|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САПР** | | Тема | оцінка | підпис |
| СПКс-11 | 5 | Розв’язання задачі комівояжера за допомогою генетичного алгоритму |  |  |
| Тураш Ю.Ю. | |
| № залікової: 1508500 | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

**Мета роботи**: Ознайомитися з основними теоретичними відомостями, вивчити еволюційні оператори схрещування та мутації, що використовуються при розв’язуванні задач комбінаторної оптимізації.

# Короткі теоретичні відомості

Завдання оптимізації - найбільш поширений і важливий для практики клас задач. Їх доводиться вирішувати кожному з нас або в побуті, розподіляючи свій час між різними справами, або на роботі, домагаючись максимальної швидкості роботи програми чи максимальної прибутковості компанії - в залежності від посади. Серед цих завдань є розв'язувані простим шляхом, але є і такі, точне рішення яких знайти практично неможливо.

Класичний приклад такого завдання, відомий як "завдання комівояжера" (Traveling Salesman Problem, TSP), формулюється так: комівояжеру потрібно об'їхати кілька міст, побувавши в кожному один раз, і повернутися у вихідну точку. Потрібно знайти найкоротший маршрут.

Найпростіший спосіб знайти оптимальне рішення - перебрати всі можливі значення параметрів. При цьому не потрібно робити ніяких припущень про властивості цільової функції, а задати її можна просто за допомогою таблиці. Однак, щоб вирішити таким способом завдання комівояжера хоча б для 20 міст, потрібно перебрати близько 1019 маршрутів, що абсолютно нереально ні для якого обчислювального центру. Таким чином, виникає необхідність в будь-якому новому методі оптимізації, придатному для практики.

Щоб використовувати генетичний алгоритм для вирішення практичних завдань, необхідно розглядати складніші варіанти введених вище понять. Пояснимо це на прикладі завдання комівояжера для 20 міст. Як індивідуумів будемо розглядати маршрути обходу. Інформацію про маршрут можна записати у вигляді однієї хромосоми - вектора довжини 20, де в першій позиції стоїть номер першого міста на шляху проходження, потім - номер другого міста і т. Д. Перше утруднення виникає, коли ми намагаємося визначити мутації для таких хромосом - стандартна операція, яка зраджує тільки одну позицію вектора, є неприпустимою, оскільки призводить до некоректного маршруту. Але можна визначити мутацію як перестановку значень двох випадково обраних генів (рисунок 3). При такому перетворенні маршрут змінюється тільки в двох містах.

За умовою завдання в розглянутих хромосомах кожен ген (номер міста) повинен зустрічатися тільки один раз. Такий різновид хромосом називається "перелічуваних хромосоми з унікальними генами" і часто використовується в комбінаторних задачах. Стандартна операція схрещування для цього типу хромосом знову ж некоректна, тому тут використовується більш складна схема двухточечного схрещування.

На малюнку 6 зображено результат роботи генетичного алгоритму в задачі комівояжера (використана демонстраційна програма до пакету GeneHunter).

Розташування 20 –ти міст

Рисунок 1 Найкоротший маршрут, знайдений генетичним алгоритмом

Знайдений маршрут, ймовірно, не є найоптимальнішим, але близький до нього по довжині - як правило, генетичні алгоритми "помиляються" не більше ніж на 2-3%. Цей недолік компенсується для комбінаторних завдань відносно високою швидкістю роботи - в нашому прикладі відповідь була отримана за 15 секунд. На практиці генетичні алгоритми нерідко використовують спільно з іншими методами, які дозволяють підвищити їх точність.

# Індивідуальне завдання

У вас є безліч міст (представлені у вигляді точок на площині з X і Y координати). Мета полягає в тому, щоб знайти найкоротший маршрут, який відвідує кожне місто рівно один раз, повертаючись в кінці своєї відправної точки.

Дано від 10 до 50 точок. Мова програмування довільна.

