Search Programming Assignment – CS 416

For this programming assignment, you will create a set of search algorithms that find solutions to the 15-puzzle.

**Problem description:**
The 15-puzzle is a slightly larger version of the 8-puzzle discussed in Russell & Norvig:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Initial state

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Goal state

The 15-puzzle has a higher average branching factor than the 8-puzzle, but the maximum branching factor is the same (4).

**Assignment summary:**
The search algorithms you are expected to implement are:

- Breadth-first search (BFS)
- Depth-first search (DFS) – depth-first search needs to check for cycles, or a solution will most likely *never* be found.
- Greedy best-first search (GBFS), for h(1) and h(2) mentioned in Russell & Norvig
- A* (AStar), for h(1) and h(2) mentioned in Russell & Norvig
- Either:
  - Depth-limited search (DLS) and Iterative deepening (ID), or
  - Bidirectional search (BDS)
- One of:
  - Iterative deepening A* (IDAstar)
  - Recursive Best-First search (RBFS)
  - Simplified Memory-bounded A* (SMAStar)

**Assignment specifics:**

You must use Microsoft’s Visual Studio C++ to implement your software. Using Visual Studio, you will want to create a “console application.” The name of the compiled program should be FifteenProblem.exe. More instructions about how to submit your code are at the end of this document. *Follow them carefully.*

**Input:** Your program will accept instructions from the command line. The program should accept the following inputs in the following format:

- “[initialstate]” [searchmethod] [options]
  - [initialstate] must contain sixteen characters, namely the digits 1-9, letters A-F, and a space, in any order.
  - [searchmethod] can be: BFS, DFS, DLS, ID, BDS, GBFS, AStar, IDAStar, RBFS, SMAStar.
• [options] are only relevant for DLS (depth-limited search), where the option specifies the maximum depth to explore, AStar, where the option specifies which heuristic to use, and SMAStar, where the options specify the heuristic to use and the maximum number of nodes to have in memory. (The nodes in memory are not just the nodes in the fringe. Ancestors of relevant nodes also need to be stored in memory.)

Examples:
• “123456789AB DEFC” BFS
• “123456789AB DEFC” DFS
• “123456789AB DEFC” DLS 2
• “123456789AB DEFC” AStar h2
• “123456789AB DEFC” SMAStar h2 4000

Output: The output should be a comma-separated list of integers listing the following statistics. These statistics should represent the state of your search tree at the moment the solution was discovered:
• [depth], [numCreated], [numExpanded], [maxFringe]
  • [depth] represents the depth in the search tree where the solution is found. The integer will be zero if the solution is at the root and it will be “-1” if a solution was not found.
  • [numCreated] is the counter that is incremented every time a node of the search tree is created (output 0 if depth == -1).
  • [numExpanded] is the counter that will be incremented every time the search algorithm acquires the successor states to the current state; i.e. every time a node is pulled off the fringe and found not to be the solution (output 0 if depth == -1).
  • [maxFringe] is the maximum size of the fringe at any point during the search (output 0 if depth == -1).

Example:
• C:\> FifteenProblem.exe "123456789ABC DFE" BFS
  3, 54, 17, 37

Important Notes:

• Technically, the order in which you expand the nodes for exploration is arbitrary, but to standardize the output of your programs, please adhere to the following convention. Expand all nodes in the following order:
  • Right – (space moves right)
  • Down – (space moves down)
  • Left – (space moves left)
  • Up – (space moves up)

• We may test your program with a randomly permuted string of the 15 tiles and the blank. This permutation may not be a legally jumbled version of the 15-puzzle. To prevent searching for a solution to these illegal test cases, use the isNormalSolution method in the available skeleton code.
**Hints:**

- The successors of a state in this problem depend greatly on the position of the blank spot. Rather than think about which tiles can move into the blank spot, try considering where the blank spot can move. Certain numerical qualities about this position will determine whether or not the blank can move left, right, up, or down. If a blank spot moves up, then its location is being swapped with the location above it.
- The heuristics can be generated by comparing the state being evaluated to the goal state. The number of misplaced tiles is easily calculated in time linear to the number of tiles in the puzzle, but the simple solution to the Manhattan distance requires time quadratic to the number of tiles in the puzzle.
- If you plan to start from scratch, spend a lot of time thinking about what data structures you will use for your fringe. Much of the variation between the algorithms comes in how to determine which elements to pull off the fringe next.
- Many of these algorithms are more similar than different. Think about how you can use polymorphism to make your life easier.

