МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий и робототехники (ФИТР)

Кафедра программного обеспечения информационных систем и технологий

**КУРСОВОЙ ПРОЕКТ**

по дисциплине: «Моделирование и оптимизация проектирования техническим систем»

на тему: «***Профилирование backend Java приложений, взаимодействующих с БД***»

Выполнил: М.А. Лосев

студент группы 10701118

Руководитель: А.А. Прихожий

проф. д.т.н.

Минск 2021

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий и робототехники (ФИТР)

Кафедра программного обеспечения информационных систем и технологий

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе**

по дисциплине: «Моделирование и оптимизация проектирования техническим систем»

на тему: «***Профилирование backend Java приложений, взаимодействующих с БД***»

Выполнил: М.А. Лосев

студент группы 10701118

Руководитель: А.А. Прихожий

проф. д.т.н.

Минск 2021

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 4](#_30j0zll)

[1 Постановка задачи](#_1fob9te) 5

[2 Модель решения задачи](#_3znysh7) 6

[3 Метод решения задачи](#_2et92p0) 7

[4 Алгоритм решения задачи](#_tyjcwt) 8

[5 Информационная модель](#_4d34og8) 11

6 [Архитектура программы](#_2p2csry) 12

7 [Графический интерфейс](#_17dp8vu) 16

8 [Тестовый пример. Тестирование программы](#_3rdcrjn) 17

9 [Демонстрация работы программы](#_3rdcrjn) 22

[ЗАКЛЮЧЕНИЕ](#_26in1rg) 26

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ](#_lnxbz9) 27

[ПРИЛОЖЕНИЕ А](#_35nkun2) 28

# **ВВЕДЕНИЕ**

На сегодняшний день довольно распространенными являются задачи транспортной логистики. Для решения данных задач применяется метод коммивояжера (коммивояжер — бродячий торговец). Суть данного метода заключается в том, что коммивояжер обходит k городов для продажи определенных предметов и в конечном итоге возвращается в исходный город. Цель метода заключается в том, что необходимо найти наиболее выгодный путь. Критерием выгоды могут быть различные варианты, например, время, расстояние, стоимость и т.д.

Для решения задачи коммивояжера нередко применяют генетический алгоритм. Данный алгоритм последовательно подбирает искомые параметры способом, похожим на биологическую эволюцию. Акцент алгоритма ставится на так называемом “скрещивании”, которое подобно тому, что есть в живой природе.

Существует ряд достопримечательностей в городе Минске, по которым можно проводить экскурсии, к ним относятся: “Верхний город”, “Красный костел”, “Национальная библиотека”, “Парк Горького”, “Площадь победы” и многие другие.

Для оптимизации перемещения между достопримечательностями был применен генетический алгоритм. В общем виде данную задачу можно свести к задаче коммивояжера.

# **1 Постановка задачи**

Необходимо найти кратчайший путь между достопримечательностями города Минск, а именно:

* Верхний город
* Красный костел
* Национальная библиотека
* Парк Горького
* Красный дворик
* Октябрьская улица
* Городские ворота
* Театр оперы и балета
* Художественный музей
* Зыбицкая улица

Возьмем условное расстояние между между достопримечательностями (в дальнейшем это расстояние можно будет корректировать), как показано на рисунке 1.1.

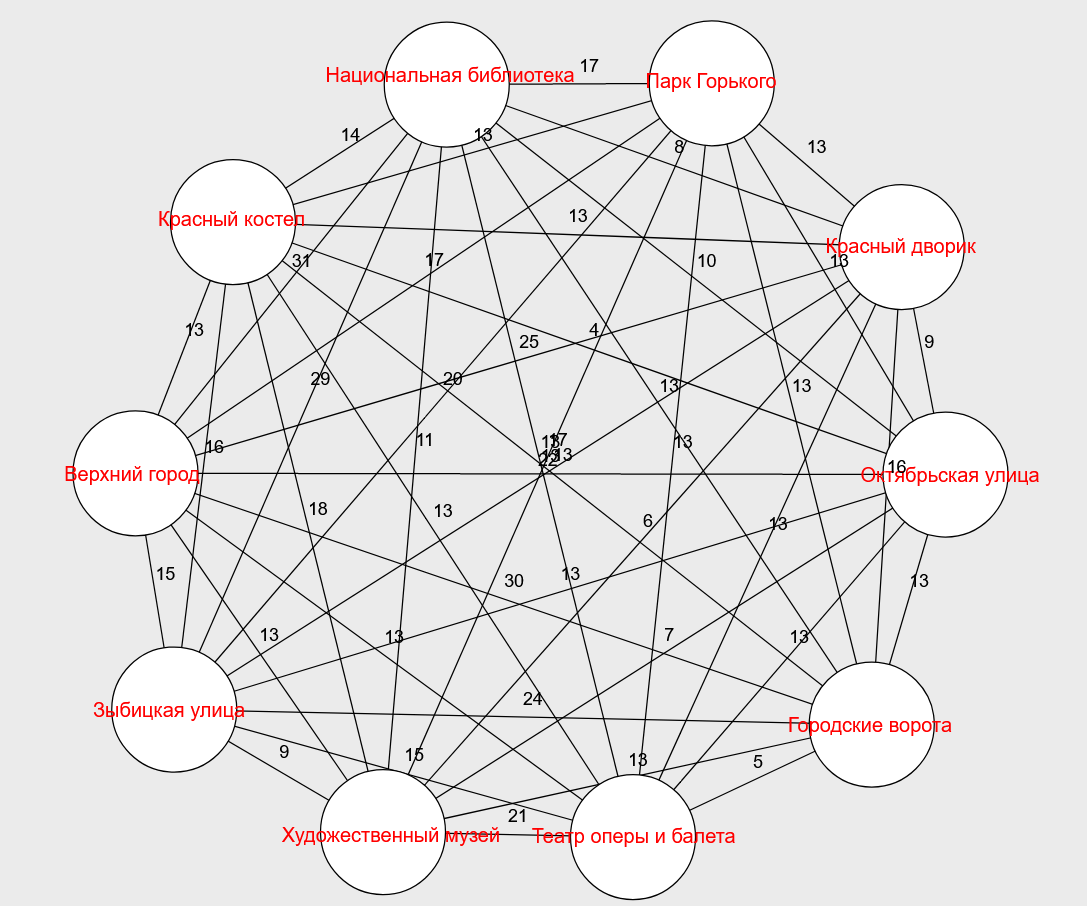


Рисунок 1.1 — Граф с условными расстояниями между достопримечательностями

Предположим, что все экскурсии будут начинаться от “Верхнего города”. Таким, образом необходимо найти кратчайший путь от заданного места, проходящий через все достопримечательности единажды, с последующим возвращением в заданную точку.

**2 Модель решения задачи**

В поставленной задаче будет рассматриваться проведение экскурсии по городу Минску. Для проведения экскурсии необходимо выполнить ряд требований:

* Объезд всех достопримечательностей
* Единоразовое посещение достопримечательности
* Возвращение в исходную точку

Также необходимо составить оптимальный план объезда достопримечательностей, а именно найти минимальный путь.

**3 Метод решения задачи**

Для решения поставленной задачи был выбран генетический алгоритм, который является представителем эволюционных алгоритмов.

В данном алгоритме все начинается с формирования начальной популяции, т.е. множество особей одного вида.

Каждая задача для генетического алгоритма старается формироваться так, чтобы особи были представлены списком целых или вещественных чисел. В этом случае мы получаем естественную биологическую интерпретацию этих списков, как хромосом, состоящим из набора ген.

Как правило для формирования начального значения ген используют датчик случайных чисел. В результате получается множество различных особей, что способствует дальнейшему процессу эволюционной адаптации всей популяции.

Как правило в процессе эволюционного отбора в популяции остаются наиболее приспособленные особи, следовательно нужно научиться определять степень приспособленности, чтобы имитировать процесс эволюции. В природе приспособленность определяется как фактор выживаемости, а в случае генетического алгоритма это получение наилучшего решения, т.е. те особи, которые предоставляют наилучшее решение, считаются более приспособленными.

Для оценки качества решения каждого конкретного индивидуума необходимо задать функцию приспособленности (целевую функцию). Значение данной функции зависит от хромосомы конкретного индивида. Для различных задач используются различные функции приспособленности.

После формирования популяции и определения функции, дающей оценку приспособленности каждой конкретной особи, происходит имитация эволюционного процесса, т.е. происходит воспроизведение изменений популяции от поколения к поколению в надежде получения все более и более приспособленных особей, а значит идти по пути улучшения решения основной задачи.

Схема работы генетического алгоритма представлена на рисунке 3.1.

