UNIVERSITÀ DEGLI STUDI
DI MILANO

Pathfinding Basics

A.I. for Video Games

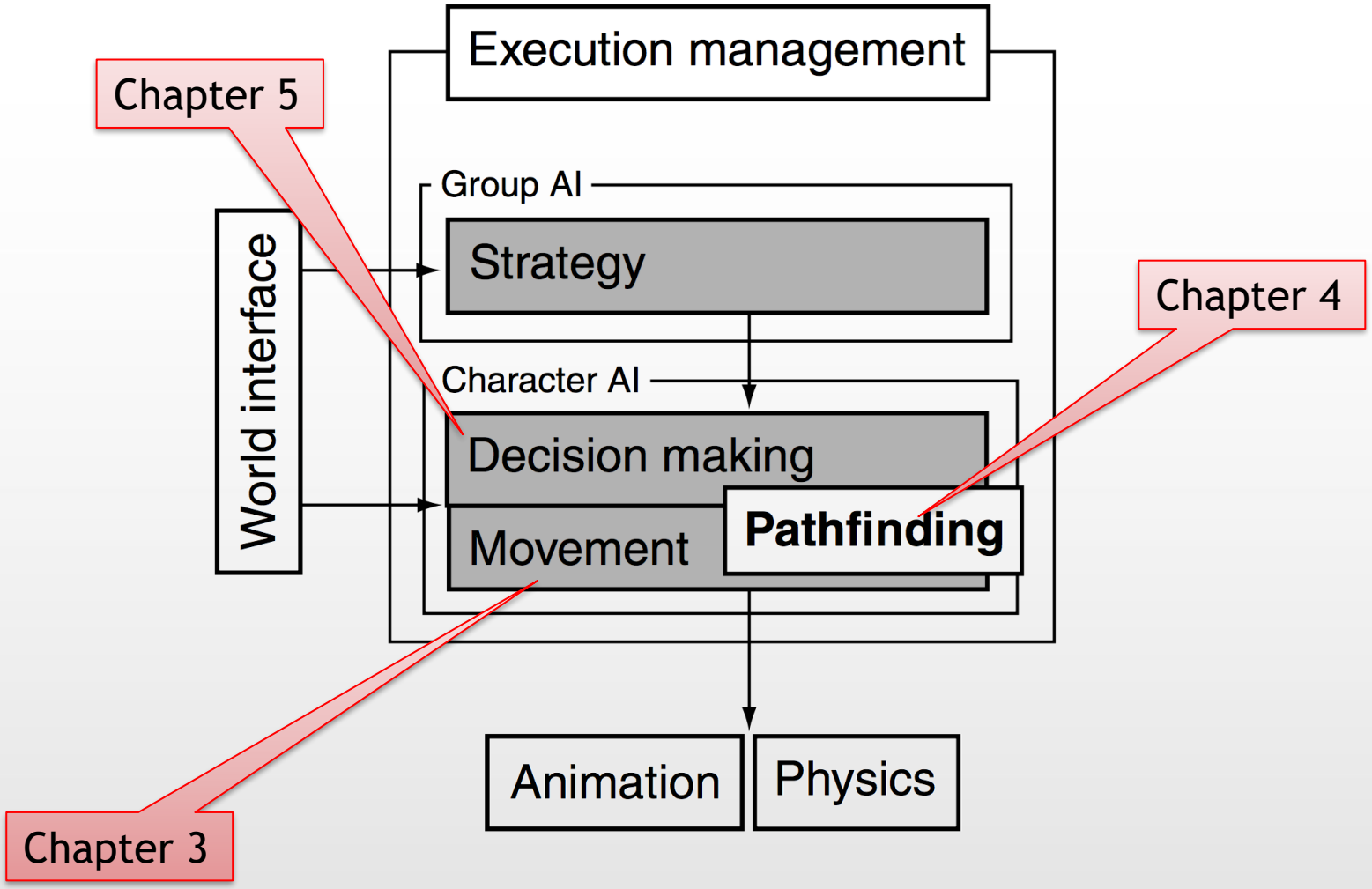
Dario Maggiorini (dario@di.unimi.it)

Foreword

- Pathfinding is all over videogames
  - It is about having your NPC strolling around … and not just that!
  - IT IS EXTREMELY IMPORTANT TO DO IT THE RIGHT WAY

- Nevertheless, pathfinding is a special case of planning
  - In video games it must be believable, it does not need to be perfect
  - Still the “Game AI” vs “Classical AI” thing, remember?

