CSG2341 Intelligent Systems

Workshop 7: Evolving Plans for Tartarus

Related Objectives from the Unit outline:

- Identify appropriate intelligent system solutions for a range of computational intelligence tasks.
- Demonstrate the ability to apply computational intelligence techniques to a range of tasks normally considered to require computational intelligence.

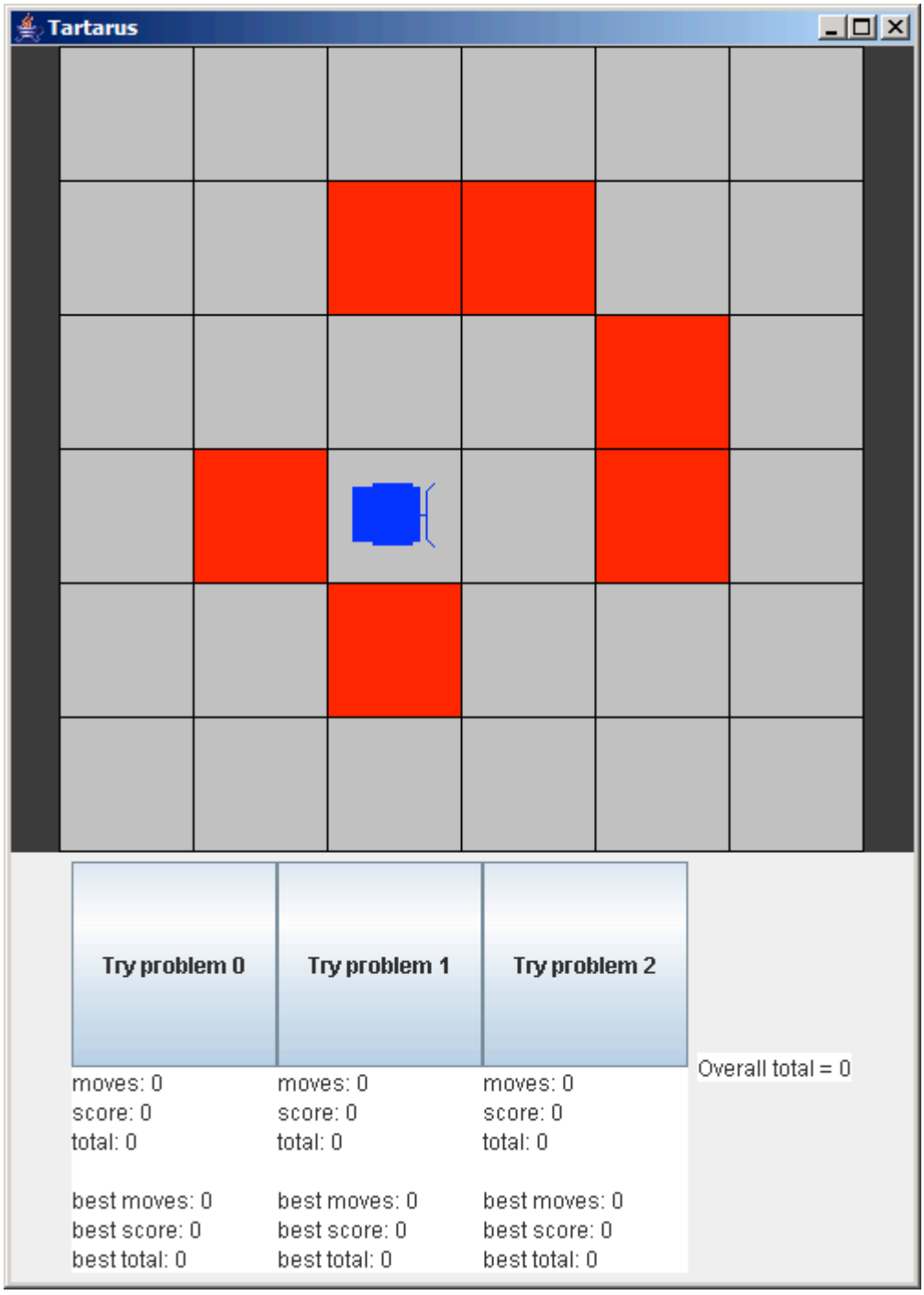
Learning Outcomes:

After completing this workshop, you should be able to demonstrate an understanding of the resources required to implement a problem solution based on evolutionary algorithms, to describe and analyse the steps in designing an evolutionary algorithm to solve an optimisation problem (in this case, for a scheduling problem) and use a computer package to implement a genetic algorithm to solve such a problem.

Warming up

This workshop shows how a genetic algorithm can be used to solve a difficult logic problem. First we will try to solve the problem by hand. The winning team will earn a fantastic prize from the tutor.

Go to BlackBoard and download the file Tartarus.jar. Double-click on it, and you should see this window open:



The top half of the window shows the Tartarus board, a 6x6 grid surrounded by walls. Some of the cells of the grid are empty (the light grey ones) and some contain boxes (the red ones). There is also a bulldozer (blue) on one of the empty cells.

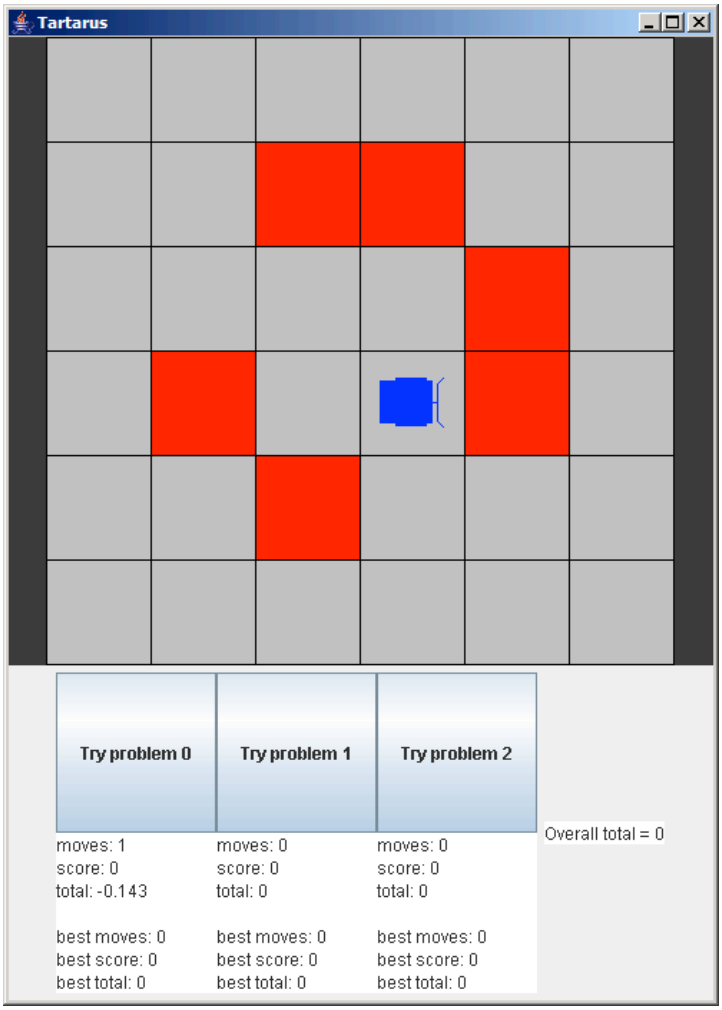
The bulldozer's job is to push the boxes into the corners, or up against the walls, using a minimum number of moves.

On each move, it can go forward, or it can turn to its right or to its left. If there is a box ahead of it, and an empty space for the box to move into, the bulldozer can push the block by going forward.