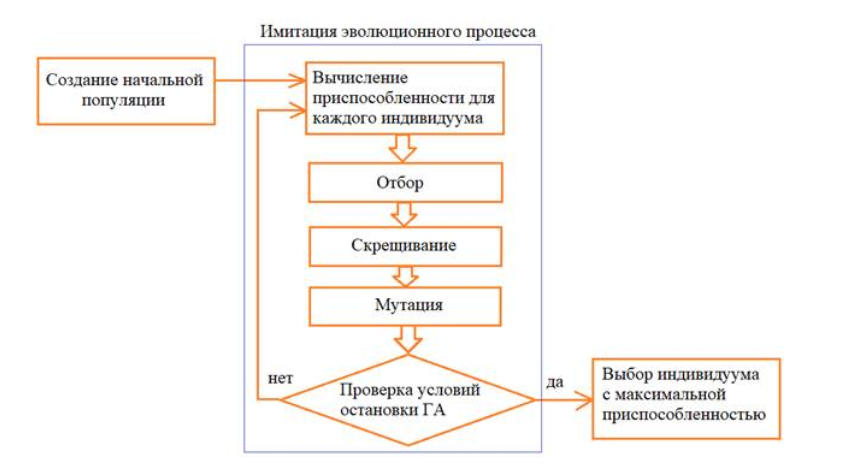


Рисунок 3.1 - Схема работы генетического алгоритма

Отбор - первые действия, выполняемые в генетическом алгоритме, выполняемые при имитации эволюционного процесса. Целью данного этапа является оставление в популяции наиболее приспособленных особей, а также сохранить популяционного разнообразия.

Скрещивание - процесс соединения наиболее приспособленного родите с наименее приспособленным. На этом этапе происходит обмен частей хромосом родителей, получая новый набор генов в хромосомах потомков.

Мутация применяется к полученной популяции и случайным образом с малой вероятностью меняет значения отдельных генов.

**4 Алгоритм решения задачи**

*Начальная популяция*

Начальная популяция генерируется обычно случайно. Единственный критерий - достаточное разнообразие особей, чтобы популяция не свалилась в ближайший экстремум.

*Оценка приспособленности*

Оценка приспособленности часто проводится в две стадии. Первая - собственно оценка, представленная формулой 4.1

, (4.1)

Вторая - дополнительные преобразования. Например, ею может быть нормировка к виду, представленному формулой 4.2

, (4.2)

где f1 и f0, соответственно, лучший и худший показатели в текущей популяции.

*Оператор скрещивания*

Чаще всего скрещивание производят над двумя лучшими особями. Результатом является также обычно две особи с компонентами, взятыми от их родителей. Цель этого оператора - распространение хороших генов в популяции и стягивание плотности популяции к тем областям, где она и так велика в том предположении, что "нас много там, где хорошо".

В одноточечном варианте, результатом скрещивания родителей, представляется формулой 4.3

, (4.3)

в k-ой популяции станут два элемента популяции k+1, такие что, будут удовлетворять формулам 4.4 и 4.5

, (4.4), (4.5)

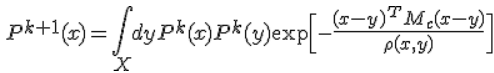
где точка *c* выбирается случайно. В двухточечном варианте, соответственно, точек пересечения будет две, и они также выбираются случайно. Легко расширить эту конструкцию и до n точек. Нужно заметить, что в случае нечетного n, происходит n+1-точечный кроссинговер с n+1-ой точкой между последней и первой компонентами.

Скрещиванием с маской является результат в виде двух потомков с компонентами, принадлежность которых определяется по битовой маске. Т.е. результатом скрещивания родителей в k-ой популяции станут два элемента популяции k+1, такие что удовлетворяют формулам 4.6 и 4.7

, (4.6), (4.7)

и противоположные условия для второго отпрыска. Маска выбирается случайно. Для простоты, ею может быть третья особь.

В непрерывном пространстве можно ввести такую аналогию для скрещивания, представленную формулой 4.8

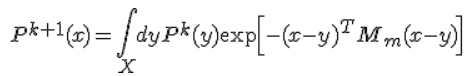
, (4.8)

где P^k(x) - плотность генофонда к-ой популяции, p(x,y) - расстояние между двумя особями с генами x и y.

*Оператор мутаций*

Оператор мутаций просто меняет произвольное число элементов в особи на другие произвольные. Фактически он является неким диссипативным элементом, с одной стороны вытягивающим из локальных экстремумов, с другой - приносящим новую информацию в популяцию.

* Инвертирует бит в случае бинарного признака.
* Изменяет на некоторую величину числовой признак. Причём, скорее на ближайший.
* Заменит на другой номинальный признак.
* В непрерывном пространстве можно ввести следующую аналогию, представленную формулой 4.9

, (4.9)

*Критерии останова*

* нахождение глобального, либо субоптимального решения;
* выходом на «плато»;
* исчерпание числа поколений, отпущенных на эволюцию;
* исчерпание времени, отпущенного на эволюцию;
* исчерпание заданного числа обращений к целевой функции.

**5 Информационная модель**

*Входные данные.*

* sightList – список достопримечательностей, экземпляр класса Sights (элемент списка имеет тип Point). Содержит координаты достопримечательностей, индекс списка – это номер достопримечательности. Число достопримечательностей отображается в поле NumberSightsValue формы приложения.
* populationSize – размер популяции (тип int), то есть списка маршрутов, улучшаемых генетическим алгоритмом. По умолчанию равен 10000.
* wrGroupSize - размер рабочей группы (тип int), которая хранится в массиве wrGroup. В массив случайным образом выбираются маршруты популяции. Массив сортируется по возрастанию длины маршрута. Первые два маршрута отсортированного массива порождают дочерний маршрут, который заменяет самый плохой маршрут в рабочей группе. Задается в диапазоне [2, populationSize - 1]. По умолчанию wrGroupSize = 5.
* mutationChance – вероятность мутации дочернего маршрута (тип int). Задается в процентах в диапазоне [0, 100]. По умолчанию равна 3%.
* maxGenerations – максимальное число поколений (тип int). Вычисления прекращаются, когда число скрещиваний (генераций дочерних маршрутов превысит) maxGenerations. По умолчанию равно 10000000.

*Выходные данные.*

* Population – популяция с измененными маршрутами (элемент списка имеет тип Tour).
* BestTour – лучший маршрут в популяции (свойство класса Population, имеет тип Tour).
* lastIterationValue – номер итерации (поколения) получения последнего лучшего маршрута (тип int).
* lastFitnessValue – длина последнего лучшего маршрута (тип double).
* Рисунок последнего лучшего маршрута.

*Промежуточные данные.*

* Списки расстояний Distances (элемент списка имеет тип double). Формируются для каждой достопримечательности и содержат расстояния до всех прочих достопримечательностей.
* Fitness – длина маршрута (свойства маршрута Tour, имеет тип double). Вычисляется для каждого маршрута.
* generation – номер текущего поколения (тип int).
* Рабочая группа – массив wrGoup (тип int). Формируется для каждого поколения из всего множества маршрутов. Содержит номера маршрутов. Два первых маршрута отсортированного массива (два его лучших маршрута) используются для создания дочернего маршрута.
* Дочерний маршрут child (тип Tour).

На каждом последующем ходу всем достопримечательностям задается параметр занятости, но хотя бы 1 из достопримечательностей обязана быть свободной. Выбор следующего шага осуществляется среди свободных достопримечательностей. Если осталась 1 пункт посещения, то занятость не рассчитывается.

**6 Архитектура программы**

В программе создаются следующие классы:

*Класс Sight.*

Имеет следующие свойства:

* Location (тип Point) – X и Y координаты достопримечательности.
* Distances – список расстояний от текущей достопримечательности до всех прочих достопримечательностей. Формируется для каждой достопримечательности.

*Класс Sights* – содержит список достопримечательностей, которые нужно посетить. Элемент списка имеет тип Sight.

Обладает следующими методами:

* OpenSightList – формируется по входным данным.
* CalculateSightDistances
* обеспечивает заполнение списка sight.Distances, содержащего расстояния от достопримечательности sight до всех прочих достопримечательностей;

*Класс Population* – содержит список маршрутов, с которыми работает генетический алгоритм. Тип элемента списка – Tour.

Имеет 2 свойства:

* BestTour – текущий лучший маршрут (тип Tour).
* Imprv - число улучшений решения (тип int).

Содержит метод CreateRandomPopulation, обеспечивающий формирование начальной популяции.

*Класс Tour* – представляет маршрут по всем достопримечательностям.

Маршрут хранится в виде списка с элементами типа int.

Имеет свойство Fitness – длина маршрута (тип double). Определяется для каждого маршрута.

Содержит следующие методы:

* DetermineFitness – определяет общую протяженность маршрута.
* Crossover – выполняет операцию скрещивания двух родительских маршрутов parent1 и parent1. Результатом скрещивания является дочерний маршрут child (тип Tour).

Первоначально child полностью совпадает с parent1. Затем из parent2 в child переносится кусок маршрута parent2.

* Mutate - выполняет операцию мутации дочернего маршрута. Вероятность мутации задается параметром mutationChance. Мутация – это либо случайный обмен 2-х достопримечательностей дочернего маршрута, либо 2-х его кусков. Регулируется переменной flipSight.

*Класс Tsp* – координирует решение задачи коммивояжера.

Содержит:

* Определение делегата NewBestTourEventHandler – обработчика события "Найден лучший маршрут".
* Определение события foundNewBestTour "Найден лучший маршрут".
* Определение датчика случайных чисел.

