Editorials

Requirements:

- graph theory:
  - DFS, BFS, Dijkstra
- IA:
  - A*

PacMan – DFS editorial

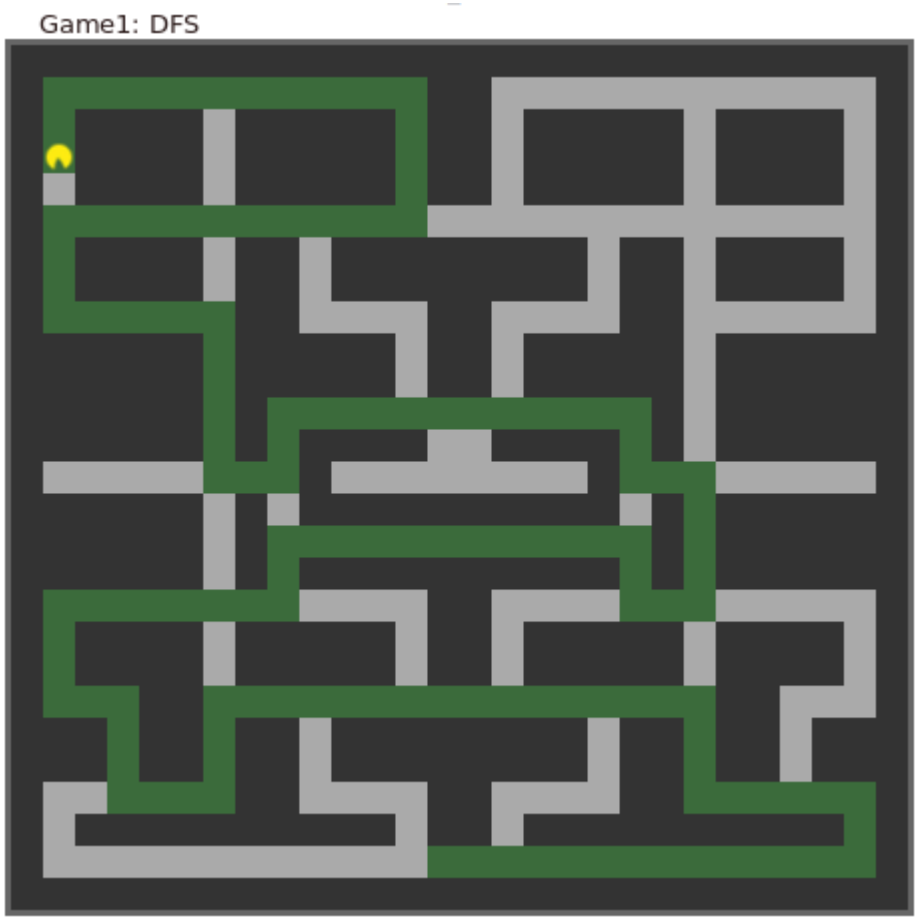
We don't need to find the shortest path in this exercise, just a path.

Code C++

```
#include <bits/stdc++.h>

using namespace std;

pair<int,int> operator+(const pair<int, int>& x, const pair<int, int>& y) {
    return make_pair(x.first+y.first, x.second+y.second);
}

void dfs (int r,
  int c,
  pair<int, int> start,
  pair<int, int> end,
  vector<char> *grid,
  vector<pair<int, int>> & expandedNodes,
  vector<pair<int, int>> & path) {
    vector<pair<int, int>> tempPath;

    map<pair<int, int>, bool> visited;

    stack<pair<int, int>> s;
    s.push(start);

    stack<vector<pair<int, int>>> stackPath;
    stackPath.push(vector<pair<int, int>>());

    pair<int, int> neighbors[4] = {{-1, 0}, {0, -1},{0, 1},{1, 0}};
```

```
 while (!s.empty()) {
  pair<int, int> current = s.top();
  s.pop();

  vector<pair<int, int>> tmpPath = stackPath.top();
  stackPath.pop();

  tmpPath.push_back(current);

  visited.insert(make_pair(current, true));

  expandedNodes.push_back(current);

  if (current == end){
   if (path.size() == 0) {
    path = tmpPath;
    break;
   }
  }

  for (auto cn: neighbors) {
   pair<int, int> neighbor = {current + cn};

   if (neighbor.first < 0 || neighbor.first >= r || neighbor.second < 0 && neighbor.second >= r)
    continue;

   if (grid[neighbor.first][neighbor.second] == '-' || grid[neighbor.first][neighbor.second] == '.') {
    if (!visited[neighbor]) {
     visited[neighbor] = true;
     s.push(neighbor);
     stackPath.push(tmpPath);
    }
   }
  }
 }
}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int pr, pc;
  cin >> pr >> pc;

  pair<int, int> start = {pr, pc};

  int fr, fc;
  cin >> fr >> fc;

  int r, c;
  cin >> r >> c;

  pair<int, int> end = {fr, fc};

  vector<char> *pacManGrid = new vector<char>[c];
  for (int ri = 0 ; ri < r ; ++ri)
    for (int ci = 0 ; ci < c ; ++ci) {
      char node;
      cin >> node;
      pacManGrid[ri].push_back(node);
    }

    vector<pair<int, int>> expandedNodes;
    vector<pair<int, int>> path;
    dfs(r, c, start, end, pacManGrid, expandedNodes, path);

    cout << expandedNodes.size() << '\n';
    for (auto node: expandedNodes)
      cout << node.first << ' ' << node.second << '\n';

    cout << path.size() - 1 << '\n';
    for (auto node: path)
      cout << node.first << ' ' << node.second << '\n';

    /*for (int ri = 0 ; ri < r ; ++ri){
      for (int ci = 0 ; ci < c ; ++ci)
        cout << pacManGrid[ri][ci] << ' ';
      cout << '\n';
    }*/

    return 0;

}
```



As you see, the DFS algorithm doesn’t give the shortest path. We need to use a better algorithm.