You can control the bulldozer by pressing keys on the keyboard:

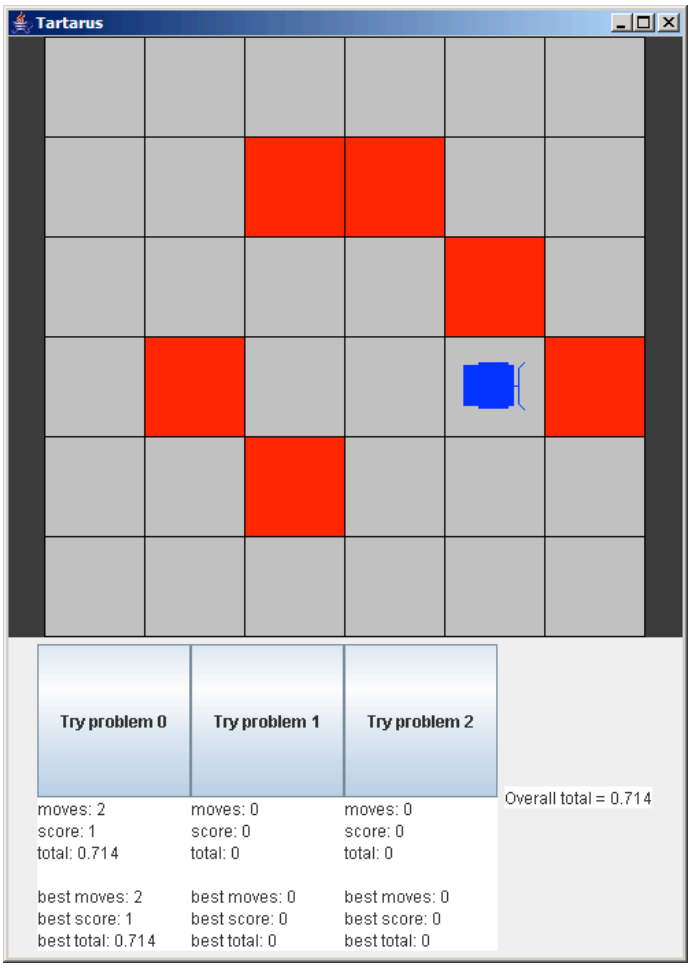
- Push "w" to go forward
- Push "a" to turn left
- Push "d" to turn right

Here's what happens if you push "w" with the board shown above:



Notice that some of the numbers near the bottom of the window have changed.

Under the button “Try problem 0”, the value of “moves” has changed to 1, and the value of “total” to -0.143. Now if you press “w” once more, you will see:

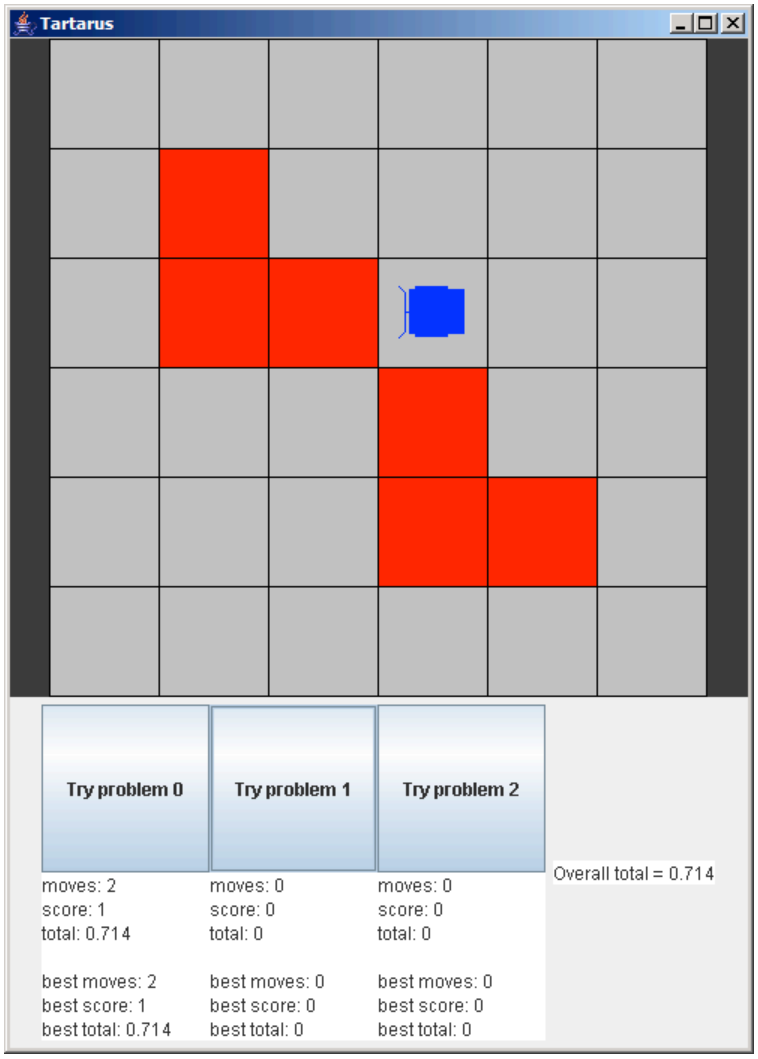


Now “moves” is 2, “score” is 1 and “total” is 0.714. Also the “best” values for problem 0 have changed, and the overall total is 0.714.

The score for a board is 2 points for each block in a corner, and 1 for each block against a wall. A perfect score is 10 – one block in each corner and the other two blocks against a wall. Each move costs 1/7th of a point, so the total for each problem is its score minus 1/7th of the number of moves to get that score.

The overall total is the sum of the totals for each problem.

You can switch to another problem by clicking on the “Try problem n” button. E.g. If we click on “Try problem 1”, we see:



Notice that the positions of the blocks and the bulldozer have changed, but the best values for problem 0 have been remembered.

You can switch between problems as often as you like. Each time you switch, you start again on that problem, but the best values are not lost. If you manage to get a better total, the best values will be updated.

That's it! The team that gets the best overall total by the end of the class wins.

Good luck!

Notes:

1. This puzzle is called “Tartarus” – to read about it, try Googling for “Tartarus Ashlock”.
2. Once you have played with it a bit, think about what it would take to write a computer program to solve the puzzle.
3. For a more difficult task, start Tartarus and press the “2” key. You will then only be able to see the cells near the bulldozer.
4. Even more difficult: press the “3” key. The view you see will always be in the middle of the window now, and will be rotated so that the bulldozer is facing upwards – that is, you see what the bulldozer sees.

The Genetic Algorithm

OK – now we are going to try to solve this problem using a genetic algorithm.

Task:

Step 1

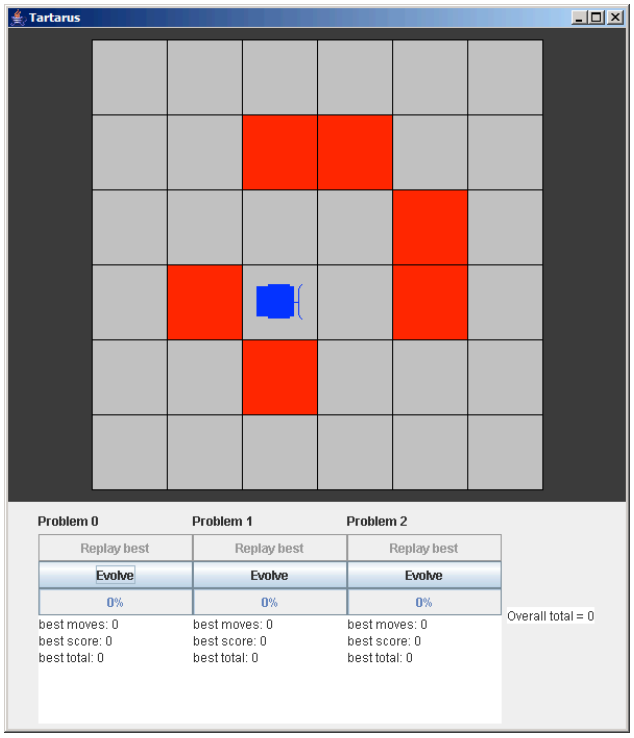
Download and unzip Tartatus.zip from BlackBoard.

