ЛАБОРАТОРНА РОБОТА №4

Назва роботи: побудова алгоритмів ефективних за часовою складністю; задача квадратичного призначення.

Мета роботи: виконати зменшення часової складності методом «гілок і границь».

1.1. Загальні відомості

В алгоритмах з експоненціальною складністю кількість операцій, необхідних для розв'язання задачі, зростає швидше, ніж поліном к-ї степені при зростанні розміру входу.

$$\lim_{n \to \infty} \frac{O(n)}{P_k(n)} > Const$$

Розв'язання задач на сучасному компютері вже для n > 20, (наприклад при O(n!)) є проблематичним. Одним із способів зменшення часової складності є використання евристичних алгоритмів. Евристичні алгоритми дозволяють вирішувати складні задачі за прийнятний час за рахунок зведення задачі з експоненціальною складністю до задачі з поліноміальною складністю, хоча такі евристичні алгоритми не завжди можуть бути застосовані.

Задача квадратичного призначення (ЗКП) розглядається як приклад побудови алгоритму ефективного за часовою складністю.

Формулювання задачі

Задані ***m*** елементів x1, x1, … , xm. Для кожної пари елементів задані вагові коефіцієнти. Вагові коефіцієнти задані матрицею R = || rij ||m x m, де rij – кількість зв'язків між елементами xi і xj

Дискретне робоче поле (ДРП) – це набір фіксованих позицій, в яких розміщуються елементи xi. Відстані між позиціями ДРП задаються в ортогональній метриці, причому відстань між сусідніми позиціями по горизонталі та вертикалі дорівнює 1. Також задана матриця відстаней ДРП: D = || dij ||n x n). Кількість елементів дорівнює кількості позицій($n = m$).

Необхідно розташувати елементи на дискретному робочому полі за критерієм мінімальної сумарної довжини з'єднань, тобто потрібно мінімізувати функцію:

$$F(p) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} r_{ij} d_{p(i)p(j)}$$

rij (i,j =1,2,…,m) – визначають степінь зв'язності елементів.

1.2. Розв'язання ЗКП методом "гілок та границь".

Основна ідея методу "гілок та границь" полягає в тому, що вся множина допустимих рішень задачі розбивається на деякі підмножини, всередені яких здійснюється впорядкований перегляд рішень з метою вибору оптимального.

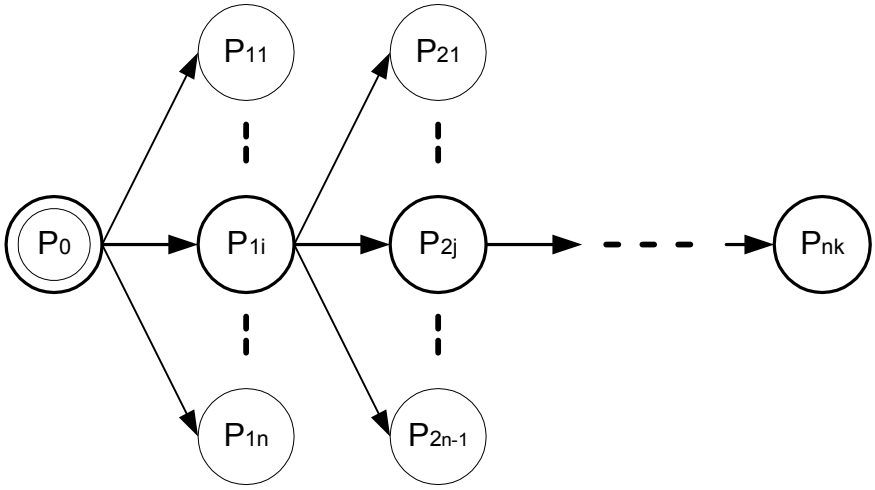


Рис. 1

Стосовно ЗКП метод "гілок та границь" полягає в наступному:

1. Кількість можливих розміщень елементів P0 розбивається на рівні за потужністю підмножини (P11…P1n) .
2. Для кожної підмножини підраховується "нижня границя" Fij.
3. Обирається та підмножина, яка має мінімальне значення "нижньої границі", решта підмножин відкидається з розгляду. Обраний елемент фіксується в позиції, яка відповідає мінімальній "нижній границі".
4. Перехід до п1. для обраної підмножини.
5. п1 –п4 повторюються, доки не будуть відкинуті всі рішення крім оптимального.

Утворення підмножин:

1. З множини нерозташованих елементів обирається будь-який елемент, наприклад перший за нумерацією (X1) і закріплюється в будь-якій вільній позиції ДРП, наприклад першій за нумерацією
2 Обраний елемент переміщується в іншу вільну позицію.
3. п.1 – п.2 повторюється, доки почерговим закріпленням не будуть охоплені всі вільні позиції ДРП.

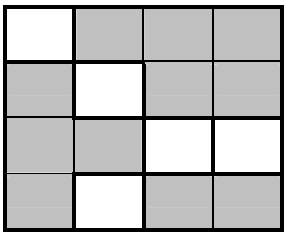
Підрахунок "нижніх границь".

Підрахунок "нижніх границь" грунтується на тому, що якщо r = (r1, r2,..., rn) і d = (d1, d2,..., dn) – два вектора, то мінімуму скалярного добутку r*d відповідає розташуванню складових вектора r у зростаючому, а складових вектора d у спадаючому порядку. Таким чином, "нижня границя" може бути отримана із верхніх половин матриць R і D, елементи яких утворюють вектори r і d , причому, складові вектора r розташовані у зростаючому порядку, а складові вектора d у спадаючому порядку.

2. Приклад.

Задане дискретне робоче поле (ДРП) і матриця зв'язності R. Розташувати елементи X1, X2, X3, X4, X5 на ДРП за критерієм мінімальної сумарної довжини з'єднань.

