Решение заданий по программированию заключительного этапа Всероссийской олимпиады среди студентов ССУЗов (Курский государственный политехнический колледж, 2014)

Замечание. Варианты решений, представленные ниже, являются правильными, но не единственными и допускают доработку.

Задача 1. Фигура

Наиболее простой подход (полный перебор) основан на том, что координаты на плоскости представлены целыми числами, соответственно можно перебрать все точки в паре вложенных циклов и добавлять очередной квадрат единичной площади к уже имеющимся в случае, если он входит в состав хотя бы одного из исходных прямоугольников. Псевдокод представлен ниже.

```
S := 0;

for I := -10000 to 10000-1 do
  for J := -10000 to 10000-1 do
    for K := 0 to N-1 do
      if КвадратВходитВПрямоугольник(I, J, I+1, J+1, Прямоугольник[K]) then begin
        Inc(S);
        break;
      end;
```

Недостатком данного подхода является его вычислительная сложность, т.к. необходимо проверить порядка $20\,000^2 N$ вхождений.

Альтернативным подходом является просмотр всех исходных прямоугольников, отметка в отдельном двумерном массиве клеток, попадающих в их состав, затем подсчет отмеченных клеток.

```
var
  A: array [-10000..10000, -10000..10000] of Boolean;

begin
  { Отметка квадратов }
  for K := 0 to ЧислоПрямоугольников-1 do
    for I := Min(Прямоугольник[K].X1, Прямоугольник[K].X2) to
             Max(Прямоугольник[K].X1, Прямоугольник[K].X2) do
      for J := Min(Прямоугольник[K].Y1, Прямоугольник[K].Y2) to
               Max(Прямоугольник[K].Y1, Прямоугольник[K].Y2) do
        A[I][J] := True;

  S := 0;

  { Подсчет числа отмеченных }
  for I := -10000 to 10000 do
    for J := -10000 to 10000 do
      if A[I][J] then
        Inc(S);
```

Данный вариант решения не сильно лучше предыдущего, т.к. он требует массива, размер которого составляет $20\,000^2$ Б $= 381$ МБ (по условию разрешалось использовать не более 128 МБ, да и в кэш он явно не влезет, соответственно операции с ним будут довольно медленными). В ходе отметки квадратов потребуется сделать порядка $\sum_{i=1}^{N} S_i$ отметок квадратов (первая группа циклов, S_i – площадь *i*-го прямоугольника), а на просмотр массива (вторая группа циклов) – $20\,000^2$ итераций. Если площади прямоугольников велики, то вычислительная сложность решения приближается к предыдущему варианту решения.

Еще одним вариантом решения является попытка построения многоугольника, который будет образован исходными прямоугольниками. Допускаю, что такой вариант решения вполне может быть разработан, однако мне он кажется очень громоздким и

требующим проверки большого числа частных случаев, например, как показано на рисунке 1.

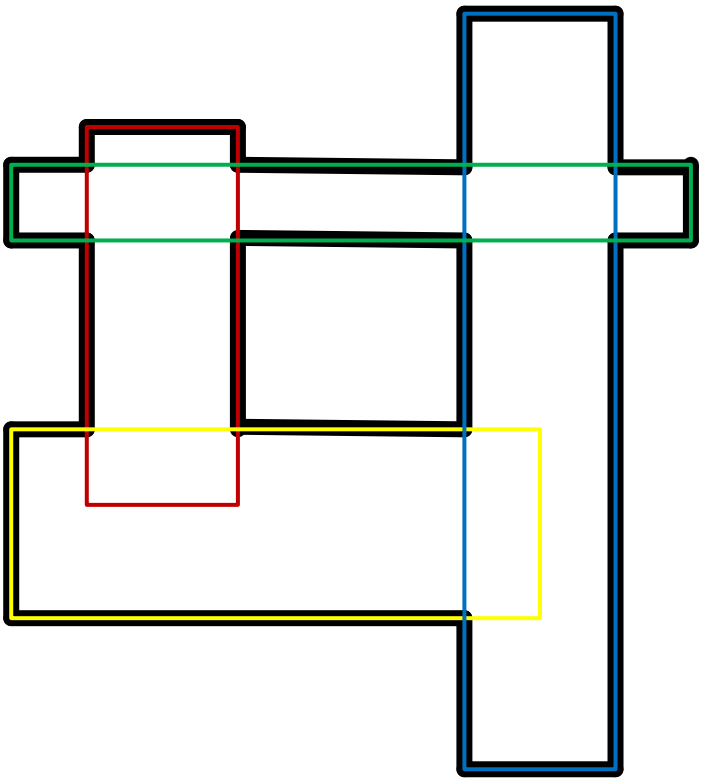

Рис. 1. Пример расположения прямоугольников. Фигура, образованная их комбинацией, имеет нетривиальную форму с множеством углов и «дырку» посередине