Step 2

Create a NetBeans project for yourself using the contents of the zip file.

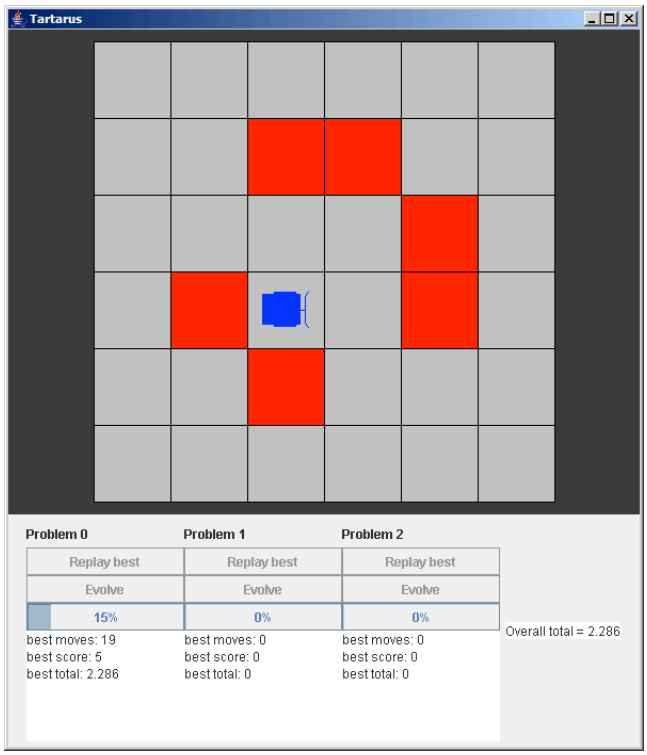
Step 3

Compile it and run it. You should see this window:

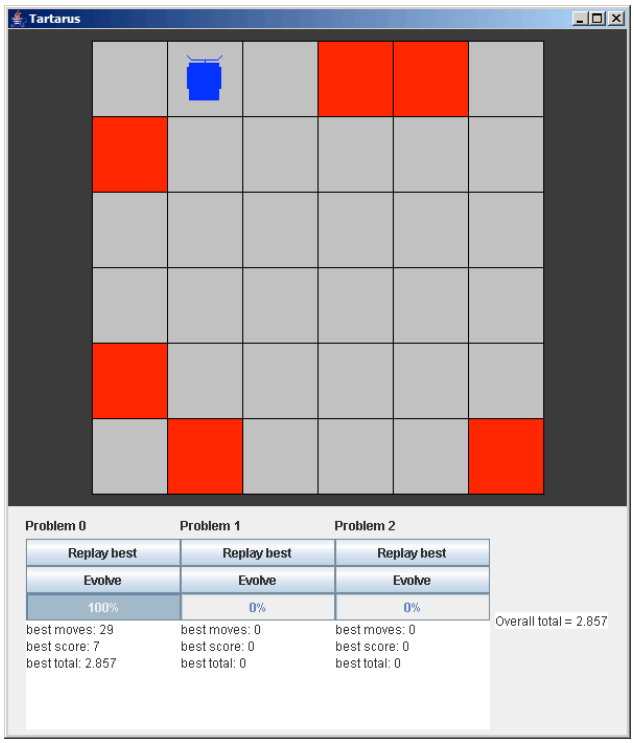


This is like the application you used in the warm-up, except that we now have an "Evolve" button and a progress bar for each Problem.

