Projektowanie efektywnych algorytmów Projekt
Etap 3

Marcin Kapuściński 248910

Luty 2021

1 Algorytm genetyczny

Algorytm genetyczny należy do zbioru algorytmów ewolucyjnych, heurystyk naśladujących selekcję naturalną. Opiera się na symulowaniu eliminacji osobników populacji przez środowisko(model Sukcesji), selekcji seksualnej(Selekcja) oraz losowych mutacji w potomstwie(Mutacja).

Konieczne jest zdefiniowanie sposobu reprezentacji osobnika, funkcji dostosowania reprezentującej przystosowanie osobnika do środowiska oraz warunku zatrzymania. Każdy osobnik(jego cechy) reprezentowany jest jako genotyp. Genotyp składa się z chromosomów, które z kolei składają się z genów. W przypadku prezentowanej implementacji algorytmu dla problemu TSP genotyp to cykl, chromosomy to kolejne miasta. Funkcja dostosowania to odwrotność kosztu cyklu. Warunek zatrzymania dla tej implementacji to przeminięcie 1 minuty.

Algorytm genetyczny swoje działanie opiera głównie na procesach występujących w przyrodzie, uproszczonych do:

1. Selekcja - proces wyboru osobników do grupy macierzystej, rozmnażającej się. Wprowadza presję ewolucyjną, właściwość sprawiającą że algorytm faworyzuje osobniki o większym stopniu przystosowania, częściej je rozmnażając i/lub przenosząc do następnej generacji(zależy od Sukcesji). Powszechne są różne modele selekcji, np. ruletkowa, turniejowa, akceptacji stochastycznej. Niektóre implementacje pozwalają wpływać na poziom presji ewolucyjnej, przykładem jest rozmiar turniejów.

2. Krzyżowanie - proces generowania potomstwa dla pary rodziców wybranych w procesie selekcji. Generuje dwóch osobników potomnych. Zachodzi z pewnym prawdopodobieństwem. Zamierzeniem jest stworzenie nowych osobników dziedziczących dobre cechy, przystosowanie, po rodzicach. Takie działanie skutkuje eksploatacją otoczenia dobrze przystosowanych osobników, intensyfikacją poszukiwań. Stosuje się wiele implementacji krzyżowania, w różnym stopniu dziedziczące cechy rodziców, przykłady to PMX (*partially mapped crossover*), OX (*order crossover*), CX (*cycle crossover*).

3. Mutacja - operacja zmienienia genotypu osobnika operatorem mutacji. Zachodzi z pewnym prawdopodobieństwem. W rozumieniu ewolucyjnym, ma na celu urozmaicenie populacji w przypadku dominacji małej grupy bardzo przystosowanych osobników. Praktycznie, zwiększa eksplorację pomagając uciec z optimów lokalnych, przeciwdziałając zbieżności algorytmu.

4. Ocena - obliczenie poziomu przystosowania osobników, obliczenie wartości funkcji przystosowania.

5. Sukcesja - określa sposób generacji nowej populacji z pokolenia na pokolenie, z każdą iteracją głównej pętli algorytmu. Najczęściej wybiera się część potomstwa i część starej populacji, w różnych proporcjach, aby przeciwdziałać zbieżności. Główne modele sukcesji to inkrementacyjny, pokoleniowy i ich pochodne.

Dane wejściowe algorytmu to rozmiar populacji, szansa na krzyżowanie, szansa na mutację jednego osobnika.

2 Implementacja

Implementacja algorytmu dla problemu TSP miała miejsce w języku `C++`. Reprezentacja osobnika to ścieżka którą uznaje się za cykl - zawiera n miast, bez powtórzeń. Jako że instancje problemu to grafy pełne, wystarczy powielić pierwszy element ścieżki na jej końcu aby uzyskać cykl. Wszystkie ścieżki generowane w ramach pierwszej generacji zaczynają się od tego samego miasta, aby uniknąć tożsamych osobników. Wszelkie modyfikacje osobników zachowują tą własność. W wielu miejscach algorytmu ma miejsce losowanie, wykorzystywany jest rozkład jednorodny.