Наиболее правильным на мой взгляд является комбинация первого и последнего подходов. Чтобы ее осуществить, необходимо каким-то образом ликвидировать большое число узлов в сетке (недостаток первого подхода) и каким-либо несложным образом закодировать фигуру из последнего способа. Для этого фигуру можно представить совокупностью элементарных прямоугольников (не квадратов единичной площади!), площадь которых в общем случае будет больше единицы. Координаты прямоугольников будут опираться не на исходную координатную сетку (–10 000, –9 999, –9 998, …, 10 000), а на более грубую, образованную множествами координат исходных прямоугольников (см. рис. 2).

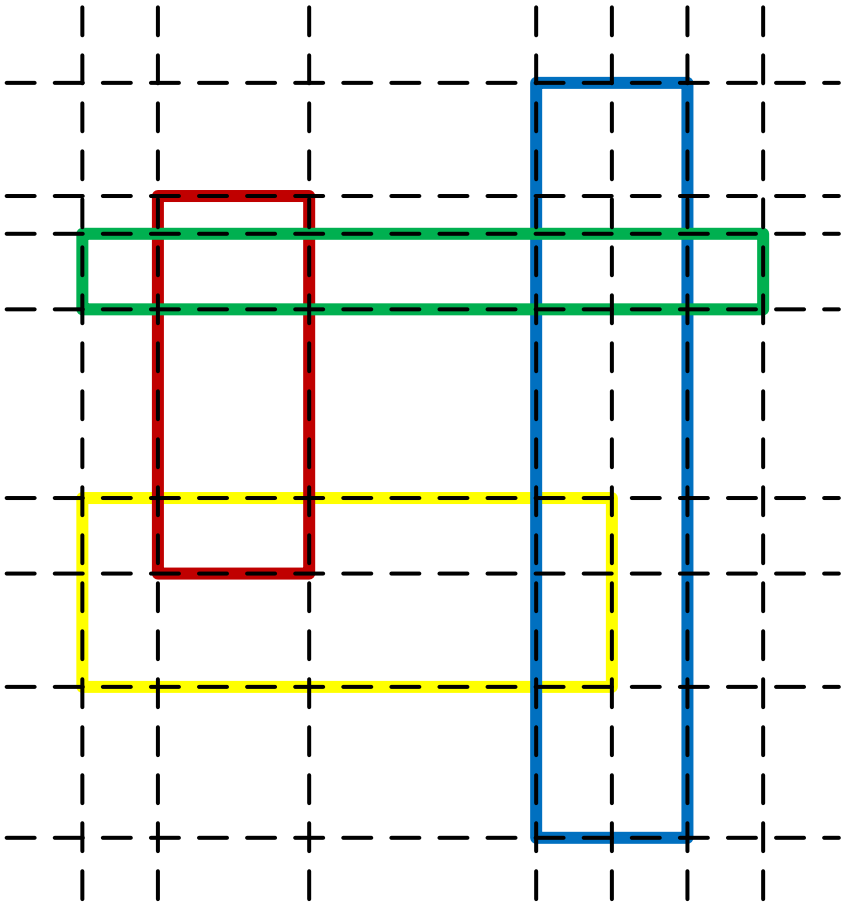

Рис. 2. Более грубая координатная сетка, образованная множествами координат прямоугольников

Данные множества удобно хранить в виде отсортированных одномерных массивов или списков без дублирования элементов. Вычисление искомой площади будет сводиться к просмотру всех прямоугольников данной грубой сетки, определению для каждого из них факта принадлежности к одному из исходных прямоугольников и нахождению суммы их площадей (рис. 3).

