14. Генетичні алгоритми

Генетичні алгоритми розв'язування задач з'явилися порівняно недавно. Часто їх розглядають як альтернативу до традиційних методів, оскільки генетичний алгоритм використовує принципово новий підхід до відшукання розв'язку задачі. Традиційні методи використовують складний і «мудрий» математичний апарат для того, щоб за мінімальну кількість кроків отримати якнайкращий результат. На противагу їм генетичний алгоритм мало турбується про оптимальні характеристики кандидатів на розв'язок задачі, проте генерує їх чимало і швидко для того, щоб відбракувати гірші і продовжити генерувати нові, враховуючи позитивний досвід, здобутий кращими кандидатами. Протягом процедури виконання такого алгоритму кандидати на розв'язок удосконалюються, поліпшують свої характеристики – еволюціонують від початкового наближення до оптимального розв'язку. Тому розв'язування задач за допомогою генетичних алгоритмів часто називають *еволюційним численням*.

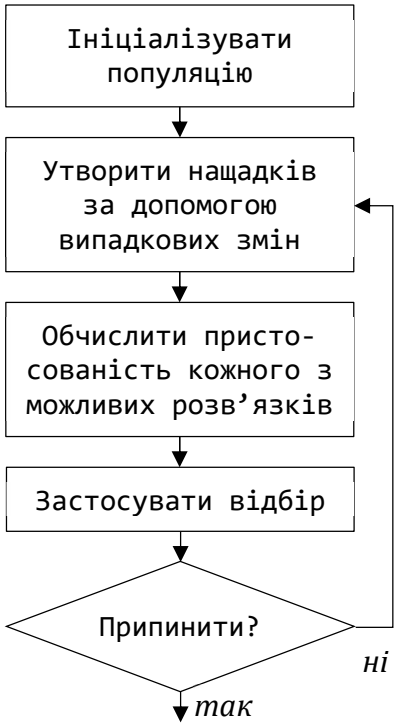

Рис. 13. Загальна схема роботи генетичного алгоритму

Природна еволюція розпочинається з первинної популяції організмів. Протягом багатьох поколінь випадкова мінливість і природний відбір формують поведінку та здібності особин популяції так, щоб максимально пристосуватись до вимог оточення. У найзагальніших термінах еволюцію можна описати як двокроковий ітераційний процес, що складається з випадкової зміни та подальшого відбору. Зв'язок між таким описом еволюції та оптимізаційним алгоритмом концептуально простий.

Як і за природної еволюції, алгоритмічний підхід започатковують з вибору первинної множини альтернативних розв'язків певної проблеми. Далі ці «батьківські» розв'язки генерують «нащадків» за допомогою вибраних засобів випадкової варіації. Усі розв'язки – і батьків, і нащадків – оцінюють з огляду на те, наскільки добре вони виконують поставлене завдання. Остаточно застосовують критерій вибору, щоб відкинути ті розв'язки, цінність яких є під рискою. Так само, як у природі «виживають найсильніші». Процес повторюють з відібраною множиною розв'язків з метою отримання наступних поколінь, аж доки не буде задоволено умову щодо отримання прийнятного розв'язку (рис. 13).

14.1. Сфера застосування

Щоб розв'язати певну проблему за допомогою традиційних алгоритмів оптимізації, дослідник має зробити низку припущень щодо способу оцінки придатності розв'язку. Наприклад, використати цільову функцію, індекс відповідності тощо. Такий вибір накладає свої обмеження на задачу. Наприклад, алгоритми лінійного програмування передбачають, що цільова функція теж є лінійною: є сумою зважених цільових доданків. За іншого традиційного підходу – градієнтного – шукають точку нуля градієнта, можливо, мінімум чи максимум цільової функції, а це вимагає, щоб вона була гладкою, диференційовною. Градієнтні методи неможливо застосувати, якщо цільова функція має стрибки чи злами.

Еволюційні алгоритми таких припущень не потребують. Важливо лише, щоб індекс відповідності дав змогу впорядкувати два конкурентні розв'язки, визначивши, який з них з

певних причин є ліпшим від іншого. Така невибагливість методу еволюційного числення дає змогу до його широкого застосування щодо тих задач, які не можна розв'язати традиційними числовими методами.

У реальному світі еволюційний підхід надає значні переваги. Адже традиційні методи використовують певні математичні моделі реальних задач. Як тільки окремі умови задачі змінюються (наприклад, вийшло з ладу обладнання на фабриці, змінилась вартість сировини чи змінився маршрут перевезень), традиційні розв'язки стають непридатними і всі обчислення треба виконувати з самого початку. На противагу цьому, в генетичному алгоритмі поточна популяція розв'язків зберігається як резервуар накопиченого знання, яке можна використати з метою пристосування до зміненого зовнішнього середовища.

Іншою перевагою еволюційного підходу є здатність за невеликий час генерувати хороші розв'язки – досить швидко для того, хто використовує цей підхід.

Щоб реалізувати еволюційний підхід, необхідно виконати такі підготовчі кроки:

- вибрати зображення розв'язку;
- винайти оператор випадкових змін;
- задати правило виживання розв'язків;
- ініціалізувати популяцію.

Внаслідок їхнього виконання, еволюційний алгоритм генеруватиме розв'язки і покращуватиме їх від покоління до покоління.

Проілюструємо ці кроки на прикладі класичної задачі відшукання туру комівояжера, яку ми розглянули у попередньому параграфі, але для великої кількості міст. Йтиметься про обчислення розв'язку для дещо спрощеної моделі, а саме: задано натуральне n – кількість міст, координати цих міст на «карті» – декартовій площині; вважається, що кожну пару міст з'єднує дорога, довжину якої можна обчислити як відстань між точками координатної площини. Потрібно відшукати замкнуту ламану найменшої довжини, що з'єднує всі міста на карті, причому кожне місто є кінцевим вузлом рівно двох сусідніх ланок ламаної. Йтиметься не про обчислення оптимального розв'язку, а тільки про дуже якісну евристику.

14.2. Підготовчі кроки

Щоб зобразити можливий розв'язок задачі засобами програми, потрібно вибрати придатну для цього структуру даних. Вона мусить бути придатною для зображення *будь-якого* розв'язку і забезпечувати можливість легко його *оцінити*. Немає одного найліпшого вибору зображення розв'язку, тому тут потрібна певна винахідливість.

Один з можливих і досить зручних способів *зображення туру* комівояжера – перестановка номерів міст. Така перестановка описує порядок відвідування міст, і будь-яка з них є більш чи менш придатним розв'язком. (Розв'язки задачі комівояжера для $n = 5$ у попередньому параграфі записано саме за допомогою перестановок). Самі перестановки можна зберігати у масиві або в зв'язному списку.

Отже, зображатимемо можливі тури за допомогою перестановок, записаних у масиви відповідного розміру. «Згенерувати новий розв'язок» означатиме створити нову перестановку. Далі потрібно визначити цільову функцію – *засіб оцінки* потенційних розв'язків. Подорож має бути найкоротшою, тому «ціною» розв'язку є довжина туру. Перевагу віддаватимемо коротшому турові порівняно з довшим. Цільова функція обчислюватиме довжину ламаної за заданими координатами вузлів. Можна використати й інші способи побудови цільової функції. Наприклад, якщо б картою був граф, як у попередньому параграфі, то для обчислення довжини можна було б використати елементи матриці суміжності.

