# Lab 3, Artificial Intelligence, spring 2017

## Table of Contents



## Introduction

This is the third and last lab of a series of labs that will result to a program in which you can play and also automatic solve an n-puzzle, where n can be 3, 8 or 15. You have an example of such a puzzle at http://mypuzzle.org/sliding. This is a classical problem used in courses about AI. In your textbook (Luger), an 8-puzzle is used as an example. Use the textbook for support and help in your work.

You have to do your coding in Java and with help of a project in NetBeans IDE. You are not allowed to use any third-party software libraries; you are limited to use the Java SE.

Although there are many solutions on the Internet you have to construct your own program that consists solely of your own written code, with the exception of the code skeleton in the section First Aid. You are allowed to use the complete skeleton or parts from it. You may not copy someone else's solution or parts of it and you may not allow someone else to copy your solution or parts of it. The fact that you send in a solution also confirms that you have read and understand the rules that apply to cheating and plagiarism, see http://www.du.se/en/Library/Academic-Writing/Copyright-and-Plagiarism/ and its sub sites. Your solution will be checked against other solutions with help of MOSS. MOSS is a system specially designed for checking program code for plagiarism. You can read more about MOSS on the following website, http://theory.stanford.edu/~aiken/moss/ .

## Hand in your solution

You shall not hand in the parts of your solution that will be the result from each separate lab. You only have to hand in your final solution that you should have after this last lab. The hand in shall consist of the project directory that is created in NetBeans plus a short report where you explain your solution. Make a zip-file containing the project catalog from NetBeans or the IDE that you have used, and the report. Up load the zip-fil to the hand in map in Fronter. Please, hand-in your solution before the end of the course, Sunday 2017-03-26. The hand-map closes at 23:59, 2017-04-16. After that time point any new hand-ins will not be accepted.

## The application

In this lab, you should add a heuristic best first search to your program. When you finish the lab your program shall;

- Execute in the command prompt
- Have a menu-system with at least following options:
  - Quit the program.
  - Create a new n-puzzle of a selectable size of n.
  - Mix the tiles by automatic move around them.
  - Manually solve the puzzle by move tile after tile.
  - Read in a state for an n-puzzle and assign this state as the start-state or goal state for automatically solving the puzzle
  - It shall be possible to select the type of search, breath, depth or best first search. If the user select depth first search, it must also be possible to limit the depth of the search. The best first search shall use your heuristic implementation.
- Consist of at least classes that represent; the menu system, the solver, breadth first search, depth first search, best first search and the puzzle.

The breadth, depth and best first search has to be implemented according to the algorithms in your textbook, see chapter three and four (Luger). When you solve the textbooks examples of the 8-puzzle, your program shall give the same output as in chapter three and four.

## A proposal of a work plan

1. Read through the algorithm for best first search, see figure 1 or chapter four of your textbook (Luger).

```
Function best_first-search;

begin
   open := [Start];
   closed := [ ];
   while open != [ ] do
      begin
         remove leftmost state from open, call it X;
         if X is a goal then return the path from Start to X
         else begin
            generate children of X;
            for each child of X do
            case
               the child is not open or closed;
                  begin
                     assign the child a heuristic value;
                     add the child to open
                  end;
               the child is already on open;
                  if the child was reached by a shorter path
                  then give the state on open the shorter path
               the child is already on closed;
                  if the child was reached by a shorter path then
                     begin
                        remove the state from closed;
                        add the child to open
                     end;
            end;
            put X on closed;
            re-order states on open by heuristic merit (best leftmost)
      end;
return FAIL
end.
```

priority case - like in the hospital - almost dead -> first
- normal sick -> has to wait

has a nother horistic part

*Figure 1: An algorithm for best first search, see chapter four in your textbook, (Luger).*

As you notice in figure 1, the open list is working as a priority queue. The state with the best heuristic value shall always be the state that is first taken out from the queue. If a child has a board equal to the board of a state that already exist in the open list, then, according to the algorithm that state in the open list should be updated with the shorter path from the child. Don't forget that the updated path affects the heuristic value of that state, and also the parent of that has to be updated. Could it be more efficient to simple replace the state in the open list with the child that has the shorter path?

2. Create a separate small program and checkup how you should implement and use a priority queue. Shall you use a linked list and search for the node with the best Heuristic value when you remove a node from the list in purpose to update X or shall you keep the list sorted all the time. Another option is to use the PriorityQueue that is in the package java.util.
Regardless if you use some type of list or the PriorityQueue it can be convenient to define a natural order for the states. You do that by implement the interface Comparable to the classes that represent a state. You find the interface Comparable in the package java.lang.

You can read about the Comparable interface at:
https://docs.oracle.com/javase/8/docs/api/index.html?java/util/package-summary.html
and how to give objects a natural order at:
https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

Look also at the code skeleton that is included to this lab, lab03.zip. In that skeleton you have an implementation of the Comparable interface for the state classes.

3. Read through the section First Aid in this lab. Decide if you shall continue with the program you constructed in the previous lab an add the new functionality to that program, or if you shall use the skeleton that is given for this lab and copy in the parts that you did in lab 1 and 2 to that skeleton, or if you only shall use some ideas from the First Aid and work out a the rest of the solution by your own.

5. Implement the best first algorithm and check if all of the three search algorithms breadth, depth and best first, give correct results for the tree of letters.

6. Add heuristic to the puzzle so it becomes possible to solve puzzles with help of the best first algorithm. Don't forget to check that the two other algorithms are still working for the updated puzzle.

7. Fine tune the implementation of the three searching algorithms. Is it possible improves the heuristic? Are there more efficient collections to use, what happens if you turn around your thinking, start the search with the goal and search for the start, and other ideas that maybe can improve the performance of the search algorithms?