**The provided skeleton:** You will be given a set of classes, written in C++, that you can use to provide the basis for these search algorithms. You may use this code as a starting skeleton or you may start from scratch if you prefer. If you choose to write your own code, **do not forget to adhere to the strict input/output specifications.** The classes provided in the skeleton are:

- **TreeSearch:** implements a simple search algorithm based on figure 3.9 in Russell & Norvig. This class should be used by all of your search algorithms, although some search algorithms (e.g., Iterative deepening and IDA*) will require multiple passes through the algorithm.
  - The class relies heavily on the classes Problem and Search to perform the search.
  - The algorithm returns a Solution, with statistics.
- **Problem:** an abstract base class that provides a basis for the FifteenProblem class you are required to implement. This class is responsible for determining what states are solutions, as well as which states are successors to other states.
  - The class relies heavily on the classes SearchNode, SearchNodeSet, and SearchState.
- **SearchState:** contains information about a state in the problem. This state matches the use of the term in Russell & Norvig, section 3.3. Both the initial state and goal state are SearchStates.
- **SearchNode:** contains information about a node in the problem. This node matches the use of the term in Russell & Norvig, section 3.3.
  - This algorithm also generates some statistics.
- **SearchNodeSet:** a linked-list of SearchNodes. This should not be confused with the Fringe class used by Search methods. This set is generated by the expand() method of the Problem class, and for this assignment will never hold more than four SearchNodes.
- **SearchAction:** contains information about the action required to get from one node to another. Seemingly irrelevant for FifteenProblem, but important for later extensibility.
Search: an abstract base class that provides a basis for the \textit{BFS, DFS, DLS, BDS, GBFS, AStar}, et al. classes you are required to implement. These classes are responsible for determining which node to explore next.

- Search relies heavily on the Fringe class.
- Some implementations of Search (e.g., \textit{GBFS, AStar}) rely on the Heuristic class.
- Some implementations of Search (but not all) will need to set the Solution class’s \textit{cutoff} to true.

Fringe: an abstract base class that contains the nodes that have been created but not yet explored. The details of the Fringe will depend greatly on the search algorithm. \textit{BFS} will want a Fringe that resembles a queue. \textit{DFS} (and others) will want a Fringe that resembles a stack. \textit{GBFS} (and others) will want a Fringe that will be able to return the node with the smallest heuristic value. This Fringe can be a binary search tree, a Fibonacci heap, or any other structure capable of returning a “smallest” node.

- Fringes will want their own nodes that will be storing:
  - a SearchNode
  - possibly a next node, or a left and right child node
  - possibly a heuristic value (this should \textit{not} be calculated multiple times for a single node)

Heuristic: an abstract base class that provides a basis for the \textit{MisplacedTiles} (AKA $h_1$) and \textit{ManhattanDistance} (AKA $h_2$) heuristics used by the informed search techniques.

Solution: contains whether or not success occurred. If success did not occur contains whether or not a cutoff occurred. If success did occur, Solution contains the final SearchNode of the solution, which will implicitly contain the entire solution, since each SearchNode refers back to its parent. Also contains the number of nodes accessed as well as maximum fringe.

- Also provided is the Helper.h/cpp functions that help with parts of the solution. The function isNormalSolution() returns if the solution is of the form “123456789ABCDEF ” or of the form “123456789ABCDFE ”. Use this function to make sure you look for the right solution to the problem. The PrintSolution function should be used for outputing the statistics.

- The skeleton can be found at: \url{http://www.cs.virginia.edu/~cs416/Assignments/FifteenSkel.zip}

\textbf{Extending the skeleton:}

- You are to create the following classes:
  - FifteenProblem: this class should inherit Problem.
  - BFS: this class should inherit Search
  - DFS: this class should inherit Search
  - One of:
    - DLS: this class should inherit Search
    - BDS: this class should inherit Search
  - GBFS: this class should inherit Search
  - AStar: this class should inherit Search
  - RBFS or IDAStar if those search methods are chosen: these classes should inherit Search
  - MisplacedTiles: this class should inherit Heuristic
  - ManhattanDistance: this class should inherit Heuristic
**Feedback:**
As with any assignment, we cannot foresee all possible sources of confusion. E-mail cs416@cs.virginia.edu with any questions or comments about the assignment.