Click on the Evolve button for Problem 1. You should see something like:



An evolutionary algorithm (actually a genetic algorithm) is running. The progress bar shows that it is 15% complete. The best solution it has found so far gives a score of 5 in 19 moves, for a total of 2.286. After a minute of so, the GA will finish and you should see something like:



The best plan now gives a total of 2.857. As evolutionary algorithms are *stochastic*, you may get a different result. You can watch the best plan in action by clicking on the “Replay best” button.

Step 4

Run the GA on the other two problems by clicking on the other Evolve buttons.

You can run the GA as many times as you like on each problem. The application will keep track of the best plans over all the runs. It shouldn’t take you long to get a very good total.

Step 5

Now we are going to use the same application to solve a slightly different version of the problem.

What we’d really like, is to get the GA to find solutions that achieve a maximum score (10) in the least possible number of moves. Unless you were very lucky, the GA probably didn’t achieve a maximum score on any of the three problems.

We can encourage the GA to do what we want by modifying the fitness function. This may be found in TestCase.java, right at the bottom:

```
private static double calcTotal(int moves, int score)
{
    return score-moves/7.0;
}
```

If we increase the constant 7 to say 25, this should put more emphasis on the score (at the moment, it is not worth increasing the score by 1 unless you can do it in less than 7 moves). Change the 7 to a 25, recompile, and rerun the application. Compare results with other class members to see what the shortest solutions are for each problem.

The class as a whole should find solutions that would take a human many hours of experimentation to equal. This is the benefit of EAs : they can search a large search space in an intelligently efficient way, trying many more solutions that a human could, even with little understanding of how to solve the problem.

Step 6

We can try to help the GA along a little. Dozer plans are represented using a string of “F’, “L” and “R”s as the genome. For example:

“FFLFRLFFLLL…….”

Represents a play that says : go forward, then forward again, then turn to your left, then go forward … and so on.

But this example contains a sequence of three “L”s, which is not very sensible – turning left three times is equivalent to turning right once, which would give the same result using fewer moves. Eventually, the GA would discover this and make the change, but we can try to speed it up by looking for three L’s in a row and replacing them with an R. A convenient place to do this is in the develop() method in DozerPlan.java. This method is called immediately after a new solution is created by crossover and mutation. At the moment it does nothing. Change it to do the following:

```
 public void develop()
 {
     if(Math.random() < CLEANUP_PROB)
     {
         char prev = '?';
         char prev2 = '?';
         for(int i = 0; i < moves.length(); i++)
         {
             if(prev2 == 'L' && prev == 'L' && moves.charAt(i) == 'L')
             {
                 moves = moves.substring(0, i-2)
                     + "R"
                     + moves.substring(i+1)
                     + "FF";
                 i -= 2;
             }
             if(i > 0)
             {
                 prev2 = moves.charAt(i-1);
             }
             else
             {
                 prev2 = '?';
             }
             if(i > -1)
             {
                 prev = moves.charAt(i);
             }
             else
             {
                 prev = '?';
             }
         }
     }
}
```

Check that all is well by compiling and testing the application. Did you get any improvement?

Step 7

Likewise, it is not sensible to do three R's in a row. Add another *if* statement to the code to replace three R's with an L. Be careful to add extra moves at the end to keep the string a constant length (these don't affect the score as the score is calculated as the maximum score at any point in the execution of the plan). Compile and test again. Again – did you get an improvement?

Step 8 (harder)

Another possible improvement comes from the observation that an “R” followed by a “L” (or the other way around) is a waste of two moves, and could be removed from the genome. Add code to make this correction (and also remove “L” followed by “R”). Again, be careful to keep the length of the genome constant.

Did this help?

Step 9 (optional)

Can you think of any other ways to improve the performance of the algorithm?

When you have completed this workshop, submit a text or document file containing your answers for steps 6, 7, 8 and 9 to your tutor using the submission facility on BlackBoard.