11. Програмування з поверненням назад

Алгоритми з поверненням назад використовують для розв'язування задач, які потребують перевірки потенційно великого, проте скінченного числа розв'язків. Такий алгоритм шукає розв'язок задачі не за заданими правилами обчислень, а шляхом спроб і помилок. Він послідовно конструює розв'язок задачі, щоразу додаючи його чергову складову частину і перевіряючи можливість продовжити побудову. Якщо такої можливості нема, алгоритм відкидає останню чи кілька останніх доданих частин, повертається на кілька етапів побудови назад і пробує знаходити інші варіанти продовження. Здебільшого такі алгоритми визначають у термінах рекурсії. Загалом увесь процес можна трактувати як побудову та обрізання дерева підзадач. Підзадача має менший розмір, ніж вихідна задача, проте на кожному етапі побудови розв'язку таких підзадач є декілька. Вибір однієї з них визначає спосіб продовження побудови розв'язку. У багатьох випадках таке дерево росте дуже швидко. Швидкість зростання дерева підзадач залежить від параметрів задачі і здебільшого буває експотенційною (2^n, 3^n тощо). Навіть у випадку обрізання 99% можливих підзадач швидкість зростання залишається чималою: $M^n/100$ для великих n зростає експотенційно.

Продемонструємо процес побудови алгоритму з поверненнями на прикладі розв'язування конкретних задач.

11.1. Тур коня

Задача 46. *Дано натуральне n. На шахівниці розміру $n \times n$ на полі з координатами (x_0, y_0) стоїть шахова фігура – кінь. Його можна пересувати за звичайними шаховими правилами. Знайти послідовність n^2-1 ходів коня, за які він обійде всі клітки дошки, побувавши у кожній лише один раз.*

Одразу запропонувати спосіб отримання розв'язку задачі нелегко, тому розв'яжемо спочатку простішу задачу: *або виконати черговий хід, або довести, що будь-який хід неможливий.* З будь-якої клітки кінь може виконати не більше восьми ходів, тому існує лише вісім можливих варіантів продовження туру. Один з цих ходів можна виконати, якщо кінь не вийде за межі дошки і стане на клітку, у якій він ще не був. Перебирати кандидатів на продовження можна послідовно, пам'ятаючи при цьому номер останнього випробуваного кандидата. Якщо всі можливі продовження ведуть на зайняті клітки або за межі дошки, то хід зробити неможливо. Алгоритм перебору кандидатів схематично записують так:

```
do {
    вибір_чергового_кандидата_зі_списку_ходів;
    if (підходить) записати_хід; }
while (!підходить && кандидати_ще_є)
```

Продовжимо міркування. Щоб розпочати перебір кандидатів, необхідно ще перед циклом якось ініціалізувати їхній вибір. Далі, якщо черговий хід вдалий, то потрібно продовжувати побудову туру (перейти до розв'язування підзадачі – побудови коротшого на одиницю туру). У протилежному випадку, якщо з даного поля хід зробити неможливо, коня необхідно повернути на попереднє поле і спробувати зробити інший хід. Отже, процедура відшукання туру коня матиме такий вигляд:

```
void TryNextMove()
{
    ініціалізувати_вибір_ходу;
    do {
        вибір_чергового_кандидата_зі_списку_ходів;
```

```
            if (підходить)
            {
                записати_хід;
                if (дошка_не_заповнена)
                {
                    TryNextMove();          // продовження побудови розв'язку
                    if (невдача)     //повернення, вибір наступного кандидата
                        стерти_попередній_хід;
                }
            }
        while (хід_був_невдалим && кандидати_ще_є);
    }
```

Щоб конкретизувати «оператори» цієї процедури, спочатку визначають спосіб зображення шахівниці. Найзручніше з цією метою використати цілочислову матрицю розмірів $n \times n$: статична або динамічна. Для початку скористаєиося статичною.

```
    const int n = 8;
    int board[n][n];
```

Її значеннями можуть бути порядкові номери ходів коня. Перед початком побудови туру матриця заповнена нулями: board[*x,y*]=0, якщо на поле (*x, y*) ще не ходили; board[*x,y*]=*i*, якщо на полі (*x, y*) кінь був на *i*-му ході. Нехай змінні *u* та *v* містять координати можливого наступного ходу.

Тепер легко конкретизувати такі оператори:

дошка_не_заповнена: перевірити умову $i<n^2$;

підходить: перевірити $(1 \le u \le n) \wedge (1 \le v \le n) \wedge (board[u,v] = 0)$;

невдача, хід_був_вдалим – для відображення цих даних використаємо додаткову змінну success логічного типу;

записати_хід – задати відповідне значення елемента масиву: *board* [*u, v*] = *i*;

знищити_хід – аналогічно до попереднього: *board* [*u, v*] = 0.

