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, such as for selecting different heuristics, simulating BFS, or setting an iteration limit, 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_Astar.cpp** – your main program should be called Blocksworld_Astar.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_Astar.py**
- **RESULTS.txt** (or RESULTS.docx) - this is a text file
  - **Describe your heuristic**. Explain how it works, and what it computes. Show two examples of states with their h(n) score to illustrate your heuristic.
  - **Give a table summarizing the performance of your best heuristic on the test cases provided** (probB03-B20). Report the solution length with your best heuristic, number of iterations (or FAILED), and maximum queue size. See the example 'statistics' shown below.
- **Transcripts** – Show 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: probB03.transcript.txt)

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 A* (based on priority queue) and heuristic look correct?
- 20% - Does the heuristic described in RESULTS.txt look like it is designed well? Does it look like it captures important aspects of how close a state is to being solved?
- 20% - Performance on test problems (probB03-probB20), as summarized in the RESULTS.txt file you submit. Are solution path lengths, number of iterations, max queue size, and other metrics reasonable?

Final reminders:

- Your program MUST have an iterative function that looks like BestFirstSearch, and a heuristic function that, given any node, estimates the remaining distance to the goal.
- Your goal is to develop a heuristic that will allow solving as many of the challenge problems as possible, in the fewest iterations.
- Don't forget to try simulating BFS BreadthFirstSearch first, by setting the heuristic to H0.
- **You must include a written document** (RESULTS.txt or .docx or .pdf) than explains the idea behind your heuristic and summarizes its performance.
- Your program does not have to exactly reproduce the output shown in my examples.

Example Transcript

```
> cat probB05.bwp
5 10 7
>>>>>>>>>>
AE
CF
D
GHIJ
B
>>>>>>>>>>
AEG
CFI
DJB

H
>>>>>>>>>>

> blocksworld_Astar probB05.bwp
(note: you can print out whatever tracing info you want here, or turn it off,
or truncate it, or only print it every 1000 iterations...)
iter=1, depth=0, pathcost=0, heuristic=15, score=0, children=20, Qsize=0
iter=2, depth=1, pathcost=1, heuristic=12, score=13, children=20, Qsize=19
iter=3, depth=2, pathcost=2, heuristic=9, score=11, children=20, Qsize=37
iter=4, depth=3, pathcost=3, heuristic=8, score=11, children=20, Qsize=55
```

```
iter=5, depth=3, pathcost=3, heuristic=8, score=11, children=20, Qsize=73
iter=6, depth=3, pathcost=3, heuristic=8, score=11, children=20, Qsize=90
iter=7, depth=3, pathcost=3, heuristic=8, score=11, children=16, Qsize=106
iter=8, depth=4, pathcost=4, heuristic=6, score=10, children=20, Qsize=120
success! iter=9, cost=5, depth=5, max queue size=139
move 0, pathcost=0, heuristic=15, f(n)=g(n)+h(n)=0
AE
CF
D
GHIJ
B
>>>>>>>>>>
move 1, pathcost=1, heuristic=12, f(n)=g(n)+h(n)=13
AE
CF
DJ
GHI
B
>>>>>>>>>>
move 2, pathcost=2, heuristic=9, f(n)=g(n)+h(n)=11
AE
CFI
DJ
GH
B
>>>>>>>>>>
move 3, pathcost=3, heuristic=8, f(n)=g(n)+h(n)=11
AE
CFI
DJB
GH

>>>>>>>>>>
move 4, pathcost=4, heuristic=6, f(n)=g(n)+h(n)=10
AE
CFI
DJB
G
H
>>>>>>>>>>
move 5, pathcost=5, heuristic=5, f(n)=g(n)+h(n)=10
AEG
CFI
DJB

H
>>>>>>>>>>
statistics: probs/probB05.bwp method Astar planlen 5 iter 9 maxq 139
```

RESULTS

using A* with my best heuristic…

```
statistics: probA03.bwp method Astar planlen 3 iter 5 maxq 16
```

```
statistics: probA04.bwp method Astar planlen 4 iter 7 maxq 19
statistics: probA05.bwp method Astar planlen 5 iter 9 maxq 29
statistics: probA06.bwp method Astar planlen 6 iter 10 maxq 29
statistics: probA07.bwp method Astar planlen 7 iter 11 maxq 34
statistics: probA08.bwp method Astar planlen 8 iter 28 maxq 75
statistics: probA09.bwp method Astar planlen 9 iter 27 maxq 82
statistics: probA10.bwp method Astar planlen 10 iter 148 maxq 358
statistics: probA11.bwp method Astar planlen 12 iter 22 maxq 67 *
* (note: my solution for probA11 has sub-optimal length, min should be 11)

statistics: probB03.bwp method Astar planlen 3 iter 6 maxq 75
statistics: probB04.bwp method Astar planlen 4 iter 6 maxq 88
statistics: probB05.bwp method Astar planlen 5 iter 9 maxq 139
statistics: probB06.bwp method Astar planlen 6 iter 7 maxq 86
statistics: probB07.bwp method Astar planlen 7 iter 13 maxq 171
statistics: probB08.bwp method Astar planlen 8 iter 18 maxq 249
statistics: probB09.bwp method Astar planlen 9 iter 131 maxq 1929
statistics: probB10.bwp method Astar planlen 10 iter 49 maxq 713
statistics: probB11.bwp method Astar planlen 11 iter 60 maxq 954
statistics: probB12.bwp method Astar planlen 12 iter 61 maxq 996
statistics: probB13.bwp method Astar planlen 13 iter 72 maxq 1025
statistics: probB14.bwp method Astar planlen 14 iter 112 maxq 1725
statistics: probB15.bwp method Astar planlen 15 iter 144 maxq 2303
statistics: probB16.bwp method Astar planlen 16 iter 136 maxq 2106
statistics: probB17.bwp method Astar planlen 17 iter 104 maxq 1649
statistics: probB18.bwp method Astar planlen 18 iter 3641 maxq 50778
statistics: probB19.bwp method Astar planlen 19 iter 7777 maxq 107512
statistics: probB20.bwp method Astar planlen 20 iter 326 maxq 5208
```

Note: my implementation of BFS was unable to find solutions for problems B10-B20, but I was able to solve them all with A*.