ДРП

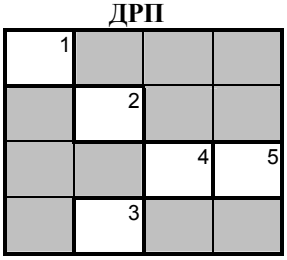


R

x	*1*	*2*	*3*	*4*	*5*
1	0	1	2	0	3
2	1	0	3	0	2
3	2	3	0	1	2
4	0	0	1	0	1
5	3	2	2	1	0

Позначимо на ДРП номера позицій в правому верхньому куті вільних позицій та визначемо матрицю відстаней D.

ДРП



D

p	*1*	*2*	*3*	*4*	*5*
1	0	2	4	4	5
2	2	0	2	2	3
3	4	2	0	2	3
4	4	2	2	0	1
5	5	3	3	1	0

Підрахуємо <u>мінімальну</u> нижню границю $F_{min} = r*d$:

r = 0 0 1 1 1 2 2 2 3 3

d = 5 4 4 3 3 2 2 2 2 1

0+0+4+3+3+4+4+4+6+3 = 31

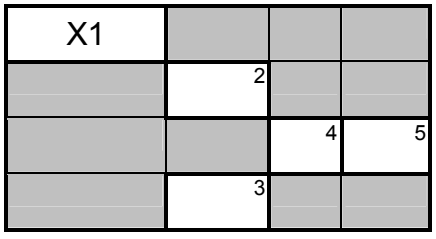
r – вектор зв'язків між елементами

d – вектор відстаней між позиціями елементів

Будемо розрізняти три типи елементів : "незакріплені", "закріплегні" в певній позиції та "зафіксовані".

<u>Вибираємо елемент X1</u>

1.1 Закріплюємо елемент X1 в позиції P1



Підраховуємо нижню границю

$F_{1(1)} = f_{н,н} + f_{1(1),н} = r_{н,н} * d_{н,н} + r_{1(1),н} * d_{1(1),н}$

$r_{1(1),н}$ = 0 1 2 3 $r_{н,н}$ = 0 1 1 2 2 3

$d_{1(1),н}$ = 5 4 4 2 $d_{н,н}$ = 3 3 2 2 2 1

$f_{1(1),н}$ = 0+4+8+6 = 18 $f_{н,н}$ = 0+3+2+4+4+3 = 16

$F_{1(1)} = 18 + 16 = 34$

$F_{1(1)}$ – нижня границя для розташування, в якому елемент X1 закріплений в позиції (1)

$f_{н,н}$ – скалярний добуток для незакріплених елементів

$f_{1(1),н}$ - скалярний добуток для закріпленого елемента X1 в позиції P1 та незакріплених елементів .

($F_{1(1)} > F_{min}$), тому переміщуємо елемент X1 в наступну вільну позицію.
Наприклад, в позицію P2.

1.2 Закріплюємо елемент X1 в позиції P2

1			
	X1		
		4	5
	3		

Підраховуємо нижню границю:

$F_{1(2)} = f_{нн} + f_{1(2),н} = r_{н,н} * d_{н,н} + r_{1(2),н} * d_{1(2),н}$

$r_{1(2),н}$ = 0 1 2 3 $r_{н,н}$ = 0 1 1 2 2 3

$d_{1(2),н}$ = 3 2 2 2 $d_{н,н}$ = 5 4 4 3 2 1

$f_{1(2),н}$ = 0+2+4+6 = 12 $f_{н,н}$ = 0+4+4+6+4+3 = 21

$F_{1(1)} = 12 + 21 = 33$

($F_{1(2)} > F_{min}$), тому переміщуємо елемент X1 в наступну вільну позицію.
Наприклад, в позицію P3.

1.3 Закріплюємо елемент X1 в позиції P3

1			
	2		
		4	5
	X1		

Підраховуємо нижню границю

$F_{1(3)} = f_{н,н} + f_{1(3),н} = r_{н,н} * d_{н,н} + r_{1(3),н} * d_{1(3),н}$

$r_{1(3),н}$ = 0 1 2 3 $r_{н,н}$ = 0 1 1 2 2 3

$d_{1(3),н}$ = 4 3 2 2 $d_{н,н}$ = 5 4 3 2 2 1

$f_{1(3),н}$ = 0+3+4+6 = 13 $f_{н,н}$ = 0+4+3+4+4+3 = 18

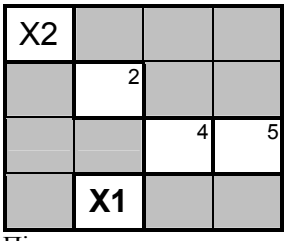
$F_{1(3)} = 13 + 18 = 31$

$F_{13} = F_{min}$ => Подальше переміщення елемента X1 припиняємо.

Фіксуємо елемент X1 в позиції P3

2. Вибираємо елемент X2.

2.1 Закріплюємо елемент X2 в позиції P1



Підраховуємо нижню границю

$F_{2(1)} = f_{н,\,н} + f_{1(3),\,н} + f_{2(1),\,н} + f_{1(3),\,2(1)}$

$F_{2(1)} = r_{н,н} * d_{н,н} + r_{1(3),\,н} * d_{1(3),\,н} + r_{2(1),\,н} * d_{2(1),\,н} + r_{1(3),\,2(1)} * d_{1(3),\,2(1)}$

$r_{2(1),н} = 0\ 2\ 3$ $\qquad$ $r_{н,н} = 1\ 1\ 2$

$d_{2(1),н} = 5\ 4\ 2$ $\qquad$ $d_{н,н} = 3\ 2\ 1$

$f_{2(1),н} = \overline{0+8+6} = 14$ $\qquad$ $f_{н,н} = \overline{3+2+2} = 7$

$r_{1(3),н} = 0\ 2\ 3$ $\qquad$ $r_{1(3),\,2(1)} = 1$

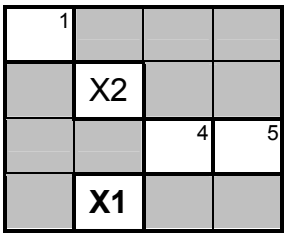
$d_{1(3),н} = 3\ 2\ 2$ $\qquad$ $d_{1(3),\,2(1)} = 4$

$F_{1(3),н} = \overline{0+4+6} = 10$ $\qquad$ $f_{1(3),\,2(1)} = \overline{4} = 4$

$F_{2(1)} = 14+7+10+4 = 35$

($F_{2(1)} > F_{min}$), тому переміщуємо елемент X2 в наступну вільну позицію. Наприклад, в позицію P2.

