Brad Pitney

12/5/2013

ECE 578

Perkowski

Homework #2: Depth-First Search for ‘Hands of Time’ Puzzle

# Introduction

For this project, I decided to use depth-first search to solve large instances of the ‘Hands of Time’ mini-game, a type of puzzle that appears within the video game ‘Final Fantasy XIII-2’ (“thirteen two”). This puzzle is essentially a node ordering problem which is presented in the form of a series of integers arranged around an analog clock face. Since the game includes only small instances of this puzzle that are meant to be solvable by hand, I wrote a puzzle generator program to create larger instances. To solve these instances, I created three solvers: a standard depth-first search solver, a solver utilizing an additional pruning optimization, and a solver using both the pruning optimization and a sorting heuristic. I then compared the performance of these solvers using a set of generated puzzle instances.

# Problem Description

The ‘Hands of Time’ puzzle consists of a ring of nodes, each containing an integer value. One node is assigned to be the currently active node. From this node, the player must select a new node, but can only select a node that is a number of spaces away equal to the integer value of the active node. For instance, if the active node shows number ‘3’, the player could move 3 nodes clockwise around the ring, or move 3 nodes counterclockwise. These integer values are never more than half of the total number of nodes in the puzzle, so no single move ever goes more than halfway around the ring. Once the player selects one of these two destination nodes, the selected node becomes the new active node, and the process repeats. A node can only be selected once, so any previously active node becomes an empty node, which cannot be selected during future moves. The player’s objective is to choose the correct sequence of clockwise and counterclockwise moves to select every node in the ring, without reaching a ‘dead end’ where no move is available. If a player moves onto a node that has no valid moves (i.e. both the clockwise and counterclockwise moves land on empty nodes) then the player loses.

The figure below shows an example of a 6-node instance of this puzzle, and the solving process:



The player starts at the top ‘2’ node and can either move two spaces clockwise to the bottom right ‘2’ node, or move counterclockwise to the bottom left ‘1’ node. In this example, the player chooses to move clockwise. At the bottom right ‘2’ node, the player can only move clockwise, since the counterclockwise move would go back to a node that has already been visited, which is not allowed. The player moves clockwise to the ‘1’ node, then counterclockwise to the bottom ‘2’ node, then to the top right ‘2’ node, then to the final ‘3’ node. At this point, the player has visited all nodes and the puzzle has been solved.

The figure below shows the same puzzle instance with a different move sequence:



This time, the player chooses the same first two moves, but on the third move chooses to move from the ‘1’ node to the ‘3’ node, instead of to the bottom ‘2’ node, as before. In this case, the player must now move three spaces, but the node in this location has already been visited (note that both the clockwise and counterclockwise moves go to the same node). The player is out of moves and fails to solve the puzzle with this move sequence.

# Design Process

The puzzle instances used in ‘Final Fantasy XIII-2’ all contain between 5 and 13 nodes, which are small enough that a computer can explore the entire search space in a fraction of a second. In order to create a set of larger, more challenging instances, I created a GeneratePuzzle method in C# and placed this within a class called ‘ClockSolver’, which provides a Windows form interface. This generator works by first creating a list of the consecutive integers, shuffling this list into random order, and inserting the starting node at the beginning of the list. It then steps through each of these integers in sequential order, calculating the number of nodes between each number and the next in the sequence, and populating a new list with these values. In this way, it generates a random puzzle instance of any size, which has at least one solution when starting at the first node. The puzzle list is then written out to a file, where it can be used repeatedly by different solvers.

To solve these puzzle instances, I wrote the SolveInstance method, which loads a puzzle file and performs a depth-first search on the instance. The method starts at the first node and calculates the indices of the next two possible nodes in the sequence. The method then calls itself recursively to check the next node in the sequence, and repeats this process to explore down the search tree. In this simple depth-first search, the search process always explores the ‘clockwise’ node before the ‘counterclockwise’ node. SolveInstance maintains a description of the puzzle state by marking activated nodes with a ‘0’ integer when moving down the tree, and restoring these back to the original value when moving up the tree. At each recursive call, the method checks for this ‘0’ value to determine if this is a valid move, returning up the tree if it is invalid. The method also tracks the depth of the recursive calls, since this corresponds to the number of nodes that have been activated without reaching a dead end. Once the recursion depth reaches the total number of nodes in the puzzle, the method recognizes that it has found a solution and returns immediately by throwing an exception. The exception is caught by the upper layer, and the solution and search time is then reported in a textbox.

