Coursework

Artificial Intelligence

Introduction

Davide Morello

40219838

Artificial Intelligence

Introduction

*This report tries to find the solution to the problem of finding the shortest path in a small network of interconnected caverns. The goal is to identify a programmatical solution to efficiently achieve a method to navigate the network in through the shortest path and in the shortest amount of time.*

# Identifying the problem

As it often happens in informatics, a problem that may appear hard at first can be easily broken down or reduced to a well-known issue. Our specific situation revolves around a small robot trying to navigate a set of caverns, and attempting to reach cavern N, which we will call “target” from its starting point, “source”.

What we know about this scenario is:

* The set of caverns are generally small, and overall the robot will face at worst a network of 20 caverns.
* Not all paths between caves are bidirectional, some may be one-way only.
* The distance between two caves, X and Y is the Euclidean distance of the coordinates of the two caves: √(𝑥2−𝑥1)2+(𝑦2−𝑦1)2

In this situation, it is fairly apparent from the start that what we are facing is small graph of interconnected nodes. Each cavern can easily be represented by a node, while paths between the various caverns, can be represented by weighted edges, where the weight of an edge spanning between two caves is the afore mentioned Euclidean formula.

Once the problem has been identified, obtaining a solution becomes quite clearer, and in this situation, can easily been reduced to finding an efficient goal-based algorithm, capable of finding the shortest path within a weighted Directed Graph, and in other words a graph where the direction of an edge is taken into consideration.

For this specific problem, we are also considering only a situation where we are not dealing with Multigraphs (graphs where two nodes might be linked by multiple edges of different weights), as the weight of each edge is identified by distance between the two nodes it links, making the use of multiple edges redundant.

Possible solutions

*Over the years in computing, many problem have been described by simply finding the sequence of actions that lead to the desired goal. Each action changes the state and tries to reach the final state, in which the goal has been achieved, starting from the initial state.*

*In graphs theory, similarly finding the shortest path within a graph is a searching problem, where each iteration attempts to achieve a state defined by the source and target nodes been linked together by a path-able set of edges. Due to the nature of the task, many different algorithms of varying complexity have been formulated, and for the goal of this report I will look into three different known solutions to the problem, that all offer a slightly different way of achieving the goal with slightly different outcomes.*

***For each one of the algorithms, I will try to give a brief overview (the length based on whether the algorithm has been explained in class), and then explain why it can be consider a suitable candidate to solve the problem.***

# Dijkstra's algorithm

## Overview

Dijkstra’s is possibly the most famous algorithm to find the shortest path with a graph. It was found by the Dutch Computer Scientist Edsger W. Dijkstra in 1959 and exists in many different variations.

At its core, Dijkstra’s runs by visiting nodes in the graph starting with source. It continues to repeatedly examine the closest and yet unvisited node and selecting its neighbors to be examined. Dijkstra’s always finds the shortest path from the source to the target, as long as all edges in the graph have a positive weight.

The algorithm is strictly greedy and can be considered a zero heuristics version of the more generalized A\*. It runs in O(N^2) where N is the amount of nodes in the graph. Dijkstra can be fairly slower than a Best-First Heuristics guided algorithm, however unlike some of its alternatives it is guaranteed to find the shortest path, rather than a path.

## A general look at the algorithm pseudocode

For a basic implementation of the algorithm, we can start by initializing three lists:

The three lists are used to properly navigate the graph while keeping an eye on the status of the search, and to iteratively understand whether the path found in the graph is better than the previous ones.

* dist[] with dist(i) ∈ ℝ+ contains the distance between a node and the source
* parent[] with parent(i)∈ V contains a node, the parent of U
* toVisit[] contains the list of nodes to visit

Once the lists are initialized, the algorithm proceeds to interactively navigate each one of the nodes which have not yet been visited, and for each one of them the neighbouring nodes are analysed completing a table of distances. If a node w connects successfully with the source, the distance is compared with the previously found ones (if any) and consequently saved.

The algorithm completes its run when all the

nodes N ∈ toVisit[] have been visited and the list is empty.

1 func Dijkstra(Graph, source):  
2  
3 toVisit = [] // The list of nodes to visit = []  
4  
5 forEach node n in Graph: // Initialization  
6 dist[n] ← ∞ // The starting distance is the to infinite  
7 parent[n] ← -1 // The previous node is instead to non-existent  
8 toVisit.push(n) // All nodes in the graph will need to be visited  
9  
10 dist[source] ← 0 // The distance from source to source is 0  
11   
12 while toVisit is not []:  
13 v ← node in toVisit with min dist[v] // Selects the node with the shortest d  
14 toVisit.pop(v)  
15   
16 for each neighbor w of v: // Checks the neighbors of v  
17 alt ← dist[v] + length(v, w)  
18 if alt < dist[v]: // Selects the new shortest path  
19 dist[w] ← alt   
20 parent[w] ← v   
21  
22 return dist[], parent[]

## Complexity

Dijkstra’s Algorithm can be expressed in function N and E respectively the number of Nodes and Edges in the graph.

