Pathfinding in radioactive environment
using Java

Rickard Sandin
2013
Pathfinding in radioactive environment

by

Rickard Sandin

Faculty of Engineering and Sustainable Development
University of Gävle

S-801 76 Gävle, Sweden

Email: Rickard_sandin@hotmail.com

Abstract

Today many people work in radioactive environments, like in nuclear power plants and areas for storing spent nuclear fuel and radioactive waste. Humans cannot perceive radioactivity, so to increase our awareness and avoid danger we need tools and devices for these purposes. IFE (Institute for Energy Technology), the company that ordered this project, have a section that develops software to visualize, plan and simulate activities in such conditions. This paper describes the implementation of a tool to find the way from Node A to Node B in a graph that exposes the worker to a minimal radiation dose, a task that can be difficult to do by hand. The programming language used for this project was JAVA. The program uses a radiation calculator written by IFE, to get radiation levels in the area. The algorithm for the search is a Uniform Cost Search that guaranties the safest way in the graph. The GUI can draw the solution, and eases debug, testing and validating the algorithm. The code works and IFE approved the project and will evaluate if they are going to use it in their products.

Keywords: Search-algorithms radiation pathfinding
## Contents

1 Introduction ......................................................................................................................... 1  
1.1 Problem ............................................................................................................................ 1  
1.2 Scope .................................................................................................................................. 1  

2 Theory .................................................................................................................................. 2  
2.1 Complexity ......................................................................................................................... 3  
2.2 Search theory .................................................................................................................... 3  
  2.2.1 Breadth-First ................................................................................................................ 4  
  2.2.2 Depth-First .................................................................................................................. 4  
  2.2.3 Best-First .................................................................................................................... 4  
  2.2.4 A* search .................................................................................................................... 5  

3 Methodology ............................................................................................................................ 5  

4 Implementation ...................................................................................................................... 5  
4.1 Radiation Calculator ............................................................................................................ 6  
4.2 Dose-Map .......................................................................................................................... 6  
4.3 Edge Cost ........................................................................................................................... 6  
4.4 Data Structures .................................................................................................................. 7  
4.5 The Search ......................................................................................................................... 8  
4.6 Search results ..................................................................................................................... 8  

5 Results ................................................................................................................................... 8  
5.1 GUI ..................................................................................................................................... 9  
5.2 Benchmarking .................................................................................................................... 12  

6 Discussion ................................................................................................................................ 13  

7 Conclusion ............................................................................................................................... 13  
7.1 Possible improvements ....................................................................................................... 14  

References ................................................................................................................................. 15  

Appendix 1 – Search code .......................................................................................................... 16
1 Introduction

I chose to do my exam work on IFE (Institute for Energy Technology) in Halden, Norway. IFE is an international research foundation for energy and nuclear technology. One of IFE’s main tasks is to develop profitable, safe and climate friendly energy systems.

In December 2010 decided the Norwegian Ministry of Foreign Affairs that IFE together with the Federal Medical Biological Center in Russia where to start a new project. The project aims to enhance supervision and safety planning in Andreeva Bay area, a temporary storage area for spent nuclear fuel and radioactive waste. The project will develop tools to optimize the waste management strategy and to improve worker radiation awareness [1].

1.1 Problem

Humans cannot perceive radioactivity, so to increase our awareness and avoid danger we need tools and devices for this purposes. My task was to create a pathfinder, a tool to find a route from a start point to a goal in an area, with the lowest radiation exposure. These types of calculations are suitable for a program but hard for humans to do manually. The radiation data comes from one of IFE’s radiation calculators. The programming language for the calculator is JAVA. The calculator can return the dose rate from every point in the scene.

Short runtime and high resolution was a sub goal.

1.2 Scope

The work is limited to searching in a two-dimensional world using X and Y coordinates even thou the calculator supports more dimensions. This is enough to use with maps or simple environments but will not find the safest way through complex multi-story buildings. Figure 1 and 2 illustrates two Pathfinding problems for the program.
Accordingly to Christer Samuelsson [2], there is a principle called ALARA which is an acronym for “As low as reasonable achievable” and means that radiation-doses are to be kept as low as reasonable possible. This principle should always be applied when planning work that is exposed to ionizing radiation.