Звичайно, завдання генерування та оцінювання турів може бути складнішим, якщо враховувати додаткові вимоги реального життя такі, як потреба зменшити час подорожі в

години пік, або відвідувати певного покупця тільки після обіду або після інших покупців. Однак складність задачі робить еволюційний алгоритм придатнішим, бо швидко вилучає задачу зі сфери застосування традиційних оптимізаційних алгоритмів. Нехай для нашого прикладу мірою якості буде просто сумарна довжина туру.

Наступним кроком є конструювання *оператора* (або операторів) *випадкових змін*, який з батьківськими генеруватиме породжені розв'язки. Для цього можна використати багато різних способів. У природі існує два основні способи відтворення: статевий і нестатевий. У статевому відтворенні двоє батьків одного виду обмінюються генетичним матеріалом, який рекомбінується у потомстві. Безстатеве відтворення за змістом є клонуванням, однак під час передачі генетичного матеріалу від батька до нащадків цей матеріал може зазнавати різноманітних змін. Такі оператори відтворення можна змоделювати в еволюційному алгоритмі. Хоча можна міркувати ширше і запропонувати оператори, для яких не існує аналогів у природі. Наприклад, рекомбінувати генетичний матеріал від трьох батьків, чи ще якісь.

Остаточний успіх еволюційного алгоритму залежить від того, на скільки добре узгоджені оператор мутацій, зображення розв'язку та цільова функція. Різні оператори можуть виявитися кращими у різних умовах. Так само як і з зображенням розв'язку, не існує одного оператора, найкращого для всіх задач.

Розглянемо можливі варіанти генерування потомства для задачі комівояжера, що використовує перестановки (рис. 14). Тут «генами» розв'язку є номери міст. Безстатеві способи полягають у випадковому виборі двох міст з батьківського туру й обміні їх місцями у породженому турі (тип I) або зміні порядку слідування між ними на протилежний (тип II). Такі способи завжди генеруватимуть допустимі тури: у них буде наявне кожне з міст, і кожне – один раз. Спеціалізовані безстатеві оператори можуть вносити мутації не у весь ланцюжок міст, а в окрему його частину. Детальний огляд публікацій, присвячених різноманітним реалізаціям генетичного алгоритму розв'язування задачі комівояжера, можна знайти в [8].

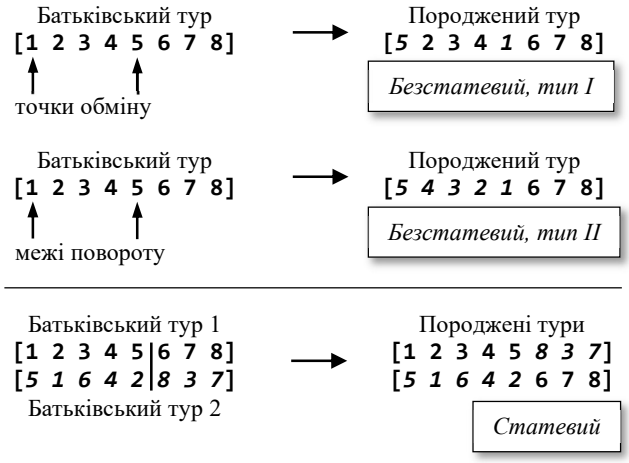

Рис. 14. Оператори генерування нових поколінь для задачі комівояжера (n=8)

Статевий спосіб з'єднує частини (виконує конкатенацію) двох батьківських турів, розітнутих у випадково вибраній точці. Він може генерувати недопустимі тури, які містять деякі міста двічі і не містять інших окремих міст. Це не означає, що в цій задачі не можна використовувати статевого розмноження – просто потрібно передбачити в операторі додаткові дії з відбракування чи виправлення генерованих турів. Виявлення дефектів геному суттєво впливає на швидкодію алгоритму, тому у нашому прикладі використано описані безстатеві оператори.

Третім кроком є визначення *правила вибору* тих розв'язків, які будуть виживати, щоб стати батьками для наступних поколінь. Так само як з варіаціями, можна розглянути багато форм вибору. Наприклад, одне просте правило: виживають найпристосованіші! За ним тільки жменька найкращих розв'язків популяції залишається, тоді як всі інші вилучаються. Такий підхід легко реалізувати програмно, якщо використати для зберігання популяції впорядковану колекцію. Тоді виживання залишає в ній задану кількість початкових елементів і вилучає хвіст.

Альтернативою може бути застосування змагального підходу, під час якого випадково створені пари (чи *m*-ки) розв'язків змагаються за виживання. Так само, як у професійному спорті, де слабший гравець чи команда виграють, бо їм пощастило у змаганні, слабші розв'язки в популяції часом виживають кілька поколінь, якщо використано такий підхід. Це може бути перевагою у складних проблемах, де буває легше поліпшити розв'язок, змінюючи гірший, ніж змінюючи тільки найкращі. Можливостей безліч, і будь-яке правило, яке загалом віддає перевагу у виживанні кращому розв'язку над гіршим, має сенс.

Останнім кроком є *вибір початкової популяції*. Якщо невідомо, як розв'язувати задачу, то розв'язки вибирають випадково з простору можливих розв'язків. Для задачі комівояжера це означало б випадкове генерування низки перестановок *n* цілих чисел, кожна з яких зображає можливий розв'язок.

Альтернативою може бути залучення до початкової популяції розв'язків, близьких до добрих, отриманих за допомогою деякого іншого алгоритму чи з урахуванням апріорної інформації щодо задачі. Якщо ці розв'язки виявляться вартими уваги, то вони виживатимуть і продукуватимуть нові покоління. Якщо ж ні, то разом з іншими – гіршими розв'язками – вони будуть вимирати. Ми використали випадкове генерування початкової популяції.

Зрозуміло, що спосіб формування початкової популяції, оператори мутації та правило виживання є «алгоритмічними параметрами» генетичного алгоритму. Ми можемо будувати різні варіанти алгоритму за рахунок зміни цих параметрів. Наприклад, безстатевий оператор другого типу було впроваджено після низки числових експериментів, які засвідчили, що оператор першого типу не в змозі виправити «петлі» – перехрещені відтинки туру. Окрім згаданих параметрів, алгоритм має ще й числові: обсяг популяції та кількості мутацій кожної особини. Їх підбирають експериментально.

14.3. Програмна реалізація

Реалізуючи генетичний алгоритм відшукання туру комівояжера програмно, ми не ставили собі за мету знайти розв'язки якоїсь конкретної задачі, тому її формулювання досить умовне.

Задача 51. *Знайти тур комівояжера, який проходить через N міст, розташованих на карті розміром Width×Height, і між кожною парою міст є пряма дорога.*

Очевидно, що за такого формулювання тур комівояжера можна зобразити за допомогою замкнутої ламаної, вершинами якої є міста з заданими координатами. Цільовою функцією виберемо довжину цієї ламаної. Кожному місту присвоїмо порядковий номер від 0 до *N*–1, тоді розв'язком задачі буде така перестановка номерів, якій відповідає найкоротша з усіх можливих ламана.

Карту в програмі можна моделювати колекцією точок координатної площини. Багато графічних бібліотек містять тип *Point* для підтримки екранних координат. Якщо плануємо обійтися власними силами, то можемо оголосити простий тип, як зображено нижче.

```
// структура поєднує абсцису та ординату точки на площині
struct Point
{
	double x;
	double y;
	Point(double a = 0, double b = 0) :x(a), y(b) {}
};

// оператор виведення перевантажено заради зручності
std::ostream& operator<<(std::ostream& os, const Point& P);
```

```
{
	(os << " (").width(4);
	(os << P.x << ',').width(4);
	os << P.y << ')';
	return os;
}
```

Тоді карта – послідовність значень типу *Point*, у найпростішому випадку – масив. Індекси елементів масиву збігаються з номерами міст. Значення можна завантажувати з текстового файлу чи генерувати програмно за допомогою давача випадкових чисел.

