```
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.IO;
5  using System.Linq;
6  using System.Reflection;
7  using System.Text;
8  using System.Threading.Tasks;
9
10 namespace _3P71_2
11 {
12     class GeneticTS
13     {
14         //Params
15         public readonly Random random;
16         public readonly City[] cities;
17         public readonly Program.ElitismMode elietismMode;
18         public readonly Program.CrossoverType crossType;
19         public readonly double crossoverRate;
20         public readonly double mutationRate;
21         public readonly int maxPopSize;
22         public readonly int maxGenerationSpan;
23         public readonly int randomSeed;
24         public readonly int tournamentSize;
25         public readonly int startCityIndex;
26         public readonly bool allowConvergence;
27         public readonly int roundDigits;
28         public readonly string experimentName;
29         List<Tour> tours;
30
31         //Convergence Counters
32         public HashSet<double> foundPaths = new HashSet<double>();
33         public double[,] connectionCosts;
34
35         //Needed variables
36         public readonly int safeZone = 0;
37         Crossover crossover;
38         //Stats
39         double currentBestCost;
40         public List<double> bestFitnesses;
41         public List<double> avgFitnesses;
42
43         public GeneticTS(Program.ElitismMode elietismMode, Program.CrossoverType
             crossType, double crossoverRate,
44             double mutationRate, int maxPopSize, int maxGenerationSpan, int
               randomSeed, int tournamentSize,
45             int startCityIndex, City[] cities, bool allowConvergence, int
               roundDigits, int safeZone)
46         {
47             //Parameters
48             this.elietismMode = elietismMode;
49             this.crossType = crossType;
```

```
50              this.crossoverRate = crossoverRate;
51              this.mutationRate = mutationRate;
52              this.maxPopSize = maxPopSize;
53              this.maxGenerationSpan = maxGenerationSpan;
54              this.randomSeed = randomSeed;
55              this.tournamentSize = tournamentSize;
56              this.startCityIndex = startCityIndex;
57              this.cities = cities;
58              this.allowConvergence = allowConvergence;
59              this.roundDigits = roundDigits;
60              this.safeZone = safeZone;
61              experimentName = string.Format("{0}-{1}-{2}-{3}-{4}({5})",
                  cities.Length, elietismMode, crossoverRate, mutationRate,
                  allowConvergence, randomSeed);
62
63
64              //Setup
65              connectionCosts = new double[cities.GetLength(0) + 1,
                  cities.GetLength(0) + 1];
66              random = new Random(randomSeed);
67              crossover = new Crossover(this);
68              avgFitnesses = new List<double>();
69              bestFitnesses = new List<double>();
70              Console.WriteLine("     Starting " + experimentName);
71              MainLoop();
72          }
73
74          /// <summary>
75          /// Does all the work
76          /// </summary>
77          private void MainLoop()
78          {
79              currentBestCost = int.MaxValue;
80              avgFitnesses = new List<double>();
81              bestFitnesses = new List<double>();
82              foundPaths.Clear();
83              tours = GenerateTours();
84              currentBestCost = tours[0].Cost;
85              Stopwatch stopwatch = Stopwatch.StartNew();
86              int convergenceCounter = 0;
87              for (int i = 0; i < maxGenerationSpan; i++)
88              {
89                  avgFitnesses.Add(tours.Sum(x => x.Cost) / tours.Count);
90                  bestFitnesses.Add(tours[0].Cost);
91
92                  switch (crossType)
93                  {
94                      case Program.CrossoverType.UOX:
95                          tours = crossover.UOXCrossover(tours);
96                          break;
97                      case Program.CrossoverType.PMX:
98                          tours = crossover.PMXCrossover(tours);
```

```
 99                          break;
100                      case Program.CrossoverType UOXPMX:
101                          tours = crossover.UOXCrossover(tours);
102                          tours = crossover.PMXCrossover(tours);
103                          break;
104                  }
105
106                  tours = Mutate(tours);
107
108                  //Convergence check. Progress after this point is basically
                       random, so might as well stop
109                  if (tours[0].Cost == bestFitnesses.Last() || Math.Abs