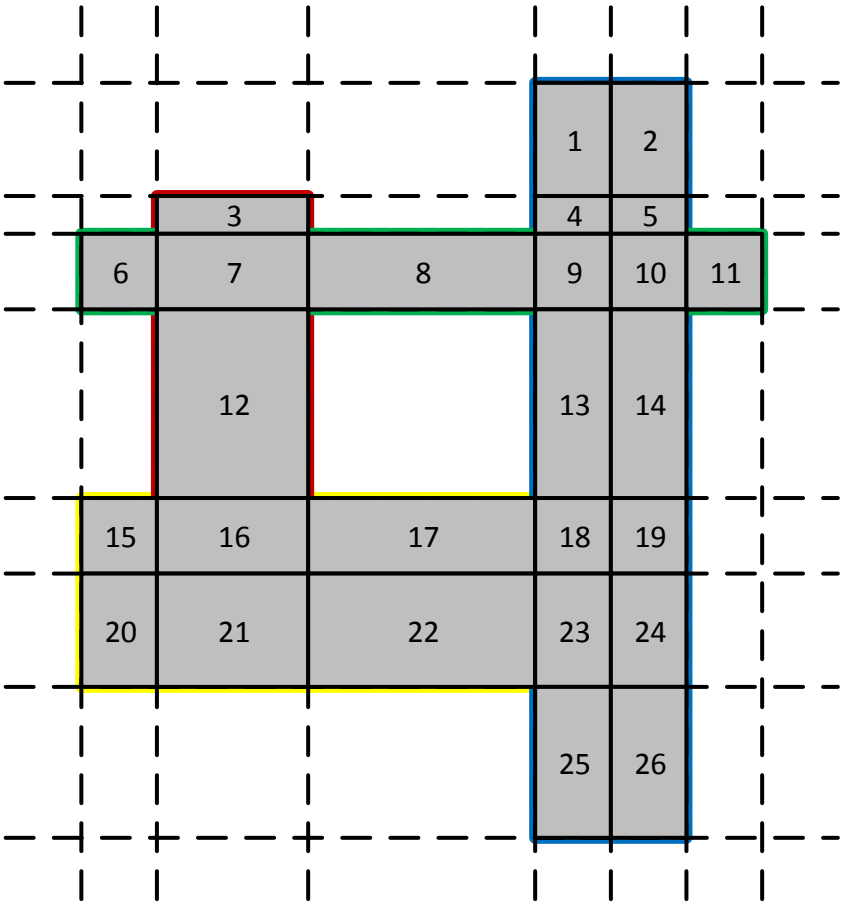

Рис. 3. Аппроксимация результирующей фигуры множеством непересекающихся прямоугольников, координаты которых привязаны к грубой сетке

Вычислительная сложность решения составляет $|X|\cdot|Y|\cdot N$, где $|X|$ и $|Y|$ – число различных координат по соответствующим осям в грубой сетке. Данный способ аппроксимации не является минимальным, однако для решения задачи этого достаточно. Подозреваю, что его можно улучшить на предмет дальнейшего снижения вычислительной сложности путем аппроксимации фигуры меньшим числом прямоугольников (см. идею на рис. 4), реализацию данного способа оставляю вам ☺.