Спроектуємо тип даних *Tour*, що моделюватиме кандидата на розв'язок задачі. Він має «знати» координати міст, зберігати перестановку їхніх номерів і повідомляти свою довжину. Довжину використовують досить часто для порівняння турів, тому є сенс обчислити її один раз і зберігати разом з перестановкою. Тур міг би також створювати потомство – реалізувати оператори мутації. Усе це можна описати мовою C++ за допомогою оголошення відповідного класу.

Турів, екземплярів такого класу, в програмі існуватиме багато. Чи доцільно в кожному з них зберігати копії координат міст? Очевидно, що ні. Достатньо було б зберегти один екземпляр карти, доступ до якого мав би кожен екземпляр класу. З цією метою можемо використати статичне поле класу, значення якого потрібно задати перед створенням екземплярів класу. Оголошення класу *Tour* матиме такий вигляд.

```
// Клас інкапсулює послідовність m_order відвідання міст, координати m_towns яких
// задано зовні. Має методи "мутування" для створення нових турів, вміє обчислити
// і зберігає довжину маршруту. Тури порівнюють за довжиною.
class Tour
{
private:
	static Point* m_towns;   // координати міст
	static int m_towns_count;
	int* m_order;            // перестановка номерів
	double m_length;         // відповідна довжина
	void CalcLength();       // метод обчислення довжини
	// новий тур містить перестановку, обчислену зовні
	Tour(int* permutation) :m_order(permutation)
	{
		CalcLength();
	}
public:
	// метод викликають перед початком використання класу
	static void InitTowns(Point* t, int n)
	{
		m_towns = t; m_towns_count = n;
	}
	~Tour() { delete[] m_order; }
	// за замовчуванням порядок відвідання - послідовний: order_[i] = i
	// за бажання можна перемішати порядок відвідання випадково
	Tour(bool permuted = false);
	// копіювання та присвоєння потрібні для роботи з контейнерами
	Tour(const Tour& t);
	Tour& operator=(const Tour& t);
	// метод отримання довжини повідомляє вже обчислене значення
	double Length() const { return m_length; }
	// методи мутації турів: обмін і поворот
	Tour Mutate();
	Tour Rotate();
```

```
        // оператори полегшують використання екземплярів класу
        bool operator<(const Tour& other) const
        {
                return this->m_length < other.m_length;
        }
        friend std::ostream& operator<<(std::ostream& os, const Tour& t);

        // Результат - побудована послідовність міст
        Point* Route() const;
};
```

Метод *CalcLength* оголошено приватним, бо він є частиною реалізації: кожен новостворений тур один раз обчислює довжину, зберігає її в полі *m_length* та готовий повідомити її за першим запитом (для отримання довжини використовують метод *Length*). Метод *CalcLength* не складний алгоритмічно, але дуже важливий з погляду правильного використання класу *Tour*. Саме він перевіряє, чи було ініціалізовано статичні поля класу, щоб можна було створити екземпляр, повідомляє про проблему запуском винятку, якщо таку виявлено.

```
// Довжина маршруту - сума довжин прямолінійних відрізків між сусідніми містами
// маршруту. Довжина відрізку - відстань між точками на координатній площині.
void Tour::CalcLength()
{
    if (m_towns == nullptr || m_towns_count == 0)
    {
        m_length = 0.0;
        throw std::runtime_error(
            "The static members of Tour class are not initialized properly");
    }
    m_length =
        sqrt(pow(m_towns[m_order[m_towns_count-1]].x - m_towns[m_order[0]].x, 2)
        + pow(m_towns[m_order[m_towns_count - 1]].y - m_towns[m_order[0]].y, 2));
    for (int i = 1; i < m_towns_count; ++i)
        m_length += sqrt(pow(m_towns[m_order[i-1]].x - m_towns[m_order[i]].x, 2)
            + pow(m_towns[m_order[i - 1]].y - m_towns[m_order[i]].y, 2));
}
```

Єдина особливість методу полягає в тому, що для індексування елементів масиву *Towns* використовують елементи перестановки номерів міст.

Конструктор *Tour(bool)* за замовчуванням генерує послідовний порядок номерів, а його виклик з аргументом *true* – випадкову перестановку цих номерів. Для перемішування конструктор використовує стандартний алгоритм *random_shuffle* бібліотеки STL. Конструктор *Tour(int*)* потрібен у методах створення потомства. Він покладається на те, що методи передають йому перестановку правильного розміру, тому його оголошено закритим від зовнішнього використання. Обидва конструктори викликають метод обчислення довжини.

```
// За замовчуванням міста відвідують послідовно від першого до останнього
// або використовують випадкову перестановку послідовності міст
Tour::Tour(bool permuted)
{
        m_order = new int[m_towns_count];
        for (int i = 0; i < m_towns_count; ++i) m_order[i] = i;
        if (permuted) std::random_shuffle(m_order, m_order + m_towns_count);
        CalcLength();
}
```

```
Tour::Tour(int* permutation) : m_order(permutation) { CalcLength(); }
```

У класі, що використовує динамічну пам'ять, обов'язково визначають деструктор (його текст є в оголошенні класу), конструктор копіювання та оператор присвоєння. Усі екземпляри класу *Tour* мають масиви перестановок однакового розміру, що спрощує копіювання: не потрібно знищувати старий і створювати новий динамічний масив.

```
// Конструктор копіювання
Tour::Tour(const Tour& t) :m_length(t.m_length)
{
	m_order = new int[m_towns_count];
	for (int i = 0; i < m_towns_count; ++i) m_order[i] = t.m_order[i];
}
// Оператор присвоєння
Tour& Tour::operator=(const Tour& t)
{
	if (this != &t)
	{
		for (int i = 0; i < m_towns_count; ++i) m_order[i] = t.m_order[i];
		m_length = t.m_length;
	}
	return *this;
}
```

Кожен тур може створювати потомство за допомогою методів, що реалізують оператори мутацій. Оператор повороту має враховувати два випадки розташування випадково згенерованих точок повороту. Якщо початкова точка менша ніж кінцева, як на рис. 14, то виконуємо звичайний обмін елементів відрізку масиву між цими точками. У протилежному випадку треба уявити, що тур насправді є не відрізком, а кільцем, і відрізок, який треба обернути, розташовано від початкової точки до кінця масиву і далі від початку масиву до кінцевої точки. Саме на такому «розімкнутому відрізку» виконуватимемо обміни.

```
// Проста мутація: обмін місцями двох випадкових міст у послідовності
Tour Tour::Mutate()
{
	int* ord = new int[m_towns_count];  // копія початкової перестановки
	for (int i = 0; i < m_towns_count; ++i) ord[i] = m_order[i];
	int k = std::rand() % m_towns_count;
	int n = std::rand() % m_towns_count;
	int to_swap = ord[k];               // зазнає змін
	ord[k] = ord[n];
	ord[n] = to_swap;
	return Tour(ord);                   // створення туру завершує конструктор
}
// Мутація напряму змінює на протилежний напрям руху між двома випадковими містами
Tour Tour::Rotate()
{
	int* ord = new int[m_towns_count];
	for (int i = 0; i < m_towns_count; ++i) ord[i] = m_order[i];
	int k = std::rand() % m_towns_count; // початкова точка повороту
	int n = std::rand() % m_towns_count; // кінцева точка
	if (k == n) return Tour(true);
	while (k < n)                        // "звичайний" поворот
	{
		int to_swap = ord[k];
```

```
            ord[k] = ord[n];
            ord[n] = to_swap;
            ++k; --n;
      }
      while (k > n)                    // поворот "по колу"
      {
            int to_swap = ord[k];
            ord[k] = ord[n];
            ord[n] = to_swap;
            ++k; k %= m_towns_count;
            --n; n = (n < 0) ? m_towns_count - 1 : n;
      }
      return Tour(ord);
}
```

