|  |
| --- |
|  |
|  |
| |  | | --- | |  | | МИНОБРНАУКИ РОССИИ | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА - Российский технологический университет****"**  **РТУ МИРЭА** | |  | |  | |
| Институт информационных технологий (ИТ) |
| Кафедра инструментального и прикладного программного обеспечения (ИиППО) |

|  |
| --- |
| **ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ 3** |
| **по дисциплине** |
| **«Программное обеспечение интеллектуальных систем**»  *Генетический алгоритм* |

|  |  |
| --- | --- |
|  |  |
| Выполнил студент группы ИКБО-02-17 | *Травин М.Б.* |
| Приняла | *Зорина Н.В.* |

|  |  |  |
| --- | --- | --- |
| Работа выполнена | «\_\_\_» \_\_\_\_\_\_\_\_2020 |  |
|  |  |  |
| «Зачтено» | «\_\_\_» \_\_\_\_\_\_\_\_2020 |  |

**Цель:** Ознакомиться с работой генетических алгоритмов на примере

создания приложений.

**Задание:**

В рамках лабораторной работы попробуйте решить одну из следующих

задач:

1. Тренировка нейросети;

2. Решение NP-полной задачи.

**Ход работы**

Данный генетическим алгоритм должен из начального набора точек найти оптимальный путь среди предложенных.

Что он делает? Он берёт и формирует начальные пути из набора точек, далее после успешного формирования пути проверяет его на оптимальность, основываясь на оптимальности опыта составления предыдущего пути, после чего из всех наборов точек выбирается оптимальный путь и предлагается как решение нп-полной задачи.

***Формализуем задачу со случайной строкой***

Входные данные: точки и длинна между ними

Выходные данные: набор длин путей между точек и общая длинна пути

Под мутацией пути S мы понимаем замену одного случайно выбранного символа из пути S на другой произвольный путь. В данной задаче мы используем только точки и расстояния между ними.

Изначальная популяция – случайный набор точек с указанной длинной между ними.

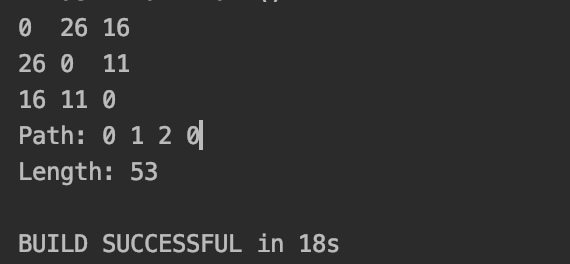
Потомки. Пусть мы зафиксировали количество мутаций одного пути на константу k1, тогда потомки — это k1 мутаций каждого пути текущего поколения.

Выжившие. Пусть мы зафиксировали размер популяции на константу k2, тогда выжившие — это k2 точек максимально похожих на оптимальный путь S.