One major limitation of this search process is the fact that the algorithm always blindly chooses the ‘clockwise’ node before the ‘counterclockwise’ node, at every step down the tree. To improve this, I started looking for heuristics could be used to make a more intelligent decision when choosing between each pair of nodes. One that came to mind was related to a trick that I had used to solve these puzzles by hand, while playing ‘Final Fantasy XIII-2’. I had found that it was often helpful to step through each node and identify every other node that is able to reach a particular node (i.e. how many “input nodes” are available to a particular node). If only one node in the puzzle is able to reach a particular node, then you know that this path must be present in the solution, otherwise the node would be left out and the solution would be invalid.

So, when choosing between two nodes, if one of the nodes has only one input node, then we know that we must choose this node. Since choosing the other node would result in a failed solution, we can skip this search path entirely and prune this branch of the search tree. To implement this feature, I added some code to SolveInstance to step through each node in the puzzle instance and count the number of input nodes available to this node. In the search routine, I added some logic to check if either of the possible target nodes has only one input node. If this is the case, we search only this node and ignore the other.

An extension of this idea can also be applied to cases where both target nodes have multiple input nodes. In the case, we should prioritize the node that has fewer inputs, since this node has a higher chance of being locked out of the solution (i.e. having no available input nodes), and causing the solution to be incomplete. To implement this heuristic, I used the new input counting code and added new logic to the search routine to choose the node with fewer inputs before exploring the other path. In this case, we can’t prune the other path, since this heuristic won’t be correct all of the time, but it does allow us to do some reasonable node sorting that should result in finding a valid solution in less time. In cases where both nodes have equal input nodes, we always choose the clockwise node first, to match the behavior of the standard solver.

# Results

To evaluate the performance of the standard depth-first search compared with these input node heuristics, I added support code SolveInstance to search in one of three modes:

* **standard mode** – basic depth-first search (DFS).
* **‘with pruning’ mode** – DFS with guaranteed selection of 1-input nodes.
* **‘with pruning and sorting’ mode** – DFS with guaranteed selection of 1-input nodes, and prioritization of nodes with fewer inputs.

I also created a SolveMultipleInstances method to call SolveInstance on multiple puzzle instance files, and added some code to GeneratePuzzle to create multiple puzzle files. I added support for measuring execution time for each search and for both displaying these results in a textbox and writing these to a text file.

With some experimentation, I found that the pruning optimization resulted in a very significant improvement to search time when compared to the standard depth-first search. The sorting heuristic also appeared to improve search time compared to the search with only pruning, although this improvement was much smaller. The following chart shows the total time for each search method to find solutions to the same set of 50 instances of 75-node puzzles:

|  |  |
| --- | --- |
| **Solver Type** | **Time (seconds)** |
| Standard DFS | 1593.4 |
| With Pruning | 12.1 |
| Pruning + Sorting | 11.4 |

To get a better assessment of the impact of the sorting heuristic, I repeated this test with 50 instances of 100-node puzzles. I didn’t include the standard DFS mode, since it would have taken days for the standard DFS to find solutions to these larger puzzle instances. Here are the results:

|  |  |
| --- | --- |
| **Solver Type** | **Time (seconds)** |
| With Pruning | 304.8 |
| Pruning + Sorting | 280.5 |

This indicates that the sorting heuristic improved the average solve time for 100-node instances by about 8%. So, the sorting heuristic has a minor impact, but is still appears to be a successful technique for improving search times.

# Conclusion

Depth-first search was used to solve large instances of the ‘Hands of Time’ puzzle from ‘Final Fantasy XIII-2’. A puzzle generator was designed to easily create multiple puzzles of a selected size. A depth-first search solver was created and then improved using a pruning optimization and a sorting heuristic. These three solvers were compared by testing each against a common set of 50 instances of 75-node and 100-node puzzles. The pruning technique provided a very large improvement in search time compared to the basic solver. The sorting heuristic provided a smaller, but noticeable, improvement in average search time compared to pruning alone.