# Варіант -13(3)

3. Селекція рулеткою

# Результат виконання роботи

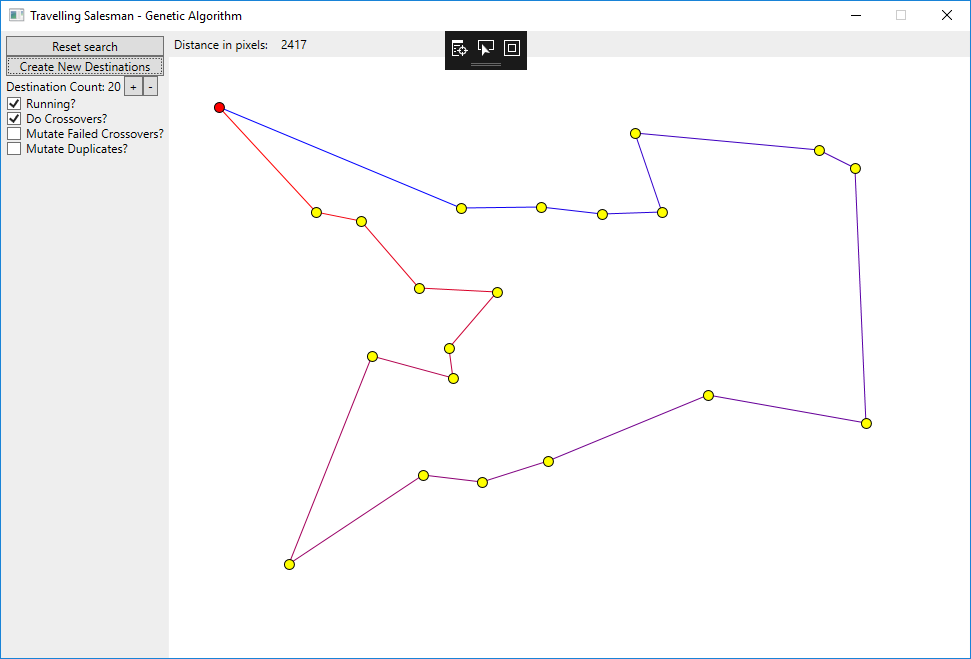


Рисунок 2 Найкоротший маршрут для 20-ти міст знайдений реалізованою програмою

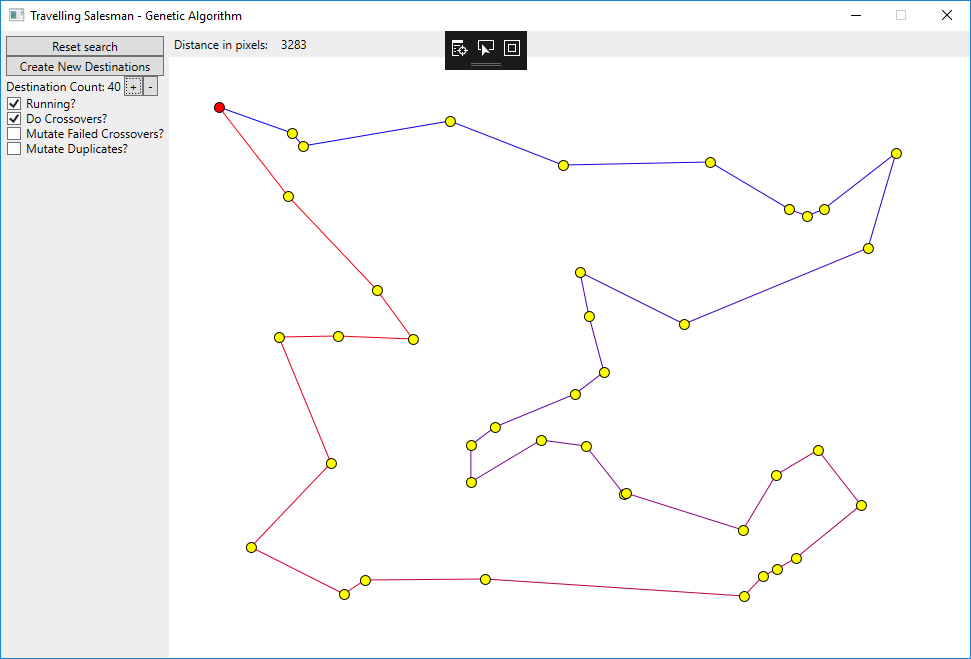


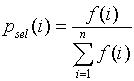
Рисунок 3 Найкоротший маршрут для 40-ка міст

Як видно на зображені вище реалізована програма шукає маршрут між вказаною кількістю міст. Їх розташування генерується випадково після кожного старту алгоритму.

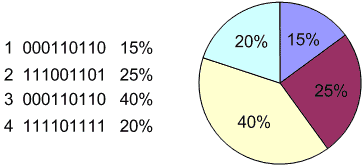
Оператор селекції здійснює відбір хромосом відповідно до значень їх функції пристосованості. В даному випадку використовується метод рулетки.