Завершимо опис типу *Tour* допоміжним методом *Route* формування послідовності міст у порядку заданої перестановки та оператором виведення туру в потік. Їх зручно використовувати для відображення обчислених турів.

```
Point* Tour::Route() const
{
      Point* r = new Point[m_towns_count + 1];
      for (int i = 0; i < m_towns_count; ++i)
            r[i] = m_towns[m_order[i]];
      r[m_towns_count] = r[0];
      return r;
}
std::ostream& operator<<(std::ostream& os, const Tour& t)
{
      os << " Length = " << t.m_length << "\n[";
      for (int i = 0; i < Tour::m_towns_count; ++i) os << ' ' << t.m_order[i];
      os << " ]\n";
      return os;
}
```

Тип даних для моделювання туру повністю готовий. Використаємо його в класі *Solver*, що реалізує генетичний алгоритм розв'язування задачі комівояжера.

Важливим питанням є спосіб зберігання популяції турів. Кожен з них має свою «цінність», яка, в нашому випадку, визначається його довжиною. Нам потрібно вилучати гірші тури і залишати кращі. Як ми будемо це робити? Змагальний підхід до відбору потребує додаткових затрат на розбиття всієї популяції на конкурсні групи, тому розглянемо простіший варіант відбору: ранжуватимемо всі тури популяції за зростанням їхньої довжини і залишатимемо в ній тільки певну кількість кращих «особин», відбраковуючи усі гірші. За такого способу відбору зручно зберігати популяцію у впорядкованому вигляді. У цьому випадку кожен новий член популяції треба додавати до неї, не порушуючи впорядкованості. Яка структура даних підходить для цього найліпше? Очевидно, дерево пошуку або впорядкований лінійний список, однак дерево працює швидше. Не будемо створювати власний контейнер, оскільки бібліотека STL пропонує декілька стандартних контейнерів, що підтримують упорядкування елементів за допомогою дерева. Можемо використати *set* або *multiset*. Відмінність лише в тому, чи дозволяти популяції містити два тури однакової довжини. Якщо так, то варто зупинити вибір на *multiset*.

Щоб утворити нове покоління, потрібно вирішити, як зберігати потомство. Нові особини не можна долучати до популяції, доки старше покоління не закінчить відтворення,

щоб різні покоління не переплуталися, бо тоді важко буде простежити, хто зі старших уже дав потомство, а хто – ні. Тому використаємо два контейнери: для накопичення потомства потрібний впорядкований контейнер, а для зберігання відібраного покоління, що має фіксовану кількість особин, підійде звичайний масив. У такому випадку виживання означатиме перенесення до масиву заданої кількості найкращих з впорядкованого контейнера та його очищення.

Щоб створити початкову популяцію, потрібно задати її розмір і згенерувати кожну особину. Вибір розміру популяції важко обґрунтувати строго. Більша популяція містить більше розмаїтого генетичного матеріалу, проте потребує більшого часу на опрацювання її особин, тому на практиці доводиться вибирати якусь «золоту середину», наприклад, враховуючи експериментальний досвід. Ми вже згадували, що генетичний алгоритм має й інші числові параметри: кількість мутацій, гранична кількість поколінь – усіх їх зробимо полями даних класу *Solver*. Методами класу будуть описані раніше кроки алгоритму.

```
class Solver
{
private:
    // координати міст, які треба відвідати
    Point* m_towns;
    int m_towns_count;

    // Розмір популяції (найпристосованіших)
    int m_population_size;
    // Кількість простих мутацій
    int m_mutation_number;
    // Кількість мутацій напряму
    int m_rotation_number;
    // Максимальна кількість поколінь
    int m_generations_limit;
    // Усе потомство автоматично сортуватиметься за зростанням довжини туру
    std::multiset<Tour> m_population;
    // Популяція найпристосованіших
    Tour* m_top_tours;
    //генерування початкової популяції:масиву випадкових перестановок номерів міст
    void StartPopulation();
    // генерування нового покоління
    void NextGeneration();
    // Природний відбір
    void Survive();
public:
    // інформування про хід обчислень
    std::function<void(int, double)> m_show;
    Solver(Point* map, int size);
    Solver(Point* map, int size, int population_size,
        int mutations, int turns, int max_generation);
    ~Solver() { delete[] m_top_tours; }
    // запуск обчислень
    int Solve(const bool& stop_required);
    // знайдений розв'язок
    Tour& Best() const { return m_top_tours[0]; }
};
```

Окремої уваги заслуговує поле функціонального типу *m_show*. Його передбачено для того, щоб головний метод *Solve* за допомогою викликів *m_show* міг інформувати користувача про хід обчислень. Стандартний тип *std::function* інкапсулює адресу об'єкта, який можна

викликати, як функцію, наприклад, лямбда-виразу. Згодом ми продемонструємо, як його налаштувати, щоб отримувати дані в зручному вигляді.

Конструктор класу *Solver* запам'ятовує координати розташування міст, числові параметри алгоритму та налаштовує клас *Tour* викликом відповідного статичного методу.

```
Solver::Solver(Point* map, int size, int population_size,
            int mutations, int turns, int max_generation)
    :m_towns(map), m_towns_count(size), m_population_size(population_size),
    m_mutation_number(mutations), m_rotation_number(turns),
    m_generations_limit(max_generation)
{
    Tour::InitTowns(m_towns, m_towns_count);
    m_top_tours = new Tour[m_population_size];
    m_show = nullptr;
}
```

Отримати початкову популяцію досить просто, адже конструктор турів сам уміє генерувати випадкові перестановки

```
void Solver::StartPopulation()
{
    for (int i = 0; i < m_population_size; ++i) m_top_tours[i] = Tour(true);
}
```

Метод створення потомства послідовно перебирає всіх членів популяції та застосовує до них оператори мутації задану кількість разів. Завдяки додаванню до впорядкованого контейнера нащадки ранжуються автоматично.

```
void Solver::NextGeneration()
{
    for (int i = 0; i < m_population_size; ++i)
    {
        // Початкові члени популяції мають шанс продовжити існування в
        // наступному поколінні
        m_population.insert(m_top_tours[i]);
        // Кожен член популяції є джерелом для мутованих "нащадків":
        // для нього потрібно виконати задане число простих мутацій ...
        for (int k = 0; k < m_mutation_number; ++k)
        {
            m_population.emplace(m_top_tours[i].Mutate());
        }
        // ... і мутацій напряму
        for (int k = 0; k < m_rotation_number; ++k)
        {
            m_population.emplace(m_top_tours[i].Rotate());
        }
    }
}
```

Метод виживання переносить найкращих до стабільного масиву й очищає впорядкований контейнер для наступного покоління.

```
void Solver::Survive()
{
    std::multiset<Tour>::iterator it = m_population.begin();
```

```
    for (int i = 0; i < m_population_size && it != m_population.end(); ++i, ++it)
    {
        m_top_tours[i] = *it;
    }
    m_population.clear();
}
```

Усі складові генетичного алгоритму збирає воєдино *Solve* – головний метод класу.

```
int Solver::Solve(const bool& stop_required)
{
    StartPopulation();
    int generation_number = 0;
    while (!stop_required && generation_number < m_generations_limit)
    {
        NextGeneration();
        Survive();
        ++generation_number;
        if (generation_number % 50 == 0)
        {
            if (m_show != nullptr) m_show(generation_number, Best().Length());
        }
    }
    return generation_number;
}
```

Кількість ітерацій у методі обмежена значенням *m_generations_limit*, проте обчислення можуть бути доволі тривалими за великого розміру популяції. Щоб дати змогу користувачеві зупинити обчислення достроково, метод приймає як аргумент змінну *stop_requiered*. Зверніть увагу: не значення змінної, а саме змінну головної програми! Перед кожною ітерацією він перевіряє, чи не змінив користувач її значення на *true*, вимагаючи зупинити процес. Як результат метод повертає кількість перевірених поколінь. Раз на п'ятдесят поколінь метод викликає *m_show* і передає йому номер покоління та довжину найкращого туру. Користувач може розпоряджатися цими даними на власний розсуд. Знайдений розв'язок задачі можна отримати методом *Solver::Best()*.