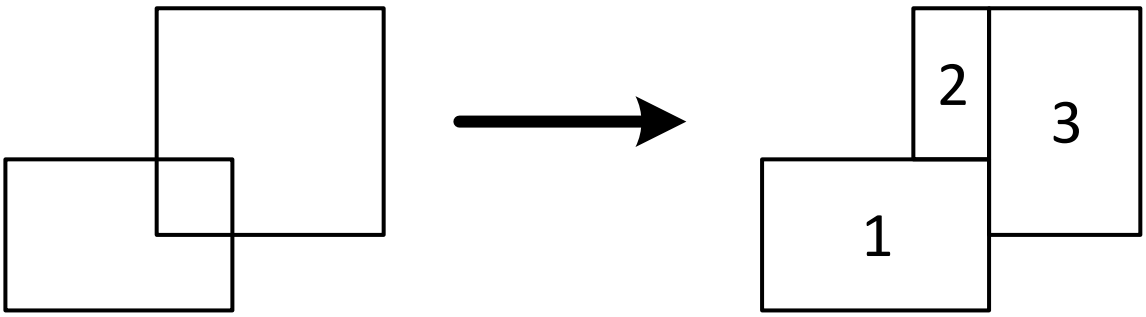

Рис. 4. Аппроксимация результирующей фигуры меньшим числом прямоугольников

Полный листинг программы приведен ниже.

```
program Task1;
{$APPTYPE CONSOLE}

uses
  Math;

type
  { Прямоугольник }
  TRect = record
```

```
    X1, Y1, X2, Y2: Integer;
  end;

  { Множество прямоугольников }
  TRectsSet = array of TRect;

  { Список координат }
  TList = array of Integer;


{ Добавление очередной координаты в список }
{ При добавлении список поддерживается отсортированным, дублирующиеся координаты отбрасываются }
procedure AddToList(var List: TList; X: Integer);
var
  I, J: Integer;
begin
  I := 0;

  { Поиск позиции }
  // Замечание: скорость работы можно повысить путем использования бинарного поиска!
  while (I < Length(List)) and (X >= List[I]) do begin
    { Проверка дублирования }
    if List[I] = X then
      exit;

    Inc(I);
  end;

  { Добавление элемента в список с сохранением сортировки }
  // Замечание: много realloc'ов, можно избежать путем задания заведомо большой длины массива!
  SetLength(List, Length(List)+1);

  for J := High(List)-1 downto I do
    List[J+1] := List[J];

  List[I] := X;
end;


{ Площадь прямоугольника }
function GetRectSquare(const R: TRect): Integer; overload;
begin
  Result := Abs( (R.X1 - R.X2) * (R.Y1 - R.Y2) );
end;


function GetRectSquare(X1, Y1, X2, Y2: Integer): Integer; overload;
begin
  Result := Abs( (X1 - X2) * (Y1 - Y2) );
end;


{ Входит ли данный прямоугольник в другой прямоугольник }
function IsRectInRect(const X1, Y1, X2, Y2: Integer; const Rect: TRect): Boolean;
var
  MinX, MaxX, MinY, MaxY: Integer;
begin
  // Замечание: данные действия можно сделать однократно вначале!
  MinX := Min(Rect.X1, Rect.X2);
  MaxX := Max(Rect.X1, Rect.X2);

  MinY := Min(Rect.Y1, Rect.Y2);
  MaxY := Max(Rect.Y1, Rect.Y2);

  Result := (MinX <= X1) and (X1 <= MaxX) and
       (MinX <= X2) and (X2 <= MaxX) and
       (MinY <= Y1) and (Y1 <= MaxY) and
       (MinY <= Y2) and (Y2 <= MaxY);
end;


{ Входит ли данный прямоугольник в состав одного из прямоугольников из заданного набора }
function IsRectInSet(X1, Y1, X2, Y2: Integer; const RectsSet: TRectsSet): Boolean;
var
  I: Integer;
begin
  Result := True;

  for I := 0 to High(RectsSet) do
    if IsRectInRect(X1, Y1, X2, Y2, RectsSet[I]) then
      exit;
```

```
  Result := False;
end;


function GetRectsSetSquare(const RectsSet: TRectsSet): Integer;
var
  XList, YList: TList;
  I, J: Integer;
begin
  Result := 0;

  { Построение списков координат }
  XList := nil;
  YList := nil;

  for I := 0 to High(RectsSet) do begin
    AddToList(XList, RectsSet[I].X1);
    AddToList(XList, RectsSet[I].X2);

    AddToList(YList, RectsSet[I].Y1);
    AddToList(YList, RectsSet[I].Y2);
  end;

  { Просмотр списков координат }
  for I := 0 to High(XList)-1 do
    for J := 0 to High(YList)-1 do
      if IsRectInSet(XList[I], YList[J], XList[I+1], YList[J+1], RectsSet) then
        Result := Result + GetRectSquare(XList[I], YList[J], XList[I+1], YList[J+1]);
end;


{ Исходные данные }
var
  RS: TRectsSet;
  S: Integer;

{ Загрузка исходных данных }
procedure LoadSourceData();
begin
  // Замечание: в исходном варианте задания загрузка должна производиться из файла!
  SetLength(RS, 2);

  with RS[0] do begin
    X1 := 1; Y1 := 1; X2 := 3; Y2 := 3;
  end;

  with RS[1] do begin
    X1 := 2; Y1 := 2; X2 := 4; Y2 := 4;
  end;
end;


begin
  { Загрузка исходных данных }
  LoadSourceData();

  { Расчет площади }
  S := GetRectsSetSquare(RS);

  { Вывод результата }
  // Замечание: в исходном варианте задания вывод результата должен производиться в файла!
  Writeln(S);
  Readln;
end.
```