Имеет следующие свойства:

* sightList – список городов (тип Sights).
* population – список всех маршрутов (тип Population).
* Halt – флаг завершения работы алгоритма (поиска новых поколений) (тип bool). Принимает значение true, если во время работы алгоритма нажать на кнопку "Стоп". В результате вычисления прерываются; надпись на кнопке меняется на "Пуск".

Имеет метод FindBestTour, выполняющий поиск решения и завершающий вычисления после генерации заданного числа поколений (параметр maxGenerations).

*Класс TspEventArgs* – устанавливает аргументы события foundNewBestTour (найден очередной лучший маршрут), при наступлении которого отображается очередной лучший маршрут.

Имеет следующие свойства:

* SightList – список достопримечательностей(тип Sights).
* BestTour – лучший на текущий момент маршрут (тип Tour).
* Generation – номер поколения, в котором появился текущий лучший маршрут (тип int).
* Done – флаг завершения алгоритма (тип bool).
* Imprv – число улучшений маршрута (тип int).

*Класс TspForm* – управляет формой приложения.

*Класс Program* – предоставляет метод Main для точки входа (entry point).

Архитектура приложения представлена на рисунке 6.1:

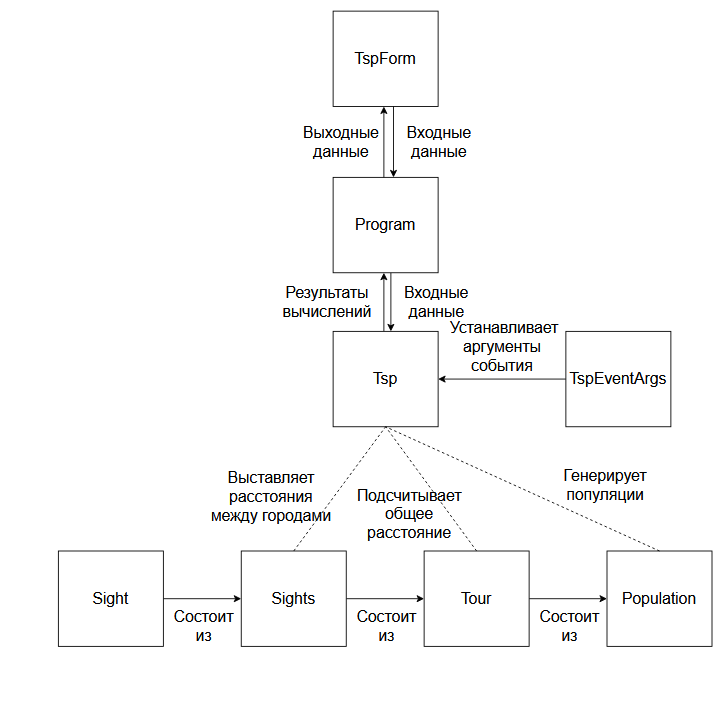


Рисунок 6.1 - Архитектура приложения

**7 Графический интерфейс**

Графический интерфейс программы представлен на рисунке 7.1:



Рисунок 7.1 - Графический интерфейс программы

На рисунке 7.1 представлены:

* 4 окна для ввода информации:
* Population size - размер популяции (тип int), то есть списка маршрутов, улучшаемых генетическим алгоритмом. По умолчанию равен 10000
* Mutati % - вероятность мутации дочернего маршрута (тип int). Задается в процентах в диапазоне [0, 100]. По умолчанию равна 3%.
* Group size - размер рабочей группы (тип int), которая хранится в массиве wrGroup. В массив случайным образом выбираются маршруты популяции. Массив сортируется по возрастанию длины маршрута. Первые два маршрута отсортированного массива порождают дочерний маршрут, который заменяет самый плохой маршрут в рабочей группе. Задается в диапазоне [2, populationSize - 1]. По умолчанию wrGroupSize = 5.
* Maximum generations - максимальное число поколений (тип int). Вычисления прекращаются, когда число скрещиваний (генераций дочерних маршрутов превысит) maxGenerations. По умолчанию равно 10000000.
* 2 кнопки:
* Clear sight list
* Start (меняется после нажатия на Stop)
* Окно для выставления положения достопримечательностей
* 3 окна вывода информации:
* Last iteration - номер итерации (поколения) получения последнего лучшего маршрута (тип int).
* Last tour length - длина последнего лучшего маршрута (тип double).
* Sights - количество достопримечательностей.

**8 Тестовый пример. Тестирование программы**

Для проверки работоспособности ПО выполнены следующие инструкции:

* Введены неверные входные значения
* Попытка построения графа без нанесения достопримечательностей
* Построение оптимального пути при маленьком значении количества итераций
* Построение оптимального пути при большом % мутаций

При вводе некорректных данных выдается предупреждение, представленное на рисунке 8.1:

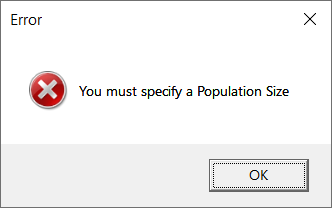


Рисунок 8.1 – Сообщение о неверно введенных данных

При попытке построения оптимального пути без указания расположения достопримечательностей выдается предупреждение, показанное на рисунке 8.1:

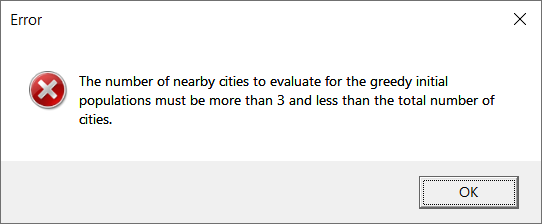


Рисунок 8.2 - Сообщение при отсутствии положения достопримечательностей

При построении оптимального пути с малым количеством итераций был построен не оптимальный путь перемещения по достопримечательностям, как показано на рисунке 8.1:

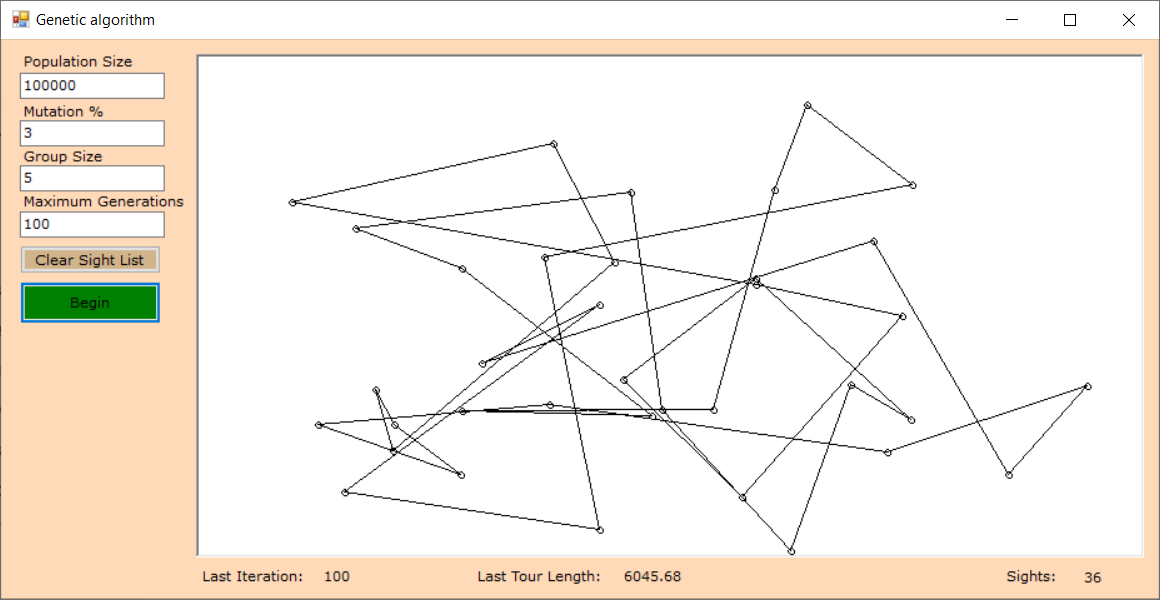


Рисунок 8.3 - Построение пути с малым количеством итераций

Однако если увеличить количество итераций, то вероятность нахождения правильного пути увеличится, как показано на рисунке 8.4:

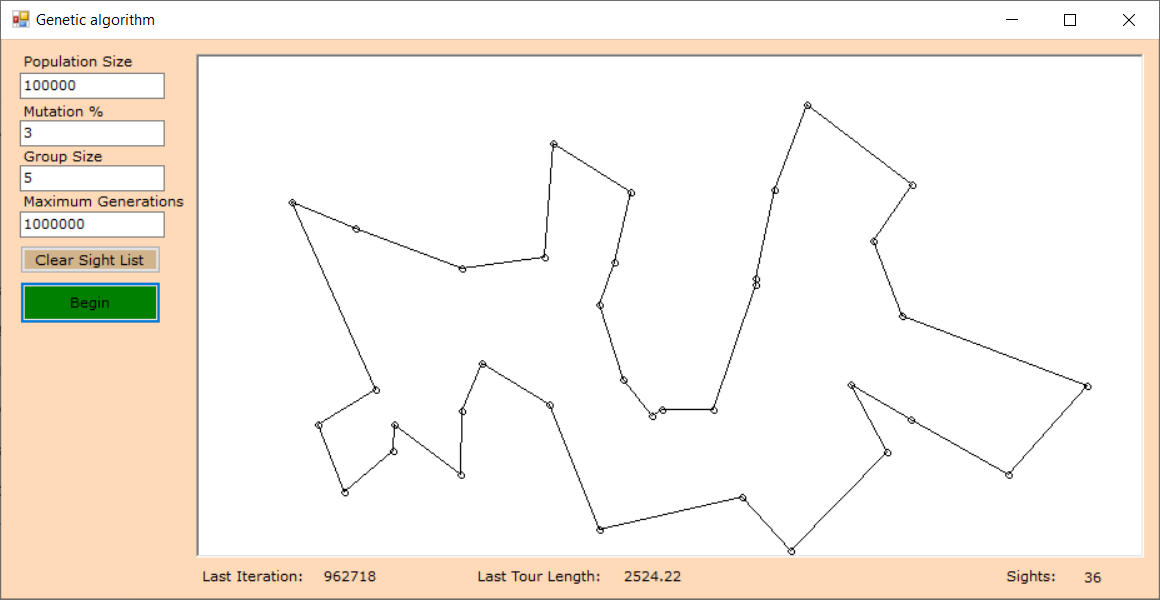


Рисунок 8.4 - Построение пути при достаточном количестве итераций

При большом количестве мутаций существует вероятность пропуска оптимального пути, как показано на рисунке 8.5:

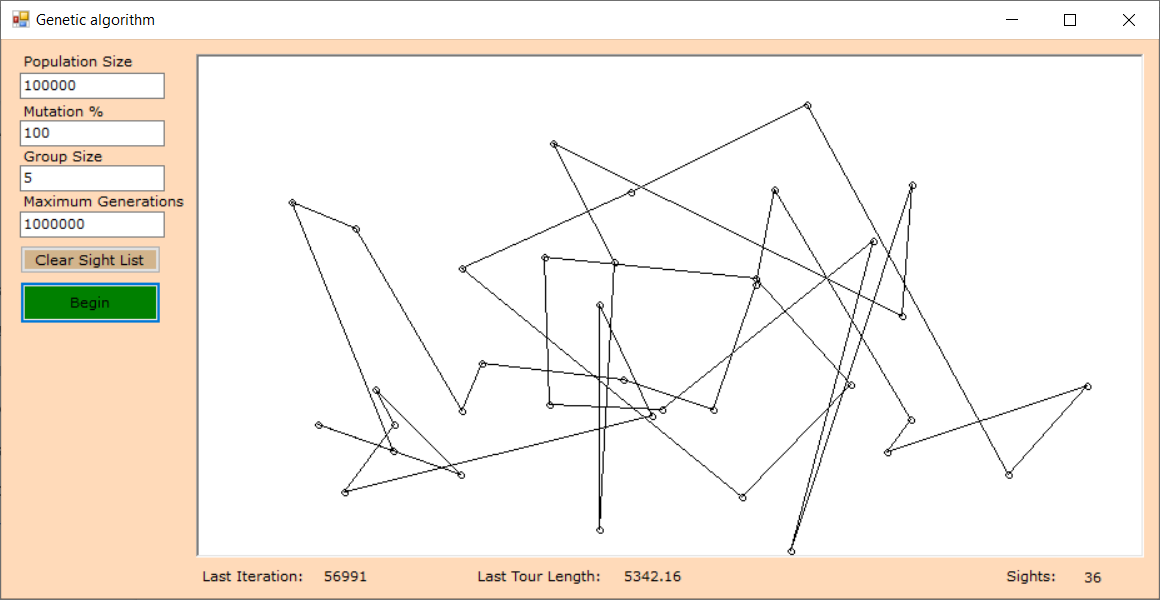


Рисунок 8.5 - Нахождение оптимального пути при большом % мутации

Для проверки эффективности алгоритма сравним его с полным перебором при количестве вершин равном 10 и 20 соответственно.

Проводить измерения будем при следующих параметрах:

* Populatio size = 100000
* Mutation = 3
* Group size = 5
* Maximum generation = 1000000

При 10 вершинах:

• Полный обход находит оптимальное решение за 362.880 итерации

• Генетический алгоритм находит оптимальное решение в среднем за 11000 итераций.

Количество итераций затраченных на нахождение оптимального маршрута представлены на рисунке 8.6

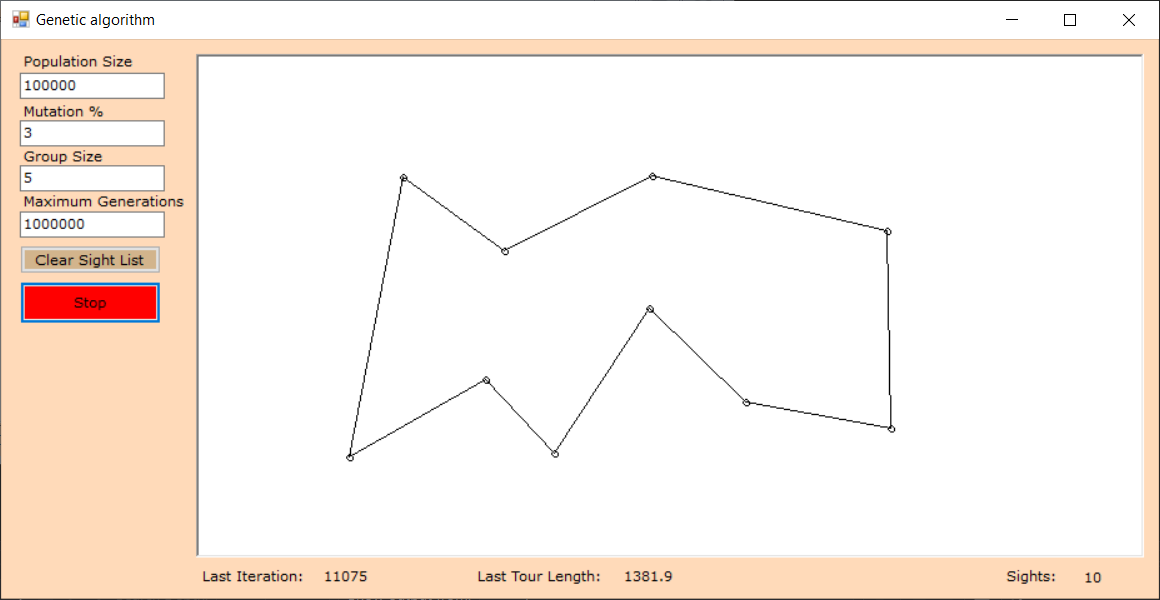


Рисунок 8.6 – Количество итераций затраченных на нахождение оптимального маршрута при n = 10

Таким образом при 10 вершинах генетическим алгоритмом был в среднем в 33 раза быстрее.

При 20 вершинах:

• Полный обход находит оптимальное решение за 1.216451e+17 итерации

• Генетический алгоритм находит оптимальное решение в среднем за 32700 итераций

Количество итераций затраченных на нахождение оптимального маршрута представлены на рисунке 8.7

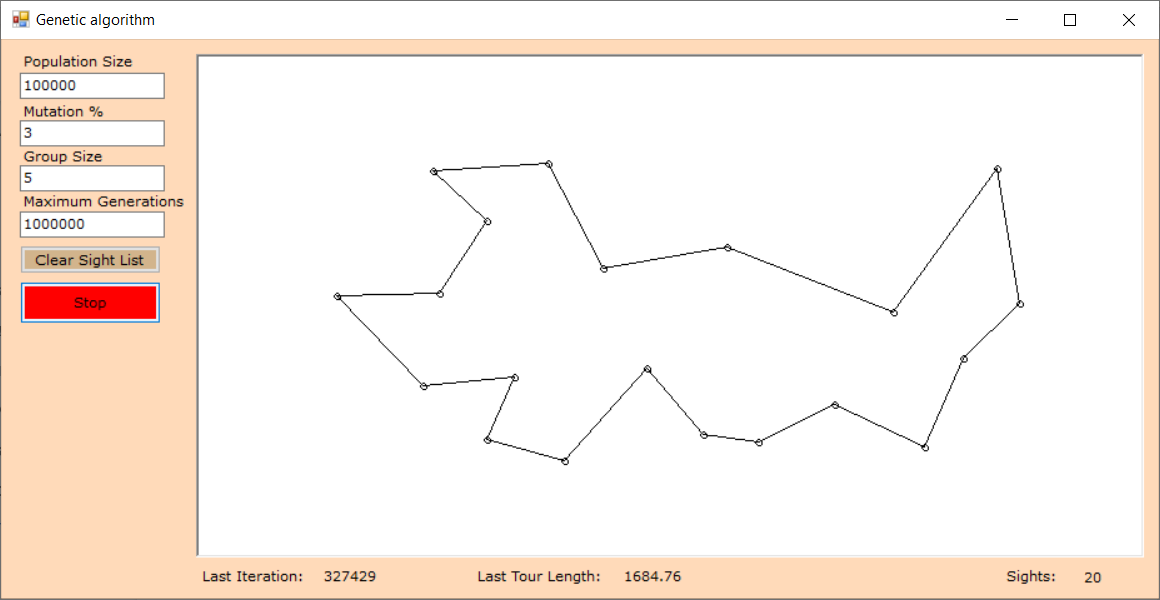


Рисунок 8.7 – Количество итераций затраченных на нахождение оптимального маршрута при n = 20

Таким образом при 20 вершинах муравьиный алгоритм в среднем в 3.7200337e+12 раз быстрее обычного перебора.