PacMan – BFS editorial

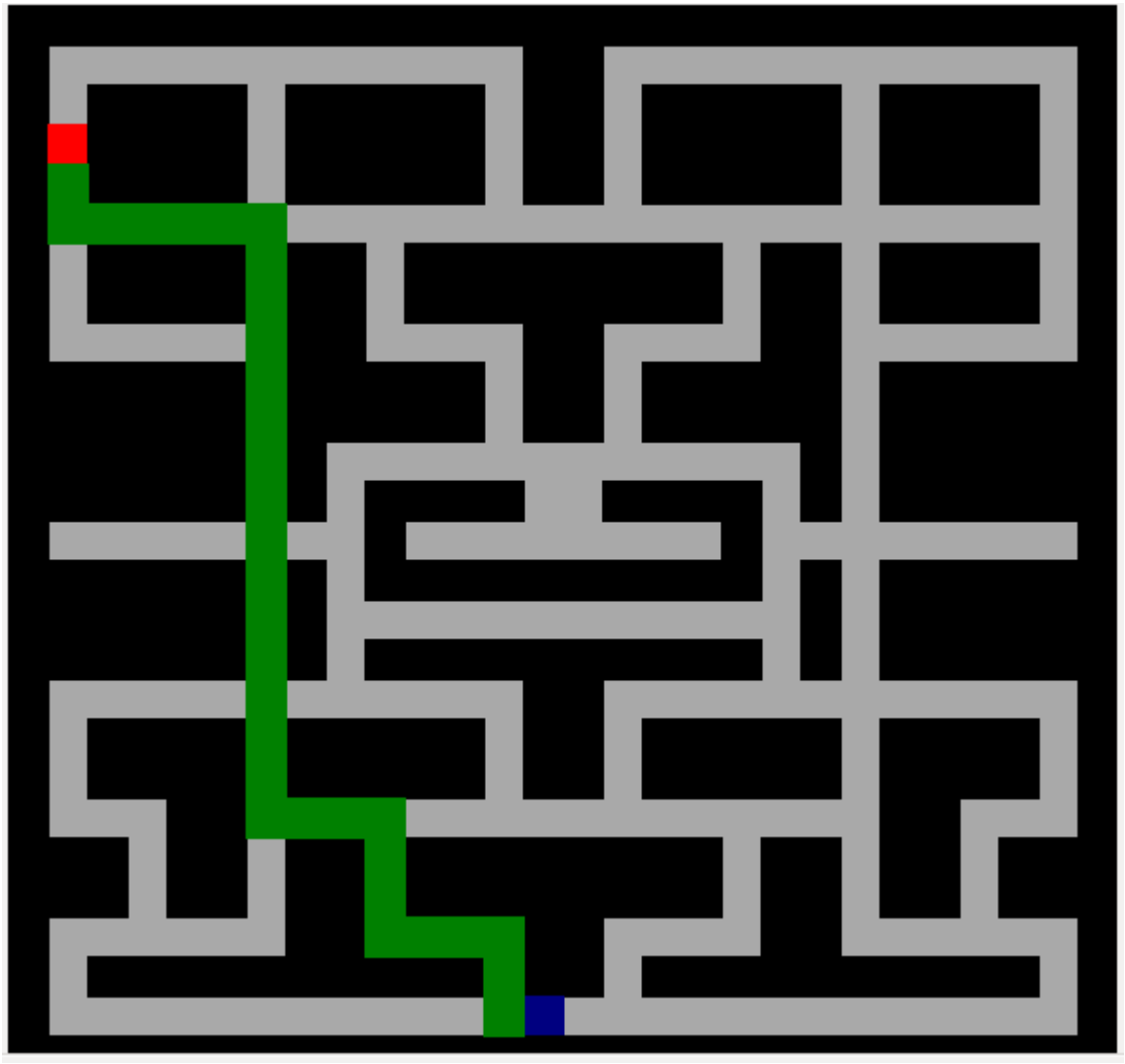
Perform the search by using the BFS algorithm.