# Clock Solver interface:



// Brad Pitney

// ECE 578

// Fall 2013

// The ClockSolver class allows for creating and solving instances of the 'Hands of Time' puzzle

// from 'Final Fantasy XIII-2'. It allows for generating multiple instances of N-node puzzles at once,

// and writes these instances to files. It solves multiple instances using depth-first search, with

// optional pruning and sorting heuristics.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

namespace ClockSolver

{

 public partial class clockSolverForm : Form

 {

 public class clockSolutionElement

 {

 public enum directionEnum

 {

 clockwise,

 counterclockwise

 }

 public directionEnum direction = directionEnum.clockwise;

 public int index = 0;

 }

 public enum solverTypeEnum

 {

 standard,

 withPruning,

 withPruningAndSorting

 }

 List<int> m\_puzzleState = new List<int>();

 List<int> m\_numberOfInputNodes = new List<int>();

 List<clockSolutionElement> m\_clockSolution = new List<clockSolutionElement>();

 string m\_solutionFoundString = "Solution Found";

 Random m\_shuffleRand = new Random();

 int m\_numberOfSolutions = 0;

 // Shuffle is based on code from http://stackoverflow.com/questions/273313/randomize-a-listt-in-c-sharp

 public void Shuffle(List<int> list)

 {

 int n = list.Count;

 while (n > 1)

 {

 n--;

 int k = m\_shuffleRand.Next(n + 1);

 int value = list[k];

 list[k] = list[n];

 list[n] = value;

 }

 }

 public clockSolverForm()

 {

 InitializeComponent();

 }

 // This method solves multiple puzzle instances by calling SolveInstance repeatedly. It tracks the time

 // spent solving each instance, displaying this data and recording the data to a results file.

 //

 // numberOfInstances - specifies the puzzle instance files that will be loaded and solved.

 // solverType - specifies if instances will be solved with the standard DFS solver, DFS with pruning

 // optimization, or DFS with both pruning and the sorting heuristic.

 // findAllSolutions - specifies whether the solver should stop when the first solution is found, or

 // continue and find every solution.

 //

 private void SolveMultipleInstances(int numberOfInstances, solverTypeEnum solverType = solverTypeEnum.standard, bool findAllSolutions = false)

 {

 List<double> resultList = new List<double>();

 DateTime startTime = DateTime.Now;

 outputTextbox.Clear();

 multiInstanceOutputTextbox.Clear();

 multiInstanceOutputTextbox.Text += "Solving " + numberOfInstances + " puzzles..." + Environment.NewLine;

 // Solve the specified number of puzzle instances.

 for (int i = 0; i < numberOfInstances; i++)

 {

 multiInstanceOutputTextbox.Text += "Solving puzzle " + (i+1) + " ... ";

 multiInstanceOutputTextbox.Refresh();

 TimeSpan solveDuration = SolveInstance(i+1, solverType, findAllSolutions);

 resultList.Add(solveDuration.TotalSeconds);

 multiInstanceOutputTextbox.Text += "Done in " + solveDuration.TotalSeconds + " seconds." + Environment.NewLine;

 }

 TimeSpan runDuration = DateTime.Now - startTime;

 multiInstanceOutputTextbox.Text += "Done. Total time: " + runDuration.TotalSeconds + " seconds." + Environment.NewLine;

 // Write the solve times out to the results file.

 string resultsFileName;

 if (solverType == solverTypeEnum.withPruning)

 {

 resultsFileName = "WIthPruningResults.txt";

 }

 else if (solverType == solverTypeEnum.withPruningAndSorting)

 {

 resultsFileName = "WithPruneAndSortResults.txt";

 }

 else

 {

 resultsFileName = "StandardResults.txt";

 }

 System.IO.File.WriteAllLines(resultsFileName, resultList.ConvertAll(Convert.ToString));

 multiInstanceOutputTextbox.Text += "Results written to: " + resultsFileName + Environment.NewLine;

 }

 // This method loads a puzzle file and calls an overloaded SolveInstance method to find a solution to this

 // puzzle instance. It also handles counting of input nodes, for solvers using the pruning and sorting heuristics.

 //

 // numberOfInstances - specifies the puzzle instance files that will be loaded and solved.

 // solverType - specifies if instances will be solved with the standard DFS solver, DFS with pruning

 // optimization, or DFS with both pruning and the sorting heuristic.

 // findAllSolutions - specifies whether the solver should stop when the first solution is found, or

 // continue and find every solution.

 //

 private TimeSpan SolveInstance(int instanceNumber = 1, solverTypeEnum solverType = solverTypeEnum.standard, bool findAllSolutions = false)

 {

 m\_puzzleState.Clear();

 m\_clockSolution.Clear();

 m\_numberOfInputNodes.Clear();

 List<int> puzzleStateOutput = new List<int>();

 // Load the specified puzzle instance from its file.