В результате тестирования был сделан вывод:

Поиск оптимального решения происходит значительно быстрее с использованием генетического алгоритма.

**9 Демонстрация работы программы**

Для демонстрации программы запустим приложения, после чего будет показан экран, представленный на рисунке 9.1:



Рисунок 9.1 - Стартовый экран

Далее, при помощи мышки, нанесем в произвольном порядке точки, которые будут обозначать расположение достопримечательностей, как показано на рисунке 9.2:

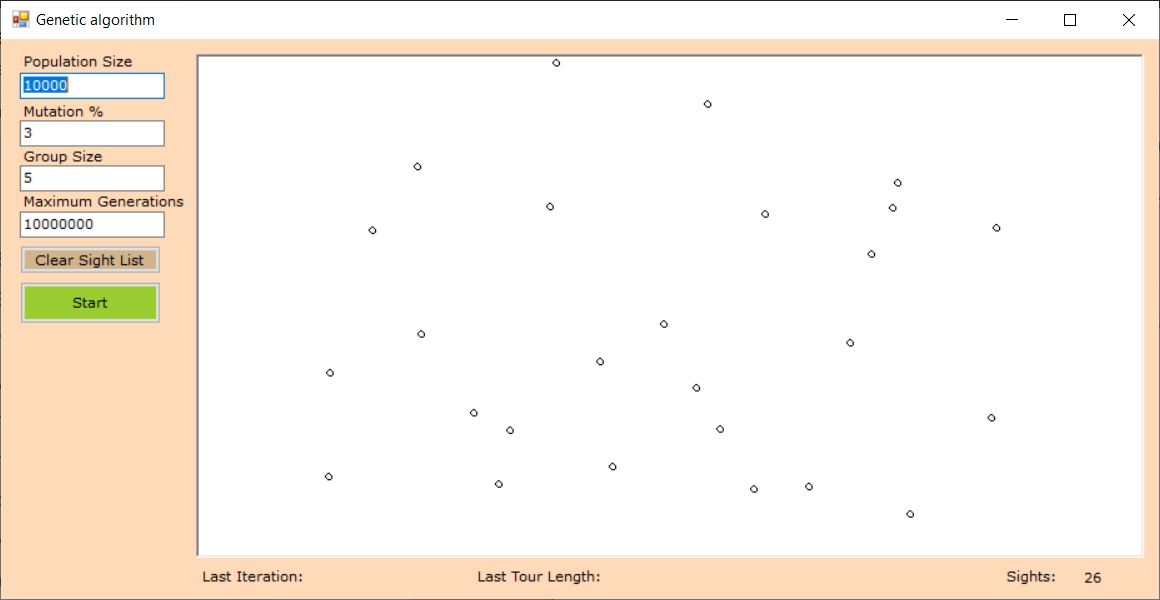


Рисунок 9.2 - Расположение достопримечательностей

После чего немного изменим входные параметры и нажмем кнопку старт.

На рисунках 9.3 и 9.4 показаны промежуточный и конечный результаты соответственно.

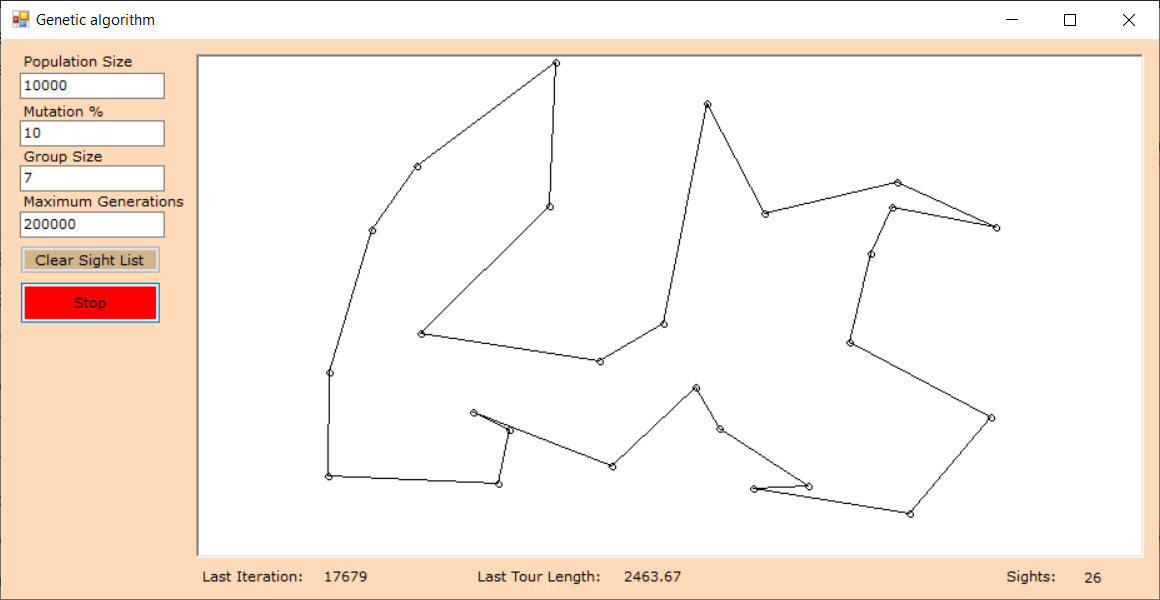


Рисунок 9.3 - Промежуточное построение пути

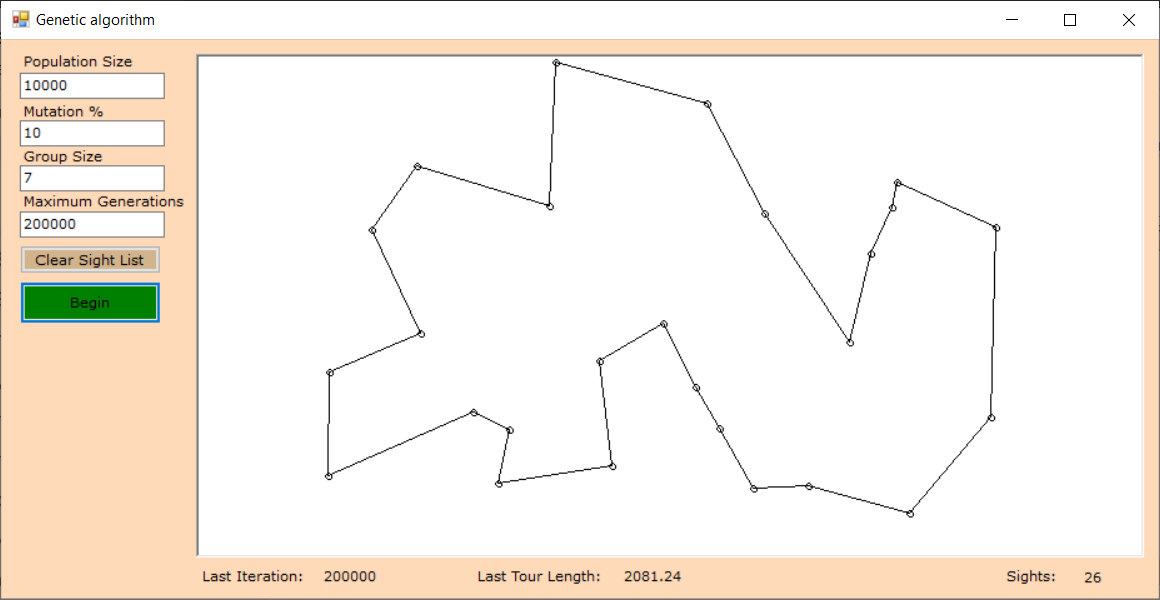


Рисунок 9.4 - Конечное построение пути

**ЗАКЛЮЧЕНИЕ**

В данной курсово работе были использованы следующие средства:

* Язык программирования C#
* Среда разработки Visual Studio 2019

Благодаря данным средствам было реализовано оконное приложение, выполняющее функциональность оптимального планирования экскурсии по городу Минск генетическим алгоритмом.

В данном приложении есть возможность ручного проставления начальных позиций достопримечательностей.