There are mainly three ways to minimize the absorbed radiation-dose.

1. Minimize time exposed.
2. Maximize distance to the source, radiation decreases fast with distance.
3. Use shielding. Put something between the source and the object. Lead, concrete or water stops most radiation efficiently.
2.1 Complexity

Mark Allen Weiss [3] writes that a common way to compare different algorithms and data structures is to use big O notation. It is a way to classify how resources like time and working space changes with input size. Some simplifications will usually be made when calculating big O; only the dominant term of the function matters, other terms and constants can be neglected. One thing that generally should be avoided if possible is the use of nested loops. Typical big O notations with ascending complexity is $O(1)$ constant, $O(\log n)$ logarithmic, $O(n)$ linear, $O(n \log n)$, $O(n^2)$ quadratic, $O(2^n)$ exponential, $O(n!)$ factorial. Figure 3 visualizes the growth rate of the different notations.

![Big O notations graph](image)

*Figure 3. Complexity graph[3, p. 224].*

2.2 Search theory

Search algorithms are an important area in computer science. A good example is Google whose success greatly depends on their search algorithms. There are algorithms for solving almost all types of problems. Here are some important words we need to know:

- Node/vertices – a point in space, have X and Y coordinates.
- Edge/link – a connection between two nodes. Can have a cost.
- Adjacent – two nodes are adjacent if they are connected by an edge.
- Graph – a set of nodes and a set of edges that connects the nodes.
- Tree – A graph without loops, only one way between two nodes.
- Uninformed search – brute force, has no guidance or external information.
- Informed search – uses outside information or a heuristic function to guide the search in the right direction.

Graphs are harder to traverse due to the possibility to get stuck in a loop. To avoid this, a search needs to remember where it has been. My problem is to find a path in a graph so I began investigate the common graph traversing algorithms from S. Russel and P. Norvigs book [4, chap. 3], which I here will explain briefly.

### 2.2.1 Breadth-First

A breadth first search visits the sibling nodes (nodes at the same level of depth) before visiting child nodes. This algorithm will always find the path with the fewest nodes between start and goal.

![Figure 4. Breadth-First Search example.](image)

### 2.2.2 Depth-First

Depth-first is the opposite of breadth-first. It goes as deep as possible before considering alternative paths. This algorithm cannot promise an optimal solution, in fact it cannot promise any solution even when one exists. The problem is that the algorithm can get stuck in a loop, but the way around that is to use iterative deepening. One advantage over the breadth-first algorithm is that it uses less memory.

![Figure 5. Depth-First Search example.](image)

### 2.2.3 Best-First

Best-first search is a general tree- or graph-search using a function $f(n)$ to evaluate which way to go or which node to expand next. Different functions result in different search strategies.
2.2.4 A* search

A* search, pronounced “A star search” is accordingly to S. Russel [4, p. 93] the most widely known form of best-first search. It uses a heuristic function to estimate the cost from node n to the goal. The evaluation function for an A* search is commonly written as you can see in Equation 1.

\[ f(n) = g(n) + h(n) \]  

(1)

Where \( g \) is total cost to get to node n and \( h \) is estimated cost to get to the goal from node n.

The heuristic function is there to point the search in the right direction. With a good heuristic function, the number of expanded nodes can be kept low. For an A* graph-search to be optimal and reliable, the heuristic function must be admissible and monotonic. Admissible means that it never underestimates the cost to the goal. Monotonic means that a node’s heuristic value never is less than its successor’s value plus the cost to get from the node to the successor.

3 Methodology

When writing a program it is usually a good idea to write and test small parts at a time. This process is called iterative development and has the advantage of getting executable code fast. Another positive thing with iterative development is that unexpected events and errors are easier to find and fix.

The programming steps known from the start was:

- Setup and use the radiation-calculator.
- Calculate edge-cost in the graph.
- Create data structures and algorithms for the search.
- Show the solution.

4 Implementation

The code was written in Eclipse in these steps:
4.1 Radiation Calculator

The first thing I needed to do was to familiarize myself with the radiation calculator. To use the calculator, the first step is to set up a scene and add one or more radioactive sources and zero or more shields. For sources, type, intensity and position must be specified. For shields material, size and position must be added. When a scene is created, the getRadiation function to get dose-rates at specified position can be used.