 String fileName = "Puzzle" + instanceNumber + ".txt";

 string[] stringArray = System.IO.File.ReadAllLines(fileName);

 m\_puzzleState = Array.ConvertAll<string, int>(stringArray, new Converter<string, int>(Convert.ToInt32)).ToList();

 // Populate arrays.

 for (int i = 0; i < m\_puzzleState.Count; i++)

 {

 puzzleStateOutput.Add(m\_puzzleState[i]);

 m\_clockSolution.Add(new clockSolutionElement());

 m\_numberOfInputNodes.Add(0);

 }

 m\_numberOfSolutions = 0;

 m\_clockSolution[0].index = 0;

 if (solverType == solverTypeEnum.withPruning)

 {

 outputTextbox.Text += "Running Solver with Pruning" + Environment.NewLine;

 }

 else if (solverType == solverTypeEnum.withPruningAndSorting)

 {

 outputTextbox.Text += "Running Solver with Pruning+Sorting" + Environment.NewLine;

 }

 else

 {

 outputTextbox.Text += "Running Standard Solver" + Environment.NewLine;

 }

 if (findAllSolutions)

 {

 outputTextbox.Text += "Instance: " + fileName + Environment.NewLine + "Finding all solutions..." + Environment.NewLine;

 }

 else

 {

 outputTextbox.Text += "Instance: " + fileName + Environment.NewLine + "Solving..." + Environment.NewLine;

 }

 outputTextbox.Refresh();

 DateTime startTime = DateTime.Now;

 TimeSpan runDuration = TimeSpan.Zero;

 // For each node, count the number of other nodes that can reach this node (i.e. input nodes).

 // This is required for the pruning/sorting heuristics.

 if (solverType != solverTypeEnum.standard)

 {

 for (int i = 0; i < m\_puzzleState.Count; i++)

 {

 int currentValue = m\_puzzleState[i];

 // Find the index of the clockwise and counterclockwise move.

 int newIndex1 = (i + currentValue + m\_puzzleState.Count) % m\_puzzleState.Count;

 int newIndex2 = (i - currentValue + m\_puzzleState.Count) % m\_puzzleState.Count;

 // Increment the counter for the target nodes.

 if (newIndex1 == newIndex2)

 {

 // If both the clockwise and counterclockwise moves point to the same node (i.e. node integer

 // is N/2), then only increment the target node counter once.

 m\_numberOfInputNodes[newIndex1]++;

 }

 else

 {

 m\_numberOfInputNodes[newIndex1]++;

 m\_numberOfInputNodes[newIndex2]++;

 }

 }

 //string inputNodesFileName = "InputNodes" + instanceNumber + ".txt";

 //System.IO.File.WriteAllLines(inputNodesFileName, m\_numberOfInputNodes.ConvertAll(Convert.ToString));

 }

 try

 {

 // Start the search. This call should either throw an exception when a solution is found, or return

 // normally when all solutions are found, depending on the state of 'findAllSolutions'.

 SolveInstance(0, 0, solverType, findAllSolutions);

 if (!findAllSolutions)

 {

 // If we return normally when findAllSolutions is not set, then something went wrong.

 outputTextbox.Text += "No solution found." + Environment.NewLine;

 return runDuration;

 }

 }

 catch (Exception exception)

 {

 if (exception.Message != m\_solutionFoundString)

 {

 outputTextbox.Text += "Unexpected exception: " + exception.Message + Environment.NewLine;

 return runDuration;

 }

 }

 if (findAllSolutions)

 {

 // Display summary of all solutions found.

 runDuration = DateTime.Now - startTime;

 outputTextbox.Text += "Search Time: " + runDuration.TotalSeconds + " seconds." + Environment.NewLine;

 outputTextbox.Text += "Found " + m\_numberOfSolutions + " solutions." + Environment.NewLine;

 }

 else

 {

 // Display details of the solution.

 runDuration = DateTime.Now - startTime;

 outputTextbox.Text += "Search Time: " + runDuration.TotalSeconds + " seconds." + Environment.NewLine;

 string solutionString = "Start at " + puzzleStateOutput[m\_clockSolution[0].index].ToString() + " at index " + m\_clockSolution[0].index.ToString() + Environment.NewLine;

 for (int i = 1; i < m\_clockSolution.Count; i++)

 {

 if (m\_clockSolution[i].direction == clockSolutionElement.directionEnum.clockwise)

 {

 solutionString += "Clockwise to " + puzzleStateOutput[m\_clockSolution[i].index].ToString() + " (index " + m\_clockSolution[i].index.ToString() + ")" + Environment.NewLine;

 }

 else

 {

 solutionString += "Counterclockwise to " + puzzleStateOutput[m\_clockSolution[i].index].ToString() + " (index " + m\_clockSolution[i].index.ToString() + ")" + Environment.NewLine;

 }

 }

 outputTextbox.Text += solutionString;

 }

 return runDuration;

 }

 // This method performs the actual depth-first search by calling itself recursively.