2.2 Закріплюємо елемент X2 в позиції P2



Підраховуємо нижню границю

$F_{2(2)} = f_{н,\,н} + f_{1(3),\,н} + f_{2(2),\,н} + f_{1(3),\,2(2)}$

$F_{2(2)} = r_{н,н} * d_{н,н} + r_{1(3),\,н} * d_{1(3),\,н} + r_{2(2),\,н} * d_{2(2),\,н} + r_{1(3),\,2(2)} * d_{1(3),\,2(2)}$

$r_{2(2),н} = 0\ 2\ 3$ $\qquad$ $r_{н,н} = 1\ 1\ 2$

$d_{2(2),н} = 3\ 2\ 2$ $\qquad$ $d_{н,н} = 5\ 4\ 1$

$f_{2(2),н} = \overline{0+4+6} = 10$ $\qquad$ $f_{н,н} = \overline{5+4+2} = 11$

$r_{1(3),\,н} = 0\ 2\ 3$ $r_{1(3),\,2(2)} = 1$

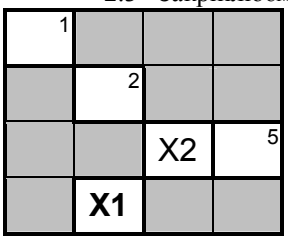
$d_{1(3),\,н} = 4\ 3\ 2$ $d_{1(3),\,2(2)} = 2$

$f_{1(3),\,н} = 0+6+6 = 12$ $f_{1(3),\,2(2)} = 2 = 2$

$F_{2(2)} = 10+11+12+2 = 35$

($F_{2(2)} > F_{min}$), тому переміщуємо елемент X2 в наступну вільну позицію.
Наприклад, в позицію P4.

2.3 Закріплюємо елемент X2 в позиції P4



Підраховуємо нижню границю

$F_{2(4)} = f_{н,\,н} + f_{1(3),\,н} + f_{2(4),\,н} + f_{1(3),\,2(4)}$

$F_{2(4)} = r_{н,н} * d_{н,н} + r_{1(3),\,н} * d_{1(3),\,н} + r_{2(4),\,н} * d_{2(4),\,н} + r_{1(3),\,2(4)} * d_{1(3),\,2(4)}$

$r_{2(4),\,н} = 0\ 2\ 3$ $r_{н,н} = 1\ 1\ 2$

$d_{2(4),\,н} = 4\ 2\ 1$ $d_{н,н} = 5\ 3\ 2$

$f_{2(4),\,н} = 0+4+3 = 7$ $f_{н,н} = 5+3+4 = 12$

$r_{1(3),\,н} = 0\ 2\ 3$ $r_{1(3),\,2(4)} = 1$

$d_{1(3),\,н} = 4\ 3\ 2$ $d_{1(3),\,2(4)} = 2$

$f_{1(3),\,н} = 0+6+6 = 12$ $f_{1(3),\,2(4)} = 2 = 2$

$F_{2(4)} = 7+12+12+2 = 33$

($F_{2(4)} > F_{min}$), тому переміщуємо елемент X2 в наступну вільну позицію - P5.

2.4 Закріплюємо елемент X2 в позиції P5

Підраховуємо нижню границю

$F_{2(5)} = f_{н,\,н} + f_{1(3),\,н} + f_{2(5),\,н} + f_{1(3),\,2(5)}$

$F_{2(5)} = r_{н,н} * d_{н,н} + r_{1(3),\,н} * d_{1(3),\,н} + r_{2(5),\,н} * d_{2(5),\,н} + r_{1(3),\,2(5)} * d_{1(3),\,2(5)}$

$r_{2(5),\,н} = 0\ 2\ 3$ $r_{н,н} = 1\ 1\ 2$

$d_{2(5),\,н} = 5\ 3\ 1$ $d_{н,н} = 4\ 2\ 2$

$f_{2(5),\,н} = 0+6+3 = 9$ $f_{н,н} = 4+2+4 = 10$

$r_{1(3),\,н} = 0\ 2\ 3$ $r_{1(3),\,2(5)} = 1$

$d_{1(3),\,н} = 4\ 2\ 2$ $d_{1(3),\,2(5)} = 3$

$f_{1(3),\,н} = 0+4+6 = 10$ $f_{1(3),\,2(5)} = 3 = 3$

$F_{2(5)} = 9+10+10+3 = 32$

$(F_{2(1)} > F_{2(2)} > F_{2(4)} > F_{2(5)} > F\min)$

Обираємо меншу нижню границю - $F_{2(5)}$

1			
	2		
		4	X2
	X1		

<u>Фіксуємо елемент X2 в позиції P5</u>

3. Вибираємо елемент X3.

3.1 Закріплюємо елемент X3 в позиції P1

X3			
	2		
		4	X2
	X1		

Підраховуємо нижню границю

$F_{3(1)} = f_{н,н} + f_{3(1),н} + \underline{f_{3(1),1(3)} + f_{3(1),2(5)}} + \underline{f_{1(3),н} + f_{2(5),н}} + f_{1(3),2(5)}$

$r_{3(1),н} = 1\ 2$ $\quad r_{н,н} = 1$ $\quad r_{1(3),2(5)} = 1$

$d_{3(1),н} = 4\ 2$ $\quad d_{н,н} = 2$ $\quad d_{1(3),2(5)} = 3$

$f_{3(1),н} = \overline{4+4} = 8$ $\quad f_{н,н} = \overline{2} = 2$ $\quad f_{1(3),2(5)} = \overline{3} = 3$

$r_{1(3),н} = 0\ 3$ $\quad r_{3(1),1(3)} = 2$

$d_{1(3),н} = 2\ 2$ $\quad d_{3(1),1(3)} = 4$

$f_{1(3),н} = \overline{0+6} = 6$ $\quad f_{3(1),1(3)} = \overline{8} = 8$

$r_{2(5),н} = 0\ 2$ $\quad r_{3(1),2(5)} = 3$

$d_{2(5),н} = 3\ 1$ $\quad d_{3(1),2(5)} = 5$

$f_{2(5),н} = \overline{0+2} = 2$ $\quad f_{3(1),2(5)} = \overline{15} = 15$

$F_{3(1)} = 8+2+3+6+8+2+15 = 44$

$(F_{3(1)} > F\min)$, тому переміщуємо елемент X3 в наступну вільну позицію. Наприклад, в позицію P2.