2.1 Operator krzyżowania

Zaimplementowano operator krzyżowania PMX. Operator losuje indeksy początkowy i końcowy podciągów rodziców. Z rodziców podciągi przepisywane są do dwójki dzieci, na adekwatne indeksy, tworząc mapowanie między dziećmi/rodzicami na przedziale wylosowanych indeksów (linia 173.). Reszta chromosomów przepisywana jest z drugiego rodzica (względem rodzica z którego jest podciąg)(linia 178.). Jeżeli podczas przepisywania powtórzy się miasto względem przepisanego wcześniej podciągu, uzyskuje się miasto zastępcze z mapy, dopóki wyeliminuje się konflikt (pętla w linii 185.). Procedura powtarzana jest dla drugiego dziecka. Maksymalna długość mapy to połowa długości ścieżki/połowa chromosomów, wartość indeksu startowego nie może przekroczyć połowy długości ścieżki, jest to błąd implementacji.

Złożoność obliczeniowa funkcji `pmx()` zależy od najdroższej pętli w liniach 178. i 191. Dla mapy o długości d pętla w linii 185. wykona się maksymalnie $2d-1$ razy, wywołując znalezienie (`indexOf()`) wprowadzanej wartości, aby sprawdzić czy występuje powtórzenie w mapie (sprawdzana jest ścieżka od początku, wystarczyłoby sprawdzać od początkowego indeksu mapy - błąd). Złożoność przeszukiwania liniowego to $O(n)$, za zamortyzowaną złożoność można przyjąć $\frac{n}{2}$, pomijając specyfikę szukania powtórzenia między konkretnymi indeksami. Sumaryczna złożoność szukania powtórzeń to $(2d-1) * \frac{n}{2}$, gdzie d wynosi maksymalnie $\frac{n}{2}$, stąd złożoność obliczeniowa funkcji `pmx()` $O(n^2)$.

```
161  pair<vector<short>, vector<short>> Genetic::pmx(vector<short> &p1, vector<short> &p2){
162      uniform_real_distribution<float> urd(0, p1.size()/2.0f);
163      //ooooooo|index1ooooooo|index2ooooooo
164
165      int index1 = (int)urd(this->generator);
166      int index2 = (urd(this->generator) + index1);
167
168      vector<short> child1;
169      child1.resize(p1.size(), -1);
170      vector<short> child2;
171      child2.resize(p1.size(), -1);
172
173      for(int it = index1; it < index2; ++it){
174          child1[it] = p2[it];
175          child2[it] = p1[it];
176      }
177
178      for(int it = 0; it < child1.size(); ++it){
179          if(child1[it] != -1){//obszar mapy
180              continue;
181          }
182
183          int pickedVal = p1[it];
184          int valIndex;
185          while( (valIndex = indexOf(child1, pickedVal)) != -1){
186              pickedVal = p1[valIndex];
187          }
188          child1[it] = pickedVal;
189      }
190
191      for(int it = 0; it < child2.size(); ++it){
192          if(child2[it] != -1){//obszar mapy
193              continue;
194          }
195
196          int pickedVal = p2[it];
197          int valIndex;
198          while( (valIndex = indexOf(child2, pickedVal)) != -1){
199              pickedVal = p2[valIndex];
200          }
201          child2[it] = pickedVal;
202      }
203
204      return pair<vector<short>, vector<short>>(child1, child2);
205  }
```

Rysunek 1: Metoda pmx()

2.2 Operator mutacji

Zaimplementowano jeden operator mutacji - inwersja części genotypu osobnika. Indeksy początku i końca inwersji są losowane. Maksymalna długość odwracanego ciągu chromosomów to połowa długości ścieżki. Maksymalna długość odwracanego ciągu to połowa długości ścieżki/połowa chromosomów, wartość indeksu startowego nie może przekroczyć połowy długości ścieżki, jest to niezamierzona właściwość.