14.4. Консольна програма

Щоб продемонструвати використання розроблених класів, потрібна нова програма. Почнемо з найпростішого варіанту – з консольної програми. Вхідних даних потрібно задати немало, тому запишемо їх до текстового файлу, наприклад, як зображено нижче.

```
600   // ширина карти
400   // висота карти
120   // кількість міст
 20   // розмір популяції
  8   // кількість обмінів міст
  5   // кількість поворотів маршруту
3000  // максимальна кількість поколінь
```

Коментарі наведено заради зручності, їх пропускають під час читання даних у програмі. Координати міст згенеруємо програмно за допомогою давача випадкових чисел.

Після створення екземпляра класу *Solver* налаштуємо його функціональне поле *m_show* за допомогою лямбда-виразу. Він виводитиме на консоль отримані від методу *Solve* дані та перевірятиме стан клавіатури: чи не натиснув користувач якусь клавішу, щоб

зупинити цикл обчислень. Вивести на консоль знайдений розв'язок допоможе оголошений раніше оператор виведення туру.

```
int main()
{
    SetConsoleOutputCP(1251);
    std::ifstream fin("input_data.txt");
    std::string comment;
    std::cout << " *** Еволюційне числення в задачі комівояжера\n\n";
    // завантаження вхідних даних
    int map_width, map_height, towns_number, population_size,
        mutations, turns, max_gen;
    fin >> map_width; std::getline(fin, comment);
    std::cout << map_width << '\t' << comment << '\n';
    fin >> map_height; std::getline(fin, comment);
    std::cout << map_height << '\t' << comment << '\n';
    fin >> towns_number; std::getline(fin, comment);
    std::cout << towns_number << '\t' << comment << '\n';
    fin >> population_size; std::getline(fin, comment);
    fin >> mutations; std::getline(fin, comment);
    fin >> turns; std::getline(fin, comment);
    fin >> max_gen; std::getline(fin, comment);
    std::cout << max_gen << '\t' << comment << '\n';

    // генерування координат міст на карті
    srand(time(0));
    Point* map = new Point[towns_number];
    std::cout << "Міста:\n";
    for (int i = 0; i < towns_number; ++i)
    {
        map[i] = Point(rand() % map_width + 1., rand() % map_height + 1.);
        std::cout << map[i];
    }
    std::cout << "\n\n";

    // розв'язування
    Solver* solver =
       new Solver(map, towns_number, population_size, mutations, turns, max_gen);
    bool need_stop = false;    // прапорець вимоги про зупинку обчислень
    solver->m_show = [&need_stop](int gen, double len)
    {
        std::cout << " * покоління: " << gen << " довжина: " << len << '\n';
        need_stop = _kbhit();
    };
    std::cout << "Щоб зупинити обчислення, натисніть довільну клавішу...\n";
    max_gen = solver->Solve(need_stop);

    std::cout << "Досліджено " << max_gen
              << " поколінь. Знайдено розв'язок:\n" << solver->Best() << '\n';
    delete solver;
    delete[] map;
}
```

Під час одного з запусків для наведених вхідних даних програма вивела такі результати:

```
*** Еволюційне числення в задачі комівояжера

600         // ширина карти
400         // висота карти
120         // кількість міст
3000        // максимальна кількість поколінь

Міста:
 ( 503, 329) ( 394, 210) (  96, 363) ( 522, 176) (  92, 360) (  64, 389) ( 597, 249)
( 574, 187) (  95, 132) (  54,  56) ( 159,  17) ( 257, 231) (  30,   9) ( 326, 344)
(  10, 191) ( 421, 262) ( 589, 307) ( 373, 266) (  51,  53) ( 531,  62) ( 358, 343)
( 163, 181) ( 180, 390) ( 420, 377) ( 436, 138) ( 240, 373) (  84, 364) ( 489, 117)
( 214, 360) ( 282,  42) (  61,  83) ( 484, 386) ( 227, 350) ( 568, 231) ( 100, 156)
( 453, 278) ( 112, 224) ( 562,  20) ( 596, 203) ( 518, 124) ( 523, 293) (  30,  76)
( 564,  57) ( 287, 385) (  92, 246) (  93, 271) (  93,  66) ( 451,  54) ( 318, 210)
( 124, 321) ( 333, 264) ( 225, 240) ( 530, 189) ( 299, 297) ( 574, 351) ( 364, 171)
( 533, 300) ( 457, 227) ( 151, 127) ( 531, 227) ( 293, 154) ( 503, 191) ( 482,  37)
( 114, 297) ( 312, 246) ( 256, 384) ( 205, 322) ( 335, 117) ( 269, 291) ( 120, 110)
( 338,  27) ( 238, 110) ( 152,   7) ( 386, 292) ( 230, 135) (  21,  51) ( 363,  77)
( 223, 195) ( 565, 148) ( 377, 101) ( 600, 323) ( 150, 301) ( 396, 370) ( 527, 123)
( 128,  28) ( 207, 314) ( 366, 342) ( 275, 349) ( 247, 367) ( 287, 198) ( 544, 296)
( 141, 217) ( 557, 182) ( 407, 209) ( 205,   6) ( 471,  65) (  22,  16) ( 460, 350)
( 547, 399) ( 496,  59) (  51, 240) ( 377,  69) ( 508, 169) ( 565, 144) ( 359, 108)
(  33, 177) ( 302, 284) (  98,  30) ( 275,  46) ( 405, 354) (  27, 212) ( 277,  38)
( 491,   6) ( 412, 140) ( 497, 194) ( 124, 174) (  94,  70) ( 485, 199) ( 267, 203)
(  82, 335)

Щоб зупинити обчислення, натисніть довільну клавішу...
 * покоління: 50 довжина: 14491.7
 * покоління: 100 довжина: 10124.3
 * покоління: 150 довжина: 8494.58
 * покоління: 200 довжина: 7241.99
 * покоління: 250 довжина: 6526.67
 * покоління: 300 довжина: 5984.39
 * покоління: 350 довжина: 5642.35
 * покоління: 400 довжина: 5485.91
 * покоління: 450 довжина: 5267.73
 * покоління: 500 довжина: 5148.96
 * покоління: 550 довжина: 5008.78
 * покоління: 600 довжина: 4907.81
 * покоління: 650 довжина: 4788.35
 * покоління: 700 довжина: 4741.89
 * покоління: 750 довжина: 4683.86
 * покоління: 800 довжина: 4651.21
 * покоління: 850 довжина: 4635.62
 * покоління: 900 довжина: 4632.46
 * покоління: 950 довжина: 4631.63
 * покоління: 1000 довжина: 4629.35
 * покоління: 1050 довжина: 4618.63
 * покоління: 1100 довжина: 4618.63
 * покоління: 1150 довжина: 4618.63

Досліджено 1200 поколінь. Знайдено розв'язок:
 Length = 4618.63
[ 44 36 115 21 91 45 63 81 49 119 4 2 26 5 22 28 32 25 88 65 43 87 13 20 86 109 82
23 97 73 50 48 64 106 53 68 66 85 51 77 11 118 89 55 113 93 1 17 15 35 57 117 114
```

```
61 102 3 52 92 33 59 90 56 40 0 31 98 54 80 16 6 38 7 78 103 83 39 27 24 99 19 42 37
112 62 95 47 101 70 76 79 104 67 60 74 71 108 29 111 94 10 72 84 107 12 96 75 41 18
9 30 46 116 69 58 8 34 105 14 110 100 ]
```