3.2 Закріплюємо елемент X3 в позиції P2

1			
	X3		
		4	X2
	X1		

Підраховуємо нижню границю

$F_{3(2)} = f_{н,н} + f_{3(2),н} + \underline{f_{3(2),1(3)} + f_{3(2),2(5)}} + \underline{f_{1(3),н} + f_{2(5),н}} + f_{1(3),2(5)}$

$r_{3(2),н} = 1\ 2$ $r_{н,н} = 1$ $r_{1(3),2(5)} = 1$

$d_{3(2),н} = 2\ 2$ $d_{н,н} = 4$ $d_{1(3),2(5)} = 3$

$f_{3(2),н} = \overline{2+4} = 6$ $f_{н,н} = \overline{4} = 4$ $f_{1(3),2(5)} = \overline{3} = 3$

$r_{1(3),н} = 0\ 3$ $r_{3(2),1(3)} = 2$

$d_{1(3),н} = 4\ 2$ $d_{3(2),1(3)} = 2$

$f_{1(3),н} = \overline{0+6} = 6$ $f_{3(2),1(3)} = \overline{4} = 4$

$r_{2(5),н} = 0\ 2$ $r_{3(2),2(5)} = 3$

$d_{2(5),н} = 5\ 1$ $d_{3(2),2(5)} = 3$

$f_{2(5),н} = \overline{0+2} = 2$ $f_{3(2),2(5)} = \overline{9} = 9$

$F_{3(2)} = 6+4+3+6+4+2+9 = 34$

($F_{3(2)} > F_{min}$), тому переміщуємо елемент X3 в наступну вільну позицію - P4.

3.3 Закріплюємо елемент X3 в позиції P4

1			
	2		
		X3	X2
	X1		

Підраховуємо нижню границю

$F_{3(4)} = f_{н,н} + f_{3(4),н} + \underline{f_{3(4),1(3)} + f_{3(4),2(5)}} + \underline{f_{1(3),н} + f_{2(5),н}} + f_{1(3),2(5)}$

$r_{3(4),н} = 1\ 2$ $r_{н,н} = 1$ $r_{1(3),2(5)} = 1$

$d_{3(4),н} = 4\ 2$ $d_{н,н} = 2$ $d_{1(3),2(5)} = 3$

$f_{3(4),н} = \overline{4+4} = 8$ $f_{н,н} = \overline{2} = 2$ $f_{1(3),2(5)} = \overline{3} = 3$

$r_{1(3),н} = 0\ 3$ $r_{3(4),1(3)} = 2$

$d_{1(3),н} = 4\ 2$ $d_{3(4),1(3)} = 2$

$f_{1(3),н} = \overline{0+6} = 6$ $f_{3(4),1(3)} = \overline{4} = 4$

$r_{2(5),н} = 0\ 2$ $r_{3(4),2(5)} = 3$

$d_{2(5),н} = 5\ 3$ $d_{3(4),2(5)} = 1$

$f_{2(5),н} = \overline{0+6} = 6$ $f_{3(4),2(5)} = \overline{3} = 3$

$F_{3(4)} = 8+2+3+6+4+6+3 = 32$

($F_{3(1)} > F_{3(2)} > F_{3(4)} > F_{min}$)

Обираємо меншу нижню границю - F3(4)

1			
	2		
		X3	X2
	X1		

Фіксуємо елемент X3 в позиції P4

4. Вибираємо елемент X4.

4.1 Закріплюємо елемент X4 в позиції P1

X4			
	2		
		X3	X2
	X1		

Підраховуємо нижню границю

$F_{4(1)} = f_{4(1),н} + f_{4(1),\,1(3)} + f_{4(1),\,1(5)} + f_{4(1),\,3(4)} +$

$f_{1(3),\,н} + f_{2(5),\,н} + f_{3(4),\,н} + f_{1(3),\,2(5)} + f_{1(3),\,3(4)} + f_{2(5),3(4)}$

$r_{4(1),н} = 1 \qquad d_{4(1),н} = 2 \qquad f_{4(1),н} = 2$

$r_{4(1),\,1(3)} = 0 \qquad d_{4(1),\,1(3)} = 4 \qquad f_{4(1),\,1(3)} = 0$

$r_{4(1),\,2(5)} = 0 \qquad d_{4(1),\,2(5)} = 5 \qquad f_{4(1),\,2(5)} = 0$

$r_{4(1),\,3(4)} = 1 \qquad d_{4(1),\,3(4)} = 4 \qquad f_{4(1),\,2(5)} = 4$

$r_{1(3),\,н} = 3 \qquad d_{1(3),\,н} = 2 \qquad f_{1(3),\,н} = 6$

$r_{2(5),\,н} = 2 \qquad d_{2(5),\,н} = 2 \qquad f_{2(5),\,н} = 4$

$r_{3(4),\,н} = 2 \qquad d_{3(4),\,н} = 3 \qquad f_{3(4),\,н} = 6$

$r_{1(3),\,2(5)} = 1 \qquad d_{1(3),\,2(5)} = 3 \qquad f_{1(3),\,2(5)} = 3$

$r_{2(5),\,3(4)} = 3 \qquad d_{2(5),\,3(4)} = 1 \qquad f_{2(5),\,3(4)} = 3$

$r_{1(3),\,3(4)} = 2 \qquad d_{1(3),\,3(4)} = 2 \qquad f_{1(3),\,3(4)} = 4$

$F_{4(1)} = 2 + 0 + 0 + 4 + 6 + 4 + 6 + 3 + 3 + 4 = 32$

Аналогічно можна підрахувати що $F_{4(2)} = 44$, тобто $F_{4(1)} > F_{4(1)}$

Фіксуємо елемент X4 в позиції P1. Для елемента X5 залишається позиція P2

X4			
	X5		
		X3	X2
	X1		

Підраховуємо сумарну довжину з'єднань

$F_{сум} = 1*3+2*2+0*4+3*2 + 3*1+0*5+2*3 + 1*4+2*2 + 1*2 =$
$3+4+0+6+3+0+6+4+4+2 = 32$