Złożoność operacji `reverse()` to dokładnie $\frac{last-first}{2}$, dla *last* i *first* jako indeksy odpowiednio ostatniego i pierwszego elementu odwracanego ciągu. Średnia długość ciągu do odwrócenia to około $\frac{n}{2}$, stąd złożoność obliczeniowa funkcji `inversion()` $O(n)$.

```
 97  void Genetic::inversion(vector<short> &path) {
 98      uniform_real_distribution<float> urd(1, path.size()/2.0f);
 99      float pos1 = urd(this->generator);
100      float pos2 = urd(this->generator);
101
102      reverse(path.begin() + round(pos1), path.begin() + round(pos1 + pos2));
103  }
```

Rysunek 2: Metoda inversion()

2.3 Funkcja przystosowania

Funkcja przystosowania (276.) to odwrotność kosztu ścieżki. Złożoność funkcji `getFitness()` i `getPathCost()` to $O(n)$.

```
276 double Genetic::getFitness(short** adjMatrix, vector<short>& path){
277     return 1.0/getPathCost(adjMatrix, path);
278 }
279
280 int Genetic::getPathCost(short** adjMatrix, vector<short>& path){
281     int cost = 0;
282     for(short it = 1; it < path.size(); ++it){
283         cost += adjMatrix[path[it-1]][path[it]];
284     }
285     cost += adjMatrix[path[path.size()-1]][path[0]];//od ostatniego do pierwszego
286     return  cost;
287 }
288
```

Rysunek 3: Metody getFitness() i getPathCost()

2.4 Populacja początkowa

Początkowa populacja generowana jest przy pomocy algorytmu *best first search*, z n ścieżkami zaczynającymi się w różnych miastach, przesuwanymi do tego samego miasta początkowego aby uniknąć tożsamych osobników. Resztę brakujących osobników do zapełnienia populacji generuje się losowo, też przesuwając do tego samego miasta początkowego. Przy generacji liczone jest przystosowanie każdego z inicjalizujących osobników, uwzględniane do najlepszego znalezionego rozwiązania. Używana w tym celu jest funkcja `reportNewFitness()`, która zapisuje ewentualne poprawy najlepszego znalezionego rozwiązania i raportuje na terminal.

```
123 void Genetic::initPopulation(short** adjMatrix, int populationSize, int n){
124     vector<short> generatedPath;
125     int pathCost = -1;
126     for(int it = 0; it < n && it < populationSize; ++it){//generacja bfs startujac w kazdym z miast w <0 ... n-1>
127         generatedPath = genPathBFS(adjMatrix, n, it);
128         shiftPathTo(generatedPath, 0);
129         pathCost = getPathCost(adjMatrix, generatedPath);
130         this->population.push_back(pair<vector<short>, double>(generatedPath, reportNewFitness(generatedPath, 1.0/pathCost, pathCost)));
131     }
132
133     while(this->population.size() < populationSize){//generacja losowych, przesuwanych w lewo do startowego miasta 0
134         generatedPath = genPathRand(n);
135         shiftPathTo(generatedPath, 0);
136         pathCost = getPathCost(adjMatrix, generatedPath);
137         this->population.push_back(pair<vector<short>, double>(generatedPath, reportNewFitness(generatedPath, 1.0/pathCost, pathCost)));
138     }
139 }
```

Rysunek 4: Metoda initPopulation()

