Basics of C++ language, assignment 2007 – 2008

Shortest king's path on chess board

Definition of assignment

The king in chess can move to each neighbouring square of its location square, also moving cornerwise is allowed. The task is to write a program to search a shortest possible path on a n*k-chess board from upper left corner (from the square (0,0)) to lower right corner (square (n-1,k-1)) of the board. Some squares are marked as occupied and these squares are not accessible for the king. For example, below is described a 6*5-board, occupied squares marked with P,

	0	**1**	**2**	**3**	**4**	**5**
0		**P**	**P**		**P**	**P**
1		**P**		**P**		**P**
2				**P**		**P**
3		**P**	**P**	**P**		**P**
4			**P**	**P**		

with a path (0,0), (0,1), (1,2), (2,1), (3,0), (4,1), (4,2),(4,3) and (5,4). This is also shortest path possible.

When the setup of board is read from a text file, a graph is formed. In the graph the nodes represent the squares of the board and a node represents that the king can move from one square to another. When the graph has been constructed, **the shortest path is found using breadth-first search**. If there are several possibilities for the shortest path, it is enough to find one. Also the situation where no path exists has to be taken into account. **The breadth-first search algorithm is described in appendix**.

Input for the program

The program reads the board from a text file; one line of file represents one row of the board such that E marks an empty square and P marks an occupied square.

The example board above is represented as follows:

```
EPPEPP
EPEPEP
EEEPEP
EPPPEP
EEPPEE
```

The program gets three command line arguments. The first argument is an option, either -show or -quiet. The first option (-show) means that program prints also in the console window and the other option (-quiet) means that program prints only in the output file. The second command line argument is the input file's name and the third command line argument is the output file's name. **All three command line arguments must be given**, otherwise program prints usage instructions. The program must be able to handle boards of different sizes, also erroneous inputs (wrong first option, input file does not exist or the file is not of proper format) must be handled.

Program's output

At first, the program prints in the console window the board such that first row and column show the coordinates. The occupied squares are represented with letter P; other squares can be left blank. Our example board would be printed as shown below:

```
        0       1       2       3       4       5
0               P       P               P       P
1               P               P               P
2                               P               P
3               P       P       P               P
4                       P       P
```

After this, program prints in the console window the path, or informs that there is no path. The path is printed by marking the steps in numerical order. For example the mentioned path on our example board is printed as follows:

```
        0       1       2       3       4       5
0       1       P       P       5       P       P
1       2       P       4       P       6       P
2               3               P       7       P
3               P       P       P       8       P
4                       P       P               9
```

If some command line arguments are missing, or if they are erroneous, the program prints usage information. If input file is erroneous or missing, information concerning this is printed. **The program prints in the console window, only if the first command line argument is -show; otherwise program prints in the output file only.**

The program prints also in a text file. The ouput is following:

1. If there is a path, the program prints path's coordinates; separated by a space such that one pair of coordinates is situated on one line. The row coordinate is the first in the pair.
2. If there is no path, the pair -1 -1 is printed.
3. If input file is erroneous or missing the pair -2 -2 is printed.
4. If a runtime error occurs, the pair -3 -3 is printed.

For example, in our example case, the program would print in the output file:

```
0 0
0 1
1 2
2 1
3 0
4 1
4 2
4 3
5 4
```

The name of the output file is the third command line argument. If file already exists, it can be overwritten without warning.

Concerning implementation

1. The board size is not known in advance; hence the data structure representing the graph must be dynamic. Statically allocated arrays are not allowed. Some standard C++ container can be, naturally, used.
2. **The program must contain a class structure. No procedural solutions are allowed.**

General requirements

The assignment is returned as a zipped file (that can be opened with pkzip) using Prolab's Puzzle system. The assignment must contain the following items:

1. readme.txt file with author information (name, department, faculty, student id and e-mail address) plus possibly some other important information for the inspector.
2. Source code with Doxygen-documentation. It is assumed that all code files are in one directory.

The inspectors will compile the code with Microsoft Visual Studio (.NET 2005). Hence the author should, prior to returning the assignment, check that the code compiles in Microsoft Studio, if some other development environment has been used. In general, this does not require much from programmer, usually one may have to make some type casts or add some header file. The inspector also runs the program in MS Visual Studio debugger; therefore author should moreover check that the program can be executed under MS Visual Studio debugger.

The inspectors execute the program with both adequate and erroneous inputs. The program should react sensibly to erroneous inputs: the program may not crash or get into eternal loop. Erroneous inputs must be informed to the user. On the course web page, there will be some input files to test the assignment program.

Moreover, program is tested for memory leaks. The programmer can check this in Visual Studio by adding in the beginning of the main program file (author can leave the following definitions in the final code)

```
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>
```

and by calling in the main function before return statement

```
_CrtDumpMemoryLeaks();
```

If there is a memory leak in the program, a message will be shown in the Studio's output window, when the program is executed in the debug mode.

Conclusion

The list below contains the most important items that should be taken into account:

1. The program is run by typing in the command prompt

```
<program_name> <-show | -quiet> <input_file_name> <output_file_name>
```

2. Program prints in the console window the board and the path, if the path is found.

3. Program prints in a text file the path coordinates, if path is found. Otherwise it prints in the text file an error code that was described in the section" Program's output".