C++ code

```
#include <bits/stdc++.h>

using namespace std;

pair<int,int> operator+(const pair<int, int>& x, const pair<int, int>& y) {
    return make_pair(x.first+y.first, x.second+y.second);
}

void bfs (int r,
  int c,
  pair<int, int> start,
  pair<int, int> end,
  vector<char> *grid,
  vector<pair<int, int>> & expandedNodes,
  vector<pair<int, int>> & path) {
    vector<pair<int, int>> tempPath;

    map<pair<int, int>, bool> visited;

    queue<pair<int, int>> q;
    q.push(start);

    queue<vector<pair<int, int>>> queuePath;
    queuePath.push(vector<pair<int, int>>());

    pair<int, int> neighbors[4] = {{-1, 0}, {0, -1},{0, 1},{1, 0}};

  while (!q.empty()) {
    pair<int, int> current =q.front();
    q.pop();

    vector<pair<int, int>> tmpPath = queuePath.front();
    queuePath.pop();

    tmpPath.push_back(current);

    visited.insert(make_pair(current, true));

    expandedNodes.push_back(current);
```

```
    if (current == end){
      if (path.size() == 0) {
        path = tmpPath;
        break;
      }
    }

    for (auto cn: neighbors) {
      pair<int, int> neighbor = {current + cn};

      if (neighbor.first < 0 || neighbor.first >= r || neighbor.second < 0 && neighbor.second >= r)
        continue;

      if (grid[neighbor.first][neighbor.second] == '-' || grid[neighbor.first][neighbor.second] == '.') {
        if (!visited[neighbor]) {
          visited[neighbor] = true;
          q.push(neighbor);
          queuePath.push(tmpPath);
        }
      }
    }
  }
}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int pr, pc;
  cin >> pr >> pc;

  pair<int, int> start = {pr, pc};

  int fr, fc;
  cin >> fr >> fc;

  int r, c;
  cin >> r >> c;

  pair<int, int> end = {fr, fc};

  vector<char> *pacManGrid = new vector<char>[c];
  for (int ri = 0 ; ri < r ; ++ri)
    for (int ci = 0 ; ci < c ; ++ci) {
      char node;
      cin >> node;
      pacManGrid[ri].push_back(node);
    }

    vector<pair<int, int>> expandedNodes;
    vector<pair<int, int>> path;
    bfs(r, c, start, end, pacManGrid, expandedNodes, path);

    cout << expandedNodes.size() << '\n';
    for (auto node: expandedNodes)
      cout << node.first << ' ' << node.second << '\n';

    cout << path.size() - 1 << '\n';
    for (auto node: path)
      cout << node.first << ' ' << node.second << '\n';

    /*for (int ri = 0 ; ri < r ; ++ri){
      for (int ci = 0 ; ci < c ; ++ci)
        cout << pacManGrid[ri][ci] << ' ';
      cout << '\n';
    }*/

    return 0;

}
```