Złożoność `genPathBFS()` to $O((n-1)*n) \approx O(n^2)$.

```
25  vector<short> Genetic::genPathBFS(short** adjMatrix, short n, short start){
26      vector<short> path = vector<short>();
27      path.reserve(n);
28      vector<bool> visited = vector<bool>();
29      visited.resize(n, false);
30      int currCity = start;
31      visited[currCity] = true;
32      path.push_back(currCity);
33      short closestCurrentCityIndex = 0;
34      short closestCurrentCityDist = SHRT_MAX;
35
36      for(short pathLength = 1; pathLength < n; ++pathLength){
37          closestCurrentCityDist = SHRT_MAX;
38          for(short dest = 0; dest < n; ++dest){
39              if(adjMatrix[currCity][dest] < closestCurrentCityDist && visited[dest] == false){
40                  closestCurrentCityDist = adjMatrix[currCity][dest];
41                  closestCurrentCityIndex = dest;
42              }
43          }
44          currCity = closestCurrentCityIndex;
45          visited[currCity] = true;
46          path.push_back(currCity);
47      }
48      //path.push_back(start);
49      return path;
50  }
```

Rysunek 5: Metoda genPathBFS()

Złożoność `shiftPathTo()` to $O(n + (n-1) * (n-1)) \approx O(n^2)$, biorąc pod uwagę pierwszą bezużyteczną pętlę - błąd implementacji, pozostaje mieć nadzieję, że optymalizacja 3 poziomu pomoże zniwelować koszt nieuwagi. Zamortyzowana złożoność metody `shiftPathTo()` to $O(n + (n-1) * \frac{n-1}{2})$, czyli wciąż $O(n^2)$.

```
52  void shiftPathTo(vector<short> &path, int startingVal){
53      int startingValIndex = 0;
54      int n = path.size();
55      for (int it = 0; it < n; ++it){
56          if (path[it] == startingVal){
57              startingValIndex = it;
58              break;
59          }
60      }
61      short currFirstEl = -1;
62      while(path[0] != startingVal){
63          currFirstEl = path[0];
64          for(int it = 0; it < n-1; ++it){
65              path[it] = path[it + 1];
66          }
67          path[n-1] = currFirstEl;
68      }
69  }
```

Rysunek 6: Metoda shiftPathTo()

Złożoność generacji losowej ścieżki w metodzie `genPathRand()` to $O(n + n) \approx O(n)$, jako że złożoność operacji `shuffle` jest liniowa względem ilość elementów.

```
71  vector<short> Genetic::genPathRand(short n){
72      vector<short> path;
73      //path.reserve(n + 1);
74      path.reserve(n);
75      for(short it = 0; it < n; ++it) {
76          path.push_back(it);
77      }
78
79      //std::mt19937 generator(std::chrono::high_resolution_clo
80      //random_device rd;
81      //std::mt19937 generator(this->rd());
82      std::shuffle(path.begin(), path.end(), this->generator);
83      //path.push_back(path[0]);
84
85      return path;
86  }
```

Rysunek 7: Metoda genPathRand()

Ostatecznie, najdroższymi operacjami w metodzie `initPopulation()` są generacje ścieżek `genPathBFS()` i `genPathRand()`, obie o złożoności obliczeniowej $O(n^2)$. Dla rozmiaru populacji mniejszego niż wielkość instancji problemu n druga pętla nie zostanie wykonana nawet raz. Rozważając dwa przypadki:

- Rozmiar populacji $<= n$: $O(rozmiarPopulacji * n^2)$
- Rozmiar populacji $> n$: $O(n*n^2+(rozmiarPopulacji-n)*n^2) \approx O(n^3+rozmiarPopulacji*n^2-(n^3)) \approx O(rozmiarPopulacji * n^2)$

Złożoność obliczeniowa funkcji `initPopulation()` to $O(rozmiarPopulacji * n^2)$.

2.5 Główna pętla algorytmu

Na początku poniższej metody inicjalizowana jest startowa populacja (210.) oraz definiowany jest obiekt służący jako 60-sekundowy stoper (211.), populacja reprezentowana jest jako niesortowana lista par osobnik - przystosowanie. Następnie, już w głównej pętli algorytmu, definiowane są rozkłady jednorodne dla selekcji (217.) oraz krzyżowania i mutacji (218.). Zastosowano model selekcji w postaci "stochastycznej akceptacji", stąd zakres losowania rozkładu jednorodnego w każdej iteracji algorytmu to od 0 do najlepszego znalezionego rozwiązania, jego współczynnika przystosowania. Podobnie jak przy selekcji ruletkowej, ta metoda daje słabo przystosowanym osobnikom szansę na przejście selekcji i faworyzuje dobrze przystosowanych osobników. W odróżnieniu od selekcji ruletkowej, nie wymaga zapamiętywania posortowanych współczynników przystosowania ani odnajdywania osobnika po losowaniu, znacznie przyspiesza to działanie algorytmu, wystarczy n losowań. Spodziewana złożoność obliczeniowa selekcji to $O(n)$.

