

## Search Programming Assignment – CS 416

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

### Problem description:

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

5	1	2	4
6	A	3	7
9		C	8
D	E	B	F

Initial state

1	2	3	4
5	6	7	8
9	A	B	C
D	E	F	

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.
- Either:
  - Depth-limited search (DLS) and Iterative deepening, or
  - Bidirectional search (BDS)
- Greedy best-first search (GBFS), for  $h(1)$  and  $h(2)$  mentioned in Russell & Norvig
- A\*, for  $h(1)$  and  $h(2)$  mentioned in Russell & Norvig
- One of:
  - Iterative deepening A\* (IDA\*)
  - Recursive Best-First search (RBFS)
  - Simplified Memory-bounded A\* (SMA\*)

### Assignment skeleton:

You will be given a set of classes, written in C++, that are to provide the basis for these search algorithms. The classes we provided are intended to serve several purposes: they should make the work somewhat easier to complete, they should reinforce the concept that all of these algorithms are more alike than different, and they should help guide your program design. **You are not to modify these classes without permission.** The classes 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, and generates some 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.
  - This class is also responsible for providing an interpretation of the Solution.
- 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: contains a linked-list of SearchNodes. This class 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 *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., *GBFS*, *AStar*) rely on the Heuristic class.
  - Some implementations of Search (but not all) will need to set the Solution class's *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. *BFS* will want a Fringe that resembles a queue. *DFS* (and others) will want a Fringe that resembles a stack. *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 *not* be calculated multiple times for a single node)
- Heuristic: an abstract base class that provides a basis for the *MisplacedTiles* (AKA  $h_1$ ) and *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.

The skeleton can be found at

<http://www.cs.virginia.edu/~cs416/Assignments/FifteenSkel.tgz> (Unix/Linux) or  
<http://www.cs.virginia.edu/~cs416/Assignments/FifteenSkel.zip> (Windows).

**Assignment specifics:**

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

You should create a program that takes as an input the initial state of the 15-problem as well as which search method to use. The main() method can either reside in the FifteenProblem.cpp file or in a driver file, if you prefer. You may use either a Unix/Linux based C++ compiler, or a Microsoft based C++ compiler. *N.B.: I have used the skeleton files in both Unix and MS C++ v6.0, but not .Net – let me know sooner rather than later if you're experiencing problems on .Net.* If you choose to use the Microsoft based C++ compiler, you will want to create a console application. The name of the compiled program should be FifteenProblem for Unix/Linux or FifteenProblem.exe for Windows.

The program should accept the following inputs in the following format:

- initialstate searchmethod options

Examples include:

- “123456789A BDEFC” BFS
- “123456789A BDEFC” DFS
- “123456789A BDEFC” DLS 2
- “123456789A BDEFC” AStar h2
- “123456789A BDEFC” SMAStar h2 4000

Input Details:

- The initial state should contain sixteen characters, namely the digits 1-9, letters A-F, and a space, in any order.
- The search method 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.)

The output should contain:

- A description of each state from initial to goal, *or* a message indicating that either a failure or a cutoff occurred. (Can a failure occur without a cutoff for this problem?) The description for the BFS example listed above would look like:

```
1234
5678
9A B
DEFC
```

```
1234
5678
9AB
DEFC
```

```
1234
5678
9ABC
DEF
```

- The depth of the solution, if a solution was found.
- The number of nodes expanded.
- The maximum size of the fringe.
- The number of states accessed.
- The number of nodes created.
- The number of nodes deleted.

**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.

**Feedback:**

As with any assignment, we cannot foresee all possible sources of confusion. E-mail [cs416@cs.virginia.edu](mailto:cs416@cs.virginia.edu) with any questions or comments about the assignment. However, in general such e-mails will *not* be answered directly. Rather, this section of this document will be updated to reflect those questions or comments worth addressing (i.e., those not addressing problems specific to a student’s particular block of code, etc.), so check back before e-mailing us to make sure your issue has not already been addressed.

- 🔒 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, for the primary reason that it will take far too long (regardless of search method) to determine that they are not solvable, unless one adds a second “goal state” (with the 14 and 15 swapped, say), which if reached implies that the board is not solvable. For those who find the assignment too easy, I recommend this enhancement. (It shouldn’t be that difficult.)
- 🔒 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. In fact, I’ll modify my SearchNode as well! An excellent suggestion. (Note, I’ve updated the skeleton files that are mentioned in this document, but you are still permitted to use the original skeletons. Further note, I still don’t want any changes to the skeleton, without our approval.)
- 🔒 I noticed that SearchState wasn’t deleting m\_desc, and that the implementation of the == operator was using a SearchState instead of a reference to a SearchState, and ...
  - ↔ Thank you for pointing out some pointer errors in my code. These errors do not affect the overall design of the skeleton (and hence should not have a major impact on the code that uses it), but are significant in that they are not properly allocating or deallocating memory. In the best cases, this is causing memory leaks. In the worst cases (if you are using it differently than I), it can lead to segmentation faults. I’ve updated the skeleton files to reflect these modifications as well. Please keep bringing any additional mistakes/oversights to my attention.