Pacman A* editorial

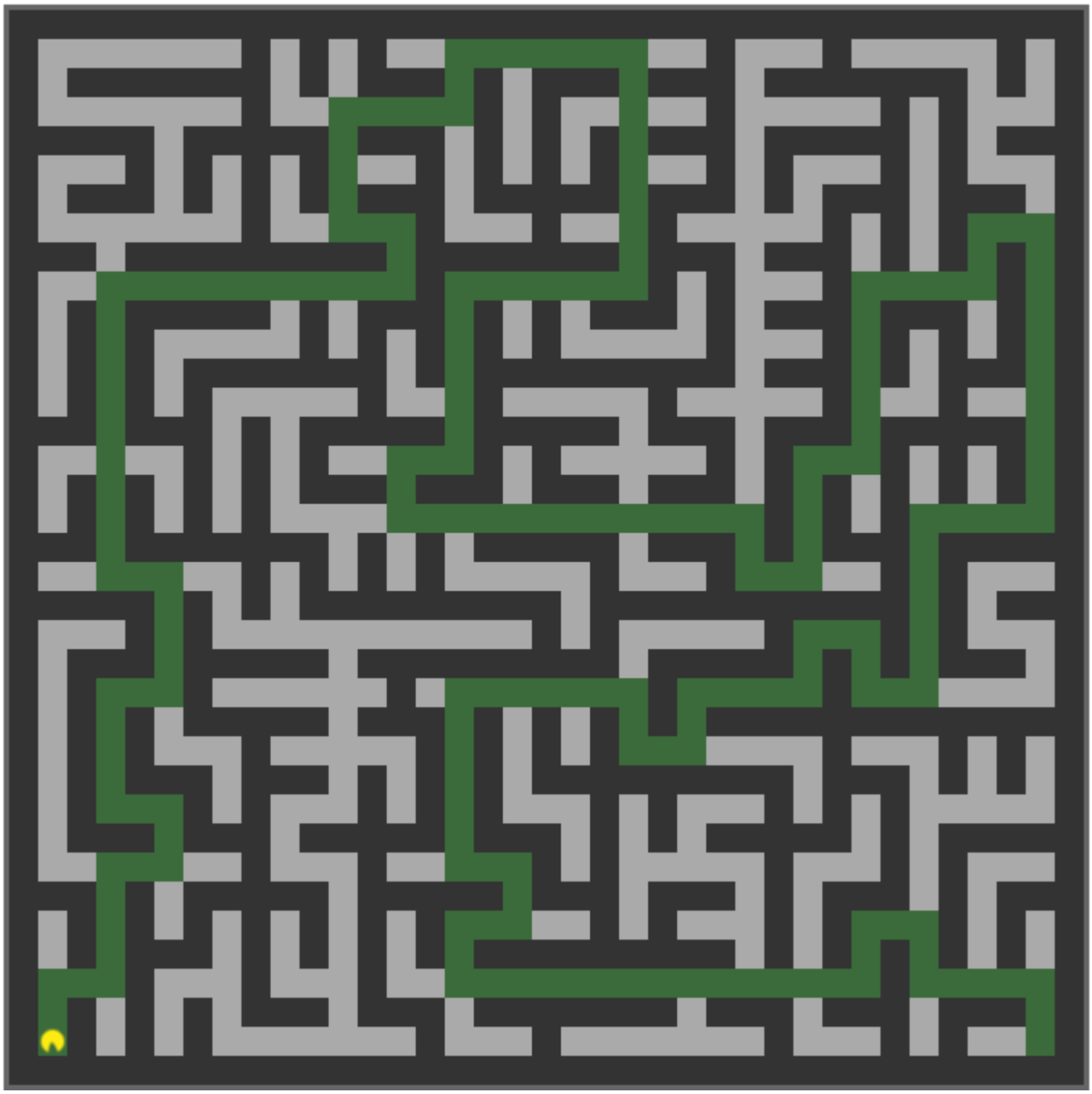
C++ code

```
#include <bits/stdc++.h>

using namespace std;

int const INF = 1e9;

//map<pair<int, int>, int> dist;

pair<int,int> operator+(const pair<int, int>& x, const pair<int, int>& y) {
    return make_pair(x.first+y.first, x.second+y.second);
}
```

```
void ASTAR (int r,
            int c,
            pair<int, int> start,
            pair<int, int> end,
            vector<char> *grid,
            list<pair<int, int>> & path,
            map<pair<int, int>, pair<int, int>> & pi) {



   map<pair<int, int>, int> dist;

   for (int i = 0 ; i < r ; ++i)
     for (int j = 0 ; j < c ; ++j) {
       pair<int ,int> node = make_pair(i, j);
       dist[node] = INF;
     }
   dist[start] = 0;

   map<pair<int, int>, bool> visited;

   priority_queue<pair<int, pair<pair<int, int>, pair<int, int>>>,
     vector<pair<int, pair<pair<int, int>, pair<int, int>>>>,
     greater<pair<int, pair<pair<int, int>, pair<int, int>>>>> q;

   int g = 1;
   int h = abs(end.first-start.first) + abs(end.second-start.second);
   int f = g + h;
   q.push({f, {{g, h}, start}});

   pair<int, int> neighbors[4] = {{-1, 0}, {0, -1},{0, 1},{1, 0}};

   while (!q.empty()) {
     pair<int, pair<pair<int, int>, pair<int, int>>> p = q.top();
     q.pop();

     pair<int, int> current = p.second.second;
     int f_current = p.first;
     int g_current = p.second.first.first;

     if(visited[current])  continue;

     visited.insert(make_pair(current, true));
```

```
    if (current == end){
      break;
    }

    for (auto cn: neighbors) {
      pair<int, int> neighbor = {current + cn};

      if (neighbor.first < 0 || neighbor.first >= r || neighbor.second < 0 && neighbor.second >= r)
        continue;

        if (grid[neighbor.first][neighbor.second] == '-' || grid[neighbor.first][neighbor.second] == '.') {
          int g_neighbor;
          if (grid[neighbor.first][neighbor.second] == '.')
            g_neighbor = 0;
          else
            g_neighbor = 1;

          h = abs(end.first-neighbor.first) + abs(end.second-neighbor.second);
          f = g_current + g_neighbor + h;

          if(dist[neighbor] > dist[current] + f) {
            dist[neighbor] = dist[current] + f;
            q.push({f, {{g_current + g_neighbor, h}, neighbor}});
            pi[neighbor] = current;
          }
        }
      }
    }

 //Buid the shortest path
 pair<int, int> node = end;
 while (true) {
   if (node == start) {
     path.push_front(start);
     break;
   }
   path.push_front(node);
   node = pi[node];
 }

}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int pr, pc;
  cin >> pr >> pc;

  pair<int, int> start = {pr, pc};

  int fr, fc;
  cin >> fr >> fc;

  int r, c;
  cin >> r >> c;

  pair<int, int> end = {fr, fc};

  vector<char> *pacManGrid = new vector<char>[c];
  for (int ri = 0 ; ri < r ; ++ri)
    for (int ci = 0 ; ci < c ; ++ci) {
      char node;
      cin >> node;
      pacManGrid[ri].push_back(node);
    }

    list<pair<int, int>> path;
    map<pair<int, int>, pair<int, int>> pi;

    ASTAR(r, c, start, end, pacManGrid, path, pi);

    cout << path.size() - 1 << '\n';
    for(auto node: path)
      cout << node.first << ' ' << node.second << '\n';

    return 0;

}
```