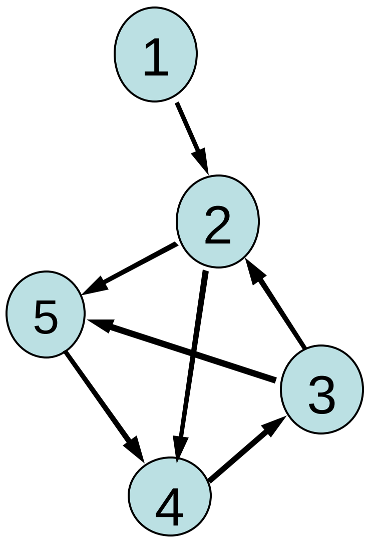
4. The assignment must contain readme.txt file and source code with Doxygen comments.

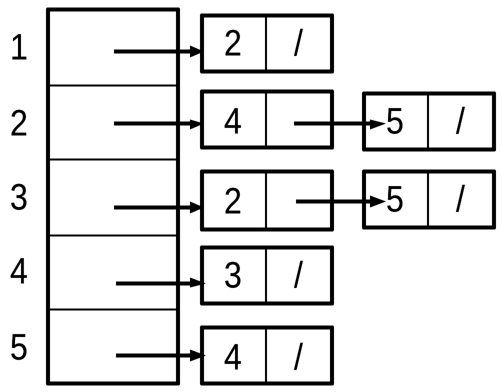
5. The code must compile in MS Visual Studio .NET 2005.

6. The program is checked for memory leaks.

Appendix: Breadth-first search algorithm

A graph is a pair G = (V,E), where V is a non-empty set of vertices and E is a set of edges, defining a relation in the set of vertices. Thus the set E defines which vertices are connected. Here we use adjacency-list representation of the graph. If there are N vertices in the graph, the adjacency list uses an array Adj, containing N linked lists. When u is a vertex, the list Adj[u] contains the neighbors of u in the graph (i.e. vertex v is in this list if and only if (u,v) is an edge in the graph). Below is an example of a directed graph and its adjacency-list representation.





Breadth-first search is one of the simplest methods to systematically investigate vertices in a graph. It can be used as a basis of many different kinds of graph algorithms. Let G = (V,E) be a graph. In the breadth-first search one selects one vertex and searches systematically edges until we encounter all the vertices that can be reached from starting vertex. In the process one also computes the distance from the starting vertex. The name of the algorithm is based on the fact that the algorithm expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. In other words, it finds first the vertices with distance 1 from the starting vertex, the the vertices with distance 2 and so on.

The algorithm uses a queue Q (FIFO). The elements are always inserted in the rear of the queue and taken from the head of the queue. An empty queue is denoted by EMPTY, the call PUSH(Q,u) inserts a vertex u in the queue and POP(Q) removes a vertex from the queue returning it. Symbol INF means an “infinite” number that is greater than any positive number. In the implementation it can be, for example, -1. The algorithm updates two arrays p and d; both contain one slot for each vertex of the graph. The array d will contain distances from the starting vertex and the array p will contain the vertice’s predecessor in a shortest path from the starting vertex. Thus after executing the algorithm, one can read from the array p one (of possibly many) shortest path to any vertex from the starting vertex. Besides these two arrays, an additional array color is used. In this array a vertex is marked white, grey or black. If a vertex is white, it has not yet been discovered; if a vertex is black it has been discovered as well as all its neighbors (which then are either black or grey). A grey vertex has been discovered but it may have undiscovered (white) neighbors. Hence grey vertices represent the interface between discovered and undiscovered vertices.

Below breadth-first search is shown as pseudo code using adjancence-list representation for the graph.

```
Input: Graph G=(V,E) and its vertex s. Graph is represented with
adjacency list.
Output: Let N be the number of vertices. Arrays p,d and color are
indexed with vertices. In the array d is computed the distances
from s. Value INF means that there is no path to the vertex. In
the array p is stored the predecessor of vertex in a shortest
path from s. Value NIL means that there is no path.

BFS(G,s)
for each u in V do
   color[u] = WHITE
   d[u] = INF
   p[u] = NIL
end for
d[s] = 0
color[s] = GRAY
Q = EMPTY
PUSH(Q,s)
while Q != EMPTY do
  u = POP(Q)
  for each v in Adj[u] do
      if color[v] == WHITE then
         color[v] = GRAY
         d[v] = d[u]+1
         p[v] = u
         PUSH(Q,v)
     end if
  end for
  color[u] = BLACK
end while
return
```

The shortest path from starting vertex can be read from the predecessor array p using following algorithm:

```
Input: Graph G=(V,E) and its vertices s and v. Algorithm
assumes that first BFS(G,s) is executed: it uses the array p
produced by BFS(G,s).
Output: The shortest path from s to v or notification that
there is no path.

BFS_PATH(G,s,v)
if v==s then
   print s
else
   if p[v] == NIL then
      print “No path from” s “to” v
   else
      BFS_PATH(G,s,p[v])
      print v
   end if
end if
return
```