Задача 2. Заправки

На мой взгляд здесь все просто, а задача может быть решена упрощенным аналогом алгоритма Дейкстры (цена бензина неотрицательна!). Для этого необходимо пометить все клетки, за исключением начальной, как еще не достигнутые. Далее для каждой уже достигнутой клетки, необходимо проверить, нельзя ли попасть в клетку справа или снизу так, что найденный путь будет являться первым найденным или будет короче предыдущего, сохраняя при этом направление, откуда пришли (соответственно слева или сверху). После итеративного повторения процесса рано или поздно улучшать станет нечего (таблица перестанет меняться), после чего найденный путь необходимо «распутать» в обратном порядке – от последней (нижней правой) клетки к первой (см. рис. 5).

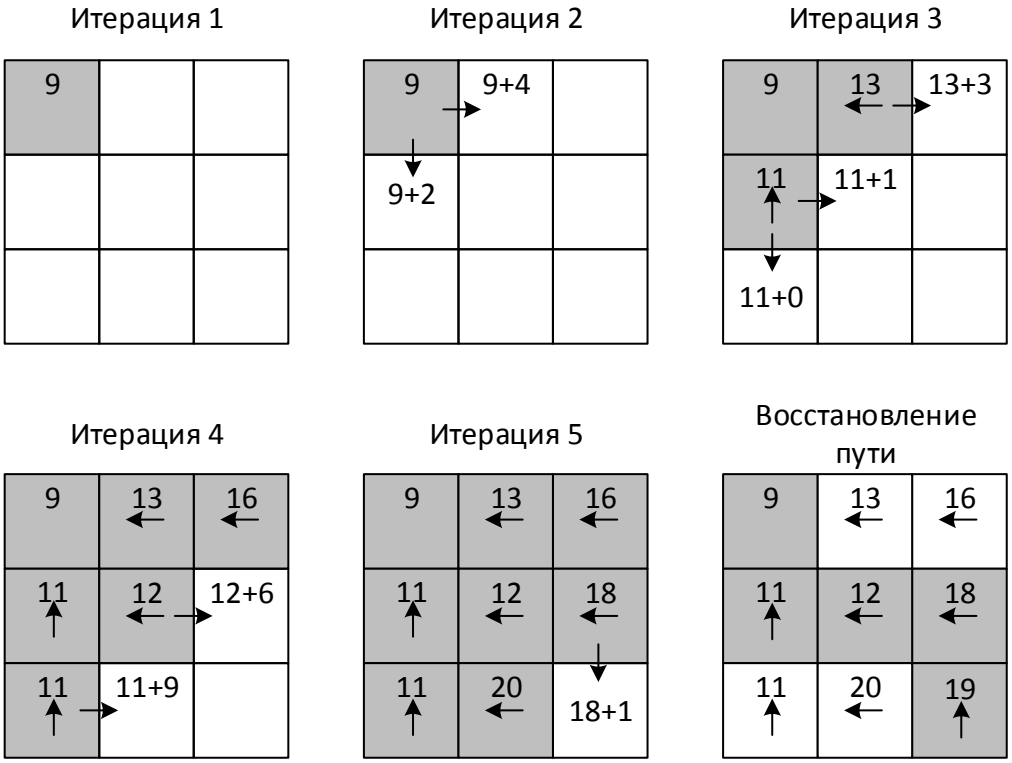

Рис. 5. Построение оптимального пути модификацией алгоритма Дейкстры

Сразу хочется уберечь от соблазна использования жадного подхода (идти в смежную с текущей клетку с минимальной стоимостью бензина), т.к. известно, что в данной задаче жадность до добра не доводит, а решения в общем случае получаются весьма плохими.