PacMan – UCS editorial

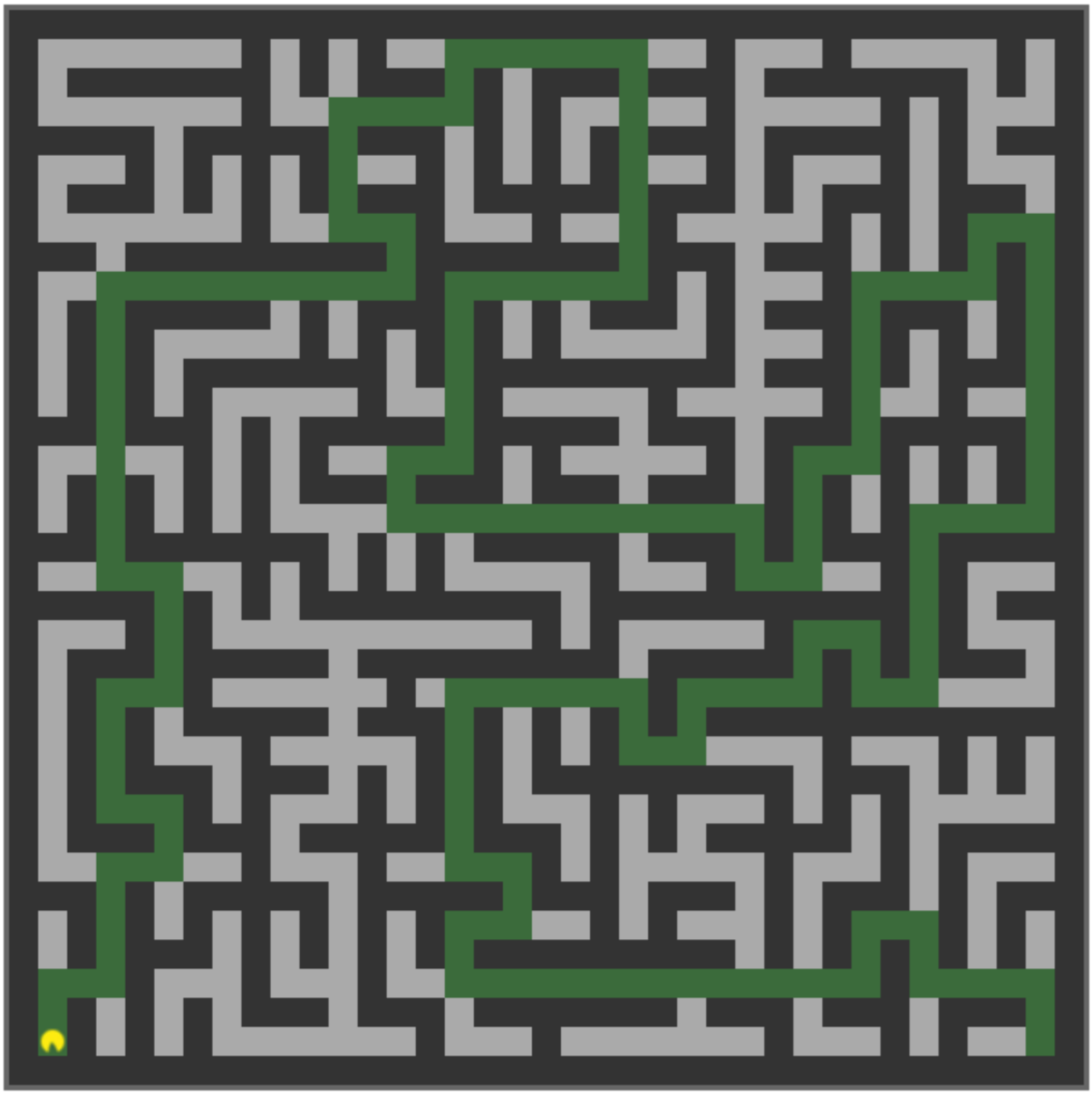
C++ Code

```
#include <bits/stdc++.h>

using namespace std;

pair<int,int> operator+(const pair<int, int>& x, const pair<int, int>& y) {
   return make_pair(x.first+y.first, x.second+y.second);
}

void ucs (int r,
          int c,
          pair<int, int> start,
          pair<int, int> end,
          vector<char> *grid,
          list<pair<int, int>> & path) {

  map<pair<int, int>, pair<int, int>> pi;
  map<pair<int, int>, bool> visited;

  priority_queue<pair<int, pair<int, int>>,
                 vector<pair<int, pair<int, int>>>,
                 greater<pair<int, pair<int, int>>>> q;

  q.push({0, start});

  pair<int, int> neighbors[4] = {{-1, 0}, {0, -1},{0, 1},{1, 0}};

  while (!q.empty()) {
    pair<int, pair<int, int>> p = q.top();
    q.pop();

    pair<int ,int> current = p.second;
    int d_current = p.first;

    visited.insert(make_pair(current, true));

    if (current == end)
      break;

    for (auto cn: neighbors) {
      pair<int, int> neighbor = {current + cn};
```

```
    if (neighbor.first < 0 || neighbor.first >= r || neighbor.second < 0 && neighbor.second >= r)
      continue;

    if (grid[neighbor.first][neighbor.second] == '-' || grid[neighbor.first][neighbor.second] == '.') {
      if (!visited[neighbor]) {
        visited[neighbor] = true;
        q.push({d_current + 1, neighbor});
        pi[neighbor] = current;
      }
    }
   }
  }

  //Buid the shortest path
  pair<int, int> node = end;
  while (true) {
   if (node == start) {
    path.push_front(start);
    break;
   }
   path.push_front(node);
   node = pi[node];
  }

}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int pr, pc;
  cin >> pr >> pc;

  pair<int, int> start = {pr, pc};

  int fr, fc;
  cin >> fr >> fc;

  int r, c;
  cin >> r >> c;

  pair<int, int> end = {fr, fc};

  vector<char> *pacManGrid = new vector<char>[c];
  for (int ri = 0 ; ri < r ; ++ri)
    for (int ci = 0 ; ci < c ; ++ci) {
      char node;
      cin >> node;
      pacManGrid[ri].push_back(node);
    }


    list<pair<int, int>> path;
    map<pair<int, int>, pair<int, int>> pi;

    ucs(r, c, start, end, pacManGrid, path);

    cout << path.size() - 1 << '\n';
    for (auto node: path)
      cout << node.first << ' ' << node.second << '\n';

    return 0;

}
```

Uniform coast search (UCS)



N Puzzle editorial

It’s not a graph problem. But, just to perform using the A* algorithm.