Усі ці рішення дають змогу виконати наступний крок деталізації:

```
bool success = false;
void TryNextMove(int step, int posX, int posY)
{
      ініціалізувати_вибір_ходу;
      do {
            обчислити_(u,v)_координати_наступного_ходу;
            if (0 <= u && u < n && 0 <= v && v < n && board[u][v] == 0)
            {
                  board[u][v] = step; //робимо хід
                  if (step < n*n)
                  {
                        TryNextMove(step + 1, u, v);   // шукаємо продовження
                        if (!success) board[u][v] = 0; // повернення на крок назад
                  }
                  else success = true;
            }
      } while (!success && кандидати_ще_є);
}
```

Отже, нашу програму майже завершено. Залишилось тільки уточнити правила ініціалізації ходу та вибору наступного ходу. Усе написане дотепер ніяк від них не залежить, тому програма працюватиме за будь-яких правил переміщення фігури.

Проте, уважний читач мав би запитати, яким чином процедура *TryNextMove* отримає доступ до змінних *board*, *success*? Ми оголосили їх перед блоком функції, тому вони є глобальними. Доступ до них матимуть усі функції, визначені далі по тексту програми. Використання глобальних змінних – не дуже хороша практика. Відомо, що неконтрольований доступ до них є надійним джерелом помилок у роботі програми. Ми могли б передавати *board* і *success* через параметри, але велика кількість параметрів рекурсивної функції навантажуватиме системний стек. Виберемо певний компроміс. Змінну *success* залишимо глобальною, а матрицю-шахівницю передаватимемо через параметри. Тоді ми зможемо створювати її динамічно того розміру, який задасть користувач. Задавати значення *success* = *false* доведеться перед кожним викликом *TryNextMove*.

	3		2	
4				1
		Kn		
5				8
	6		7	

Рис. 10. Можливі ходи коня

Відомо, що кінь ходить «буквою Г». На рис. 10 цифрами позначено клітки, куди може походити кінь з клітки *Kn*. Як зміняться його координати після виконання ходу? Наприклад, якщо він походить на клітку «1», то значення *x* зросте на 2, а значення *y* – на 1. Такі зміни координат можна задати для кожного можливого ходу і записати у масиви, а для їхнього перебору використати змінну-лічильник ходів. Тепер легко завершити деталізацію програми: ініціалізація вибору ходу полягатиме у присвоєнні нуля лічильнику ходів, вибір наступного кандидата – збільшення лічильника і зміна поточних координат коня на відповідні зміщення. Усіх кандидатів на наступний хід буде вичерпано, коли лічильник досягне максимального значення 8. Остаточно отримаємо:

```
// створює динамічну матрицю nxn і заповнює її нулями
int** buildBoard(int n)
{
    int** board = new int*[n];
    *board = new int[n*n];
    for (int i = 0, *curr = *board; i < n*n; ++i, ++curr) *curr = 0;
    for (int i = 1; i < n; ++i) board[i] = board[i - 1] + n;
    return board;
}

// звільняє динамічну пам'ять від матриці
void eraseBoard(int**& board)
{
    delete[] * board;
    delete[] board;
    board = nullptr;
}

// виводить на друк матрицю-шахівницю; рядки - в оберненому порядку,
// оскільки початок координат шахівниці - в лівому нижньому кутку
void writeBoard(int** board, int n)
{
    for (int i = n - 1; i >= 0; --i)
    {
        cout.width(4); cout << i + 1 << " |";
        for (int j = 0; j < n; ++j)
        {
            cout.width(4); cout << board[j][i];
        }
        cout << '\n';
    }
```

```
    cout << "      " << std::string(n * 4 + 1, '-') << "\n      ";
    for (int j = 0; j < n; ++j)
    {
        cout.width(4); cout << char('a' + j);
    }
    cout << '\n';
}

// правила переміщення коня задано глобальними змінними
const int dx[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
const int dy[] = { 1, 2, 2, 1, -1, -2, -2, -1 };
const int dsize = sizeof dx / sizeof *dx;

bool success;           // ознака того, чи знайдено розв'язок

void tryNextMove(int** board, int n, int step, int posX, int posY)
{
    int length = n * n; // кількість ходів незмінна
    int k = 0;          // лічильник можливих продовжень
    do
    {
        int u = posX + dx[k];       // можливий хід коня
        int v = posY + dy[k]; ++k;
        if (0 <= u && u < n && 0 <= v && v < n && board[u][v] == 0)
        {
            board[u][v] = step;     //робимо хід
            if (step < length)
            {
                tryNextMove(board, n, step + 1, u, v); // шукаємо продовження
                if (!success) board[u][v] = 0;    // повернення на крок назад
            }
            else success = true;
        }
    } while (!success && k < dsize);
}

void tourByRecursion()
{
    int n;                                  // розмір шахівниці
    cout << "Введіть розмір шахівниці: "; cin >> n;
    int** board = buildBoard(n);            // побудова порожньої шахівниці
    int posX, posY;                         // початкові координати коня
    cout << "Введіть координати x y першого поля (1<=x,y<=" << n << "): ";
    cin >> posY >> posX;
    board[--posX][--posY] = 1;              // поставили коня
    success = false;                        // розв'язку ще не знайшли
    tryNextMove(board, n, 2, posX, posY); // пробуємо знайти тур коня
    // виведення результатів і звільнення пам'яті
    if (success)
    {
        cout << "\n    Знайдено тур коня\n----------------------------\n\n";
        writeBoard(board, n);
    }
    else cout << "\n    Розв'язку не знайшли\n";
    eraseBoard(board);
}
```