Після 1050-го покоління довжина туру перестала зменшуватися, обчислення було зупинено. Різні запуски програми дають різні розв'язки, близькі за довжиною. На це і варто було сподіватися, адже генетичний алгоритм знаходить евристику, а не точний розв'язок. Висока швидкість виконання програми дає змогу провести низку експериментів і обрати серед знайдених найкращу евристику.

Очевидно, що виведення на консоль не дає повної уяви про розташування міст і конфігурацію знайденого туру. Хотілося б побачити карту і маршрут на ній у графічному поданні. З цією метою можна використати довільну графічну бібліотеку. Класи *Tour* і *Solver* спроектовано так, що кожен з них виконує лише власне завдання і практично не залежать від середовища використання. Покажемо це на прикладі в наступному параграфі.

14.5. Віконний застосунок

Операційна система Windows надає графічне середовище для виконання застосунків, тому достатньо написати віконний застосунок, що запускатиме описаний раніше генетичний алгоритм. Є кілька способів написання віконних програм мовою C++. Ми могли б безпосередньо взаємодіяти з функціями операційної системи через API Win32, або використати котрусь з бібліотек для створення застосунків: Microsoft Foundation Classes (MFC), Windows Forms, Qt тощо. Кожен зі способів потребує від програміста вивчення додаткових засобів, опис яких виходить далеко за межі цього посібника. Ми використаємо MFC і опишемо найважливіші моменти створення віконного застосунку.

Microsoft Foundation Classes – класична бібліотека мовою C++, що утворює тонку об'єктну оболонку навколо API Win32, суттєво полегшує процес написання програм. Це давня бібліотека, проте й сучасні версії Visual Studio підтримують її, а в інтернеті щотижня з'являються публікації, присвячені використанню MFC.

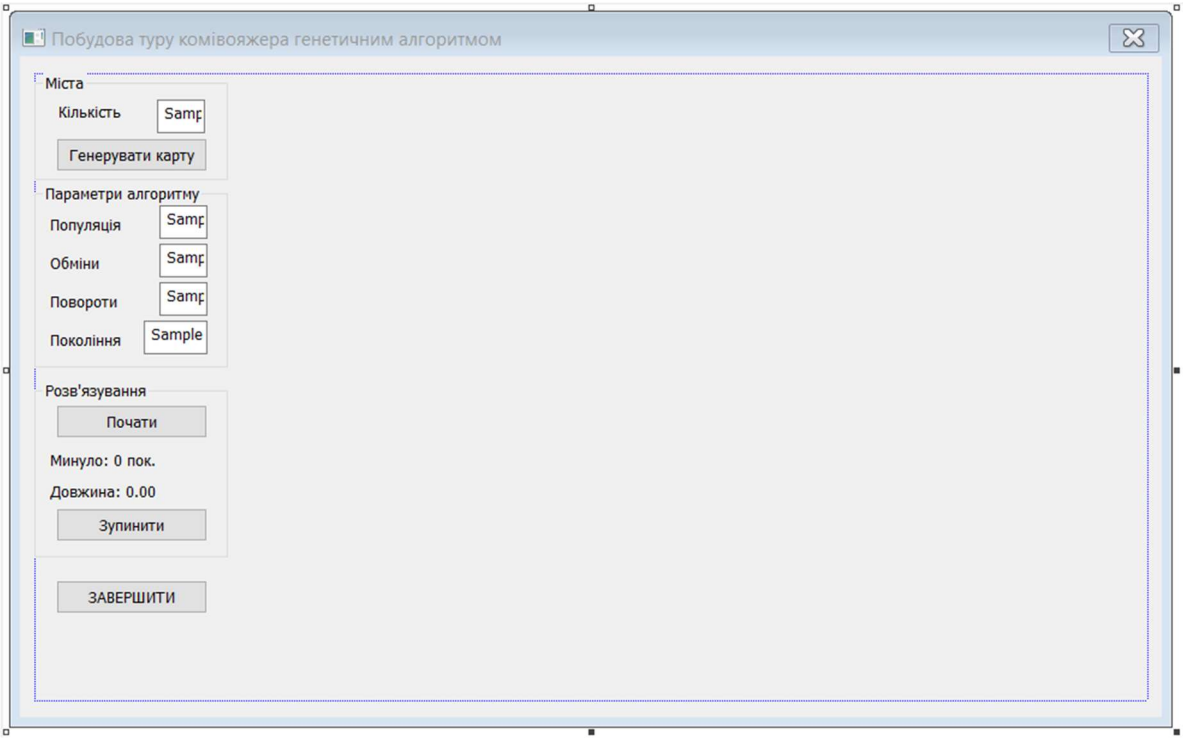

Рис. 15. Вікно застосунку на етапі проектування

Ми створимо застосунок найпростішого вигляду – заснований на використанні вікна діалогу. Класи *Tour* і *Solver* використаємо майже без змін: достатньо замінити власний тип *Point* на *CPoint*, що входить до складу MFC. До стандартного вікна, запропонованого середовищем програмування, додамо низку елементів керування: написи, рядки введення числових параметрів задачі, кнопки. Вікно на етапі проектування зображено на рис. 15. Вільну частину вікна використовуватимемо для виведення карти і найкращого зі знайдених турів. Розмір карти прив'язано до розмірів вікна.

Під час написання віконного застосунку з використанням стандартних бібліотек (MFC чи іншої) значну частину коду створює середовище програмування. Зокрема, якщо проєкт назвати *EvoSalesmanTour*, студія створить клас *CEvoSalesmanTourDlg*, відповідальний за відображення вікна та взаємодію з ним. Програмістові залишиться доповнити його оголошення полями даних, потрібними для взаємодії з елементами керування, та визначити методи опрацювання подій натискання на кнопки, перемальовування вікна тощо. Загалом програмування застосунку полягає у створенні методів реагування на події інтерфейсу користувача, адже Windows – система, керована потоком подій.