Также пользователь может самостоятельно ввести следующие критерии:

* Размер популяции
* Вероятность мутации
* Размер рабочей группы
* Максимальное число поколений
* Число городов-соседей

На основе полученных данных будет запущен генетический алгоритм, который будет формировать поколения пока не найдет оптимальное решение или не выполнит заданное количество итераций.

Во время поиска решения пользователь сможет визуально наблюдать за нахождением оптимального пути благодаря графическому интерфейсу, а также определить число итераций, необходимых для нахождения кратчайшего пути, и кратчайший путь.

**СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Задачи комивояжера генетическим алгоритмом [Электронный ресурс] – Режим доступа: <https://dic.academic.ru/dic.nsf/ruwiki/42438> – Дата доступа 27.11.2021, свободный – Загл. с экрана. – Яз.рус.
2. Генетический алгоритм [Электронный ресурс] – Ре-жим доступа: [https://studfilies.net/preview/4494386/](https://studfiles.net/preview/4494386/) – Дата доступа 27.11.2021, свободный – Загл. с экрана. – Яз.рус.
3. Сравнение алгоритмов нахождения пути [Электронный ресурс] – Режим доступа: <https://ru.wikipedia.org/wiki/Рефакторинг> – Дата доступа 27.11.2021, свободный – Загл. с экрана. – Яз.рус.

ПРИЛОЖЕНИЕ А

Листинг исходных кодов

*Файл Sight.cs*

using System;

using System.Collections.Generic;

using System.Text;

using System.Drawing;

namespace Tsp

{

/// <summary>

/// An individual City in our tour.

/// </summary>

public class City

{

/// <summary>

/// Constructor that provides the city location.

/// </summary>

/// <param name="x">X position of the city.</param>

/// <param name="y">Y position of the city.</param>

public City(int x, int y)

{

Location = new Point(x, y);

}

/// <summary>

/// Private copy of the location of this city.

/// </summary>

private Point location;

/// <summary>

/// The location of this city.

/// </summary>

public Point Location

{

get

{

return location;

}

set

{

location = value;

}

}

/// <summary>

/// Private copy of the distance from this city to every other city.

/// The index in this array is the number of the city linked to.

/// </summary>

private List<double> distances = new List<double>();

/// <summary>

/// The distance from this city to every other city.

/// </summary>

public List<double> Distances

{

get

{

return distances;

}

set

{

distances = value;

}

}

/// <summary>

/// Private copy of the list of the cities that are closest to this one.

/// </summary>

private List<int> closeCities = new List<int>();

/// <summary>

/// A list of the cities that are closest to this one.

/// </summary>

public List<int> CloseCities

{

get

{

return closeCities;

}

}

/// <summary>

/// Find the cities that are closest to this one.

/// </summary>

/// <param name="numberOfCloseCities">When creating the initial population of tours, this is a greater chance

/// that a nearby city will be chosen for a link. This is the number of nearby cities that will be considered close.</param>

public void FindClosestCities( int numberOfCloseCities )

{

double shortestDistance;

int shortestCity = 0;

double[] dist = new double[Distances.Count];

Distances.CopyTo(dist);

if (numberOfCloseCities > Distances.Count - 1)

{

numberOfCloseCities = Distances.Count - 1;

}

closeCities.Clear();

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

{

shortestDistance = Double.MaxValue;

for (int cityNum = 0; cityNum < Distances.Count; cityNum++)

{

if (dist[cityNum] < shortestDistance)

{

shortestDistance = dist[cityNum];

shortestCity = cityNum;

}

}

closeCities.Add(shortestCity);

dist[shortestCity] = Double.MaxValue;

}

}

}

}

*Файл Sights.cs*

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

using System.Data;

using System.IO;

using System.Globalization;

namespace Tsp

{

/// <summary>

/// This class contains the list of cities for this test.

/// Each city has a location and the distance information to every other city.

/// </summary>

public class Cities : List<City>

{

/// <summary>

/// Determine the distances between each city.

/// </summary>

/// <param name="numberOfCloseCities">When creating the initial population of tours, this is a greater chance

/// that a nearby city will be chosen for a link. This is the number of nearby cities that will be considered close.</param>

public void CalculateCityDistances( int numberOfCloseCities )

{

foreach (City city in this)

{

city.Distances.Clear();

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

{

city.Distances.Add(Math.Sqrt(Math.Pow((double)(city.Location.X - this[i].Location.X), 2D) +

Math.Pow((double)(city.Location.Y - this[i].Location.Y), 2D)));

}

}

foreach (City city in this)

{

city.FindClosestCities(numberOfCloseCities);

}

}

/// <summary>

/// Open the XML file that contains the list of cities.

/// </summary>

/// <param name="fileName">Name of the XML file.</param>

/// <returns>The city list.</returns>

/// <exception cref="FileNotFoundException">fileName parameter is invalid.</exception>

/// <exception cref="InvalidCastException">XML File is not properly formatted.</exception>

public void OpenCityList(string fileName)

{

DataSet cityDS = new DataSet();

try

{

this.Clear();

cityDS.ReadXml(fileName);

DataRowCollection cities = cityDS.Tables[0].Rows;

foreach (DataRow city in cities)

{

this.Add(new City(Convert.ToInt32(city["X"], CultureInfo.CurrentCulture), Convert.ToInt32(city["Y"], CultureInfo.CurrentCulture)));

}

}

finally

{

cityDS.Dispose();

}

}

}

}

*Файл Tsp.cs*

using System;

using System.Collections.Generic;

using System.Windows.Forms;

using System.Text;

using System.Data;

using System.Drawing;

namespace Tsp

{

/// <summary>

/// This class performs the Travelling Salesman Problem algorithm.

/// </summary>

class Tsp

{

/// <summary>

/// Delegate used to raise an event when a new best tour is found.

/// </summary>

/// <param name="sender">Object that generated this event.</param>

/// <param name="e">Event arguments. Contains information about the best tour.</param>

public delegate void NewBestTourEventHandler(Object sender, TspEventArgs e);

/// <summary>

/// Event fired when a new best tour is found.

/// </summary>

public event NewBestTourEventHandler foundNewBestTour;

/// <summary>

/// Random number generator object.

/// We allow the GUI to set the seed for the random number generator to assist in debugging.

/// This allows errors to be easily reproduced.

/// </summary>

Random rand;

/// <summary>

/// The list of cities. This is only used to calculate the distances between the cities.

/// </summary>

Cities cityList;

/// <summary>

/// The complete list of all the tours.

/// </summary>

Population population;

/// <summary>

/// Private copy of a flag that will stop the TSP from calculating any more generations.

/// </summary>

private bool halt = false;

/// <summary>

/// The GUI sets this flag to true to stop the TSP algorithm and allow the Begin() function to return.

/// </summary>

public bool Halt

{

get

{

return halt;

}

set

{

halt = value;

}

}

/// <summary>

/// Default Constructor

/// </summary>

public Tsp()

{

}

/// <summary>

/// Starts the TSP algorithm.

/// To stop before all generations are calculated, set <see cref="Halt"/> to true.

/// </summary>

/// <param name="populationSize">Number of random tours to create before starting the algorithm.</param>

/// <param name="maxGenerations">Number of times to perform the crossover operation before stopping.</param>

/// <param name="groupSize">Number of tours to examine in each generation. Top 2 are chosen as the parent tours whose children replace the worst 2 tours in the group.</param>

/// <param name="mutation">Odds that a child tour will be mutated..</param>

/// <param name="seed">Seed for the random number generator.</param>

/// <param name="chanceToUseCloseCity">The odds (out of 100) that a city that is known to be close will be used in any given link.</param>

/// <param name="cityList">List of cities in the tour.</param>

public void Begin(int populationSize, int maxGenerations, int groupSize, int mutation, int seed, int chanceToUseCloseCity, Cities cityList)

{

rand = new Random(seed);

this.cityList = cityList;

population = new Population();

population.CreateRandomPopulation(populationSize, cityList, rand, chanceToUseCloseCity);

displayTour(population.BestTour, 0, false);

bool foundNewBestTour = false;

int generation;

for (generation = 0; generation < maxGenerations; generation++)

{

if (Halt)

{

break; // GUI has requested we exit.

}

foundNewBestTour = makeChildren(groupSize, mutation);

if (foundNewBestTour)

{

displayTour(population.BestTour, generation, false);

}

}

displayTour(population.BestTour, generation, true);

}

/// <summary>

/// Randomly select a group of tours from the population.

/// The top 2 are chosen as the parent tours.

/// Crossover is performed on these 2 tours.

/// The childred tours from this process replace the worst 2 tours in the group.