Before that, we must check if the puzzle is solvable or not. By computing the number of inversions of the input puzzle (the empty tile not count)

"What is an inversion here?
If we assume the tiles written out in a single row (1D Array) instead of being spread in N-rows (2D Array), a pair of tiles (a, b) form an inversion if a appears before b but a > b.
For above example, consider the tiles written out in a row, like this:
2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 X
The above grid forms only 1 inversion i.e. (2, 1)."

https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/

if #inversions is even, then the puzzle is solvable.

To solve that, we have to compute the coast of each state, in order to know the most nearest state to the goal state (final puzzle).

As you know the A* algorithm deals with heuristics and coasts.

The formula will be:

$$coast = f(x) + h(x)$$

$where$:

$f(x)$: $\#number\ of\ steps\ to\ get\ the\ state\ x\ from\ the\ start\ state.$

$h(x)$: $a\ heuristic\ function,\ computes\ the\ coast\ to\ get\ goal\ state\ from\ x.$

In my solution, $h(x) = \#misplaced\ tile\ in\ x\ state\ by\ comparing\ with\ the\ goal\ state.$

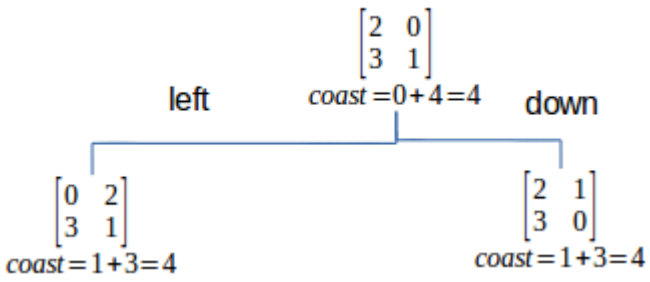
In every iteration, we pick the puzzle that have the minimum coast. If the puzzle is not seen, then we store it in a structure π, that indicates the predecessor or every puzzle with the made move.

At the end, we build the solution, using π.

let’s take an example with size 2 (4-puzzle)

$$\begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix}$$

from the start state, we can slide the empty tile (0) to the left or down:

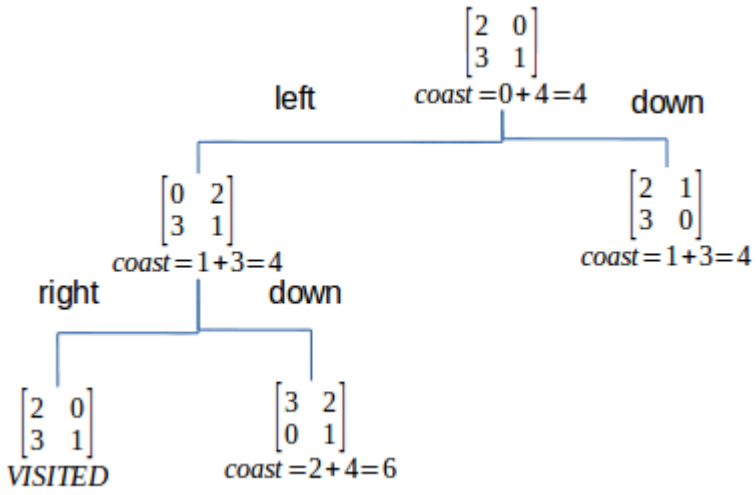


Don’t forget to update the visited and π maps:

visited	
$\begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix}$	true
$\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$	true
$\begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix}$	true

π	
$\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$	$\{\begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix}$,LEFT$\}$
$\begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix}$	$\{\begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix}$,DOWN$\}$

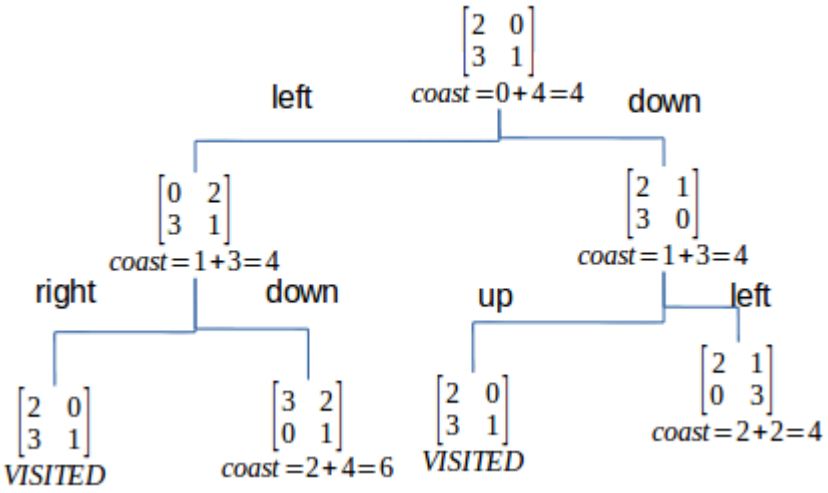
the puzzle to expand is that one who has the minimum coast, here both are equal, we can pick anyone:



visited	
$\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix}$	true
$\begin{bmatrix}0 & 2\\3 & 1\end{bmatrix}$	true
$\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix}$	true
$\begin{bmatrix}3 & 2\\0 & 1\end{bmatrix}$	true