Для $n = 8$ отримано такий розв'язок задачі про відшукання туру коня:

```
Введіть розмір шахівниці: 8
Введіть координати x y першого поля (1<=x,y<=8): 1 1

       Знайдено тур коня
--------------------------------

   8 |  64  17   8  29  20  15   6  13
   7 |   9  30  19  16   7  12  25  22
   6 |  18  63  28  11  24  21  14   5
   5 |  31  10  33  62  41  26  23  54
   4 |  34  61  40  27  50  55   4  45
   3 |  39  32  37  42   3  46  53  56
   2 |  60  35   2  49  58  51  44  47
   1 |   1  38  59  36  43  48  57  52
      --------------------------------
        a   b   c   d   e   f   g   h
```

Зауважимо, що степінь дерева розв'язків цієї задачі є досить великим – він дорівнює 8. Тому час, необхідний для відшукання розв'язку зростає зі збільшенням розмірів дошки дуже швидко. Наприклад, для $n = 8$ кількість можливих варіантів дорівнює $2^{192} \approx 6{,}277 \times 10^{57}$, а для $n = 10$ автор цих рядків знайшов розв'язок швидше за комп'ютер.

Характерною особливістю записаного алгоритму є те, що він робить кроки у напрямі відшукання загального розв'язку. Всі кроки записують так, щоб можна було повернутись назад і відкинути ті з них, які заводять у тупик. Такий процес називають *поверненням назад* (англійською – *backtracking*).

За допомогою алгоритму процедури *TryNextMove* виводять універсальну схему розв'язування задач зі скінченною кількістю можливих розв'язків:

```
void Try();
{   ініціалізація_вибору_кандидатів;
    do {
        вибір_чергового_кандидата;
        if (підходить) {
            запис_кандидата;
            if (розв'язок_неповний) {
                Try();
                if (невдача) стерти_запис;
            }
        }
    } while (невдача && кандидати_ще_є);
}
```

Абстрактні «оператори» цієї схеми уточнюють для кожної конкретної задачі.

11.2. Тур коня без рекурсії

Ми уже говорили, що складність алгоритму з поверненнями є експотенційною. Дещо покращити ситуацію зі швидкодією могла б відмова від рекурсивних викликів: адже звертання до підпрограми вимагає додаткових затрат на розташування локальних змінних і налаштування адресів. А з якою метою використано рекурсію у процедурі *tryNextMove*? Зрозуміло, що ми звели задачу «знайди тур довжини n^2» до двох задач: «зроби хід» і «знайди тур довжини $n^2 - 1$», однак що ще? За рахунок чого відбуваються повернення і відкидання тупикових варіантів?

Очевидно, що в сегменті стеку програми зберігаються копії локальних змінних усіх викликаних екземплярів процедури *tryNextMove*. Наприклад, змінна *step* «пам'ятає» номер шуканого ходу, змінні *posX* та *posY* – координати фігури, *k* – кількість випробуваних кандидатів. Саме завдяки використанню цієї інформації *tryNextMove* здатна виконувати повернення. Проте для її зберігання зовсім не обов'язково використовувати системний стек. Ми могли б запровадити власний стек, наприклад, за допомогою використання масиву відповідного типу та розміру і зберігати у ньому всі необхідні дані.