3. Приклад програми

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define SIZE 4
#define ELEMENTS_COUNT 5
#define PERMITTED_POSITIONS_COUNT ELEMENTS_COUNT

#define PERMITTED_POSITION_FLAG 0x00000001
#define DEFINED_POSITION_FLAG 0x00000002

#define ELEMENT_INDEX_TO_ELEMENT_ID(INDEX) ((INDEX) + 1)
#define POSITION_INDEX_TO_POSITION_ID(INDEX) ((INDEX) + 1)

#define POSITIONS_DEFINITIONS {
\
{{ PERMITTED_POSITION_FLAG }, { 0 }, { 0 }, { 0 }}, \
{{ 0 }, { PERMITTED_POSITION_FLAG }, { 0 }, { 0 }}, \
{{ 0 }, { 0 }, { PERMITTED_POSITION_FLAG }, { PERMITTED_POSITION_FLAG }}, \
{{ 0 }, { PERMITTED_POSITION_FLAG }, { 0 }, { 0 }}                        \
}

#define R_DEFINITIONS { \
{0, 1, 2, 0, 3}, \
{1, 0, 3, 0, 2}, \
{2, 3, 0, 1, 2}, \
{0, 0, 1, 0, 1}, \
{3, 2, 2, 1, 0}         \
}

typedef struct PositionStruct{
	unsigned int positionIndex;
	unsigned int positionIIndex;
	unsigned int positionJIndex;
	unsigned int elementIndex;
} Position;

typedef struct TablePositionStruct{
	unsigned int flags;
	unsigned int positionIndex;
	unsigned int elementIndex;
} TablePosition;

int compareForNormalSorting(const void * a, const void * b){
	long long int difference = (long long int)*(unsigned int*)a - (long long int)*(unsigned
int*)b;
	difference < INT_MIN ? difference = INT_MIN : 0;
	difference > INT_MAX ? difference = INT_MAX : 0;
	return (int)difference;
}

int compareForReverseSorting(const void * a, const void * b){
	long long int difference = (long long int)*(unsigned int*)a - (long long int)*(unsigned
int*)b;
	difference < INT_MIN ? difference = INT_MIN : 0;
	difference > INT_MAX ? difference = INT_MAX : 0;
	return (int)-difference;
}
```

```
int comparePositionsForNormalSorting(const void * a, const void * b){
	long long int difference = (long long int)((Position*)a)->positionIndex - (long long
int)((Position*)b)->positionIndex;
	difference < INT_MIN ? difference = INT_MIN : 0;
	difference > INT_MAX ? difference = INT_MAX : 0;
	return (int)difference;
}

void printTablePositions(const char * const space, TablePosition(*tablePositions)[SIZE]){
	TablePosition currentTablePosition;
	for (unsigned int iIndex = 0; iIndex < SIZE; iIndex++){
		printf("%s|", space);
		for (unsigned int jIndex = 0; jIndex < SIZE; jIndex++){
			currentTablePosition = tablePositions[iIndex][jIndex];
			if (!(currentTablePosition.flags ^ (DEFINED_POSITION_FLAG |
PERMITTED_POSITION_FLAG))){
				printf("| %4d |",
ELEMENT_INDEX_TO_ELEMENT_ID(currentTablePosition.elementIndex));
			}
			else if (currentTablePosition.flags & PERMITTED_POSITION_FLAG){
				printf("|      |");
			}
			else{
				printf("|XXXXXX|");
			}
		}
		printf("|\n");
	}
}

void markFreePositions(TablePosition(*tablePositions)[SIZE], unsigned int firstJ){
	unsigned int positionIndex = 0;
	for (unsigned int iIndex = 0; iIndex < SIZE; ++iIndex){
		for (unsigned int jIndex = 0; jIndex < SIZE; ++jIndex){
			TablePosition  * currentTablePositionPointer = firstJ ?
&tablePositions[jIndex][iIndex] : &tablePositions[iIndex][jIndex];
			if (currentTablePositionPointer->flags & PERMITTED_POSITION_FLAG){
				currentTablePositionPointer->positionIndex = positionIndex++;
			}
		}
	}
}

unsigned int getPositionsByFlags(Position * usedPositions,
TablePosition(*tablePositions)[SIZE], unsigned int flagsFilter, unsigned int flags){
	unsigned int usedPositionsIndex = 0;
	TablePosition currentTablePosition;
	for (unsigned int iIndex = 0; iIndex < SIZE; ++iIndex){
		for (unsigned int jIndex = 0; jIndex < SIZE; ++jIndex){
			currentTablePosition = tablePositions[iIndex][jIndex];
			if (!(currentTablePosition.flags & flagsFilter ^ flags)){
				usedPositions[usedPositionsIndex].elementIndex =
currentTablePosition.elementIndex;
				usedPositions[usedPositionsIndex].positionIndex =
currentTablePosition.positionIndex;
				usedPositions[usedPositionsIndex].positionIIndex = iIndex;
				usedPositions[usedPositionsIndex].positionJIndex = jIndex;
				usedPositionsIndex++;
			}
		}
	}
	return usedPositionsIndex;
}

void genTableD(unsigned int(*tableD)[PERMITTED_POSITIONS_COUNT],
TablePosition(*tablePositions)[SIZE]){
	TablePosition currentTablePositionForFirstPosition;
	TablePosition currentTablePositionForSecondPosition;
	for (unsigned int iIndexForFirstPosition = 0; iIndexForFirstPosition < SIZE;
```