 //

 // index - used for tracking the array index of the currently selected node.

 // depth - used for tracking the current depth of the search (recursion depth).

 // solverType - specifies if instances will be solved with the standard DFS solver, DFS with pruning

 // optimization, or DFS with both pruning and the sorting heuristic.

 // findAllSolutions - specifies whether the solver should stop when the first solution is found, or

 // continue and find every solution.

 //

 private void SolveInstance(int index, int depth, solverTypeEnum solverType, bool findAllSolutions)

 {

 if (m\_puzzleState[index] == 0)

 {

 // This path is a dead-end, so go back. '0' indicates that this node has already been visited.

 return;

 }

 m\_clockSolution[depth].index = index;

 if (depth == m\_puzzleState.Count - 1)

 {

 // All nodes have been visited, so a solution has been found.

 if (findAllSolutions)

 {

 m\_numberOfSolutions++;

 return;

 }

 else

 {

 throw new Exception(m\_solutionFoundString);

 }

 }

 int currentValue = m\_puzzleState[index];

 m\_puzzleState[index] = 0; // Mark the current node as 'visited'.

 // Find the index of the clockwise and counterclockwise move.

 int newIndex1 = (index + currentValue + m\_puzzleState.Count) % m\_puzzleState.Count;

 int newIndex2 = (index - currentValue + m\_puzzleState.Count) % m\_puzzleState.Count;

 if (newIndex1 == newIndex2)

 {

 // Both moves point to the same node (i.e. node integer is N/2).

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 }

 else

 {

 // Apply solver type specific logic.

 if (solverType == solverTypeEnum.withPruning)

 {

 // If 'pruning' is selected, then automatically select any node with a single input node,

 // and skip the other path entirely. I.e. the 1-input node will be left out of the solution

 // if we don't select it, so selecting the other node is guaranteed to cause the solution to fail.

 if (newIndex1 != 0 && m\_numberOfInputNodes[newIndex1] == 1)

 {

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.clockwise;

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 }

 else if (newIndex2 != 0 && m\_numberOfInputNodes[newIndex2] == 1)

 {

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.counterclockwise;

 SolveInstance(newIndex2, depth + 1, solverType, findAllSolutions);

 }

 else

 {

 // Otherwise, always search both paths, starting with the clockwise move.

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.clockwise;

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.counterclockwise;

 SolveInstance(newIndex2, depth + 1, solverType, findAllSolutions);

 }

 }

 else if (solverType == solverTypeEnum.withPruningAndSorting)

 {

 // If 'pruning + sorting' is selected, perform the pruning optimization, but also prioritize

 // the move with fewer input nodes, since the one with fewer inputs is more likely to run out

 // of possible inputs and cause the solution to fail.

 if (newIndex1 != 0 && m\_numberOfInputNodes[newIndex1] == 1)

 {

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.clockwise;

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 }

 else if (newIndex2 != 0 && m\_numberOfInputNodes[newIndex2] == 1)

 {

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.counterclockwise;

 SolveInstance(newIndex2, depth + 1, solverType, findAllSolutions);

 }

 else if (m\_numberOfInputNodes[newIndex1] <= m\_numberOfInputNodes[newIndex2])

 {

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.clockwise;

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.counterclockwise;

 SolveInstance(newIndex2, depth + 1, solverType, findAllSolutions);

 }

 else

 {

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.counterclockwise;

 SolveInstance(newIndex2, depth + 1, solverType, findAllSolutions);

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.clockwise;

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 }

 }

 else

 {

 // If the standard solver is selected, then search both paths, starting with the clockwise move.

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.clockwise;

 SolveInstance(newIndex1, depth + 1, solverType, findAllSolutions);

 m\_clockSolution[depth + 1].direction = clockSolutionElement.directionEnum.counterclockwise;

 SolveInstance(newIndex2, depth + 1, solverType, findAllSolutions);

 }

 }

 // Restore the current node to its actual integer value.

 m\_puzzleState[index] = currentValue;

 }

 // This method generates one or more puzzle instance files with the specified number of nodes.