Następnie ma miejsce faktyczna selekcja. Pętla w linii 228. rozpatruje każdego osobnika, jeżeli jego przystosowanie jest większe od wylosowanej wartości, uznawany jest za część populacji macierzystej. Natychmiast weryfikowana jest szansa na krzyżowanie dla danego osobnika, analogicznie jak dla selekcji (232.). Jeżeli osobnik się krzyżuje, dodawany jest do bieżącej pary (233.). Jeżeli w parze nie ma jeszcze pierwszego osobnika, aktualnie rozpatrywany osobnik się nim staje i zostaje zapamiętany (261.). Jeżeli w parze znajduje się już pierwszy osobnik, następuje krzyżowanie z obecnym (234.).

Mutacja (237., 249.) lub jej brak ma miejsce osobno dla każdego z dwóch potomków. Sprawdzenie szansy na mutację jest analogiczne do spełniania selekcji i krzyżowania. Niezależnie czy mutacja przebiegła, sprawdzane jest czy przystosowanie potomka jest większe od rodzica z którego otrzymał chromosomy poza mapą (244., 256.). Jeżeli potomek jest lepiej przystosowany, zastępuje rodzica w populacji (245., 257.). Taki zabieg przebiega dla obu potomków i ich rodziców.

```
207 vector<short> Genetic::findShortestHCycle(short** adjMatrix, short n, int lightestHCycle){
208     this->lightestHCycle = lightestHCycle;
209     this->iteration = 0;
210     initPopulation(adjMatrix, this->populationSize, n);
211     Timer timer(60);
212     int currPopSize = this->populationSize;
213 
214     while(!timer.elapsed()){
215         ++this->iteration;
216         //selection
217         uniform_real_distribution<float> urdSel(0, this->highestFitness);//
218         uniform_real_distribution<float> urdCrossMut(0, 1);
219 
220         vector<short> firstSpecimen;
221         int firstSpecimenIndex;
222         pair<vector<short>, double> specimen;
223         bool firstSpecimenSet = false;
224 
225         pair<vector<short>, vector<short>> children;
226         int pathCost;
227         double pathFitness;
228         for(int it = 0;it < this->population.size(); ++it){
229             specimen = this->population[it];//
230             if(specimen.second > urdSel(this->generator)){//zostal wylosowany
231 
232                 if(this->crossoverRate > urdCrossMut(this->generator)){
233                     if(firstSpecimenSet){
234                         children = pmx(firstSpecimen, specimen.first);
235                 //mutacja
236                         if (this->mutationRate > urdCrossMut(this->generator)){//mutuje1
237                             inversion(children.first);
238 
239                         } else {
240                             //nie mutuje1
241                         }
242                         pathCost = getPathCost(adjMatrix, children.first);
243                         pathFitness = reportNewFitness(children.first, 1.0/pathCost, pathCost);
244                         if(pathFitness >= this->population[firstSpecimenIndex].second){//*0.99
245                             this->population[firstSpecimenIndex] =
246                                 pair<vector<short>, double>(children.first, pathFitness);
247                         }
248                         if (this->mutationRate > urdCrossMut(this->generator)){//mutuje2
249                             inversion(children.second);
250 
251                         } else {
252                             //nie mutuje2
253                         }
254                         pathCost = getPathCost(adjMatrix, children.second);
255                         pathFitness = reportNewFitness(children.second, 1.0/pathCost, pathCost);
256                         if(pathFitness >= this->population[it].second){//*0.99
257                             this->population[it] =
258                                 pair<vector<short>, double>(children.second, pathFitness);
259                         }
260                         firstSpecimenSet = false;
261                     }else{
262                         firstSpecimen = specimen.first;
263                         firstSpecimenIndex = it;
264                         firstSpecimenSet = true;
265                     }
266                 }else{//nie rozmnaza sie
267 
268                 }
269             }
270         }
271     }
272     cout << firstImprov << endl;
273     return this->bestCurrPath;
274 }
```