```
++iIndexForFirstPosition){
		for (unsigned int jIndexForFirstPosition = 0; jIndexForFirstPosition < SIZE;
++jIndexForFirstPosition){
			currentTablePositionForFirstPosition =
tablePositions[iIndexForFirstPosition][jIndexForFirstPosition];
			if (currentTablePositionForFirstPosition.flags &
PERMITTED_POSITION_FLAG){
				for (unsigned int iIndexForSecondPosition = 0;
iIndexForSecondPosition < SIZE; ++iIndexForSecondPosition){
					for (unsigned int jIndexForSecondPosition = 0;
jIndexForSecondPosition < SIZE; ++jIndexForSecondPosition){
						currentTablePositionForSecondPosition =
tablePositions[iIndexForSecondPosition][jIndexForSecondPosition];
						if (currentTablePositionForSecondPosition.flags &
PERMITTED_POSITION_FLAG){
							unsigned int d = 0;
							iIndexForFirstPosition >
iIndexForSecondPosition ? (d += iIndexForFirstPosition - iIndexForSecondPosition) : (d +=
iIndexForSecondPosition - iIndexForFirstPosition);
							jIndexForFirstPosition >
jIndexForSecondPosition ? (d += jIndexForFirstPosition - jIndexForSecondPosition) : (d +=
jIndexForSecondPosition - jIndexForFirstPosition);

	tableD[currentTablePositionForFirstPosition.positionIndex %
PERMITTED_POSITIONS_COUNT][currentTablePositionForSecondPosition.positionIndex %
PERMITTED_POSITIONS_COUNT] = d;

	tableD[currentTablePositionForSecondPosition.positionIndex %
PERMITTED_POSITIONS_COUNT][currentTablePositionForFirstPosition.positionIndex %
PERMITTED_POSITIONS_COUNT] = d; // same
						}
					}
				}
			}
		}
	}
}
void printTableR(const char * const space, unsigned int(*tableR)[ELEMENTS_COUNT]){
	for (unsigned int iIndex = 0; iIndex < ELEMENTS_COUNT; iIndex++){
		printf("%s|", space);
		for (unsigned int jIndex = 0; jIndex < ELEMENTS_COUNT; jIndex++){
			printf("| %5d|", tableR[iIndex][jIndex]);
		}
		printf("|\n");
	}
}
void printTableD(const char * const space, unsigned int(*tableD)[PERMITTED_POSITIONS_COUNT]){
	for (unsigned int iIndex = 0; iIndex < PERMITTED_POSITIONS_COUNT; iIndex++){
		printf("%s|", space);
		for (unsigned int jIndex = 0; jIndex < PERMITTED_POSITIONS_COUNT; jIndex++){
			printf("| %5d|", tableD[iIndex][jIndex]);
		}
		printf("|\n");
	}
}

unsigned int computeF(unsigned int(*tableR)[ELEMENTS_COUNT], unsigned
int(*tableD)[PERMITTED_POSITIONS_COUNT], TablePosition(*tablePositions)[SIZE], unsigned int
firstFreeElementIndex){
	unsigned int sum = 0;

	Position usedPositions[SIZE * SIZE] = { 0 };
	unsigned int usedPositionsCount = 0;
	Position freePositions[SIZE * SIZE] = { 0 };
	unsigned int freePositionsCount = 0;

	unsigned int sortedRs[SIZE * SIZE] = { 0 };
	unsigned int sortedRsCount = 0;
	unsigned int sortedDs[SIZE * SIZE] = { 0 };
```

```
	unsigned int sortedDsCount = 0;

	usedPositionsCount = getPositionsByFlags(usedPositions, tablePositions,
PERMITTED_POSITION_FLAG | DEFINED_POSITION_FLAG, PERMITTED_POSITION_FLAG |
DEFINED_POSITION_FLAG);
	freePositionsCount = getPositionsByFlags(freePositions, tablePositions,
PERMITTED_POSITION_FLAG | DEFINED_POSITION_FLAG, PERMITTED_POSITION_FLAG);

	// for used
	for (Position * beginUsedPositionPointer = usedPositions, *beyondUsedPositionPointer =
usedPositions + usedPositionsCount; beginUsedPositionPointer < beyondUsedPositionPointer;
beginUsedPositionPointer++){
		for (Position * usedPositionPointer = beginUsedPositionPointer + 1;
usedPositionPointer < beyondUsedPositionPointer; usedPositionPointer++){
			sum +=
				tableR[beginUsedPositionPointer->elementIndex %
PERMITTED_POSITIONS_COUNT][usedPositionPointer->elementIndex % PERMITTED_POSITIONS_COUNT]
				*
				tableD[beginUsedPositionPointer->positionIndex %
PERMITTED_POSITIONS_COUNT][usedPositionPointer->positionIndex % PERMITTED_POSITIONS_COUNT];
		}
	}
	// for free
	for (unsigned int beginFreeElementIndex = firstFreeElementIndex; beginFreeElementIndex
< ELEMENTS_COUNT; beginFreeElementIndex++){
		for (unsigned int freeElementIndex = beginFreeElementIndex + 1; freeElementIndex
< ELEMENTS_COUNT; freeElementIndex++){
			sortedRs[sortedRsCount++] =
				tableR[beginFreeElementIndex][freeElementIndex];
		}
	}
	for (Position * beginFreePositionPointer = freePositions, *beyondFreePositionPointer =
freePositions + freePositionsCount; beginFreePositionPointer < beyondFreePositionPointer;
beginFreePositionPointer++){
		for (Position * freePositionPointer = beginFreePositionPointer + 1;
freePositionPointer < beyondFreePositionPointer; freePositionPointer++){
			sortedDs[sortedDsCount++] =
				tableD[beginFreePositionPointer->positionIndex %
PERMITTED_POSITIONS_COUNT]
				[freePositionPointer->positionIndex % PERMITTED_POSITIONS_COUNT];
		}
	}
	//if (sortedRsCount != sortedDsCount){
	//	printf("Bad table of positions\r\n");
	//	exit(-1);
	//}
	qsort(sortedRs, sortedRsCount, sizeof(unsigned int), compareForNormalSorting);
	qsort(sortedDs, sortedDsCount, sizeof(unsigned int), compareForReverseSorting);
	for (unsigned int index = 0; index < sortedRsCount; ++index){
		sum += sortedRs[index] * sortedDs[index];
	}
	// for used<->free
	for (Position * usedPositionPointer = usedPositions, *beyondUsedPositionPointer =
usedPositions + usedPositionsCount; usedPositionPointer < beyondUsedPositionPointer;
usedPositionPointer++){
		sortedRsCount = 0;
		sortedDsCount = 0;
		for (unsigned int freeElementIndex = firstFreeElementIndex; freeElementIndex <
ELEMENTS_COUNT; freeElementIndex++){
			sortedRs[sortedRsCount++] =
				tableR[usedPositionPointer->elementIndex %
PERMITTED_POSITIONS_COUNT][freeElementIndex];
		}
		for (Position * freePositionPointer = freePositions, *beyondFreePositionPointer
= freePositions + freePositionsCount; freePositionPointer < beyondFreePositionPointer;
freePositionPointer++){
			sortedDs[sortedDsCount++] =
				tableD[usedPositionPointer->positionIndex %
PERMITTED_POSITIONS_COUNT]
```

