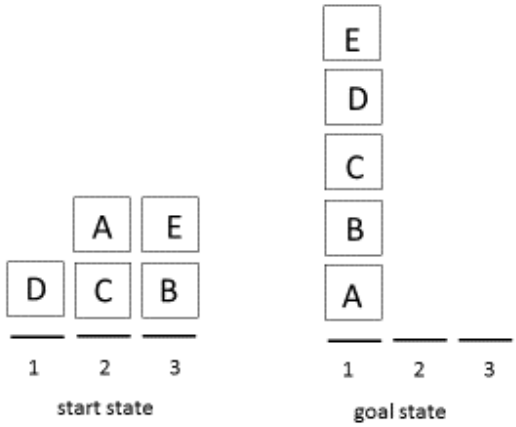
CSCE 420 - Spring 2021
Programing Assignment #1
due: Mon, Feb 15, 2021, 12:00pm (noon)

Objective

The overall goal of this assignment is to implement Breadth-First Search (BFS) and use it to solve a classic AI search problem: the "Blocksworld" (BW). This problem involves stacking blocks from a random initial configuration into a target configuration. You are to write a C++ program to solve random instances of this problem using your own implementation of BFS, based on the iterative queue-based BFS algorithm in the textbook.

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…)



The difficulty of solving Blocksworld problems depends on the path length of the solution (i.e. number of moves). Given the branching-factor for this problem, you will probably only be able to solve simple instances of the problem requiring up to around 7 moves with BFS. (In Project 2, we will extend this to solve harder Blocksworld problems using A* search).

In this domain, there are many sequences of moves that can generate the same states. Thus, you will have to specifically implement **GraphSearch**, that is, keeping track of **visited states**. (Your code for BFS should be written from scratch by you; however, you may use data structures from the STL such as *queue* and *unordered_map*.)

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:

```
### blocks1.bwp ###
3 5 -1
>>>>>>>>>>
D
CA
BE
>>>>>>>>>>
ABCDE


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

This describes the same problem as shown in the figure above. Note the two empty stacks in the goal. The goal will not always have the blocks in order in a single stack - any goal configuration is 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 block characters, etc.

State class

You will have to write at least 2 class in C++: one for State, and one for Node. The State class stores an internal representation of a configuration of Blocks. The BW specified here is designed to make the representation easy: either as a vector of vector of characters, or possibly a vector of strings. Don't use arrays; remember that there can be a variable number of stacks and blocks (though names of the blocks will be limited to A-Z, so you can assume they are just characters). You need to write a constructor function that will take the stacks (S lines) you read from the input file and create a state (with S stacks).

In addition, here are some important functions you will need to write for the State class:

```
void State::print(); // for printing out a state (in the horizontal format)
bool State::match(State*); // tell whether 2 states are equal, for goal-testing
Type State::hash(); // generate a "key" unique to each state for tracking Visited
vector<State*> successors(); // generate all children of this state given all legal
moves
```

The print() and match() functions are easy. The hash() function will be used for keeping track of visited states (see below). Basically, you need to generate a key of some Type (probably int or string) that uniquely represents each state. One suggestion is to "serialize" the object into a string by concatenating the stacks separated by a character like ';'. For example, the initial state

above could be represented by “D;CA;BE”. But there are other ways to do it. This can be used to keep track of visited states with an *unordered_map*.

The successors() function is the most important function. When you call state->successors(), it should construct and return a vector of the children states by all possible moves. This means you will have to create approximately S*(S-1) new states (copies) where the top block of each stack is moved to each of the other stacks.

Node class

Next, you will have to write a Node class. Effectively, a Node contains (or is a wrapper) around a State class. However, there is an important distinction: a Node represents a particular place in the search space. It is Nodes that the BFS algorithm operates on (stores in a queue and processes; see below). In particular, a Node has a parent, which is a pointer to the Node from which it was generated (or nullptr for the initial state, at the root of the search space). This allows you to retain the connectivity of the search space, so you can trace a path from the goal node (once found) back through a sequence of moves to the initial state. In fact, the same state can be represented by multiple paths in the search tree; i.e. different sequences of moves can produce the same state. A Node is also useful for holding other information, such as depth in the tree (or pathcost and heuristic score, as we will see in Proj 2).

The Node class should have the following member functions:

```
// checks whether this state matches another (like the goal)
bool Node::goal_test(State*);
// gets the successors of the state, and wraps them in Nodes
vector<Node*> Node::successors();
Type Node::hash(); // return hash key of state
int Node::print_path(); // print sequence of states from root down to this
```

The first 3 functions are basically wrappers that call similar functions in the State class. print_path() is used at the end, when you succeed in finding the goal. You want to trace back up the chain of parent pointers (recursively) till you hit the root of the search space, and then print out the states on the path in order from initial to goal state, representing all the move to solve the BW problem. You will have to figure the other functions you want to add, such as constructors. Remember, it is critical to store a pointer to the parent and update the depth when creating a new Node for a State.

BFS() function

Given classes for State and Node above, we are now ready to implement the main BFS search routine. You should follow the pseudocode for GraphSearch in the textbook. It should be an iterative loop, where you store unexplored nodes in a FIFO queue (aka the ‘frontier’ or ‘agenda’). In each pass, you pop the next node of the top of the queue, and then perform a goal test (to see if it matches the goal state). If it does not match the goal, then call successors() to get its children, and push them into the queue and iterate.

If the popped node does match the goal state, **print out a success message and the solution path** (sequence of moves from initial to goal state). Also **print out summary statistics, like: total number of goal tests, depth of the solution (path length), and maximum queue size during the search**.

If the queue ever becomes empty, you would break out of the loop and **report failure** (path to goal could not be found). Since for harder BW problems, the algorithm could run for a long time (for many iterations, taking up a lot of memory), it will be convenient to keep track of the number of iterations and build in stopping criterion where the algorithm gives up after a maximum number of iterations (**MAX_ITERS**) has been reached. You can set this depending on the performance of your machine, how long you are willing to wait, and how much memory you have (I set my MAX_ITERS to 100,000, which gives up after about 1 minute). It is OK to report failure for some problems that can't be solved in a reasonable amount of time.

In order to avoid searching states you've searched before, you will have to implement *GraphSearch*. This just means that you need to use a data structure to keep track of visited states. This should be done with an *unordered_map* (in the STL). This is where it is useful to have a hash() function you can call to generate a unique key for each state. For example, it could be an *unordered_map<string,Node*>*, where the string is the one described above for the State class (e.g. "D;CE;BA"), and you can store a pointer to the first Node representing such a State.

BW Problem Generator

In order to test your program, I have created a Blocksworld problem generator, implemented as a simply python script, blocksgen.py (checked into the course project on the TAMU Github Server). When you run it, you specify 3 numbers on the command-line:

```
> python blockgen.py <S:int> <B:int> <N:int>

Input arguments:
  S: number of stacks (int)
  B: number of blocks (int)
  N: number of scrambling steps or moves (int)
```

This generates a random initial state, then makes N random moves to generate a goal state, and then prints them out in the .bwp file format described above. You can use this to facilitate testing your program.

Files for this Project on TAMU Github Server

If you have not already done so, you can clone the course github project using this command:

```
> git clone https://github.tamu.edu/ioerger/cs420-spr21.git
```

Otherwise, to get updated files, do a 'pull' in the cloned directory like this:

```
> git pull
```

In the subdirectory "Proj1/", you will see this handout (.docx), the problem generator blockgen.py, and the test problems: testcase1.bwp … testcase4.bwp.

What to Turn In

Create a subdirectory in your (private) Github project for the class (on the TAMU Github server) called 'Proj1'. In Proj1/, you should have at least 4 files:

- **README** – should explain how your program works (e.g. command-line arguments), and any important parameters or constraints
- **RESULTS** – this is a text file containing printouts (transcript) of your runs on the Test Cases (see below)
- **makefile** – we must be able to compile your C++ code on linux2.cs.tamu.edu by simply call 'make'
- **BlocksworldBFS.cpp** – your main program should be called BlocksworldBFS, 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

Remember to make me (ioerger@tamu.edu) a collaborator on your project, so I can check out your code and grade it.

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:

- 20% - does it compile and run without problems?
- 40% - does the implementation (code) look correct?
- 20% - does it correctly solve the Test Cases?
- 20% - does it correctly solve Other Cases (unknown BW problems we will test it on)

Test Cases

The input files for these test cases will be provided in the Github directory.

```
################## Test Case1 ####################

> cat testcase1.bwp
2 3 3
>>>>>>>>>>
ABC

>>>>>>>>>>

CBA
>>>>>>>>>>

> BlocksworldBFS testcase1.bwp
success! iter=4, depth=3, max queue size=1
move 0
ABC

>>>>>>>>>>
move 1
```

```
AB
C
>>>>>>>>>>
move 2
A
CB
>>>>>>>>>>
move 3

CBA
>>>>>>>>>>

################## Test Case2 ####################

> cat testcase2.bwp
4 6 5
>>>>>>>>>>
BCF

A
DE
>>>>>>>>>>
B
DFC
AE

>>>>>>>>>>

> BlocksworldBFS testcase2.bwp
success! iter=990, depth=4, max queue size=2207
move 0
BCF

A
DE
>>>>>>>>>>
move 1
BCF

AE
D
>>>>>>>>>>
move 2
BCF
D
AE

>>>>>>>>>>
move 3
BC
DF
AE

>>>>>>>>>>
move 4
B
DFC
AE

>>>>>>>>>>

################## Test Case3 ####################
```

```
> cat testcase3.bwp
5 10 8
>>>>>>>>>>
EGH
AFI
BJ
C
D
>>>>>>>>>>
EG

BJ

DIFCAH
>>>>>>>>>>

> BlocksworldBFS testcase3.bwp
iter=1000, agenda size=6675, curr depth=3
iter=2000, agenda size=12364, curr depth=3
iter=3000, agenda size=17336, curr depth=4
...
iter=62000, agenda size=265416, curr depth=5
success! iter=62593, depth=5, max queue size=267914
move 0
EGH
AFI
BJ
C
D
>>>>>>>>>>
move 1
EGH
AF
BJ
C
DI
>>>>>>>>>>
move 2
EGH
A
BJ
C
DIF
>>>>>>>>>>
move 3
EGH
A
BJ

DIFC
>>>>>>>>>>
move 4
EGH

BJ

DIFCA
>>>>>>>>>>
move 5
EG

BJ
```

DIFCAH
>>>>>>>>>>