Номер шуканого ходу та ознака успіху є загальними даними, їх можна зберігати в змінних програми, а до стеку необхідно буде занести координати виконаного ходу і кількість кандидатів, випробуваних на цьому ході. Отже, ми могли б використати такі оголошення:

```
struct State
{
	int x, y, k;
	State(int X = 0, int Y = 0, int K = 0) :x(X), y(Y), k(K) {}
};
const int length = n * n;
State steps[length];
```

Тут *length* – як і раніше, довжина туру, масив *steps* буде використано як стек для зберігання інформації щодо виконаних ходів.

З метою побудови розв'язку ми використаємо такі ж підходи, як і в попередній програмі, тому без зайвих пояснень просто наведемо її текст:

```
bool knightTour(int** board, int n, int posX, int posY)
{
	int length = n * n - 1;                // кількість ходів
	State* moves = new State[length + 1]; // внутрішній стек
	int step = 0;                          // ходів ще не робили
	board[--posX][--posY] = 1;
	State curr(posX, posY, 0);             // початкове розташування коня
	moves[step] = curr;                    // і стан дошки
	do
	{	bool success = false;  // успішність чергового ходу
		int u, v;              // координати чергового ходу
		while (curr.k < 8 && !success)
		{                      // шукаємо можливе продовження
			u = curr.x + dx[curr.k];
			v = curr.y + dy[curr.k];
			success = 0 <= u && u < n && 0 <= v && v < n && board[u][v] == 0;
			++curr.k;
		}
		if (success)
		{            // знайшли продовження
			moves[++step] = curr;      // запам'ятовуємо хід
			board[u][v] = step + 1;
			curr = State(u, v, 0);     // новий стан дошки
		}
		else         // повертаємося на хід назад
		{
			board[curr.x][curr.y] = 0; // звільнили клітинку
			curr = moves[step--];      // повернули стан
		}
	} while (step < length && step >= 0);
```

```
      delete[] moves;
      return step == length;
}
void solveTour()
{
      int n;                                  // розмір шахівниці
      cout << "Введіть розмір шахівниці: "; cin >> n;
      int** board = buildBoard(n);            // побудова порожньої шахівниці
      int posX, posY;                         // початкові координати коня
      cout << "Введіть координати x y першого поля (1<=x,y<=" << n << "): ";
      cin >> posY >> posX;
      bool success = knightTour(board, n, posX, posY); // шукаємо тур коня
      // виведення результатів і звільнення пам'яті
      if (success)
      {
            cout << "\n   Знайдено тур коня\n-------------------------------\n\n";
            writeBoard(board, n);
      }
      else cout << "\n   Розв'язку не знайшли\n";
      eraseBoard(board);
}
```

Програма *solveTour* знаходить такі ж розв'язки, як попередня, оскільки перебирає можливі ходи у такому самому порядку.

У цій програмі для контролю за тим, чи вже отримано розв'язок, використано змінну *step* – вона містить поточну довжину туру. Якщо в умові циклу залишити тільки умову *step* >= 0, а в циклі друкувати матрицю *board* кожного разу, коли *step* = *length*, то програма надрукує **всі** тури коня, що починаються на клітці з координатами (*posX*, *posY*). Наприклад, для задачі з розміром дошки *n*=5 і стартовою кліткою (1, 1) усіх турів є 304 (без урахування їхньої симетрії).

11.3. Задача про 8 ферзів

Наведемо ще один приклад побудови алгоритму з поверненнями. І знову задача буде пов'язана з шахівницею. Нехай у читача не складеться хибне враження, що алгоритми з поверненнями застосовні виключно до шахових фігур – ні, таким способом можна будувати Гамільтонів цикл, розфарбовувати карту, розв'язувати судоку та ще багато інших задач. Наша мета – продемонструвати, що виведена загальна схема побудови розв'язку діє.

Задача 47. *Розташувати на порожній шахівниці 8 ферзів так, щоб ні один з них не перебував під боєм іншого.*

Перш за все потрібно вибрати спосіб зображення розташування фігур на дошці. Відомо, що клітинки шахівниці позначають парою *БуквaЦифра*. Ми могли б оголосити відповідний тип *cell* і зобразити розташування ферзів масивом елементів цього типу.

```
struct cell
{
      char x; int y;
      cell(char a = 'a', int b = 1) :x(a), y(b) {}
};
cell queens[8];
```