```
                        [freePositionPointer->positionIndex % PERMITTED_POSITIONS_COUNT];
                }
                if (sortedRsCount != sortedDsCount){ // REMOVE!
                        printf("Bad table of positions\r\n");
                        exit(-1);
                }
                qsort(sortedRs, sortedRsCount, sizeof(unsigned int), compareForNormalSorting);
                qsort(sortedDs, sortedDsCount, sizeof(unsigned int), compareForReverseSorting);
                for (unsigned int index = 0; index < sortedRsCount; ++index){
                        sum += sortedRs[index] * sortedDs[index];
                }
        }
        return sum;
}
void tablePositionsProcessing(unsigned int(*tableR)[ELEMENTS_COUNT], unsigned
int(*tableD)[PERMITTED_POSITIONS_COUNT], TablePosition(*tablePositions)[SIZE]){
        Position freePositions[SIZE * SIZE] = { 0 };
        unsigned int freePositionsCount = getPositionsByFlags(freePositions, tablePositions,
PERMITTED_POSITION_FLAG | DEFINED_POSITION_FLAG, PERMITTED_POSITION_FLAG);
        qsort(freePositions, freePositionsCount, sizeof(Position),
comparePositionsForNormalSorting);
        if (freePositionsCount != PERMITTED_POSITIONS_COUNT) {
                printf("Bad table of positions\r\n");
                exit(-1);
        }
        unsigned int minF = computeF(tableR, tableD, tablePositions, 0);
        printf("Fmin = %d\n\n", minF);
        for (unsigned int elementIndex = 0; elementIndex < ELEMENTS_COUNT; elementIndex++){
                printf("%d:\n", ELEMENT_INDEX_TO_ELEMENT_ID(elementIndex));
                unsigned int currentMinF = INT_MAX;
                unsigned int verifiedPositionForElementIIndex = 0;
                unsigned int verifiedPositionForElementJIndex = 0;
                unsigned int etap = 0;
                for (Position * freePositionPointer = freePositions, *beyondFreePositionPointer
= freePositions + freePositionsCount; freePositionPointer < beyondFreePositionPointer;
freePositionPointer++){
                        unsigned int positionForElementIIndex = freePositionPointer-
>positionIIndex % SIZE;
                        unsigned int positionForElementJIndex = freePositionPointer-
>positionJIndex % SIZE;
                        if
(!(tablePositions[positionForElementIIndex][positionForElementJIndex].flags &
(PERMITTED_POSITION_FLAG | DEFINED_POSITION_FLAG) ^ PERMITTED_POSITION_FLAG)){

        tablePositions[positionForElementIIndex][positionForElementJIndex].flags |=
DEFINED_POSITION_FLAG;

        tablePositions[positionForElementIIndex][positionForElementJIndex].elementIndex =
elementIndex;
                                unsigned int currentF = computeF(tableR, tableD, tablePositions,
elementIndex + 1);
                                printf("     %d.%d. Set element X%d in position %d:\r\n",
ELEMENT_INDEX_TO_ELEMENT_ID(elementIndex), ++etap, ELEMENT_INDEX_TO_ELEMENT_ID(elementIndex),
POSITION_INDEX_TO_POSITION_ID(freePositionPointer->positionIndex));
                                printTablePositions("     ", tablePositions);
                                printf("     F = %d\n\n", currentF);

        tablePositions[positionForElementIIndex][positionForElementJIndex].flags &=
~DEFINED_POSITION_FLAG;
                                currentMinF > currentF ? (
                                        currentMinF = currentF,
                                        verifiedPositionForElementIIndex =
positionForElementIIndex,
                                        verifiedPositionForElementJIndex = positionForElementJIndex
                                        ) : 0;
                                if (minF == currentF) { // <=
                                        break;
                                }
                        }
```

```
                }
        tablePositions[verifiedPositionForElementIIndex][verifiedPositionForElementJIndex].flag
s |= DEFINED_POSITION_FLAG;

        tablePositions[verifiedPositionForElementIIndex][verifiedPositionForElementJIndex].elem
entIndex = elementIndex;
                printf("   ->%d.%d. Fix element X%d in position %d:\r\n",
ELEMENT_INDEX_TO_ELEMENT_ID(elementIndex), ++etap,
ELEMENT_INDEX_TO_ELEMENT_ID(verifiedPositionForElementIIndex),
POSITION_INDEX_TO_POSITION_ID(verifiedPositionForElementJIndex));
                printTablePositions("     ", tablePositions);
                minF = currentMinF;
                printf("     Fmin = F = %d\r\n", minF);
                printf("\n");