4.2 Dose-Map

Once the calculator worked, the next step was to visualize the radioactivity from the sources. This was first done creating a two dimensional array containing radioactivity levels in mSv/h, where each element in the array represented a square meter. After printing the array to the console, the data was pasted in to an excel-sheet and then using conditional formatting to obtain a map like the one in figure 7.

![Figure 7. Dose-Map in Excel.](image)

4.3 Edge Cost

The cost for an edge is the absorbed dose one would get traveling on that edge. One way to compute the absorbed dose is to calculate the average dose for the edge and multiply it with the time it takes to travel the edge. The average dose rate is the sum of the sampled dose rates divided by the number of samples. For the average to be accurate, samples must be taken at a rate so that big radiation peaks will not fit between two samples. The problem is similar to find a Riemann sum under a graph, which Håkan Lennnerstad explains in one of his books [5]. In the program, there is a variable to handle resolution in the calculations. With higher resolution, the results are more accurate but computation slower. Figure 8 shows calculations of a ten meters edge passing a radioactive source, with different resolution.
Exposure time equals distance traveled divided by traveling speed. In the calculations, a slow walking speed of 1 m/s is used. If you walk twice as fast, you get half the dose.

Measurements performed with different resolution indicate that sampling data at least ten times per second gives accuracy greater than 90%. Figure 9 shows the result data for four different graphs with different resolution.

**4.4 Data Structures**

In this program, three custom data structures are used. They are:

- Nodes – points in a graph with X and Y coordinates.
- Edges - a line between two nodes, have a length, a cost and a default speed of 1 m/s.
- Search Nodes – Keep track of current node, previous node (the parent) and the cost so far to get to the current node.
4.5 The Search

The original plan was to implement an A* search algorithm but I never managed to find a heuristic function to my problem. The reason is that heuristic functions often use a straight-line distance to the goal as guidance, for this problem, distance is not important. What matters is accumulated dose, and sometimes that means moving in the opposite direction from the goal, walk around and then approach from a safe angle.

The algorithm finally used was a Uniform-Cost search, which is a variant of a best-first search and works in a similar way as the more known Dijkstra's algorithm. The main difference between them is that Dijkstra’s algorithm works without a goal; it calculates the cost to get to every node in the graph while the uniform-cost search stops when it found the way to the goal.

The uniform-cost search algorithm uses two lists to keep track of its progress. One standard ArrayList to store all visited search nodes to prevent revisiting them, and a PriorityQueue where the next possible moves are stored, sorted with the cheapest move first.

Here follows the pseudo code for a uniform-cost search from Wikipedia[6]:

```plaintext
procedure UniformCostSearch(Graph, root, goal)
  node := root, cost = 0
  frontier := priority queue containing node only
  explored := empty set
  do
    if frontier is empty
      return failure
    node := frontier.pop()
    if node is goal
      return solution
    explored.add(node)
    for each of node's neighbors n
      if n is not in explored
        if n is not in frontier
          frontier.add(n)
        else if n is in frontier with higher cost
          replace existing node with n
```

4.6 Search results

The search returns a list with all the (Search) Nodes to get from start to goal. At first, the solution-list was printed in the console, but it was a lot of manual work to verify the coordinates so I decided to implement a Graphical User Interface, using Java swing components, to visualize the results.

5 Results

This is what I achieved:
5.1 GUI

The first version of the GUI is shown in figure 10 blow. In this scene, there are two radioactive sources and one shield. The blue dots are nodes, the blue lines are edges and the black lines represent the solution. The information on the labels are: node number, x coordinate, y coordinate.

![GUI version 1](image)

*Figure 10. GUI version 1.*

The next version was implemented to view bigger and more complex areas using a grid as graph. The tiles represent one square meter and there is a node on each tile. From a node, there are eight edges in the directions; up, down, left, right and diagonally.

In this version the search pattern and solution is shown with an animation when a button is clicked. The three following figures are from an animation. Initially, the start and goal are marked on the map.