Rysunek 8: Metoda findShortestHCycle()

W powyższej implementacji zastosowano podejście inkrementacyjne. Nie ma wyraźnego etapu oceny czy sukcesji, populacja nie zostaje wyraźnie zastąpiona nową, ulega modyfikacji. Potomstwo ulega ocenie po narodzinach, przed poddaniem sukcesji, która to polega na porównaniu z jednym z rodziców i ewentualnej faktycznej sukcesji.

3 Pomiary

Poprawność działania algorytmu jest wnioskowana na podstawie poprawnie obliczanych kosztów ścieżek, poprawnego działania operatorów krzyżowania i mutacji na danych przykładowych oraz raportowania jedynie coraz to lepszych znalezionych rozwiązań.

Pomiary zostały wykonane po skompilowaniu z opcją optymalizacji 3 poziomu, na procesorze o maksymalnym taktowaniu 3.4 GHz. Na rzecz pomiarów przyjęto warunek stopu w postaci limitu czasu wykonania do 60 sekund. Pomiar dla każdej konfiguracji został wykonany 5 razy. Kolumny tablic reprezentują poziomy prawdopodobieństwa mutacji, wiersze odpowiadają wielkości populacji.

3.1 gr96.tsp

	0.01	0.05	0.1	0.2	0.3	0.5
50	10.3	6.69	6.98	6.29	5.49	6.39
100	9.23	7.16	6.38	7.15	6.53	5.91
200	8.20	7.46	7.24	5.79	5.88	7.16
300	8.49	7.27	7.20	6.40	6.63	7.24
500	8.49	7.27	7.20	6.40	6.63	7.24

Rysunek 9: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji gr96.tsp i szansy krzyżowania 0.6

	0.01	0.05	0.1	0.2	0.3	0.5
50	8.32	7.59	6.93	6.86	6.52	6.61
100	9.03	7.19	5.37	6.19	6.58	5.90
200	8.84	7.97	7.36	7.21	6.28	5.47
300	9.37	8.24	6.70	5.24	7.44	6.76
500	11.5	8.83	7.80	7.62	6.14	6.47

Rysunek 10: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji gr96.tsp i szansy krzyżowania 0.8

	0.01	0.05	0.1	0.2	0.3	0.5
50	1.24	0.88	1.01	0.92	0.84	0.97
100	1.02	1.00	1.19	1.16	0.99	1.00
200	0.93	0.94	0.98	0.80	0.94	1.31
300	0.91	0.88	1.08	1.22	0.89	1.07
500	0.74	0.82	0.92	0.84	1.08	1.12

Rysunek 11: Mapa ciepła dla wyniku dzielenia wartości komórki w tablicy szansy krzyżowania 0.6 na wartość odpowiadającej komórki w tabeli szansy krzyżowania 0.8

3.2 gr120.tsp

	0.01	0.05	0.1	0.2	0.3	0.5
50	11.4	11.3	10.3	9.51	8.69	8.69
100	11.3	9.75	10.0	10.0	10.2	8.31
200	10.9	10.1	9.27	7.89	9.55	7.76
300	12.5	9.39	9.29	9.10	8.52	8.25
500	11.8	10.1	8.71	8.70	9.24	9.31

Rysunek 12: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji gr120.tsp i szansy krzyżowania 0.6

