Artificial Intelligence	Assignment #2	Due: start of class, Fri., 2/27

This is an individual assignment. Do your own work. You may not discuss any aspect of the problem (design, approach, code) with anyone but the course instructor.

This assignment will have you implement, test, and compare the performance of several different uninformed and informed search strategies using F# and Visual Studio. Part 1 will use a console project written entirely in F#. Part 2 will use a Windows Forms GUI project in which the F# problem solving code is put in a library project which is accessed by a C# project that implements the GUI.

Part 1

1. **Using the provided program2_part1_start project** (similar to the search2 project from class, but with a couple of modifications), create a new function in the TreeSearch module that uses an appropriate combiner function and the general tree search function to **implement the A* heuristic search.** The combiner function must set each node's value field and sort the frontier based on each node's value field similar to how the provided greedy best first search (gbfs) works. But the value will be different for A*.
2. Define another search function that implements **depth-limited-search** using a combiner and treeSearch. This search function should take an additional parameter for the maximum depth, and the combiner should remove any nodes whose depth exceeds that limit. **Use List.filter and and an anonymous function** to implement your combiner. List.filter's first parameter is a function (anonymous for this assignment) that should return true for the list items you want to keep in the list. The list to process is the second/last parameter of List.filter so you can (and are required to for this assignment) **use the forward pipeline operator** to pass it in from the left. The items for which the function returns true will be in the resulting list; other items will be "filtered out" / removed.
3. Use your depth-limited search created in step 2 to **implement an iterative-deepening-search function**. Unlike the searches above, this one won't use treeSearch directly. It will just call your implementation of depth-limited search repeatedly. Start with a limit of 1 and increase it by 1 with each iteration until the problem is solved.
4. Locate the 10 lines of code before the final ReadLine call **at the end of the NumberProblem module**. Make 5 more copies of them and **modify** them so that you're not creating the mutable variables again (not possible), but just assigning them new values in each of the copies. Modify the comment and the call of the search function **so that all the search strategies except depth-limited search are tested.**
5. Run the program (e.g., with the start debugging / play button) and after all 6 searches have finished. In the console window displaying the results, right-click the title bar and choose Edit > Select All. Then hit Enter to copy all the output to the clipboard. Then paste it all into a text file (e.g., into a Notepad document). **Save the resulting text file and print it.**
6. **Write a short analysis of your results**. Which searches (2 or 3) were fastest? Which ones were slowest? Which ones used the least space? Which ones used the most space?

Which ones found the path with the fewest steps? Which ones found the path with the minimum weighted length/cost?

7. **Make the EightPuzzle module the project's entry point** (move it to the bottom of the list of source files in the solution explorer window using alt-downarrow). Run the program and note that it solves the 8 puzzle pretty quickly with breadth-first search, but it only requires 6 steps for the current start state. It can require as many as 31, so the time and space needed for breadth-first search will be prohibitive due to the exponential growth of both with the depth of the goal node. **Change the search being used to your implementation of A***. Now you should be able to **change the file name used to set the start state** (before the definition of the problem near the bottom of the file) **to "start12.txt"**. Run the program again. Depending on the speed of your computer, it should find the best solution (fewest steps) in 5-10 seconds. To handle instances of the problem that require more steps to find a solution, you'll need to make some additional changes.
8. Now you will **implement graph search versions of all the strategies for which you've been using tree search**. Copy the searches you added to the bottom of the TreeSearch module to the GraphSearch module. Change the open statement at the top of the EightPuzzle module so that it uses GraphSearch rather than TreeSearch. You can change the name of the treeSearch function and all the calls of it in the GraphSearch module to graphSearch (e.g., with ctrl-h for search & replace), but you don't have to. Use a mutable variable called expanded whose value starts as an empty list defined in the main search function (treeSearch or graphSearch). **Before expanding the current node at the front of the frontier, add the state stored in that node to expanded** (list of expanded states). Modify the expand function so that it takes this list as an additional parameter. Before adding/yielding a child node to the list of children created by expand, make sure the state you would use to create the new node isn't on the expanded list. You might want to look at the documentation for List.tryFind and only yield a node if that function returns None (indicates it wasn't found). Using graph search with A*, you should be able to solve the 12-step puzzle in a few hundredths of a second or with breadth-first search in less than a second.
9. Now **change the file name used to create the start state to "start20.txt"**. A* should still find a solution in 1-2 seconds, but a better heuristic will make it faster. **Define a second heuristic function in addition to the given function h1.** The new function should use what's called the Manhattan distance metric (distance using only North-South or East-West moves) added up over all the tiles (how many "blocks" is each tile away from where it would be in the goal state). This is more accurate and larger than the h1 heuristic, which is just the count of how many tiles are out of place. Using the new heuristic should make A* more directed to the goal, resulting in less exploration and quicker discovery of the shortest path to the goal, i.e., it should now be able to solve the 20-step puzzle in a few hundredths of a second. **Use File > Save All to save your project and print all 4 files in the project.**

Part 2

10. Now you'll complete a multi-project, multi-language solution that lets the user or your AI program solve the 8 puzzle using a GUI and some basic animation.

11. **Open the provided program2_part2_start project**. Note there are two projects in the solution explorer window. EightPuzzleSearch at the bottom is an F# library project that compiles to a dynamic link library (.dll) file. 8Puzzle at the top is a C# Windows Forms application project that uses the F# library to solve the 8 puzzle. You won't need to know anything about C# to make this work. **Double-click the Solver.fs file** in the F# project to view the code. For the library project the module contains both the search code and the problem specification and is defined with = after the module name and everything after that is indented (needed for a library like this). It's also a little different from the version in part 1 in that each node doesn't just store the parent state (which really isn't enough to extract the solution / sequence of actions needed to solve the problem) but a list of indices into the problem's actions array that represents the actions needed to get there from the start node. That way we can create an array of action functions, action names, or action numbers corresponding to the solution. To solve the 20-step puzzle in a reasonable amount of time, **add your implementation of the A* search you made in step 1 after gbfs**.
12. **Add the modifications you made in step 8** to make functions graphSearch and expand perform a real graph search using an expanded list.
13. **Add your second heuristic function from step 9** and use it in creating the problem structure. The F# solver is now complete. You need to build a new version of the dll file with your changes. **Right-click the F# project in the solution explorer window and choose Rebuild from the pop-up menu**.
14. **Double-click the Form1.cs file** in the C# 8Puzzle project. You'll see the window with a solve button on the bottom and a square button used as a model for generating the numbered tiles with code on start up. Use the **View > Code** menu command to see the C# code. In the first line of the last method in this file **change gbfs** in the call of EightPuzzleSearch.gbfs **to the name of your function for doing A***. With your changes, the solve button should solve the problem quickly, reset the tiles if needed to the start state, and use the actions used to reach the goal node to demonstrate the solution that was found. Run the program to confirm this. **Use the Save All command from the File menu to save your modifed solution.**

Submit your printouts of all 4 source files from the part 1 project, your output from step 5 and your analysis from step 6. Make a folder named after you in the drop box of our class folder and copy the final version of both Visual Studio solution folders into it. Alternatively, you may upload one or two zip files containing the solutions, which you may upload from off campus using the address pathfinder.bloomu.edu in your browser.