![Animation of GUI version 2](image)
Second, when the button is clicked, thin branches expand out from the start node searching for the goal while trying to avoid yellow and red zones.

Once the goal is found, the backtracking with a thicker line from the goal starts. The backtracking ends when the start is reached.

In the third version, there is an image as background in the bottom layer. The image is a screenshot from one of IFE’s test models and is shown in figure 14. For scale, see the man in the left top corner. On top of this image, a transparent dose-map is added. Radioactive sources are simulated in some of the barrels and other objects. The two walls in the middle are created with shielding abilities. Some of the edges in the graph are removed to prevent the path to go through objects.
Figure 14. Test scene from CREATE.

Figure 15 shows the transparent dose-map over the background image. There are six radioactive sources in this scene.

Figure 15. Scene image with transparent dose-map.
In this scenario a worker starts in the top left corner and are supposed to get his tools on the yellow barrel in the middle of the map. The path with the lowest dose is shown in the figure 16. The received dose for this path is almost 0.005 mSv.

![Figure 16. The safest way to the yellow barrel in the middle for the manikin.](image)

At the bottom of the page there is a graph showing the total received dose for each node along the path. Figure 17 shows the graph for this scenario. As the figure shows, approximately half the dose is from the last five meters next to the goal barrel.

![Figure 17. Received dose for walking the path from previous figure.](image)

5.2 Benchmarking

A sub goal for this program was fast and accurate results, and the way to accomplish that is to use efficient data structures and smart algorithms. The first prototypes was not very efficient, instead all focus where on functionality. Later in the progress, when bigger data input where used, like a thousand nodes and thousands of edges, it became clear that the algorithm was sluggish. The running time seemed to grow quadratic to the input size. To find the bottleneck in the algorithm timers were used together with XProf for profiling. In this way the most time consuming functions could be found and optimized. In one place the use of a Linked List was found. A Linked List is usually a very inefficient data structure to use. So exchanging the Linked List to an ArrayList reduced the complexity to something like n log n.

Figure 18 shows running times for graphs of different sizes, the scene have 30 sources and use 10 samples per second as resolution.
Figure 18. Running time.

The running time of the search algorithm depends mainly on the graph size and complexity, number of sources in the scene and the resolution in the calculations. Scene complexity and resolution affects running time like a constant. Figure 19 shows two graphs with scene complexity respectively sample resolution related to running time.

Figure 19. Running time with different indata.

6 Discussion

This has been an interesting and challenging project. Thanks to the office place at the company I was able to focus better on the task than if I would have done it from home. And thanks to the “on site” programming, I was able to get fast answers to questions that came up along the way. My schedule worked even though time past very fast the last weeks.

The work with the GUI took more time than originally planned. Both the implementation and when it was finished. It turned out that the randomly created search patterns was rather interesting to watch and caught both my and others interest. The time was well spent but as a note to self: If part of the code are going to get tweaked and modified a lot, try to make (variable-)changes easy to apply.

7 Conclusion

The algorithm works fine and handles all of the defined problems from chapter 1.1. The company approved my work and liked the presentation I held for them.
7.1 Possible improvements

- Easier handling of input and output, perhaps using Google earth or something similar.
- Automatic adjustment of resolution in edge cost calculations.
- Search in 3 dimensions.
References