 //

 // numberOfElements - the number of nodes in each puzzle instance.

 // numberOfPuzzles - the number of puzzle instance files to create.

 //

 void GeneratePuzzle(int numberOfElements = 10, int numberOfPuzzles = 1)

 {

 outputTextbox.Clear();

 List<int> nodeOrderList = new List<int>();

 List<int> puzzleList = new List<int>();

 outputTextbox.Text += "Generating " + numberOfPuzzles + " puzzles with " + numberOfElements + " nodes..." + Environment.NewLine;

 outputTextbox.Refresh();

 for (int puzzleNumber = 0; puzzleNumber < numberOfPuzzles; puzzleNumber++)

 {

 nodeOrderList.Clear();

 puzzleList.Clear();

 // Create an ordered list of numbered nodes.

 for (int i = 1; i < numberOfElements; i++)

 {

 nodeOrderList.Add(i);

 puzzleList.Add(0);

 }

 // Randomize the node list, but insert the start node at the beginning of the list.

 Shuffle(nodeOrderList);

 nodeOrderList.Insert(0, 0);

 puzzleList.Insert(0, 0);

 // Create the puzzle by converting the numbered nodes into a list of integers representing

 // the distance to the next node in the sequence.

 int currentPosition = 0;

 for (int i = 1; i < numberOfElements; i++)

 {

 int nextPosition = nodeOrderList.IndexOf(i);

 puzzleList[currentPosition] = Math.Min(Math.Abs(nextPosition - currentPosition), numberOfElements - Math.Abs(currentPosition - nextPosition));

 currentPosition = nextPosition;

 }

 // Randomize the value of the last node, so that it looks like all the other nodes.

 puzzleList[currentPosition] = m\_shuffleRand.Next(1, numberOfElements / 2);

 // Store the puzzle instance in a file.

 string fileName = "Puzzle" + (puzzleNumber+1) + ".txt";

 System.IO.File.WriteAllLines(fileName, puzzleList.ConvertAll(Convert.ToString));

 //System.IO.File.WriteAllLines("NodeList.txt", nodeOrderList.ConvertAll(Convert.ToString));

 }

 outputTextbox.Text += "Puzzles created." + Environment.NewLine;

 }

 // Button events ////////////////////////////////////////////

 private void generateButton\_Click(object sender, EventArgs e)

 {

 int numberOfElements, numberOfInstances;

 if (!int.TryParse(numberOfNodesTextbox.Text, out numberOfElements))

 {

 numberOfElements = 10;

 //outputTextbox.Text += "Using default number of nodes: " + numberOfElements + Environment.NewLine;

 }

 if (!int.TryParse(numberOfInstancesTextbox.Text, out numberOfInstances))

 {

 numberOfInstances = 1;

 }

 GeneratePuzzle(numberOfElements, numberOfInstances);

 }

 private void solveFileButton\_Click(object sender, EventArgs e)

 {

 int numberOfInstances;

 if (!int.TryParse(numberOfInstancesTextbox.Text, out numberOfInstances))

 {

 numberOfInstances = 1;

 }

 SolveMultipleInstances(numberOfInstances, solverTypeEnum.standard, false);

 }

 private void findAllSolutionsButton\_Click(object sender, EventArgs e)

 {

 int numberOfInstances;

 if (!int.TryParse(numberOfInstancesTextbox.Text, out numberOfInstances))

 {

 numberOfInstances = 1;

 }

 SolveMultipleInstances(numberOfInstances, solverTypeEnum.standard, true);

 }

 private void withPruningButton\_Click(object sender, EventArgs e)

 {

 int numberOfInstances;

 if (!int.TryParse(numberOfInstancesTextbox.Text, out numberOfInstances))

 {

 numberOfInstances = 1;

 }

 SolveMultipleInstances(numberOfInstances, solverTypeEnum.withPruning, false);

 }

 private void withPruningAndSortingButton\_Click(object sender, EventArgs e)

 {

 int numberOfInstances;

 if (!int.TryParse(numberOfInstancesTextbox.Text, out numberOfInstances))

 {

 numberOfInstances = 1;

 }

 SolveMultipleInstances(numberOfInstances, solverTypeEnum.withPruningAndSorting, false);

 }

 }

}