	0.01	0.05	0.1	0.2	0.3	0.5	0.8	1
50	10.4	9.61	9.85	9.64	10.8	8.87	8.87	9.06
100	11.2	9.73	9.35	9.65	9.16	9.14	7.87	8.30
200	12.4	9.26	9.22	8.60	8.25	8.01	8.48	7.85
300	11.8	8.60	9.05	8.97	9.35	8.70	7.99	8.15
500	15.2	8.30	11.4	10.5	8.60	8.95	8.61	9.85

Rysunek 13: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji gr120.tsp i szansy krzyżowania 0.8

	0.01	0.05	0.1	0.2	0.3	0.5
50	1.09	1.18	1.04	0.99	0.80	0.98
100	1.01	1.00	1.07	1.04	1.11	0.91
200	0.88	1.09	1.01	0.92	1.16	0.97
300	1.06	1.09	1.03	1.01	0.91	0.95
500	0.78	1.22	0.76	0.83	1.08	1.04

Rysunek 14: Mapa ciepła dla wyniku dzielenia wartości komórki w tablicy szansy krzyżowania 0.6 na wartość odpowiadającej komórki w tabeli szansy krzyżowania 0.8

4 Wnioski

Pożądany rozmiar populacji dla tej implementacji zdaje się być przynajmniej większy od wielkości instancji problemu, np. dla tabeli 13 widoczny jest wzrost jakości rozwiązań dla wielkości populacji 200. Jest to prawdopodobnie spowodowane urozmaiceniem startowej populacji o osobniki wygenerowane losowo. W porównaniu do populacji wygenerowanej tylko metodą *best first search* może to skutkować lepszym pokryciem przestrzeni rozwiązań, prowadzącym do efektywniejszej eksploracji.

Można również zaobserwować względny brak różnicy między szansą rozmnażania 0.6 i 0.8. Rozważając ekstremum w postaci szansy krzyżowania 1.0, jeżeli osobnik przejdzie selekcję, będzie uczestniczył w krzyżowaniu. Dla szansy krzyżowania bliskiej 0.0, nawet jeżeli bardzo dużo osobników będzie przechodzić selekcję, bardzo niewiele z nich będzie się rozmnażać - oznacza to bezowocne generowanie liczb losowych i porównania które nie skutkują rozwojem populacji. Jako że model sukcesji to podejście inkrementacyjne, nie ma alternatywy dla krzyżowania, potencjalny rodzic pozostaje w populacji, niezmienny aż do faktycznego krzyżowania. Ostatecznie, dla specyfiki reszty implementacji, szansa na krzyżowanie jest zbędna ponieważ jedyne spowalnia algorytm i bezcelowo zmienia i odracza pary, te funkcje w pewnym stopniu pełni już proces selekcji. W ramach selekcji populacja przeglądana jest liniowo ale model selekcji wystarcza aby wprowadzić różnorodność w krzyżowanych parach, zapewnia że pary nie będą formowane sekwencyjnie, dwa osobniki za dwoma.

Na podstawie powyższych wyników można stwierdzić konieczność ustawiania wysokiego prawdopodobieństwa

mutacji. Co zaskakujące, najlepsze wyniki osiągnięto dla bardzo wysokiego prawdopodobieństwa mutacji, bliskiego lub równego 1. Jest to prawdopodobnie związane z faktem, iż populacja może tylko się polepszać - potomstwo gorzej przystosowane niż rodzice jest odrzucane. Sprawia to, że algorytm jest bardzo zbieżny, otrzymanie lepiej przystosowanego potomstwa eksplorującego inne maksimum lokalne jest mało prawdopodobne. Jedyną możliwością uzyskania eksploracji jest "przeskoczenie" z jednego dobrego rozwiązania na lepsze ale w innej części przestrzeni rozwiązań, dążące do innego maksimum lokalnego - jest to ułatwione mutacją. Z pomiarów wnioskowane jest, że bardzo wysokie prawdopodobieństwo mutacji jest potrzebne aby utrzymać eksplorację, jak i silnie ją rozpocząć przy starcie algorytmu, dywersyfikując populację.