                       (avgFitnesses.Last() - bestFitnesses.Last()) < 1)
110                  {
111                      convergenceCounter++;
112                      if (convergenceCounter > maxGenerationSpan / 5)
113                      {
114                          Console.WriteLine("         breaking after " + i);
115                          break;
116                      }
117                  }
118                  else
119                  {
120                      convergenceCounter = 0;
121                  }
122              }
123              bestFitnesses = TrimTail(bestFitnesses);
124
125          }
126          /// <summary>
127          /// Generates a string array contaaining all the important information
               about this GeneticTS run
128          /// </summary>
129          /// <returns>Information on this experiment</returns>
130          public string[] GetExperimentInfo()
131          {
132              return new string[]
133              {
134                  (",,Data set: " + cities.Length),
135                  (",,startCityIndex: " + startCityIndex),
136                  (",,crossoverRate: " + crossoverRate),
137                  (",,mutationRate: " + mutationRate),
138                  (",,maxGenerationSpan: " + maxGenerationSpan),
139                  (",,maxPopSize: " + maxPopSize),
140                  (",,Tournement Size: " + tournamentSize),
141                  (",,Seed: \"" + randomSeed + "\""),
142                  (",,Final Path Cost: " + tours[0].Cost),
143                  (",,Tour: " + PathToString(tours[0].Path)),
144              };
145          }
146          /// <summary>
147          /// Trims the tail of a list of doubles so that the final value, if it
```

```
            repeats, repeats no more than list.Count/20 times
148         /// </summary>
149         /// <param name="list"> list to trim</param>
150         /// <returns>trimd list</returns>
151         private List<double> TrimTail(List<double> list)
152         {
153             int firstIndexOf = list.IndexOf(list.Last()) + 1 + (list.Count / 20);
154             if (firstIndexOf < list.Count - 1)
155             {
156                 list.RemoveRange(firstIndexOf, list.Count - firstIndexOf);
157             }
158             return list;
159         }
160         /// <summary>
161         /// Mutates the a number of random tours in the given list.
162         /// Mutation is done through random swapping.
163         /// mutateCount is calculated to dramatically redice the number of
            Math.Random calls that would be needed.
164         /// On average, it will do the same number of mutations as just doing
            Math.Random.Next(2) < mutatuinRate
165         /// </summary>
166         /// <param name="tourList"></param>
167         /// <returns></returns>
168         private List<Tour> Mutate(List<Tour> tourList)
169         {
170             int mutateCount = (int)(mutationRate * tourList.Count) / 2;
171             for (int i = 0; i < mutateCount; i++)
172             {
173                 int parentIndex = TournementSelect(tourList);
174                 int[] child = (int[])tourList[parentIndex].Path.Clone();
175
176                 //mutate ( swap )
177                 int swapIndexA = random.Next(1, child.Length - 1);
178                 int swapIndexB = random.Next(1, child.Length - 1);
179                 int tempVal = child[swapIndexA];
180                 child[swapIndexA] = child[swapIndexB];
181                 child[swapIndexB] = tempVal;
182
183                 tourList = AddChild(tourList, child, parentIndex);
184             }
185             return Prioritize(tourList);
186         }
187         /// <summary>
188         /// Generates tour paths randomly
189         /// </summary>
190         /// <returns>List of randomly generated tours</returns>
191         private List<Tour> GenerateTours()
192         {
193             tours = new List<Tour>();
194             int[] curPath = new int[cities.Length];
195             double cost;
196             for (int i = 0; i < curPath.Length; i++)
```

```
197             {
198                 curPath[i] = i + 1;
199             }
200
201             for (int i = 0; i < maxPopSize; i++)
202             {
203                 curPath = Shuffle(curPath);
204                 cost = CalcTourCost(curPath);
205                 if (foundPaths.Add(cost))
206                 {
207                     tours.Add(new Tour(curPath, cost));
208                 }
209             }
210
211             return Prioritize(tours);
212         }
213         /// <summary>
214         /// Sorts list of tours from min -> max based on tour cost.