Листинг программы приведен ниже.

```
program Task2;
{$APPTYPE CONSOLE}

type
  { Матрица }
  TMatrix = array of array of Integer;

  { Маршрут }
  // Замечание: можно обойтись списком вместо двумерного массива!
  TWay = array of array of Boolean;

  { Направление, из которого в ячейку пришел текущий кратчайший путь }
  TDirection = (drUnknown, drFromTop, drFromLeft);

  { Параметры ячейки матрица }
  TCell = record
    Visited: Boolean;        { Достигнута ли ячейки }
```

```
    CurrPathLen: Integer;    { Длина кратчайшего пути до ячейки }
    From: TDirection;        { Откуда пришел кратчайший путь }
  end;

  TCells = array of array of TCell;


procedure FindWay(const M: TMatrix; var W: TWay; var L: Integer);
var
  C: TCells;
  I, J: Integer;
  Changed: Boolean;
  CurrX, CurrY: Integer;
begin
  SetLength(C, Length(M), Length(M));

  { Исходная информация: ни одна ячейка не посещена }
  for I := 0 to High(C) do
    for J := 0 to High(C) do
      with C[I][J] do begin
        Visited := False;
        CurrPathLen := High(Integer);    { Заведомо большое значение }
        From := drUnknown;
      end;

  { Кроме верхней левой }
  with C[0][0] do begin
    Visited := True;
    CurrPathLen := M[0][0];
  end;

  repeat
    Changed := False;

    { Просмотр ячеек таблицы, корректировка длин маршрутов }
    // Замечание вместо просмотра N^2 ячеек каждый раз можно организовать список, в котором хранить
    // "волновой фронт" — список обновленных клеток с прошлого прохода!
    for I := 0 to High(M) do
      for J := 0 to High(M) do begin
        { Непосещенные клетки не рассматриваем }
        if not C[I][J].Visited then
          continue;

        { Шаг вправо }
        if (I < High(M)) and                                          { Есть куда идти вправо }
           (C[I][J].CurrPathLen + M[I+1][J] < C[I+1][J].CurrPathLen)  { Путь вправо короче
                                                                        предыдущего }
        then with C[I+1][J] do begin
          Visited := True;
          CurrPathLen := C[I][J].CurrPathLen + M[I+1][J];
          From := drFromLeft;
          Changed := True;
        end;

        { Шаг вниз }
        if (J < High(M)) and                                          { Есть куда идти вправо }
           (C[I][J].CurrPathLen + M[I][J+1] < C[I][J+1].CurrPathLen)  { Путь вправо короче
                                                                        предыдущего }
        then with C[I][J+1] do begin
          Visited := True;
          CurrPathLen := C[I][J].CurrPathLen + M[I][J+1];
          From := drFromTop;
          Changed := True;
        end;
      end;
  until not Changed;

  { Построение пути }
  CurrX := Length(M)-1;
  CurrY := Length(M)-1;

  SetLength(W, Length(M), Length(M));

  for I := 0 to High(W) do
    for J := 0 to High(W) do
      W[I][J] := False;

  W[0][0] := True;

  while not ((CurrX = 0) and (CurrY = 0)) do begin
    W[CurrX][CurrY] := True;
```

```
    case C[CurrX][CurrY].From of
      drFromTop: CurrY := CurrY - 1;
      drFromLeft: CurrX := CurrX - 1;
    else
      ASSERT(False);
    end;
  end;

  { Длина пути }
  L := C[High(C)][High(C)].CurrPathLen;

  C := nil;
end;


procedure PrintWay(const W: TWay);
var
  I, J: Integer;
begin
  for I := 0 to High(W) do begin
    for J := 0 to High(W) do
      Write(Byte(W[I][J]));

    Writeln;
    end;
end;


{ Исходные данные }
var
  M: TMatrix;
  W: TWay;
  L: Integer;


{ Загрузка исходных данных }
procedure LoadSourceData();
begin
  SetLength(M, 3, 3);

  M[0][0] := 9; M[0][1] := 4; M[0][2] := 3;
  M[1][0] := 2; M[1][1] := 1; M[1][2] := 6;
  M[2][0] := 0; M[2][1] := 9; M[2][2] := 1;
end;


begin
  { Загрузка исходных данных }
  LoadSourceData();

  { Поиск решения }
  FindWay(M, W, L);

  { Вывод решения }
  Writeln(L);
  Writeln;
  PrintWay(W);

  Readln;
end.
```

Задача 3. Динамический приз

Прежде всего необходимо оценить количество монет, которые получает игрок, выбравший в качестве начальной клетку доски с координатами (i, j). Для этого необходимо определить множество клеток, достижимых из начальной клетки не более чем за P шагов, для чего можно воспользоваться волновым алгоритмом. На рисунке 6 показан вариант построения окрестности диаметром $P = 3$, при этом в каждой клетке сохраняется минимальное число шагов d_{ij}, необходимое шахматному коню для ее достижения из начальной клетки.