Після виконання всіх доповнень оголошення класу *CEvoSalesmanTourDlg* матиме вигляд, зображений нижче.

```
class CEvoSalesmanTourDlg : public CDialogEx
{
public:
	CEvoSalesmanTourDlg(CWnd* pParent = NULL);	// standard constructor
// Dialog Data
	enum { IDD = IDD_EVOSALESMANTOUR_DIALOG };
protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV support

// Implementation
protected:
	HICON m_hIcon;

	// Generated message map functions
	virtual BOOL OnInitDialog();			// початкове налаштування вікна
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();				// перемальовування вікна
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
public:
	// Кількість міст на карті
	int m_amount;
	// Розмір популяції потенційних розв'язків
	int m_population;
	// Кількість мутацій-обмінів
	int m_exchanges;
	// Кількість мутацій-поворотів
	int m_rotations;
	// Максимальна кількість поколінь
	int m_generation_limit;

	// Ділянка побудови карти
	RECT m_map_rectangle;
	// Кількість опрацьованих поколінь
	int m_generations;
	// Поточна найменша довжина
	double m_shortest;
```

```
	// Масив m_amount точок для побудови міст і відшукання туру
	CPoint* m_towns;
	// Масив m_amount+1 точок для побудови туру
	CPoint * m_route;
	// Зв'язок з написом - кількості випробуваних поколінь
	CStatic m_Exemined;
	// Зв'язок з написом - довжина туру
	CStatic m_Length;
protected:
	afx_msg LRESULT OnUpdateGui(WPARAM wParam, LPARAM lParam);
	// Потік для виконання обчислень
	CWinThread* m_pWinThread;
	// Метод обчислень, що буде виконаний в окремому потоці
	static UINT WorkerThread(LPVOID lpParam);
	// Прапорець зупинки потоку виконання
	bool m_StopReguired;
	// Ознака виконання потоку обчислень
	bool m_ThreadRunning;
public:
	afx_msg void OnBnClickedStop();
	CButton m_GenerateButton;
	CButton m_StartButton;
	CButton m_StopButton;
	// Методи реагування на натискання кнопок
	afx_msg void OnBnClickedGenerate();
	afx_msg void OnBnClickedStart();
	afx_msg void OnBnClickedCancel();
};
```

Метод ініціалізації діалогу визначить, зокрема, розміри прямокутника, придатного для відображення карти. Продемонструємо написаний для цього фрагмент коду, опустивши автоматично згенеровану частину.

```
BOOL CEvoSalesmanTourDlg::OnInitDialog()
{
	CDialogEx::OnInitDialog();

	// ... автоматично згенерований код пропущено ...
	// TODO: Add extra initialization here
	UpdateData(FALSE);
	CWnd* group = GetDlgItem(IDGROUP_TOWNS);
	RECT rect;
	group->GetWindowRect(&rect);           // межі групи елементів керування
	this->GetClientRect(&m_map_rectangle); // доступна частина вікна діалогу
	m_map_rectangle.left = rect.right + 3;
	--m_map_rectangle.right;
	--m_map_rectangle.bottom;
	srand(time(0));                    // запуск давача випадкових чисел
	m_StartButton.EnableWindow(FALSE);// деякі кнопки спочатку недоступні
	m_StopButton.EnableWindow(FALSE);
	return TRUE;
}
```

Зауважимо, що булівські константи *TRUE* та *FALSE*, записані великими літерами – не помилка, а звичайна практика написання MFC-застосунків.

Метод перемальовування вікна – один з найважливіших для нас. Наведемо його частину, відповідальну за зображення карти й маршруту. Міста зобразимо кружечками, а маршрут – ламаною. Зауважимо, що MFC містить усі потрібні для цього графічні методи.

```
void CEvoSalesmanTourDlg::OnPaint()
{
    // частина, згенерована автоматично
    if (IsIconic())
    {
        // ... автоматично згенерований код пропущено ...
    }
    else
        // код, написаний власноруч
        CPaintDC dc(this);   // графічний контекст вікна виконує всі побудови
        dc.SelectStockObject(BLACK_PEN);    // малюватимемо чорним
        dc.SelectStockObject(WHITE_BRUSH);  // по білому
        dc.Rectangle(&m_map_rectangle);     // Рамка
        if (m_route != nullptr)             // Маршрут
        {
            dc.Polyline(m_route, m_amount + 1);
            delete[] m_route;
            m_route = nullptr;
        }
        if (m_towns != nullptr)             // Міста
        {
            for (int i = 0; i < m_amount; ++i) dc.Ellipse(
                m_towns[i].x-4, m_towns[i].y-4, m_towns[i].x+3, m_towns[i].y+3);
        }
        CDialogEx::OnPaint();
    }
}
```

Генерування координат міст відрізняється від випадку консольної програми лише викликом методу *InvalidateRect*, який перемальовує ту частину вікна, де зображена карта. Подивимось на метод опрацювання кнопки «Генерувати карту».

```
void CEvoSalesmanTourDlg::OnBnClickedGenerate()
{
      UpdateData(TRUE);               // прочитати задані користувачем значення
      delete[] m_towns;
      m_towns = new CPoint[m_amount]; // утворити масив точок
      int width = m_map_rectangle.right - m_map_rectangle.left - 10;
      int height = m_map_rectangle.bottom - m_map_rectangle.top - 10;
      for (int i = 0; i < m_amount; ++i)
      {                          // надати їм випадкових координат у межах карти
            m_towns[i].x = rand() % width + m_map_rectangle.left + 5;
            m_towns[i].y = rand() % heigth + m_map_rectangle.top + 5;
      }
      InvalidateRect(&m_map_rectangle);      // перемалювати карту
      m_StartButton.EnableWindow(TRUE);
}
```

Дещо складніше організувати відшукання розв'язку задачі. Ми б хотіли спостерігати за змінами маршруту під час виконання генетичного алгоритму. Для цього ми мали б налаштувати *solver.m_show* так, щоб він викликав перемальовування вікна. Але є одне «але». Під час тривалих обчислень вікно застосунку перемальовуватися не буде. Така вже особливість

його функціонування. Щоб перемальовування стало можливим, обчислення треба запустити в окремому потоці виконання. Засобами бібліотеки MFC це можна зробити функцією *AfxBeginThread*. Першим параметром вона приймає адресу методу, який треба виконати в потоці, другим – вказівник на довільний об'єкт. Зазвичай другим параметром передають вказівник на екземпляр діалогу, що створив потік, щоб надати доступ до даних діалогу.

Час від часу потік обчислень мав би ініціювати перемальовування вікна застосунку. І тут виникає ще одне «але». Перемальовування вікна можна викликати лише з потоку вікна, а для потоку обчислень такий виклик заборонено. Але потік обчислень може надіслати застосункові Windows-повідомлення, опрацювання якого відбудеться у потоці вікна. Цією можливістю і скористаємося.