**Метод рулетки**

Roulette-wheel selection - відбирає особин за допомогою n "запусків" рулетки. Колесо рулетки містить по одному сектору для кожного члена популяції. Розмір i-ого сектора пропорційний відповідає величині *Psel(i)* обчислюється за формулою:



При такому відборі члени популяції з більш високою пристосованістю з більшою ймовірністю будуть частіше вибиратися, ніж особини з низькою пристосованістю.

  
Рисунок 4. Оператор селекції типу колеса рулетки з

пропорційними функціями пристосованості секторами

# Код програми

**Locationn.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Pfz.TravellingSalesman

{

public sealed class Location

{

public Location(int x, int y)

{

X = x;

Y = y;

}

// We could add other properties, like the name, the description

// or anything similar that we consider useful.

public int X { get; private set; }

public int Y { get; private set; }

public double GetDistance(Location other)

{

int diffX = X - other.X;

int diffY = Y - other.Y;

return Math.Sqrt(diffX\*diffX + diffY\*diffY);

}

public static double GetTotalDistance(Location startLocation, Location[] locations)

{

if (startLocation == null)

throw new ArgumentNullException("startLocation");

if (locations == null)

throw new ArgumentNullException("locations");

if (locations.Length == 0)

throw new ArgumentException("The locations array must have at least one element.", "locations");

foreach(var location in locations)

if (location == null)

throw new ArgumentException("The locations array can't contain null values.");

double result = startLocation.GetDistance(locations[0]);

int countLess1 = locations.Length-1;

for(int i=0; i<countLess1; i++)

{

var actual = locations[i];

var next = locations[i+1];

var distance = actual.GetDistance(next);

result += distance;

}

result += locations[locations.Length-1].GetDistance(startLocation);

return result;

}

public static void SwapLocations(Location[] locations, int index1, int index2)

{

if (locations == null)

throw new ArgumentNullException("locations");

if (index1 < 0 || index1 >= locations.Length)

throw new ArgumentOutOfRangeException("index1");

if (index2 < 0 || index2 >= locations.Length)

throw new ArgumentOutOfRangeException("index2");

var location1 = locations[index1];

var location2 = locations[index2];

locations[index1] = location2;

locations[index2] = location1;

}

// Moves an item in the list. That is, if we go from position 1 to 5, the items

// that were previously 2, 3, 4 and 5 become 1, 2, 3 and 4.

public static void MoveLocations(Location[] locations, int fromIndex, int toIndex)

{

if (locations == null)

throw new ArgumentNullException("locations");

if (fromIndex < 0 || fromIndex >= locations.Length)

throw new ArgumentOutOfRangeException("fromIndex");

if (toIndex < 0 || toIndex >= locations.Length)

throw new ArgumentOutOfRangeException("toIndex");

var temp = locations[fromIndex];

if (fromIndex < toIndex)

{

for(int i=fromIndex+1; i<=toIndex; i++)

locations[i-1] = locations[i];

}

else

{

for(int i=fromIndex; i>toIndex; i--)

locations[i] = locations[i-1];

}

locations[toIndex] = temp;

}

public static void ReverseRange(Location[] locations, int startIndex, int endIndex)

{

if (locations == null)

throw new ArgumentNullException("locations");

if (startIndex < 0 || startIndex >= locations.Length)

throw new ArgumentOutOfRangeException("startIndex");

if (endIndex < 0 || endIndex >= locations.Length)

throw new ArgumentOutOfRangeException("endIndex");

if (endIndex < startIndex)

{

int temp = endIndex;

endIndex = startIndex;

startIndex = temp;

}

while(startIndex<endIndex)

{

Location temp = locations[endIndex];

locations[endIndex] = locations[startIndex];

locations[startIndex] = temp;

startIndex++;

endIndex--;

}

}

public override string ToString()

{

return X + ", " + Y;

}

}

}

**RandomProvider.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Diagnostics;

namespace Pfz.TravellingSalesman

{

public static class RandomProvider

{

private static readonly Random \_random = new Random();

public static int GetRandomValue(int limit)

{

return \_random.Next(limit);

}

public static Location[] GetRandomDestinations(int count)

{

if (count < 2)

throw new ArgumentOutOfRangeException("count");

Location[] result = new Location[count];

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

{

int x = GetRandomValue(700) + 50;

int y = GetRandomValue(500) + 50;

result[i] = new Location(x, y);

}

return result;

}

public static void MutateRandomLocations(Location[] locations)

{

if (locations == null)

throw new ArgumentNullException("locations");

if (locations.Length < 2)

throw new ArgumentException("The locations array must have at least two items.", "locations");

// I opted to give up to 10% of the chromosome size in number of mutations.