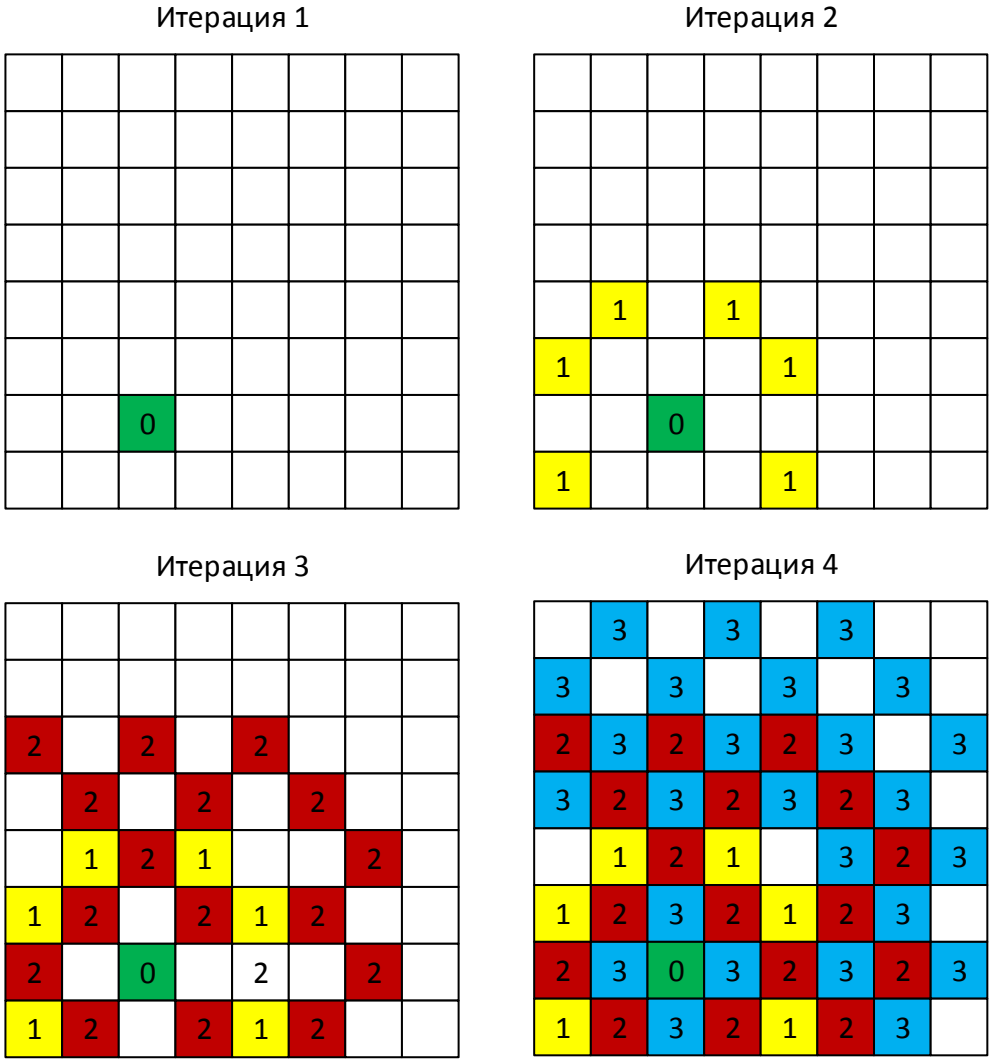

Рис. 6. Пример построения окрестности текущего положения коня

Оценка суммарного числа монет может быть произведена по формуле $C = \sum_{i=1}^{8}\sum_{j=1}^{8} 2^{P-d_{ij}} c_{ij}$, где $c_{ij} = 1$ в случае ее достижимости за число шагов, не превышающее P, и $c_{ij} = 0$ в противном случае.

Далее необходимо перебрать все возможные клетки шахматной доски, произвести оценку числа монет для каждой из них и выбрать ту, для которой $C < M$ и $C \to \max$. При этом можно заметить, что имеет место ряд симметрий и расположение коня в одной из симметричных клеток даст решение того же качества (рис. 7).