Остаточно, щоб запустити обчислення разом з перемальовуванням, треба виконати такі кроки: оголосити окремий метод, наприклад, *WorkerThread*, що створює екземпляр класу *Solver*, налаштовує його член *m_show* і викликає метод *Solve*; у методі опрацювання натискання кнопки «Почати» запрограмувати створення потоку обчислень, що виконає *WorkerThread*; зареєструвати і використати нестандартне Windows-повідомлення, наприклад, *WM_UPDATE_GUI* та викликати в методі його опрацювання перемальовування вікна.

```
// Метод опрацювання натискання кнопки
void CEvoSalesmanTourDlg::OnBnClickedStart()
{
    int amount = m_amount;
    UpdateData(TRUE);
    if (amount != m_amount)
    {
        MessageBox(
            L"Кількість міст змінено.\nГенеруйте нову карту перед розв'язуванням",
            L"Помилка");
        m_GenerateButton.EnableWindow(TRUE);
        m_GenerateButton.SetFocus();
        m_StartButton.EnableWindow(FALSE);
        return;
    }
    m_GenerateButton.EnableWindow(FALSE); // налаштування елементів інтерфейсу
    m_StartButton.EnableWindow(FALSE);
    m_StopButton.EnableWindow(TRUE);
    m_StopReguired = false;                // налаштування прапорця зупинки
    // створення і запуск окремого потоку виконання
    m_pWinThread = AfxBeginThread(&CEvoSalesmanTourDlg::WorkerThread,
         (LPVOID)this, THREAD_PRIORITY_NORMAL, CREATE_SUSPENDED, 0, NULL);
    m_pWinThread->m_bAutoDelete = TRUE;
    m_pWinThread->ResumeThread();
}

// Метод виконання обчислень (для окремого потоку)
UINT CEvoSalesmanTourDlg::WorkerThread(LPVOID lpParam)
{
    CEvoSalesmanTourDlg* pDlg = (CEvoSalesmanTourDlg*)lpParam;
    // якщо немає доступу до вікна діалогу, потік завершиться з помилкою
    if (NULL == pDlg || NULL == pDlg->GetSafeHwnd())
        return 1;
    // створення екземпляра класу, що реалізує генетичний алгоритм
    Solver* solver = new Solver(pDlg->m_towns, pDlg->m_amount, pDlg->m_population,
        pDlg->m_exchanges, pDlg->m_rotations, pDlg->m_generation_limit);
    // налаштування відображення поточних змін
    solver->m_show = [pDlg, solver](int gen, double len)
```

```
	{
		pDlg->m_generations = gen;
		pDlg->m_shortest = len;
		pDlg->m_route = solver->Best().Route();
		::PostMessage(pDlg->GetSafeHwnd(), WM_UPDATE_GUI, 0, 0);
	};
	// запуск обчислень, метод пильнує за прапорцем зупинки
	pDlg->m_generations = solver->Solve(pDlg->m_StopReguired);
	// отримання розв'язку
	pDlg->m_shortest = solver->Best().Length();
	pDlg->Route = solver->Best().Route();
	delete solver;
	pDlg->OnUpdateGui(0, 0);		// встановлення остаточного вигляду вікна
	pDlg->m_ThreadRunning = false;
	pDlg->m_GenerateButton.EnableWindow(TRUE);
	pDlg->m_StartButton.EnableWindow(TRUE);
	pDlg->m_StopButton.EnableWindow(FALSE);
	return 0;
}

// Метод оновлення вигляду вікна діалогу
afx_msg LRESULT CEvoSalesmanTourDlg::OnUpdateGui(WPARAM wParam, LPARAM lParam)
{	// оновлюємо написи
	CString g; g.Format(L"Минуло: %d пок.", m_generations);
	m_Exemined.SetWindowTextW(g);
	CString s; s.Format(L"Довжина: %.2f", m_shortest);
	m_Length.SetWindowTextW(s);
	InvalidateRect(&m_map_rectangle);  // перемальовуємо карту
	return 0;
}
```

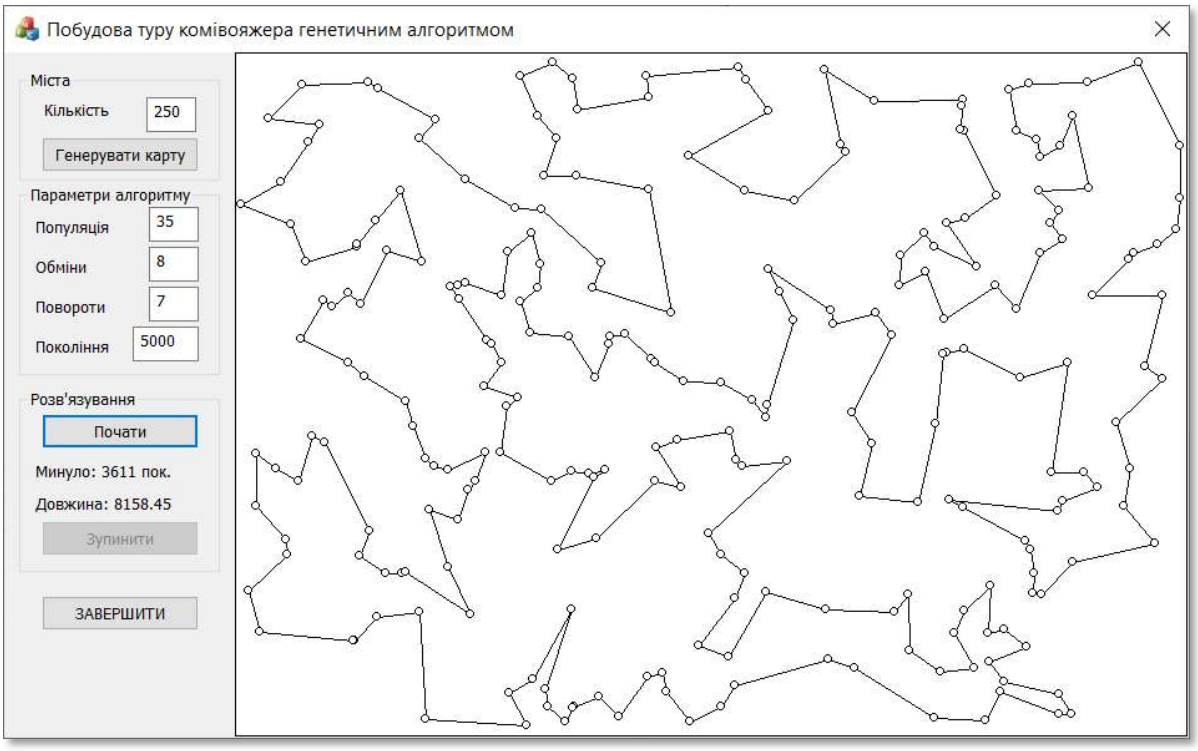

Рис. 16. Результати розв’язування задачі комівояжера для 250 міст

Найлегшим для реалізації виявився метод опрацювання натискання кнопки «Зупинити»: він змінює стан прапорця зупинки та доступність кнопок вікна діалогу.

```
void CEvoSalesmanTourDlg::OnBnClickedStop()
{
	m_StopReguired = true;
	m_GenerateButton.EnableWindow(TRUE);
	m_StartButton.EnableWindow(TRUE);
	m_StopButton.EnableWindow(FALSE);
}
```

На рис. 16 зображено евристичний розв'язок задачі комівояжера для великої кількості міст – для 250. Розв'язати таку задачу традиційними методами не вдалося б, а для генетичного алгоритму вистачило трохи більше хвилини на сучасному ноутбуці.

Сфера застосування еволюційного числення постійно розширюється, охоплюючи економічні, фізичні та інші задачі. Бажаємо читачам випробувати його можливості для власних потреб.

14.6. Запитання та завдання для самоперевірки

1. Перелічіть основні кроки генетичного алгоритму в загальному випадку.
2. Охарактеризуйте сферу застосовності генетичних алгоритмів.
3. Опишіть підготовчі кроки для побудови генетичного алгоритму.
4. Які оператори випадкових змін було використано в описаному генетичному алгоритмі розв'язування задачі комівояжера?
5. Що таке цільова функція генетичного алгоритму?
6. Що таке правило виживання в генетичному алгоритмі?
7. У описаних програмах членами популяції розв'язків задачі комівояжера є екземпляри класу *Tour*. Пригадайте, як вони дізнаються координати міст на карті.
8. Які структури даних використано в описаних програмах для зберігання популяції та потомства? Чим зумовлено їхній вибір?
9. Перелічіть числові параметри генетичного алгоритму розв'язування задачі комівояжера. Поміркуйте, як їхні значення впливають на швидкодію програми та на точність розв'язку.
10. Як віконний застосунок зберігає здатність реагувати на дії користувача під час тривалого виконання обчислень?
11. Завантажте програми за наведеним посиланням, запустіть їх на виконання.
12. Запропонуйте та випробуйте власні зміни та доповнення до програм. Наприклад, використайте двостатевий оператор мутацій, змініть правило виживання на змагальне, дослідіть, як зміни вплинули на швидкодію алгоритму й точність розв'язку.