// Maybe I should find a better number of make this configurable.

int mutationCount = GetRandomValue(locations.Length/10) + 1;

for(int mutationIndex=0; mutationIndex<mutationCount; mutationIndex++)

{

int index1 = GetRandomValue(locations.Length);

int index2 = GetRandomValue(locations.Length-1);

if (index2 >= index1)

index2++;

switch(GetRandomValue(3))

{

case 0: Location.SwapLocations(locations, index1, index2); break;

case 1: Location.MoveLocations(locations, index1, index2); break;

case 2: Location.ReverseRange(locations, index1, index2); break;

default: throw new InvalidOperationException();

}

}

}

public static void FullyRandomizeLocations(Location[] locations)

{

if (locations == null)

throw new ArgumentNullException("locations");

int count = locations.Length;

for(int i=count-1; i>0; i--)

{

int value = GetRandomValue(i+1);

if (value != i)

Location.SwapLocations(locations, i, value);

}

}

internal static void \_CrossOver(Location[] locations1, Location[] locations2, bool mutateFailedCrossovers)

{

// I am not validating parameters because this method is internal.

// If you want to make it public, you should validate the parameters.

var availableLocations = new HashSet<Location>(locations1);

int startPosition = GetRandomValue(locations1.Length);

int crossOverCount = GetRandomValue(locations1.Length - startPosition);

if (mutateFailedCrossovers)

{

bool useMutation = true;

int pastEndPosition = startPosition + crossOverCount;

for (int i=startPosition; i<pastEndPosition; i++)

{

if (locations1[i] != locations2[i])

{

useMutation = false;

break;

}

}

if (useMutation)

{

MutateRandomLocations(locations1);

return;

}

}

Array.Copy(locations2, startPosition, locations1, startPosition, crossOverCount);

List<int> toReplaceIndexes = null;

int index = 0;

foreach(var value in locations1)

{

if (!availableLocations.Remove(value))

{

if (toReplaceIndexes == null)

toReplaceIndexes = new List<int>();

toReplaceIndexes.Add(index);

}

index++;

}

// Finally we will replace duplicated items by those that are still available.

// This is how we avoid having chromosomes that contain duplicated places to go.

if (toReplaceIndexes != null)

{

// To do this, we enumerate two objects in parallel.

// If we could use foreach(var indexToReplace, location from toReplaceIndexex, location1) it would be great.

using(var enumeratorIndex = toReplaceIndexes.GetEnumerator())

{

using(var enumeratorLocation = availableLocations.GetEnumerator())

{

while(true)

{

if (!enumeratorIndex.MoveNext())

{

Debug.Assert(!enumeratorLocation.MoveNext());

break;

}

if (!enumeratorLocation.MoveNext())

throw new InvalidOperationException("Something wrong happened.");

locations1[enumeratorIndex.Current] = enumeratorLocation.Current;

}

}

}

}

}

}

}

**TravellingSalesmanAlgorithm.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Pfz.TravellingSalesman

{

public sealed class TravellingSalesmanAlgorithm

{

private readonly Location \_startLocation;

private readonly KeyValuePair<Location[], double>[] \_populationWithDistances;

public TravellingSalesmanAlgorithm(Location startLocation, Location[] destinations, int populationCount)

{

if (startLocation == null)

throw new ArgumentNullException("startLocation");

if (destinations == null)

throw new ArgumentNullException("destinations");

if (populationCount < 2)

throw new ArgumentOutOfRangeException("populationCount");

if (populationCount % 2 != 0)

throw new ArgumentException("The populationCount parameter must be an even value.", "populationCount");

\_startLocation = startLocation;

destinations = (Location[])destinations.Clone();

foreach(var destination in destinations)

if (destination == null)

throw new ArgumentException("The destinations array can't contain null values.", "destinations");

\_populationWithDistances = new KeyValuePair<Location[], double>[populationCount];

// Create initial population.

for(int solutionIndex=0; solutionIndex<populationCount; solutionIndex++)

{

var newPossibleDestinations = (Location[])destinations.Clone();

// Try commenting the next 2 lines of code while keeping the \_GetFakeShortest active.