π	
$\begin{bmatrix}0 & 2\\3 & 1\end{bmatrix}$	$\{\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix}$, LEFT$\}$
$\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix}$	$\{\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix}$, DOWN$\}$
$\begin{bmatrix}3 & 2\\0 & 1\end{bmatrix}$	$\{\begin{bmatrix}0 & 2\\3 & 1\end{bmatrix}$, DOWN$\}$

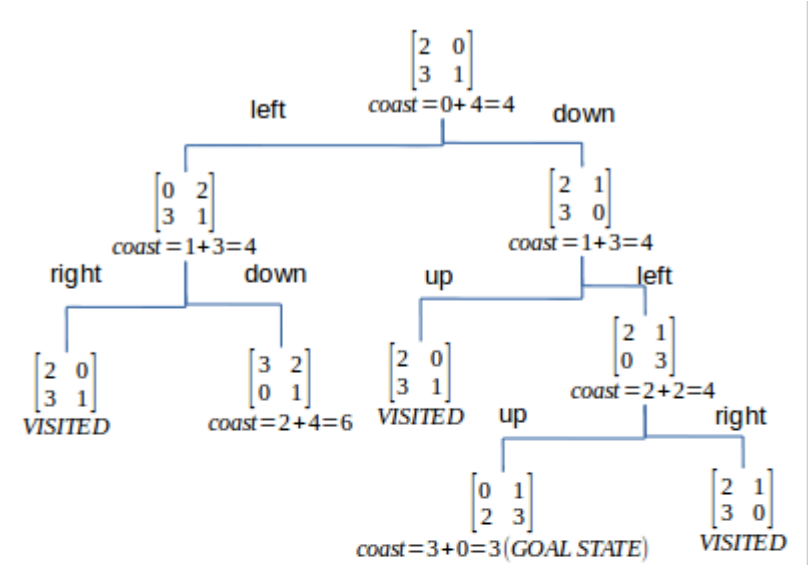
the puzzle to expand is that one who has the minimum coast:



visited	
$\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix}$	true
$\begin{bmatrix}0 & 2\\3 & 1\end{bmatrix}$	true
$\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix}$	true
$\begin{bmatrix}3 & 2\\0 & 1\end{bmatrix}$	true
$\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}$	true

π	
$\begin{bmatrix}0 & 2\\3 & 1\end{bmatrix}$	$\{\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix}, \text{LEFT}\}$
$\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix}$	$\{\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix}, \text{DOWN}\}$
$\begin{bmatrix}3 & 2\\0 & 1\end{bmatrix}$	$\{\begin{bmatrix}0 & 2\\3 & 1\end{bmatrix}, \text{DOWN}\}$
$\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}$	$\{\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix}, \text{LEFT}\}$

the puzzle to expand is that one who has the minimum coast:



visited	
$\begin{bmatrix}2&0\\3&1\end{bmatrix}$	true
$\begin{bmatrix}0&2\\3&1\end{bmatrix}$	true
$\begin{bmatrix}2&1\\3&0\end{bmatrix}$	true
$\begin{bmatrix}3&2\\0&1\end{bmatrix}$	true
$\begin{bmatrix}2&1\\0&3\end{bmatrix}$	true

π	
$\begin{bmatrix}0&2\\3&1\end{bmatrix}$	$\{\begin{bmatrix}2&0\\3&1\end{bmatrix}, LEFT\}$
$\begin{bmatrix}2&1\\3&0\end{bmatrix}$	$\{\begin{bmatrix}2&0\\3&1\end{bmatrix}, DOWN\}$
$\begin{bmatrix}3&2\\0&1\end{bmatrix}$	$\{\begin{bmatrix}0&2\\3&1\end{bmatrix}, DOWN\}$
$\begin{bmatrix}2&1\\0&3\end{bmatrix}$	$\{\begin{bmatrix}2&1\\3&0\end{bmatrix}, LEFT\}$
$\begin{bmatrix}0&1\\2&3\end{bmatrix}$	$\{\begin{bmatrix}2&1\\0&3\end{bmatrix}, UP\}$

we can rebuild the moves from the goal state to the start state using the π map:

$$\pi[\begin{bmatrix}0&1\\2&3\end{bmatrix}]=\{\begin{bmatrix}2&1\\0&3\end{bmatrix}, UP\}$$

moves: UP

0

$$\pi[\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}]=\{\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix},\text{LEFT}\}$$

moves:

UP	LEFT
0	*1*

$$\pi[\begin{bmatrix}2 & 1\\3 & 0\end{bmatrix}]=\{\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix},\text{DOWN}\}$$

moves:

UP	LEFT	DOWN
0	*1*	*2*

The moves are: DOWN, LEFT and UP.

$$\begin{bmatrix}2 & 0\\3 & 1\end{bmatrix} \xrightarrow{\text{DOWN}} \begin{bmatrix}2 & 1\\3 & 0\end{bmatrix} \xrightarrow{\text{LEFT}} \begin{bmatrix}2 & 1\\0 & 3\end{bmatrix} \xrightarrow{\text{UP}} \begin{bmatrix}0 & 1\\2 & 3\end{bmatrix}$$