Исходный код

1. **import** java.util.ArrayList;
2. **import** java.util.Collections;
3. **import** java.util.List;
4. **import** java.util.Random;
6. **public** **class** FitnessSalesman {
8. **private** **int** generationSize;
9. **private** **int** genomeSize;
10. **private** **int** numberOfCities;
11. **private** **int** reproductionSize;
12. **private** **int** maxIterations;
13. **private** **float** mutationRate;
14. **private** **int** tournamentSize;
15. **private** SelectionType selectionType;
16. **private** **int**[][] travelPrices;
17. **private** **int** startingCity;
18. **private** **int** targetFitness;
20. **public** FitnessSalesman(**int** numberOfCities, SelectionType selectionType, **int**[][] travelPrices, **int** startingCity, **int** targetFitness){
21. **this**.numberOfCities = numberOfCities;
22. **this**.genomeSize = numberOfCities-1;
23. **this**.selectionType = selectionType;
24. **this**.travelPrices = travelPrices;
25. **this**.startingCity = startingCity;
26. **this**.targetFitness = targetFitness;
28. generationSize = 5000;
29. reproductionSize = 200;
30. maxIterations = 1000;
31. mutationRate = 0.1f;
32. tournamentSize = 40;
33. }
35. **public** List<SalesmanGenome> initialPopulation(){
36. List<SalesmanGenome> population = **new** ArrayList<>();
37. **for**(**int** i=0; i<generationSize; i++){
38. population.add(**new** SalesmanGenome(numberOfCities, travelPrices, startingCity));
39. }
40. **return** population;
41. }
43. **public** List<SalesmanGenome> selection(List<SalesmanGenome> population){
44. List<SalesmanGenome> selected = **new** ArrayList<>();
45. SalesmanGenome winner;
46. **for**(**int** i=0; i<reproductionSize; i++){
47. **if**(selectionType == SelectionType.ROULETTE){
48. selected.add(rouletteSelection(population));
49. }
50. **else** **if**(selectionType == SelectionType.TOURNAMENT){
51. selected.add(tournamentSelection(population));
52. }
53. }
55. **return** selected;
56. }
58. **public** SalesmanGenome rouletteSelection(List<SalesmanGenome> population){
59. **int** totalFitness = population.stream().map(SalesmanGenome::getFitness).mapToInt(Integer::intValue).sum();
60. Random random = **new** Random();
61. **int** selectedValue = random.nextInt(totalFitness);
62. **float** recValue = (**float**) 1/selectedValue;
63. **float** currentSum = 0;
64. **for**(SalesmanGenome genome : population){
65. currentSum += (**float**) 1/genome.getFitness();
66. **if**(currentSum>=recValue){
67. **return** genome;
68. }
69. }
70. **int** selectRandom = random.nextInt(generationSize);
71. **return** population.get(selectRandom);
72. }
74. **public** **static** <E> List<E> pickNRandomElements(List<E> list, **int** n) {
75. Random r = **new** Random();
76. **int** length = list.size();
78. **if** (length < n) **return** **null**;
80. **for** (**int** i = length - 1; i >= length - n; --i)
81. {
82. Collections.swap(list, i , r.nextInt(i + 1));
83. }
84. **return** list.subList(length - n, length);
85. }
87. **public** SalesmanGenome tournamentSelection(List<SalesmanGenome> population){
88. List<SalesmanGenome> selected = pickNRandomElements(population,tournamentSize);
89. **return** Collections.min(selected);
90. }
92. **public** SalesmanGenome mutate(SalesmanGenome salesman){
93. Random random = **new** Random();
94. **float** mutate = random.nextFloat();
95. **if**(mutate<mutationRate) {
96. List<Integer> genome = salesman.getGenome();
97. Collections.swap(genome, random.nextInt(genomeSize), random.nextInt(genomeSize));
98. **return** **new** SalesmanGenome(genome, numberOfCities, travelPrices, startingCity);
99. }
100. **return** salesman;
101. }
103. **public** List<SalesmanGenome> createGeneration(List<SalesmanGenome> population){
104. List<SalesmanGenome> generation = **new** ArrayList<>();
105. **int** currentGenerationSize = 0;
106. **while**(currentGenerationSize < generationSize){
107. List<SalesmanGenome> parents = pickNRandomElements(population,2);
108. List<SalesmanGenome> children = crossover(parents);
109. children.set(0, mutate(children.get(0)));
110. children.set(1, mutate(children.get(1)));
111. generation.addAll(children);
112. currentGenerationSize+=2;
113. }
114. **return** generation;
115. }
117. **public** List<SalesmanGenome> crossover(List<SalesmanGenome> parents){
118. *// housekeeping*
119. Random random = **new** Random();
120. **int** breakpoint = random.nextInt(genomeSize);
121. List<SalesmanGenome> children = **new** ArrayList<>();
123. *// copy parental genomes - we copy so we wouldn't modify in case they were*
124. *// chosen to participate in crossover multiple times*
125. List<Integer> parent1Genome = **new** ArrayList<>(parents.get(0).getGenome());
126. List<Integer> parent2Genome = **new** ArrayList<>(parents.get(1).getGenome());
128. *// creating child 1*
129. **for**(**int** i = 0; i<breakpoint; i++){
130. **int** newVal;
131. newVal = parent2Genome.get(i);
132. Collections.swap(parent1Genome,parent1Genome.indexOf(newVal),i);
133. }
134. children.add(**new** SalesmanGenome(parent1Genome,numberOfCities,travelPrices,startingCity));
135. parent1Genome = parents.get(0).getGenome(); *// reseting the edited parent*
137. *// creating child 2*
138. **for**(**int** i = breakpoint; i<genomeSize; i++){
139. **int** newVal = parent1Genome.get(i);
140. Collections.swap(parent2Genome,parent2Genome.indexOf(newVal),i);
141. }
142. children.add(**new** SalesmanGenome(parent2Genome,numberOfCities,travelPrices,startingCity));
144. **return** children;
145. }
147. **public** SalesmanGenome optimize(){
148. List<SalesmanGenome> population = initialPopulation();
149. SalesmanGenome globalBestGenome = population.get(0);
150. **for**(**int** i=0; i<maxIterations; i++){
151. List<SalesmanGenome> selected = selection(population);
152. population = createGeneration(selected);
153. globalBestGenome = Collections.min(population);
154. **if**(globalBestGenome.getFitness() < targetFitness)
155. **break**;
156. }
157. **return** globalBestGenome;
158. }
160. **public** **void** printGeneration(List<SalesmanGenome> generation ){
161. **for**( SalesmanGenome genome : generation){
162. System.out.println(genome);
163. }
164. }
165. }

***Результаты работы программы***



Видно, что за 4 итераций нам удалось добиться оптимального пути.

**Вывод**

В ходе выполнения лабораторной работы мы ознакомились с работой генетического алгоритма, а также применили свои знания на практике.

Ссылка на GitHub: <https://github.com/SailentzZz/POIS-lab3-SalesmanTravelerProblem>

**Список литературы**

1. Себастьян Рашка. Python и машинное обучение / - М.: Изд-во Вильямс, 2019

2. Петер Флах. Машинное обучение. Наука и искусство построения алгоритмов, которые извлекают знания из данных / - М.: Изд-во ДМК Пресс, 2019

3. Вирсански Эйял. Генетический алгоритм на Python / - М.: Изд-во ДМК Пресс, 2020

4. Леонид Гладков. Генетические алгоритмы / - М.: Отдельное издание 2010

5. Материалы лекций