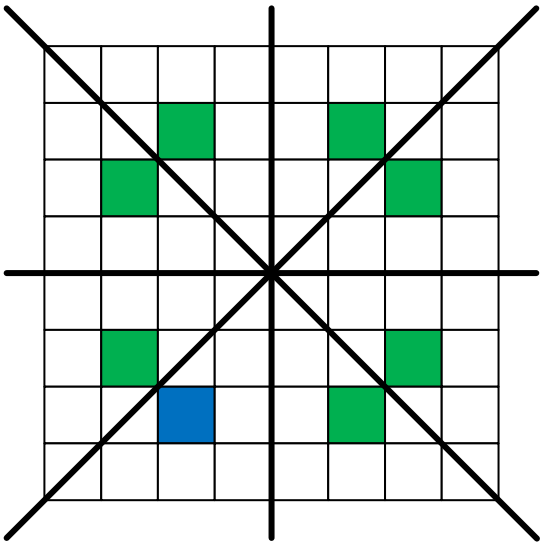

Рис. 7. Симметрии на шахматной доске

Учет симметрий позволяет производить обход не всей шахматной доски (64 клетки), а лишь 10 клеток любого из восьми треугольных сегментов. Соответствующий код программы приведен ниже.

```
program Task3;
{$APPTYPE CONSOLE}

const
  N = 8;

type
  { Шахматная доска }
  TBoard = array [1..N, 1..N] of Integer;


procedure MarkBoard(X, Y, P: Integer; var Board: TBoard);

  procedure MarkCell(X, Y, Mark: Integer);
  begin
    if (0 < X) and (X <= N) and    { Клетка в пределах доски }
       (0 < Y) and (Y <= N) and
       (Board[X][Y] < 0)           { Клетка еще не посещена }
    then
      Board[X][Y] := Mark;
  end;

  procedure MarkPositions(CurrMark: Integer);
  var
    I, J: Integer;
  begin
    for I := 1 to N do
      for J := 1 to N do begin
        { Пропускаем клетки доски, отметка которых не совпадает с заданной }
        if Board[I][J] <> CurrMark then
          continue;

        MarkCell(I+1, J+2, CurrMark+1);
        MarkCell(I+1, J-2, CurrMark+1);

        MarkCell(I-1, J+2, CurrMark+1);
        MarkCell(I-1, J-2, CurrMark+1);

        MarkCell(I+2, J+1, CurrMark+1);
        MarkCell(I+2, J-1, CurrMark+1);

        MarkCell(I-2, J+1, CurrMark+1);
        MarkCell(I-2, J-1, CurrMark+1);
      end;
  end;

var
  I, J, K: Integer;
begin
  { Очистка доски }
```

```
  for I := 1 to N do
    for J := 1 to N do
      Board[I][J] := -1;

  { Начальное расположение коня }
  Board[X][Y] := 0;

  { Определение достижимости }
  for K := 0 to P-1 do
    MarkPositions(K);
end;


function GetCoinsCnt(const Board: TBoard; P: Integer): Integer;
var
  I, J: Integer;
begin
  Result := 0;

  for I := 1 to N do
    for J := 1 to N do
       if Board[I][J] >= 0 then
        Result := Result + 1 shl (P - Board[I][J]);
end;


var
  B: TBoard;
  I, J, C, P, M: Integer;
  BestC, BestI, BestJ: Integer;


{ Загрузка исходных данных }
procedure LoadSourceData();
begin
  P := 1;
  M := 5;
end;


begin
  { Загрузка исходных данных }
  LoadSourceData();

  BestC := -1;

  { Просмотр части доски }
  for I := 1 to N div 2 do
    for J := I to N div 2 do begin
      { Расчет достижимости для начальной клетки (I, J) }
      MarkBoard(I, J, P, B);

      { Расчет числа монет }
      C := GetCoinsCnt(B, P);

      if (C > BestC) and (C <= M) then begin
        BestC := C;
        BestI := I;
        BestJ := J;
      end;
    end;

  { Вывод решения }
  if BestC >= 0 then begin
    Writeln(BestC);
    Writeln(BestI, ' ', BestJ);
  end else
    Writeln('Not found');

  Readln;
end.
```