8. When you are satisfied with your solution, run the start and goal state that you have in the file testPuzzle.zip a few times, and write in your best results into the table with search results.

9. Write a short report in which you explain and describe your solution.

10. Hand in your solution; see the section Hand in your solution, in this document.

## First aid!

If you have troubles to find a useful design of your program, maybe this section can be helpful.

Figure 2 shows a class-diagram over simple solution of the application you are going to construct.

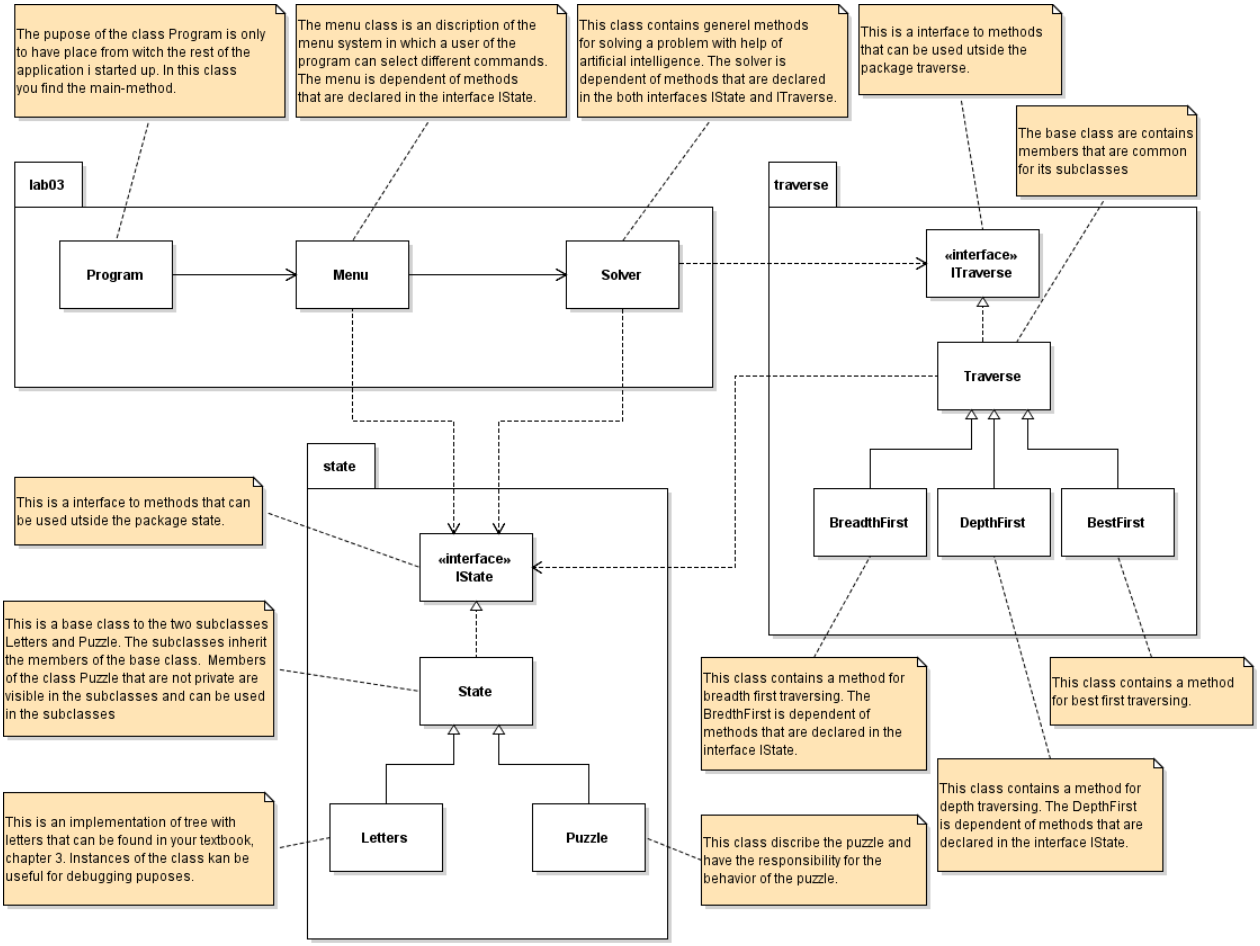


*Figure 2: A class diagram that show a design of the application.*

One different from the privies class diagram in lab 2 and this diagram in figure 2 is that package traverse contains now a class for best first search. Another different is that the concrete classes for different type of traversing's has been subclasses to the base class Travers. You can find a similar construction in the package state.

Every member that are equal in all the subclasses been moved up to the base class. A subclass can use these members as if the where written directly into the subclass. The advance of this is that you avoid writing the same code in several different classes. Only members that are special for each subclass are left into the subclasses. You use to say that a subclass inherent the members of its base class. These inherited members become visible in the subclass, if they not are private in the base class. In Java you use the reserved word inherent to declare that a class inherent from another class.

You find a proposal of a code skeleton for the above-described design in the file-archive lab03.zip. If you want, you can download this code skeleton and continue to develop it to a solution for lab 3. The content of the file lab03.zip is a project directory that you can open up and continue to work with in NetBeans.

It is very easy to rename packages, classes, methods and variables in NetBeans. Highlight the name you want to change, right click and select refactor. In that way you can change the name of the package that you created in the previous lab. It is possible to move a class from one package to another by dragging the file of the class to the package that you want the class belong to.

## Literature

Luger G. F., (2009) *Artificial intelligence Structures and Strategies for Complex Problem Solving*, sixth edition, Pearson Education Inc., ISBN-13: 978-0-13-209001-8.