/// </summary>

/// <param name="groupSize">Number of tours in this group.</param>

/// <param name="mutation">Odds that a child will be mutated.</param>

bool makeChildren(int groupSize, int mutation)

{

int[] tourGroup = new int[groupSize];

int tourCount, i, topTour, childPosition, tempTour;

// pick random tours to be in the neighborhood city group

// we allow for the same tour to be included twice

for (tourCount = 0; tourCount < groupSize; tourCount++)

{

tourGroup[tourCount] = rand.Next(population.Count);

}

// bubble sort on the neighborhood city group

for (tourCount = 0; tourCount < groupSize - 1; tourCount++)

{

topTour = tourCount;

for (i = topTour + 1; i < groupSize; i++)

{

if (population[tourGroup[i]].Fitness < population[tourGroup[topTour]].Fitness)

{

topTour = i;

}

}

if (topTour != tourCount)

{

tempTour = tourGroup[tourCount];

tourGroup[tourCount] = tourGroup[topTour];

tourGroup[topTour] = tempTour;

}

}

bool foundNewBestTour = false;

// take the best 2 tours, do crossover, and replace the worst tour with it

childPosition = tourGroup[groupSize - 1];

population[childPosition] = Tour.Crossover(population[tourGroup[0]], population[tourGroup[1]], cityList, rand);

if (rand.Next(100) < mutation)

{

population[childPosition].Mutate(rand);

}

population[childPosition].DetermineFitness(cityList);

// now see if the first new tour has the best fitness

if (population[childPosition].Fitness < population.BestTour.Fitness)

{

population.BestTour = population[childPosition];

foundNewBestTour = true;

}

// take the best 2 tours (opposite order), do crossover, and replace the 2nd worst tour with it

childPosition = tourGroup[groupSize - 2];

population[childPosition] = Tour.Crossover(population[tourGroup[1]], population[tourGroup[0]], cityList, rand);

if (rand.Next(100) < mutation)

{

population[childPosition].Mutate(rand);

}

population[childPosition].DetermineFitness(cityList);

// now see if the second new tour has the best fitness

if (population[childPosition].Fitness < population.BestTour.Fitness)

{

population.BestTour = population[childPosition];

foundNewBestTour = true;

}

return foundNewBestTour;

}

/// <summary>

/// Raise an event to the GUI listener to display a tour.

/// </summary>

/// <param name="bestTour">The best tour the algorithm has found so far.</param>

/// <param name="generationNumber">How many generations have been performed.</param>

/// <param name="complete">Is the TSP algorithm complete.</param>

void displayTour(Tour bestTour, int generationNumber, bool complete)

{

if (foundNewBestTour != null)

{

this.foundNewBestTour(this, new TspEventArgs(cityList, bestTour, generationNumber, complete));

}

}

}

}

*Файл Tour.cs*

using System;

using System.Collections.Generic;

using System.Text;

namespace Tsp

{

/// <summary>

/// This class represents one instance of a tour through all the cities.

/// </summary>

public class Tour : List<Link>

{

/// <summary>

/// Constructor that takes a default capacity.

/// </summary>

/// <param name="capacity">Initial size of the tour. Should be the number of cities in the tour.</param>

public Tour(int capacity)

: base(capacity)

{

resetTour(capacity);

}

/// <summary>

/// Private copy of this fitness of this tour.

/// </summary>

private double fitness;

/// <summary>

/// The fitness (total tour length) of this tour.

/// </summary>

public double Fitness

{

set

{

fitness = value;

}

get

{

return fitness;

}

}

/// <summary>

/// Creates the tour with the correct number of cities and creates initial connections of all -1.

/// </summary>

/// <param name="numberOfCities"></param>

private void resetTour(int numberOfCities)

{

this.Clear();

Link link;

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

{

link = new Link();

link.Connection1 = -1;

link.Connection2 = -1;

this.Add(link);

}

}

/// <summary>

/// Determine the fitness (total length) of an individual tour.

/// </summary>

/// <param name="cities">The cities in this tour. Used to get the distance between each city.</param>

public void DetermineFitness(Cities cities)

{

Fitness = 0;

int lastCity = 0;

int nextCity = this[0].Connection1;

foreach (Link link in this)

{

Fitness += cities[lastCity].Distances[nextCity];

// figure out if the next city in the list is [0] or [1]

if (lastCity != this[nextCity].Connection1)

{

lastCity = nextCity;

nextCity = this[nextCity].Connection1;

}

else

{

lastCity = nextCity;

nextCity = this[nextCity].Connection2;

}

}

}

/// <summary>

/// Creates a link between 2 cities in a tour, and then updates the city usage.

/// </summary>

/// <param name="tour">The incomplete child tour.</param>

/// <param name="cityUsage">Number of times each city has been used in this tour. Is updated when cities are joined.</param>

/// <param name="city1">The first city in the link.</param>

/// <param name="city2">The second city in the link.</param>

private static void joinCities(Tour tour, int[] cityUsage, int city1, int city2)

{

// Determine if the [0] or [1] link is available in the tour to make this link.

if (tour[city1].Connection1 == -1)

{

tour[city1].Connection1 = city2;

}

else

{

tour[city1].Connection2 = city2;

}

if (tour[city2].Connection1 == -1)

{

tour[city2].Connection1 = city1;

}

else

{

tour[city2].Connection2 = city1;

}

cityUsage[city1]++;

cityUsage[city2]++;

}

/// <summary>

/// Find a link from a given city in the parent tour that can be placed in the child tour.

/// If both links in the parent aren't valid links for the child tour, return -1.

/// </summary>

/// <param name="parent">The parent tour to get the link from.</param>

/// <param name="child">The child tour that the link will be placed in.</param>

/// <param name="cityList">The list of cities in this tour.</param>

/// <param name="cityUsage">Number of times each city has been used in the child.</param>

/// <param name="city">City that we want to link from.</param>

/// <returns>The city to link to in the child tour, or -1 if none are valid.</returns>

private static int findNextCity(Tour parent, Tour child, Cities cityList, int[] cityUsage, int city)

{

if (testConnectionValid(child, cityList, cityUsage, city, parent[city].Connection1))

{

return parent[city].Connection1;

}

else if (testConnectionValid(child, cityList, cityUsage, city, parent[city].Connection2))

{

return parent[city].Connection2;

}

return -1;

}

/// <summary>

/// Determine if it is OK to connect 2 cities given the existing connections in a child tour.

/// If the two cities can be connected already (witout doing a full tour) then it is an invalid link.

/// </summary>

/// <param name="tour">The incomplete child tour.</param>

/// <param name="cityList">The list of cities in this tour.</param>

/// <param name="cityUsage">Array that contains the number of times each city has been linked.</param>

/// <param name="city1">The first city in the link.</param>

/// <param name="city2">The second city in the link.</param>

/// <returns>True if the connection can be made.</returns>

private static bool testConnectionValid(Tour tour, Cities cityList, int[] cityUsage, int city1, int city2)

{

// Quick check to see if cities already connected or if either already has 2 links

if ((city1 == city2) || (cityUsage[city1] == 2) || (cityUsage[city2] == 2))

{

return false;

}

// A quick check to save CPU. If haven't been to either city, connection must be valid.

if ((cityUsage[city1] == 0) || (cityUsage[city2] == 0))

{

return true;

}

// Have to see if the cities are connected by going in each direction.

for (int direction = 0; direction < 2; direction++)

{

int lastCity = city1;

int currentCity;

if (direction == 0)

{

currentCity = tour[city1].Connection1; // on first pass, use the first connection

}

else

{

currentCity = tour[city1].Connection2; // on second pass, use the other connection

}

int tourLength = 0;

while ((currentCity != -1) && (currentCity != city2) && (tourLength < cityList.Count - 2))

{

tourLength++;

// figure out if the next city in the list is [0] or [1]

if (lastCity != tour[currentCity].Connection1)

{

lastCity = currentCity;

currentCity = tour[currentCity].Connection1;

}

else

{

lastCity = currentCity;

currentCity = tour[currentCity].Connection2;

}

}

// if cities are connected, but it goes through every city in the list, then OK to join.

if (tourLength >= cityList.Count - 2)

{

return true;

}

// if the cities are connected without going through all the cities, it is NOT OK to join.

if (currentCity == city2)

{

return false;

}

}

// if cities weren't connected going in either direction, we are OK to join them

return true;

}

/// <summary>

/// Perform the crossover operation on 2 parent tours to create a new child tour.

/// This function should be called twice to make the 2 children.

/// In the second call, the parent parameters should be swapped.