Appendix 1 – Search code

```java
package pathcalculator;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;
/**Best-First-Search class<br>* Contains getSolution method<br>* Author: Rickard Sandin<br>*/
public class BFS {
    private List<SearchNode> closedList;
    private PriorityQueue<SearchNode> openList;
    private int searchProgress = -1; // in percent
    private int numberOfNodes; // nodes in graph
    /**Finds the safest way from start to goal in a graph using best-first-search.<br>* @param Start node<br>* @param Goal node<br>* @param Edge List<br>* @return Solution (SearchNode list)<br>*/
    public List<SearchNode> getSolution(Node startNode, Node goalNode, List<Edge> edgeList) {
        Comparator<SearchNode> comparator = new SearchNodeComp(); //custom comparator, compares G-value for search nodes
        closedList = new ArrayList<SearchNode>(); //visited/expanded nodes
        openList = new PriorityQueue<SearchNode>(100, comparator); //the frontier. Possible next node
        numberOfNodes = countNodes(edgeList); //number of nodes = countNodes(edgeList);
        openList.add(new SearchNode(startNode, null, 0)); //add starting node to open list, no parent and no dose
        while (!openList.isEmpty()) {
            SearchNode current = openList.remove(); //get first node
            closedList.add(current); //add current node to closed list
            printProgress(); //prints in console search progress in percent
            if (current.getNode().getxPosition() == goalNode.getxPosition() && current.getNode().getyPosition() == goalNode.getyPosition()) {
                return extractSolution(closedList); //extract solution path from closed list.
            }
            List<SearchNode> adjacentNodes = findCurrentNodesAdjacentNodes(current, edgeList);
            addNewAdjacentNodesToOpenList(adjacentNodes);
        }
        return null; //no solution found
    }
    private void addNewAdjacentNodesToOpenList(List<SearchNode> adjacentNodes) {
        for (int i = 0; i < adjacentNodes.size(); i++) {
            SearchNode currentAdj = adjacentNodes.get(i);
            boolean newNode = true; //in closed?
            for (int j = 0; j < closedList.size(); j++) {
                if (closedList.get(j).getNode().equals(currentAdj.getNode())) {
                    newNode = false;
                }
            } //in open?
            List<SearchNode> tempList = new ArrayList<SearchNode>(openList);
            for (int j = 0; j < tempList.size(); j++) {
                if (tempList.get(j).getNode().equals(currentAdj.getNode())) {
                    if (currentAdj.getG() >= tempList.get(j).getG()) {
                        newNode = false;
                    }
                    if (currentAdj.getG() > tempList.get(j).getG()) {
                        openList.remove(tempList.get(j));
                    }
                }
            }
            if (newNode) {
                openList.add(currentAdj);
            }
        }
    }
    private List<SearchNode> findCurrentNodesAdjacentNodes(SearchNode current, List<Edge> edgeList) {
        List<SearchNode> adjacentNodes = new ArrayList<SearchNode>();
        for (int i = 0; i < edgeList.size(); i++) {
            if (current.getNode() == edgeList.get(i).getNodeA()) {
                float G = (float) (current.getG() + edgeList.get(i).getDose());
                adjacentNodes.add(new SearchNode(edgeList.get(i).getNodeB(), current.getNode(), G));
            }
            if (current.getNode() == edgeList.get(i).getNodeB()) {
                float G = (float) (current.getG() + edgeList.get(i).getDose());
                adjacentNodes.add(new SearchNode(edgeList.get(i).getNodeA(), current.getNode(), G));
            }
        }
        return adjacentNodes;
    }
    private void printProgress() {
        int tempProgress = (int) 100 * closedList.size() / numberOfNodes;
        if (searchProgress != tempProgress) {
            searchProgress = tempProgress;
            System.out.println(searchProgress + " %");
        }
    }
    // some other methods...
}
```

private int countNodes(List<Edge> edgeList) {
    Set<Node> nodeSet = new HashSet<Node>();
    for (int i = 0; i < edgeList.size(); i++) {
        nodeSet.add(edgeList.get(i).getNodeA());
        nodeSet.add(edgeList.get(i).getNodeB());
    }
    return nodeSet.size();
}

/** Extracts solution from closedList wish contains all expanded nodes. *
 * @param closedList *
 * @return List with solution from start to goal *
 */
private static List<SearchNode> extractSolution(List<SearchNode> closedList) {
    System.out.println("n Number of Nodes expanded: " + closedList.size() + ", \\
    SearchNode current = closedList.get(closedList.size()-1); // the goal
    List<SearchNode> path = new ArrayList<SearchNode>();
    path.add(current);
    boolean done = false;
    while(!done){
        if(current.getParent() == null) // start node has no parent
            done = true;
        for(int i = 0; i < closedList.size(); i++) {
            if (closedList.get(i).getNode().equals(current.getParent())){
                current = closedList.get(i); // add first in list
                path.add(0, current);
                break;
            }
        }
    }
    return path;
}

// used to print searched node
public List<SearchNode> getClosedList(){
    return closedList;
}