Шуканими є поля *x* та *y* елементів масиву *queens*. Проте, ми могли б відразу спростити пошук розв'язку, адже всі ферзі мусять стояти в різних вертикалях шахівниці, щоб не загрожувати один одному. Тому можна відразу покласти *queens*[0].*x* = '*a*'; *queens*[1].*x* = '*b*'; ... *queens*[7].*x* = '*h*'; і шукати тільки номери горизонталей. Ці номери знаходитемо поступово від першого до останнього. Для першого ферзя можна вибрати довільну горизонталь. Щоб поставити другого, потрібно перевірити, чи не загрожує він першому. Щоб поставити третього – чи не загрожує він першому і другому, щоб поставити *k*-го – чи не загрожує він усім попереднім. Таку перевірку можна описати функцією.

```
// перевіряє, чи загрожує k-ва королева попереднім
bool safe(cell* queens, int k)
{
    for (int i = k - 1; i >= 0; --i)      // пам'ятаємо, що вертикалі різні, тому
        if (queens[i].y == queens[k].y || // порівнюємо горизонталі та
            queens[i].x - queens[i].y == queens[k].x - queens[k].y || // діагоналі
            queens[i].x + queens[i].y == queens[k].x + queens[k].y) return false;
    return true;
}
```

Тепер можемо конкретизувати «узагальнені оператори» схеми алгоритму з поверненнями. *Ініціалізація вибору кандидата* означає присвоїти значення 0 полю *y* чергового ферзя так, ніби він стоїть у своїй вертикалі перед дошкою. *Вибір чергового кандидата* означає збільшення *y* – просування ферзя на одну горизонталь догори. *Кандидати ще є* – ферзь не досяг останньої горизонталі (*y* < 8). Чи *підходить*, перевіряє функція *safe*. *Розв'язок неповний*, якщо розташували ще не всіх ферзів. Якщо масив *queens* передати рекурсивній функції побудови розв'язку, то *записати кандидата* і *стерти кандидата* відбуватиметься автоматично. Зауважимо також, що ми можемо не обмежуватися розміром шахівниці 8×8, а записати функцію для довільного натурального *n*.

```
// шукає безпечне розташування n ферзів
bool tryPutQueen(cell* queens, int n, int k)
{
      // k-ва фігура відразу стає у свою вертикаль
      queens[k].x = 'a' + k;
      queens[k].y = 0;      // горизонталь підбиратимемо (є n спроб)
      bool success = false;
      do
      {     ++queens[k].y;    // чергова горизонталь
            if (safe(queens, k))
            {   // розташування k-ї фігури - безпечне
                  if (k + 1 == n) success = true; // знайшли розв'язок
                  else                            // шукаємо продовження
                        success = tryPutQueen(queens, n, k + 1);
            }
      } while (!success && queens[k].y < n);
      return success;
}
```

Правильно викликати *tryPutQueen* допоможе наступна процедура

```
void putQueens()
{
      int n;                                  // розмір шахівниці
      cout << "Введіть розмір шахівниці: "; cin >> n;
      cell * queens = new cell[n];            // шуканий роз'язок
      bool success = tryPutQueen(queens, n, 0);
```

```
        // виведення результатів і звільнення пам'яті
        if (success)
        {
              int**board = buildBoard(n);     // матриця допоможе зобразити шахівницю
              cout << "\n       Знайдено розташування"
                    << "\n--------------------------------\n\n      ";
              for (int i = 0; i < n; ++i)
              {
                    cout << "  " << queens[i];    // друкуємо знайдене розташування
                    // і "розставляємо" фігури на шахівниці:
                    // порожня клітинка містить 0, клітинка з ферзем - 13
                    board[queens[i].x - 'a'][queens[i].y - 1] = 13;
              }
              cout << "\n\n";
              writeBoard(board, n);             // друк шахівниці
              eraseBoard(board);
        }
        else cout << "\n       Розв'язку не знайшли\n";
        delete[] queens;
}
```

З'ясувалося, що задача розташування n ферзів має розв'язок для усіх натуральних n крім 2 і 3. Ось один з отриманих розв'язків. Порожню клітинку шахівниці позначено нулем, а розташування ферзя – числом 13 (воно віддалено нагадує корону).

```
Введіть розмір шахівниці: 8

       Знайдено розташування
--------------------------------

        a1  b5  c8  d6  e3  f7  g2  h4

   8 |   0   0  13   0   0   0   0   0
   7 |   0   0   0   0   0  13   0   0
   6 |   0   0   0  13   0   0   0   0
   5 |   0  13   0   0   0   0   0   0
   4 |   0   0   0   0   0   0   0  13
   3 |   0   0   0   0  13   0   0   0
   2 |   0   0   0   0   0   0  13   0
   1 |  13   0   0   0   0   0   0   0
      --------------------------------
         a   b   c   d   e   f   g   h
```