A typical implementation of the algorithm such as the one shown in the codes, implement the search through a simple list containing the set of nodes to visit. For this reason, we can consider the complexity of the algorithm a simply O(N^2) as for each node K in the list of nodes to visit, in the worst-case scenario, the given node K will have N-1 neighbors.

The complexity can however be further reduced by the usage of a Fibonacci Heap. Moreover, in case of repeated searches, the overall speed of the algorithm can be further increased by the usage of memoing applied to the distance between nodes which can be pre-calculated or store.

## Correctness

Being a greedy algorithm, Dijkstra focuses on one simple idea: the local best solution, is always formulated by selecting the local best at each iteration. In our situation, the best path to a given node n in a graph, will be explored at each iteration by finding the shortest path from the source, at a given iteration, and attempting to move towards N from that specific point.

Consequently, Dijkstra’s correctness is proved by induction.

In order to do so, we generally consider these invariant lemmas:

* For each node n visited, dist[n] is the shorted distance from the source to the node n if a path is available, otherwise infinite
* For each node v not yet visited, dist[v] is the shortest distance between the source and the node v if a path is available, otherwise infinite

If only the source has been visited, the lemma will hold, as dist[source] is zero, and all nodes are set to an infinite distance.

Working on node N-1 where N is the number of nodes in the source, we can now consider and edge E where the distance dist[v] for the edge n-v is the smallest distance of eny unvisited node and nv is the distance dist[v] is equal to dist[n] + length[n, v]. In this scenario, dist[v] must be the shortest distance from the source as if there was a shortest one, it would have been visited fist. Therefore dist[w] < dist[v] would create a contradiction, just like an existing shorter path to v, different from dist[v] would have been less than dist[n] + length[n, u]. For this reason, even after v has been visited, for each node been visted, the distance distance[w] from w to the source is the shortest distance, using the nodes what have already been visited. Moreover if there is a shorter path using v, this will be updated.

## Conclusion

For our caverns exploration problem, Dijkstra is an optimal candidate.  
The greedy approach offers use a situation, where generally speaking the solution is reached quickly, while the overall complexity of the calculation, given the maximum number of nodes (caverns) being less or equal to twenty, ensures the that the iteration will be completely quickly regardless of input.

Dijkstra’s weaknesses are often related to its inability to obtain correct results when a path contains negative edges, which due to the nature of the iterations could potentially break the logic. This is often solves by the implementation of different algorithms, or variations such as the well known Bellam-Ford that can solve efficiently and correctly graphs with negative edges.

This is however not a problem in our scenario, considering in fact that the distance between two nodes A and B is as simple as the Euclidean distance between Ax, Ay and Bx, By, it is impossible to end up with negative edges and consequently all of our edges will have a positive weight.

# A\*

## Overview

*Since the algorithm has been throaty explained in class, I will only beefy touch on a few points and focus on why it can be considered one of the optimal candidates to solve the caverns issue.*

In 1968, Peter Hart, Nils Nilsson and Bertram Raphael at Stanford expanded Dijkstra’s algorithm, and creating what is now known as A\*.

A\* is what we can consider as an optimization, and generalization of Dijkstra’s Shortest-Path algorithm, that achieves better performance throughout the usage of heuristics to navigate the research. In other words, the selection of the node to expand will be operated by a function *f(n)* which will focus on evaluating which nodes is the best one to expand.

As an optimization or improvement over Dijkstra, A\* can be classified as informed or Best-Firsts search algorithm which uses heuristics to guide itself in finding the shortest path.

Dijkstra’s vs A\*  
As mentioned a couple of times, Dijkstra’s can be considered a special case for A\* where the heuristics used to determine the which node to expand is zero. In other words, and down to the behavior of the algorithm this determines a fairly different way of expanding nodes.  
While Dijkstra will given a node n ∈ of N, try choose to expand based on the distance between the node, its neighbour and the total distance from the source, A\* will instead use a more refined approach where an estimate of the distance from the node n to the target t is taken in consideration.

This means, that given a node n, the last node being visited, A\* will instead choose how to proceed based on the formula: ***f(n) = g(n) + h(n),*** where g(n) is the cost so far to reach node n, and more importantly h(n) is a heuristic used to estimate the cost of the shortest path admissible to reach the target node t from the current node n.

In order to work however, A\* requires the function that calculates the path to always underestimate the final cost (or distance).

## Complexity

In general, we can say that the time complexity for the A\*, just like it’s Space complexity can be very different based on the implementation.

The time complexity, can vary based on the heuristics used to evaluate the node to expand.

In an optimal scenario, such as search tree the complexity can be polynomial or linear, however it can grow to be exponential to the length of the solution, such as when the calculation revolves around the shortest path. Generally speaking, a good heuristic function will give A\* a great edge over Dijkstra’s and other more basic algorithms, reducing the branching factoring (number of nodes expanded) and reaching an optimal solution much quicker.

Space on the other hand, can represent a slightly worse obstacle, in order to operate quickly, and based on the implementation A\* similarly to Dijkstra’s, does keep information for all the expanded nodes in memory.

## Correctness and Optimality