C++ Code

```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<int>> vii;
typedef pair<int, int> pii;

class puzzle {
  public:
    vii grid;
    int x_empty_tile;
    int y_empty_tile;
    int step_from_start;
    int misplaced_tiles;
    int coast;
    string move;

  public:
    puzzle(vii grid,
           int x_empty_tile,
           int y_empty_tile,
           int step_from_start,
           int misplaced_tiles,
           int coast,
           string move) {

      this->grid = grid;
      this->x_empty_tile = x_empty_tile;
      this->y_empty_tile = y_empty_tile;
      this->step_from_start = step_from_start;
      this->misplaced_tiles = misplaced_tiles;
      this->coast = coast;
      this->move = move;
  }
};

typedef pair<int, puzzle*> pqt;
```

```
int heuristic(int k, vii p1, vii p2){
  int h = 0;
  int tile = 0;
  for (int i = 0 ; i < k ; ++i){
      for (int j = 0 ; j < k ; ++j){
        if (p1[i][j] != tile) h++;
        tile++;
      }
  }
  return h;
}
```

```
//Compute the number of all moved puzzle using A*
void a_star(int k, puzzle *start, puzzle *goal, map<vii, pair<vii, string>> & pi){

  priority_queue<pqt, vector<pqt>, greater<pqt>> q;
  q.push({start->coast, start});

  map<vii, bool> visited;
  while (!q.empty()){
    puzzle *current_puzzle = q.top().second;
    q.pop();

    vii current_grid = current_puzzle->grid;

    if (visited[current_grid]) continue;
    visited[current_grid] = true;

    if (current_puzzle->misplaced_tiles == 0) break;

    //Do the four moves (when is possible)
    map<pii, string> moves;
    moves.insert({{-1, 0}, "UP"});
    moves.insert({{0, -1}, "LEFT"});
    moves.insert({{0, 1}, "RIGHT"});
    moves.insert({{1, 0}, "DOWN"});

    for (auto move: moves){
      int x = current_puzzle->x_empty_tile + move.first.first;
      int y = current_puzzle->y_empty_tile + move.first.second;

      if ( x < 0 || x >= k || y < 0 || y >= k) continue;

      vii next_grid = current_grid;
      int tmp = next_grid[x][y];
      next_grid[x][y] = 0;
      next_grid[current_puzzle->x_empty_tile][current_puzzle->y_empty_tile] = tmp;

      if (!visited[next_grid]){
        int f = current_puzzle->step_from_start + 1;
        int h = heuristic(k, next_grid, goal->grid);
        int coast = f + h;

        puzzle *next_puzzle = new puzzle(next_grid, x, y, f, h, coast, move.second);

        pi[next_grid] = {current_grid, move.second};
```

```
        q.push({coast, next_puzzle});
      }

    }
  }
}

void display_grid(int k, vii grid){
  for (int i = 0 ; i < k ; ++i){
    for (auto tile: grid[i])
      cout << tile << ' ';
    cout << '\n';
  }
}

void rebuild_path(int k, map<vii, pair<vii, string>> pi, vii start_grid, vii goal_grid, vector<string>
& path){
  if (!pi.empty()) {
    vii p = pi[goal_grid].first;
    path.push_back(pi[goal_grid].second);
    path.push_back(pi[p].second);
    while (p != start_grid){
      p = pi[p].first;
      if (!pi[p].second.empty())
        path.push_back(pi[p].second);
    }
  }
}

bool is_solvable(int k, vector<int> arr){
  int less = 0;
  for (int i = 0 ; i < k * k ; ++i){
    for(int j = i + 1 ; j < k * k ; ++j ){
      if (arr[j] != 0 && arr[j] < arr[i]) less++;
    }
  }

  return less % 2 == 0;
}
```

```
int main() {
  ios::basic_ios::sync_with_stdio(false);
  int k;
  cin >> k;

  vii goal_p (k, vector<int> (k));
  int tile = 0;
  for (int i = 0 ; i < k ; ++i){
    for (int j = 0 ; j < k ; ++j) {
      goal_p[i][j] = tile++;
    }
  }
  puzzle *goal_puzzle = new puzzle(goal_p, 0, 0, 0, 0, 0, "NONE");

  vii start_p (k, vector<int> (k));
  vector<int> arr;
  int x_empty_tile = 0, y_empty_tile = 0;
  for (int i = 0 ; i < k ; ++i){
    for (int j = 0 ; j < k ; ++j) {
      int tile;
      cin >> tile;
      if (tile == 0){
        x_empty_tile = i;
        y_empty_tile = j;
      }
      else
        arr.push_back(tile);

      start_p [i][j] = tile;
    }
  }

  if (!is_solvable(k, arr)) {
    cout << "0\n";
    return 0;
  }

  int h0 = heuristic(k, start_p, goal_p);
  puzzle *start_puzzle = new puzzle(start_p, x_empty_tile, y_empty_tile, 0, h0, 0 + h0, "NONE");

  map<vii, pair<vii, string>> pi;
  a_star(k, start_puzzle, goal_puzzle, pi);

  vector<string> path;
```

```
  rebuild_path(k, pi, start_puzzle->grid, goal_puzzle->grid, path);

  cout << path.size() << '\n';
  if (!path.empty())
    for (auto it = path.crbegin() ; it != path.crend() ; ++it)
      cout << *it << '\n';

  return 0;
}
```

```
mayar:5-N-PUZZLE$ ./N-puzzle
2
2
0
3
1
3
DOWN
LEFT
UP
```