215         /// Also prune's list to ensure there are no more than maxPopSize tours
216         /// </summary>
217         /// <param name="tourList">tourList to sort</param>
218         /// <returns>Sorted and potentially shrunk list</returns>
219         public List<Tour> Prioritize(List<Tour> tourList)
220         {
221             List<Tour> tours = new List<Tour>(tourList.OrderBy(tour =>
                  tour.Cost));
222             if (tourList.Count > maxPopSize)
223             {
224                 tours.RemoveRange(maxPopSize, (tours.Count - maxPopSize));
225             }
226             return tours;
227         }
228         /// <summary>
229         /// Converts an array of integers into a single string with arrows
230         /// </summary>
231         /// <param name="path">integer array representing path</param>
232         /// <returns>Path in string form</returns>
233         private string PathToString(int[] path)
234         {
235             string output = "";
236             for (int i = 0; i < path.Length - 1; i++)
237             {
238                 output += path[i] + " -> ";
239                 CostBetween(path[i], path[i + 1]);
240             }
241             output += path[path.Length - 1];
242             return output;
243         }
244
245         /// <summary>
246         /// Iterates through given path and calculates its overall cost
247         /// </summary>
```

```
248         /// <param name="path">Path to iterate</param>
249         /// <returns>cost of entire tour</returns>
250         public double CalcTourCost(int[] path)
251         {
252             double cost = 0;
253             for (int i = 0; i < path.Length - 1; i++)
254             {
255                 cost += CostBetween(path[i], path[i + 1]);
256             }
257             return cost;
258         }
259         /// <summary>
260         /// Calculates the cost between two cities, given their indexes
261         /// </summary>
262         /// <param name="from">first city</param>
263         /// <param name="to"> second city</param>
264         /// <returns>cost between cities</returns>
265         double CostBetween(int from, int to)
266         {
267             double val = connectionCosts[from, to];
268             if (val == 0)
269             {
270                 val = Math.Sqrt(Math.Pow(cities[from - 1].x - cities[to - 1].x,
                      2) + Math.Pow(cities[from - 1].y - cities[to - 1].y, 2));
271                 connectionCosts[from, to] = val;
272                 connectionCosts[to, from] = val;
273             }
274             return val;
275         }
276         /// <summary>
277         /// Knuff shuffle of a given array of integers
278         /// </summary>
279         /// <param name="cities">array of integers, in order from 1 to
              cities.length - 1</param>
280         /// <returns> shuffled array</returns>
281         private int[] Shuffle(int[] cities)
282         {
283             List<int> output = new List<int>();
284             for (int i = 0; i < cities.Length; i++)
285             {
286                 if (cities[i] != startCityIndex)
287                 {
288                     output.Add(cities[i]);
289                 }
290             }
291 
292             for (int i = output.Count; i > 1; i--)
293             {
294                 int k = random.Next(i);
295                 int value = output[k];
296                 output[k] = output[i - 1];
297                 output[i - 1] = value;
```

```
298             }
299
300             output.Insert(0, startCityIndex);
301             output.Add(startCityIndex);
302
303             return output.ToArray();
304         }
305         /// <summary>
306         /// Selects a tour through tournement selection. Size of tournement is
              set in initialization
307         /// </summary>
308         /// <param name="tourList">List to choose from</param>
309         /// <returns>index of winner of tournement</returns>
310         public int TournementSelect(List<Tour> tourList)
311         {
312             int bestIndex = random.Next(0, tourList.Count - 1);
313             for (int i = 0; i < tournamentSize - 1; i++)
314             {
315                 int ranIndex = random.Next(0, tourList.Count - 1);
316                 if (tourList[bestIndex].Cost > tourList[ranIndex].Cost)
317                 {
318                     bestIndex = ranIndex;
319                 }
320             }
321             return bestIndex;
322         }
323         /// <summary>
324         /// Adds given child to given List, and if it's parent is not protected
              by current elitism mode, deletes given parent
325         /// </summary>
326         /// <param name="tourList">list to append</param>
327         /// <param name="ch">child</param>
328         /// <param name="parentIndex">index of parent in tourList</param>
329         /// <returns>new tourList</returns>
330         public List<Tour> AddChild(List<Tour> tourList, int[] ch, int
              parentIndex)
331         {
332             double cost = Math.Round(CalcTourCost(ch), roundDigits); ;
333             if (allowConvergence || foundPaths.Add(cost))
334             {
335                 if (parentIndex > safeZone)
336                 {
337                     tourList.RemoveAt(parentIndex);
338                 }
339                 tourList.Add(new Tour(ch, cost));
340             }
341             return tourList;
342         }
343
344     }
345 }
```