// If you avoid the algorithm from running and press reset, you will see that it always

// start with a path that seems "good" but is not the best.

for(int randomIndex=0; randomIndex<newPossibleDestinations.Length; randomIndex++)

RandomProvider.FullyRandomizeLocations(newPossibleDestinations);

var distance = Location.GetTotalDistance(startLocation, newPossibleDestinations);

var pair = new KeyValuePair<Location[], double>(newPossibleDestinations, distance);

\_populationWithDistances[solutionIndex] = pair;

}

Array.Sort(\_populationWithDistances, \_sortDelegate);

}

private Location[] \_GetFakeShortest(Location[] destinations)

{

Location[] result = new Location[destinations.Length];

var currentLocation = \_startLocation;

for(int fillingIndex=0; fillingIndex<destinations.Length; fillingIndex++)

{

int bestIndex = -1;

double bestDistance = double.MaxValue;

for(int evaluatingIndex=0; evaluatingIndex<destinations.Length; evaluatingIndex++)

{

var evaluatingItem = destinations[evaluatingIndex];

if (evaluatingItem == null)

continue;

double distance = currentLocation.GetDistance(evaluatingItem);

if (distance < bestDistance)

{

bestDistance = distance;

bestIndex = evaluatingIndex;

}

}

result[fillingIndex] = destinations[bestIndex];

currentLocation = destinations[bestIndex];

destinations[bestIndex] = null;

}

return result;

}

private static readonly Comparison<KeyValuePair<Location[], double>> \_sortDelegate = \_Sort;

private static int \_Sort(KeyValuePair<Location[], double> value1, KeyValuePair<Location[], double> value2)

{

return value1.Value.CompareTo(value2.Value);

}

public IEnumerable<Location> GetBestSolutionSoFar()

{

foreach(var location in \_populationWithDistances[0].Key)

yield return location;

}

public bool MustMutateFailedCrossovers { get; set; }

public bool MustDoCrossovers { get; set; }

public void Reproduce()

{

var bestSoFar = \_populationWithDistances[0];

int halfCount = \_populationWithDistances.Length / 2;

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

{

var parent = \_populationWithDistances[i].Key;

var child1 = \_Reproduce(parent);

var child2 = \_Reproduce(parent);

var pair1 = new KeyValuePair<Location[], double>(child1, Location.GetTotalDistance(\_startLocation, child1));

var pair2 = new KeyValuePair<Location[], double>(child2, Location.GetTotalDistance(\_startLocation, child2));

\_populationWithDistances[i\*2] = pair1;

\_populationWithDistances[i\*2 + 1] = pair2;

}

// We keep the best alive from one generation to the other.

\_populationWithDistances[\_populationWithDistances.Length-1] = bestSoFar;

Array.Sort(\_populationWithDistances, \_sortDelegate);

}

public void MutateDuplicates()

{

bool needToSortAgain = false;

int countDuplicates = 0;

var previous = \_populationWithDistances[0];

for(int i=1; i<\_populationWithDistances.Length; i++)

{

var current = \_populationWithDistances[i];

if (!previous.Key.SequenceEqual(current.Key))

{

previous = current;

continue;

}

countDuplicates++;

needToSortAgain = true;

RandomProvider.MutateRandomLocations(current.Key);

\_populationWithDistances[i] = new KeyValuePair<Location[], double>(current.Key, Location.GetTotalDistance(\_startLocation, current.Key));

}

if (needToSortAgain)

Array.Sort(\_populationWithDistances, \_sortDelegate);

}

private Location[] \_Reproduce(Location[] parent)

{

var result = (Location[])parent.Clone();

if (!MustDoCrossovers)

{

// When we are not using cross-overs, we always apply mutations.

RandomProvider.MutateRandomLocations(result);

return result;

}

int otherIndex = RandomProvider.GetRandomValue(\_populationWithDistances.Length/2);

var other = \_populationWithDistances[otherIndex].Key;

RandomProvider.\_CrossOver(result, other, MustMutateFailedCrossovers);

if (!MustMutateFailedCrossovers)

if (RandomProvider.GetRandomValue(10) == 0)

RandomProvider.MutateRandomLocations(result);

return result;

}

}

}

**Висновок:** під час виконання лабораторної роботи, я отримав практичні навички необхідні для програмної реалізації алгоритму, який дозволяє вирішувати задачу комівояжера з використанням генетичного алгоритму та методом відбору рулеткою.