/// </summary>

/// <param name="parent1">The first parent tour.</param>

/// <param name="parent2">The second parent tour.</param>

/// <param name="cityList">The list of cities in this tour.</param>

/// <param name="rand">Random number generator. We pass around the same random number generator, so that results between runs are consistent.</param>

/// <returns>The child tour.</returns>

public static Tour Crossover(Tour parent1, Tour parent2, Cities cityList, Random rand)

{

Tour child = new Tour(cityList.Count); // the new tour we are making

int[] cityUsage = new int[cityList.Count]; // how many links 0-2 that connect to this city

int city; // for loop variable

int nextCity; // the other city in this link

for (city = 0; city < cityList.Count; city++)

{

cityUsage[city] = 0;

}

// Take all links that both parents agree on and put them in the child

for (city = 0; city < cityList.Count; city++)

{

if (cityUsage[city] < 2)

{

if (parent1[city].Connection1 == parent2[city].Connection1)

{

nextCity = parent1[city].Connection1;

if (testConnectionValid(child, cityList, cityUsage, city, nextCity))

{

joinCities(child, cityUsage, city, nextCity);

}

}

if (parent1[city].Connection2 == parent2[city].Connection2)

{

nextCity = parent1[city].Connection2;

if (testConnectionValid(child, cityList, cityUsage, city, nextCity))

{

joinCities(child, cityUsage, city, nextCity);

}

}

if (parent1[city].Connection1 == parent2[city].Connection2)

{

nextCity = parent1[city].Connection1;

if (testConnectionValid(child, cityList, cityUsage, city, nextCity))

{

joinCities(child, cityUsage, city, nextCity);

}

}

if (parent1[city].Connection2 == parent2[city].Connection1)

{

nextCity = parent1[city].Connection2;

if (testConnectionValid(child, cityList, cityUsage, city, nextCity))

{

joinCities(child, cityUsage, city, nextCity);

}

}

}

}

// The parents don't agree on whats left, so we will alternate between using

// links from parent 1 and then parent 2.

for (city = 0; city < cityList.Count; city++)

{

if (cityUsage[city] < 2)

{

if (city % 2 == 1) // we prefer to use parent 1 on odd cities

{

nextCity = findNextCity(parent1, child, cityList, cityUsage, city);

if (nextCity == -1) // but if thats not possible we still go with parent 2

{

nextCity = findNextCity(parent2, child, cityList, cityUsage, city); ;

}

}

else // use parent 2 instead

{

nextCity = findNextCity(parent2, child, cityList, cityUsage, city);

if (nextCity == -1)

{

nextCity = findNextCity(parent1, child, cityList, cityUsage, city);

}

}

if (nextCity != -1)

{

joinCities(child, cityUsage, city, nextCity);

// not done yet. must have been 0 in above case.

if (cityUsage[city] == 1)

{

if (city % 2 != 1) // use parent 1 on even cities

{

nextCity = findNextCity(parent1, child, cityList, cityUsage, city);

if (nextCity == -1) // use parent 2 instead

{

nextCity = findNextCity(parent2, child, cityList, cityUsage, city);

}

}

else // use parent 2

{

nextCity = findNextCity(parent2, child, cityList, cityUsage, city);

if (nextCity == -1)

{

nextCity = findNextCity(parent1, child, cityList, cityUsage, city);

}

}

if (nextCity != -1)

{

joinCities(child, cityUsage, city, nextCity);

}

}

}

}

}

// Remaining links must be completely random.

// Parent's links would cause multiple disconnected loops.

for (city = 0; city < cityList.Count; city++)

{

while (cityUsage[city] < 2)

{

do

{

nextCity = rand.Next(cityList.Count); // pick a random city, until we find one we can link to

} while (!testConnectionValid(child, cityList, cityUsage, city, nextCity));

joinCities(child, cityUsage, city, nextCity);

}

}

return child;

}

/// <summary>

/// Randomly change one of the links in this tour.

/// </summary>

/// <param name="rand">Random number generator. We pass around the same random number generator, so that results between runs are consistent.</param>

public void Mutate(Random rand)

{

int cityNumber = rand.Next(this.Count);

Link link = this[cityNumber];

int tmpCityNumber;

// Find which 2 cities connect to cityNumber, and then connect them directly

if (this[link.Connection1].Connection1 == cityNumber) // Conn 1 on Conn 1 link points back to us.

{

if (this[link.Connection2].Connection1 == cityNumber)// Conn 1 on Conn 2 link points back to us.

{

tmpCityNumber = link.Connection2;

this[link.Connection2].Connection1 =link.Connection1;

this[link.Connection1].Connection1 = tmpCityNumber;

}

else // Conn 2 on Conn 2 link points back to us.

{

tmpCityNumber = link.Connection2;

this[link.Connection2].Connection2 = link.Connection1;

this[link.Connection1].Connection1 = tmpCityNumber;

}

}

else // Conn 2 on Conn 1 link points back to us.

{

if (this[link.Connection2].Connection1 == cityNumber)// Conn 1 on Conn 2 link points back to us.

{

tmpCityNumber = link.Connection2;

this[link.Connection2].Connection1 = link.Connection1;

this[link.Connection1].Connection2 = tmpCityNumber;

}

else // Conn 2 on Conn 2 link points back to us.

{

tmpCityNumber = link.Connection2;

this[link.Connection2].Connection2 = link.Connection1;

this[link.Connection1].Connection2 = tmpCityNumber;

}

}

int replaceCityNumber = -1;

do

{

replaceCityNumber = rand.Next(this.Count);

}

while (replaceCityNumber == cityNumber);

Link replaceLink = this[replaceCityNumber];

// Now we have to reinsert that city back into the tour at a random location

tmpCityNumber = replaceLink.Connection2;

link.Connection2 = replaceLink.Connection2;

link.Connection1 = replaceCityNumber;

replaceLink.Connection2 = cityNumber;

if (this[tmpCityNumber].Connection1 == replaceCityNumber)

{

this[tmpCityNumber].Connection1 = cityNumber;

}

else

{

this[tmpCityNumber].Connection2 = cityNumber;

}

}

}

}

*Файл Population.cs*

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace Tsp

{

class Population : List<Tour>

{

/// <summary>

/// Private copy of the best tour found so far by the Genetic Algorithm.

/// </summary>

private Tour bestTour = null;

/// <summary>

/// The best tour found so far by the Genetic Algorithm.

/// </summary>

public Tour BestTour

{

set

{

bestTour = value;

}

get

{

return bestTour;

}

}

/// <summary>

/// Create the initial set of random tours.

/// </summary>

/// <param name="populationSize">Number of tours to create.</param>

/// <param name="cityList">The list of cities in this tour.</param>

/// <param name="rand">Random number generator. We pass around the same random number generator, so that results between runs are consistent.</param>

/// <param name="chanceToUseCloseCity">The odds (out of 100) that a city that is known to be close will be used in any given link.</param>

public void CreateRandomPopulation(int populationSize, Cities cityList, Random rand, int chanceToUseCloseCity)

{

int firstCity, lastCity, nextCity;

for (int tourCount = 0; tourCount < populationSize; tourCount++)

{

Tour tour = new Tour(cityList.Count);

// Create a starting point for this tour

firstCity = rand.Next(cityList.Count);

lastCity = firstCity;

for (int city = 0; city < cityList.Count - 1; city++)

{

do

{

// Keep picking random cities for the next city, until we find one we haven't been to.

if ((rand.Next(100) < chanceToUseCloseCity) && ( cityList[city].CloseCities.Count > 0 ))

{

// 75% chance will will pick a city that is close to this one

nextCity = cityList[city].CloseCities[rand.Next(cityList[city].CloseCities.Count)];

}

else

{

// Otherwise, pick a completely random city.

nextCity = rand.Next(cityList.Count);

}

// Make sure we haven't been here, and make sure it isn't where we are at now.

} while ((tour[nextCity].Connection2 != -1) || (nextCity == lastCity));

// When going from city A to B, [1] on A = B and [1] on city B = A

tour[lastCity].Connection2 = nextCity;

tour[nextCity].Connection1 = lastCity;

lastCity = nextCity;

}

// Connect the last 2 cities.

tour[lastCity].Connection2 = firstCity;

tour[firstCity].Connection1 = lastCity;

tour.DetermineFitness(cityList);

Add(tour);

if ((bestTour == null) || (tour.Fitness < bestTour.Fitness))

{

BestTour = tour;

}

}

}

}

}

*Файл TspEventArgs.cs*

using System;

using System.Collections.Generic;

using System.Windows.Forms;

using System.Drawing;

namespace Tsp

{

/// <summary>

/// Event arguments when the TSP class wants the GUI to draw a tour.

/// </summary>

public class TspEventArgs : EventArgs

{

/// <summary>

/// Default Constructor.

/// </summary>

public TspEventArgs()

{

}

/// <summary>

/// Constructor that sets all the properties.

/// </summary>

/// <param name="cityList">The list of cities to draw.</param>

/// <param name="bestTour">The tour that connects all the cities.</param>

/// <param name="generation">Which generation is this.</param>

/// <param name="complete">Is this the last update before we are done.</param>

public TspEventArgs(Cities cityList, Tour bestTour, int generation, bool complete)

{

this.cityList = cityList;

this.bestTour = bestTour;

this.generation = generation;

this.complete = complete;

}

/// <summary>Private copy of the list of cities.</summary>

private Cities cityList;

/// <summary>Public property for list of cities.</summary>

public Cities CityList

{

get

{

return cityList;

}

}

/// <summary>Private copy of the tour of the cities.</summary>

private Tour bestTour;

/// <summary>Public property for the tour of the cities.</summary>

public Tour BestTour

{

get

{

return bestTour;

}

}

/// <summary>Private copy for which generation this tour came from.</summary>

private int generation;

/// <summary>Public property for which generation this tour came from.</summary>

public int Generation

{

get

{

return generation;

}

set

{

generation = value;

}

}

/// <summary>Private copy indicating if the TSP algorithm is complete.</summary>

private bool complete = false;

/// <summary>Public property indicating if the TSP algorithm is complete.</summary>

public bool Complete

{

get

{

return complete;

}

set

{

complete = value;

}

}

}

}