- 🔒 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.
  
- 🔒 How can I use the SearchNodeSet when there is no setNext() or setValue() methods?

  - ↪ It's all in the constructor. Sample code that properly uses it is:
 

```
SearchNodeSet * sample = 0;
if (x)
    sample = new SearchNodeSet(newNode, sample);
if (y)
    sample = new SearchNodeSet(anotherNewNode, sample);
```

If x and y are both true, sample will now look like:

anotherNewNode

➔

newNode

➔ Null

Be sure you know what sample will look like if one or both are false!
  
- 🔒 I'm getting the following error in MS C++ v6.0: error C2665: 'delete' : none of the 2 overloads can convert parameter 1 from type 'const char \*'. What should I do?

  - ↪ If you're on a computer which you can update, download the latest patch for MS C++. That should fix the problem. If not (e.g., you're on a computer in Thornton), change "const char \* m\_desc" to be "char \* m\_desc" and "const char \* desc" to be "char \* desc" in SearchState.cpp and SearchState.h.
  
- 🔒 How should I specify which heuristic to use for AStar, etc.?

  - ↪ I've updated the assignment specifics to provide this information.
  
- 🔒 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:
 

```
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.
  
- 🔒 What exactly does number of states accessed mean?

- ⇒ The number of states accessed refers to:
  1. The number of states compared against the goal state, plus
  2. The number of states compared against other states (e.g., when determining if a cycle exists)

This is important in helping to understand how checking for cycles adds to the time complexity.

- 🔒 Are you going to compile and run our solutions after we turn them in? That's the impression I was under, but in class on Tuesday it was brought up that it would be sufficient to demonstrate our solution from the lab, whether it be running on blue, cobra, or our home machine.

- ⇒ I reserve the option to compile and run the solutions you turn in. My desire is to not have to do this if your solutions are perfect during the demonstration. If I have to compile, I will compile first on cobra. If that doesn't work, I will then compile on blue. (I didn't mention cobra, because I'm expecting most students don't have access to cobra.) For Windows solutions, I'll be using my office computer. If that doesn't work, I'll try a computer in Thornton stacks.

- 🔒 Could you elaborate on the supposed memory leak that was touched on Thursday in class?

- ⇒ The memory leak was being caused because when children nodes are being deleted, the parent nodes were not being deleted, even if they don't have any more children. (They should NOT be deleted if they have other children.)

- 🔒 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. *N.B.: this can be problematic since only a shallow copy was made of the parent in the first place. Whatever processed generated the parent, therefore cannot delete it or else the object will be deleted twice, leading to unpredictable (and usually bad) behavior.*
3. SearchNodeSet will only delete its next 'set node' (as opposed to a SearchNode) if the pointer still exists to it. In reality, this probably never happens.
4. 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.

5. TreeSearch deletes the expanded SearchNodeSet after everything has been removed from it.

Now, I'll mention the classes that you are responsible for creating:

6. Your Search is responsible for deleting its Fringe.
7. 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.*
8. Your main() routine is responsible for deleting its FifteenProblem, its Solution, its Search method, and its TreeSearch.
9. 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.

🔒 It seems I'm getting a type mismatch with regards to whether or not the SearchNode should be const in:

```
SearchNode(const SearchState * state, SearchNode * parent,
           const SearchAction * action);
```

Should parent be const?

⇒ At one point, parent was const, but since the children are now responsible for changing the parent's m\_numChildren, this is no longer the case. Code that you wrote before this change will have to be updated to reflect this necessary change.

🔒 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.

🔒 The assignment operator (which is used by the copy constructor) assumes that m\_parent will exist, but this will not be true if I'm copying the root.

↪ True. I will fix this in a later release, if more bugs are uncovered. In the meantime, if this error impacts you, please change the line:

```
m_parent->m_numChildren++;
```

to:

```
if (m_parent)
    m_parent->m_numChildren++;
```

🔒 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 Wednesday, February 19<sup>th</sup>, or Thursday, February 20<sup>th</sup> between 3 and 5 PM. 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 source code and supporting files need to be submitted in compressed form no later than 9 AM on Monday, February 24<sup>th</sup>. Use the URL <http://www.cs.virginia.edu/~cs416/submit.html> to submit your actual assignment.

- If you are submitting a Linux/Unix solution:
  - The name of the file you submit should be FifteenProblem.tgz. As the name suggests, it should be a gzipped tarball. To create a gzipped tarball, type “`gtar -cvzf FifteenProblem.tgz README Makefile *.cpp *.h`”
  - The solution needs to be able to compile and run on blue.unix.
  - Be sure to include a Makefile, so that all I need to type to compile your submission is “make”.
  - The name of the executable should be FifteenProblem. Upper case “F”, lower case “ifteen”, upper case “P”, lower case “roblem”.
  - The README file should contain your name and any information that might be helpful in the assessment of your submission.
- If you are submitting a Microsoft solution:
  - 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.

You may submit either the file FifteenProblem.tgz for Linux/Unix solutions or FifteenProblem.zip for Windows versions.

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