        }
}
int main(void){
        unsigned int tableR[ELEMENTS_COUNT][ELEMENTS_COUNT] = R_DEFINITIONS;
        unsigned int tableD[PERMITTED_POSITIONS_COUNT][PERMITTED_POSITIONS_COUNT];
        TablePosition tablePositions[SIZE][SIZE] = POSITIONS_DEFINITIONS;

        markFreePositions(tablePositions, 1);
        genTableD(tableD, tablePositions);

        printf("Positions:\n");
        printTablePositions("", tablePositions);
        printf("\n");
        printf("R:\n");
        printTableR("", tableR);
        printf("\n");
        printf("D:\n");
        printTableD("", tableD);
        printf("\n");

        tablePositionsProcessing(tableR, tableD, tablePositions);

        printf("Processed positions:\n\n");
        printTablePositions("", tablePositions);
        printf("F = %d:\n", computeF(tableR, tableD, tablePositions, ELEMENTS_COUNT));

        return 0;
}
```

4. Варіанти завдань

1

	X	X	X
X		X	X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	2	0	3
x2		0	3	0	2
x3			0	1	2
x4				0	1
x5					0

2

	X	X	X
X		X	X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	3	0	2
x2		0	3	0	2
x3			0	2	1
x4				0	1
x5					0

3

	X	X	X
X	X		X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	3	0	3
x2		0	3	0	2
x3			0	1	2
x4				0	1
x5					0

4

	X	X	X
X	X		X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	2	0	3
x2		0	2	0	3
x3			0	2	1
x4				0	1
x5					0

5

	X	X	
X		X	X
X	X		X
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	2	0	3
x2		0	3	0	2
x3			0	1	2
x4				0	1
x5					0

6

	X	X	
X		X	X
X	X		X
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	3	0	2
x2		0	2	0	3
x3			0	2	1
x4				0	1
x5					0

7

	X	X	
X		X	X
X	X		X
X	X		X

R

	x1	x2	x3	x4	x5
x1	0	1	2	0	3
x2		0	3	0	2
x3			0	1	2
x4				0	1
x5					0

8

	X	X	
X		X	X
X	X		X
X	X		X

R

	x1	x2	x3	x4	x5
x1	0	1	3	0	2
x2		0	2	0	3
x3			0	2	1
x4				0	1
x5					0

9

	X		X
X		X	X
X	X	X	
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	2	0	3
x2		0	3	0	2
x3			0	1	2
x4				0	1
x5					0

10

	X		X
X		X	X
X	X	X	
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	1	3	0	2
x2		0	2	0	3
x3			0	2	1
x4				0	1
x5					0

11

31

4	X	X	X
X	3	X	X
X	X	5	2
X	1	X	X

36

R

	x1	x2	x3	x4	x5
x1	0	2	3	0	4
x2		0	4	0	3
x3			0	2	3
x4				0	2
x5					0

12

	X	X	X
X		X	X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	4	0	3
x2		0	3	0	4
x3			0	3	2
x4				0	2
x5					0

13

	X	X	X
X	X		X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	3	0	4
x2		0	4	0	3
x3			0	2	3
x4				0	2
x5					0

14

	X	X	X
X	X		X
X	X		
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	4	0	3
x2		0	3	0	4
x3			0	3	2
x4				0	2
x5					0

15

	X	X	
X		X	X
X	X		X
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	3	0	4
x2		0	4	0	3
x3			0	2	3
x4				0	2
x5					0

16

	X	X	
X		X	X
X	X		X
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	4	0	3
x2		0	3	0	4
x3			0	3	2
x4				0	2
x5					0

17

	X	X	
X		X	X
X	X		X
X	X		X

R

	x1	x2	x3	x4	x5
x1	0	2	3	0	4
x2		0	4	0	3
x3			0	2	3
x4				0	2
x5					0

18

	X	X	
X		X	X
X	X		X
X	X		X

R

	x1	x2	x3	x4	x5
x1	0	2	4	0	3
x2		0	3	0	4
x3			0	3	2
x4				0	2
x5					0

19

	X		X
X		X	X
X	X	X	
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	3	0	4
x2		0	4	0	3
x3			0	2	3
x4				0	2
x5					0

20

	X		X
X		X	X
X	X	X	
X		X	X

R

	x1	x2	x3	x4	x5
x1	0	2	4	0	3
x2		0	3	0	4
x3			0	3	2
x4				0	2
x5					0

21

		X	X
	X		X
X	X	X	
X	X	X	X

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	0	2
x3			0	1	2
x4				0	5
x5					0

22

		X	X
	X		X
X	X	X	
X	X	X	X

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	3	2
x3			0	1	2
x4				0	1
x5					0

23

	X	X	X
		X	X
X	X		X
X	X	X	

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	0	2
x3			0	1	2
x4				0	5
x5					0

24

	X	X	X
		X	X
X	X		X
X	X	X	

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	3	2
x3			0	1	2
x4				0	1
x5					0

25

		X	X
X	X		X
	X	X	
X	X	X	X

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	0	2
x3			0	1	2
x4				0	5
x5					0

26

		X	X
X	X		X
	X	X	
X	X	X	X

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	3	2
x3			0	1	2
x4				0	1
x5					0

27

	X	X	X
X		X	X
	X		X
X	X	X	

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	0	2
x3			0	1	2
x4				0	5
x5					0

28

	X	X	X
X		X	X
	X		X
X	X	X	

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	3	2
x3			0	1	2
x4				0	1
x5					0

29

		X	X
X	X		X
X	X	X	
	X	X	X

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	0	2
x3			0	1	2
x4				0	5
x5					0

30

		X	X
X	X		X
X	X	X	
	X	X	X

R

	x1	x2	x3	x4	x5
x1	0	5	3	0	1
x2		0	3	3	2
x3			0	1	2
x4				0	1
x5					0

5. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.