- Do we need to use a particular algorithm to implement the individual searches in Bidirectional search?
  - No

- Since I know that h2 dominates h1, do I need to calculate h1 anyway?
  - Yes

- Can I use the fact that h1 & h2 always return integers?
  - Yes

- Should the program expect inputs on the command line, or via user interaction?
  - Command line

- Will all submitted boards be solvable?
  - Yes, because with the use of the function “isNormalSolution()” you can always use the proper goal state. All puzzles are solvable to either of the two states. (“123456789ABCDEF “ or “123456789ABCDFE “)

- I’ve created a new SearchAction (SearchActionWithDesc), and for the 15-problem, I only need four of them – one for each possible type of move. My problem is that the SearchNode destructor is deleting its SearchAction. I know I could create a new one for each SearchNode, but (1) that wastes space and time, and (2) it requires implementing an overloaded == operator if I want to determine if two SearchActions are equal. May I modify SearchNode so that it does not delete its SearchAction?
  - Absolutely. You should definitely do this.

- I’m having some problems with the statistics that my program is generating. Is there anything I should check for?
  - Some of the functions in the base code you are given increment counters. The statistics output by the program come from these counters at the end of execution. Therefore, if you decide to use the provided base code, you should take great care to make sure you understand what each function does. There are several things that can cause bad numbers. First, if you re-implement a function instead of using an existing function in the base code but forget to increment a counter you will probably get wrong numbers. Second, you might be calling a function twice when it only needs to be called once. This again can cause bad numbers. An important part of scientific inquiry is keeping accurate statistics in order to validate important claims about an algorithm.

- Do I have to make sure that my algorithm is not leaking any memory?
  - Absolutely. You should definitely do this. Every good algorithm should be able to clean up all of the memory it used. If memory leaks are allowed, the program may exhibit unstable behavior for harder problems. Thus programs may be graded on residual memory leaks. If you would like to check if your program leaks memory you may use this method:
    - Put an extra set of brackets around main() and add call to _CrtDumpMemoryLeaks():
      ```
      #include<crtDBG.h>
      int main(int argc, char **argv) {
      { ... your main code goes here... }
      ```
I’m using STL containers within my fringe classes for ease of coding. These classes have a built-in size() function that I’d like to use, but that means adding a size() function to my fringe so that I can access the STL container’s size() function. May I add this function?

Several points about this:
- STL containers are definitely not required, but are allowed.
- My BFS (for example) class increments the m_fringeSize variable whenever anything is inserted into the fringe, and decrements it whenever anything is removed from the fringe, so it does not actually need to calculate the size of the fringe when the fringeSize() method is invoked. You do NOT want to use a size() function that has to do this, but the STL containers are probably savvy enough that they are likewise maintaining an internal variable rather than calculating the size when asked.
- Finally, you can (and should) modify your BFSFringe (for example) without actually modifying Fringe if you want to add this functionality.

Can you give me any idea about how our main should work?

Sure. After figuring out which options to use, you should have code that is similar to:

```cpp
FifteenProblem * test = new FifteenProblem(argv[1]);
Search * method = new DLS(atoi(argv[3]));
TreeSearch * solve = new TreeSearch(test, method);
Solution * soln = solve->findSolution();
cout << test->SolutionDesc(soln);
```

For clarity’s sake, I’ve removed the code that determines which method to use, as well as code that verifies the arguments are valid, etc.

Are you going to compile and run our solutions after we turn them in?

I reserve the option to compile and run the solutions you turn in.

Can you explain how the different destructors do/should work?

Sure. I’ll try to do it from a bottom-up approach.

First, I’ll mention the classes that are in the skeleton:

1. SearchState makes a deep copy of the description passed to it during its construction, which means that it is responsible for deleting this copy when it is deleted. (By deep copy I mean that it allocates new memory and copies the data into that new memory location. A shallow copy means just storing the pointer, which really isn’t a copy at all.)
2. SearchNode makes a deep copy of the SearchState passed to it during its construction, which means that it is responsible for deleting this copy when it is deleted. It also deletes its parent, if it is an only child.
3. Solution will delete the node it contains even though it only had a shallow copy of it. This node should never have a child, so we don’t have to worry about it being deleted by that process, but we do want to make sure it’s not deleted by some other process. Depending on the order that other things are deleted (e.g., your Fringe), deleting this node might cause all of its ancestors to get deleted as well.
4. TreeSearch deletes the expanded SearchNodeSet after everything has been removed from it.
Now, I’ll mention the classes that you are responsible for creating:

5. Your Search is responsible for deleting its Fringe.
6. Your Fringe is responsible for deleting all SearchNodes in it. (If you have a root FringeNode, you can delete this and have it recurse.) There are two reasons that you must be careful here:
   1. **You do not want SearchNodes to be deleted when you’re removing FringeNodes for purpose of expansion.**
   2. The Fringe (or FringeNode) only made a shallow copy, so you do not want the process that created them, or any other process, deleting them. Normally, you shouldn’t have to worry about them being deleted by their children because nodes in the Fringe typically don’t have any children. Depending on your implementation, this might become a concern for RBFS.

7. Your main() routine is responsible for deleting its FifteenProblem, its Solution, its Search method, and its TreeSearch.
8. Of course, as there are many possible ways to implement this, there might be other deletion concerns as well.

☐ Can we create a FringeNode that will hold both a SearchNode and its heuristic? If so, how can our Fringe’s insert() method accept a heuristic, since Fringe is part of the skeleton?
   ◐ You can create any type of class that will make it easier to solve the assignment. I can tell you that I created an AStarFringeNode that did indeed have a heuristic value stored with it. In my case, AStarFringe’s insert() method did not accept a heuristic, but rather calculated the heuristic value and passed it on to AStarFringeNode. You could, however, add an additional insert() method to AStarFringe and have it take a heuristic if you like. Or, in the other direction, you could have your AStarFringeNode calculate the heuristic value in its constructor. Of course, this means that at whatever level you do this, that level will have to know what the heuristic is. Again, for my solution, my constructor for AStarFringe takes a heuristic as an argument.

☐ How should we call TreeSearch multiple times for iterative deepening and its relatives?
   ◐ ID can use the DLS search method with increasing levels of depth. You don’t actually even have to create an ID search class if you find that it is unnecessary.

☐ How do I read command line arguments?
   ◐ For the signature of your main() routine, use
     ```
     int main(int argc, char * argv[])
     ```
     Then argc is the number of command line arguments, including the name of the program, argv[0] is the name of the program, argv[1] is the first argument following the program name, etc.

☐ Can you be more specific about deletions? Specifically, when should SearchNodes be deleted?
   ◐ Theoretically, there are three times you might need to delete a SearchNode:
     1. If a SearchNode cannot be expanded, it should be deleted.
     2. If a SearchNode cannot be added to the fringe (e.g., because it contains a cycle), it should be deleted.
     3. When your fringe is deleted, all SearchNodes in it should be deleted.
     I say theoretically, because for the 15-problem, the first case will never happen. So, what if when a SearchNode is expanded none of its children can be added to
the fringe? When the last of those children is deleted (due to the second case), the parent will be deleted by that child’s destructor. (See SearchNode.cpp).

Submission: You must demonstrate a working solution for at least 2 of the search methods (although it would strongly benefit you to demonstrate working solutions for all search methods) on either Thursday February 9th or Friday the 10th between 3 and 5 PM. At Thornton Stacks, there will be eight 30-minute slots on these days containing a maximum of 10 people each. Slots will be assigned in a first-come, first-serve manner. Naturally, these demonstrations cannot take more than 3 minutes each on average. If you are unable to find a slot that matches your schedule please let us know as soon as possible as this will provide an extra burden on us. Although this demonstration is required, it is intended primarily for your benefit and will not affect your final grade on the assignment.

The executable, source code, README, and any supporting files need to be submitted in compressed form no later than 9 AM on Monday, February 20th. Use the URL http://www.cs.virginia.edu/~cs416/submit.html to submit your actual assignment.

- The name of the file you submit should be FifteenProblem.zip. As the name suggests, it should be a zip file. You can use WinZip, or even regular “zip” on blue.unix (or even gtar). If you don’t know how to do this, see the Linux/Unix directions, but be sure to include any other files that might be necessary for me to compile your solution. (E.g., *.dsp and *.dsw)
- The name of the executable should be FifteenProblem.exe. Upper case “F”, lower case “ifteen”, upper case “P”, lower case “roblem.exe”.
- The README file should contain your name, what version of MS C++ you were using (i.e., v6.0 or .NET – let me know if neither of these work for you), and any information that might be helpful in the assessment of your submission.

Students have five late days to use during the course of the semester. Each late day buys you a 24 hour extension.