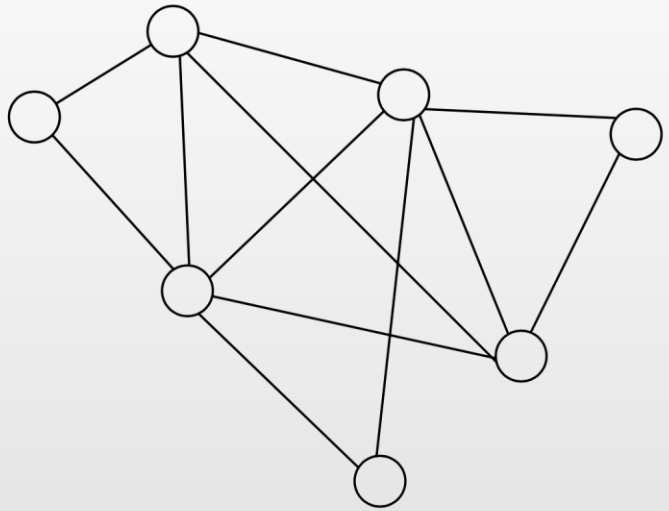
Mapping Keywords to the Book



Pathfinding in a Nutshell

1. You make a **representation** of your space
2. You run an algorithm to understand “the best” way to go from point A to point B

- Be (very) aware
  - Representation means GRAPH (this is **not** a game data structure)
  - You need to define “best”



Graphs

- A graph is a formal mathematical structure defined as follows:

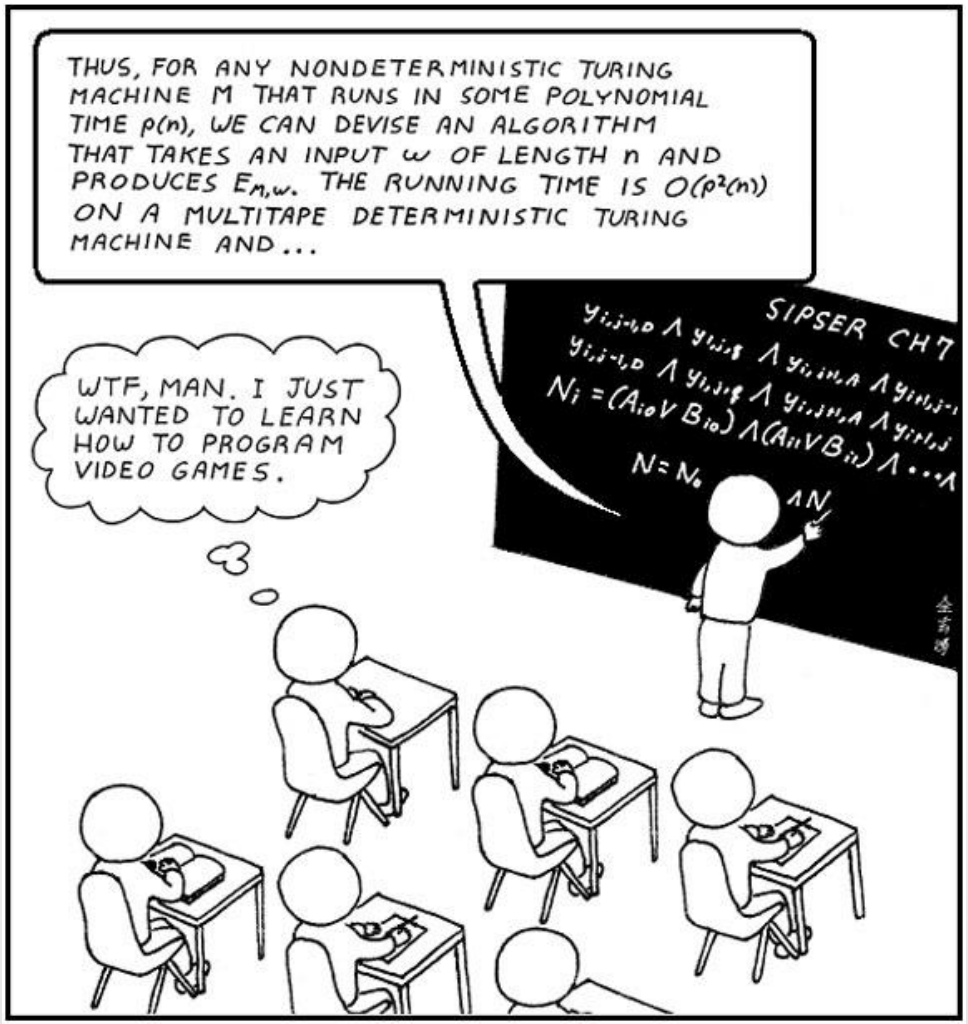
$$G = (V, E)$$

- Where
  - V is a set if vertices (or nodes)
  - E is a set if edges

- Every edge induces a symmetric relation denoted as $\sim$ on V
  - $\sim$ is called *adjacency relation*
  - $\forall \{x, y\} \in \mathrm{V}^2, \mathrm{x} \sim \mathrm{y} \rightarrow x$ and y are adjacent to one another

- If two nodes, x and y in the graph are adjacent, then there is an edge linking them together

I know Your Feeling



Many Kinds of Graphs

- Simple / Undirected
  - What we just described
- Directed
  - The set $\{x, y\}$ is ordered (only one direction is allowed)
  - We are not excluding $\{x, y\}$ and $\{y, x\}$ to exist at the same time
  - We draw the graph with arrows
- Oriented
  - Same as directed, but only one is allowed between $\{x, y\}$ and $\{y, x\}$
- Mixed
  - Some edged are directed and some are not
- Multigraph
  - We have more edges connecting the same two nodes
- Weighted
  - Every edge has associated a numerical value

Many Kinds of Graphs … for Games

- Simple / Undirected
  - An open world or a labyrinth

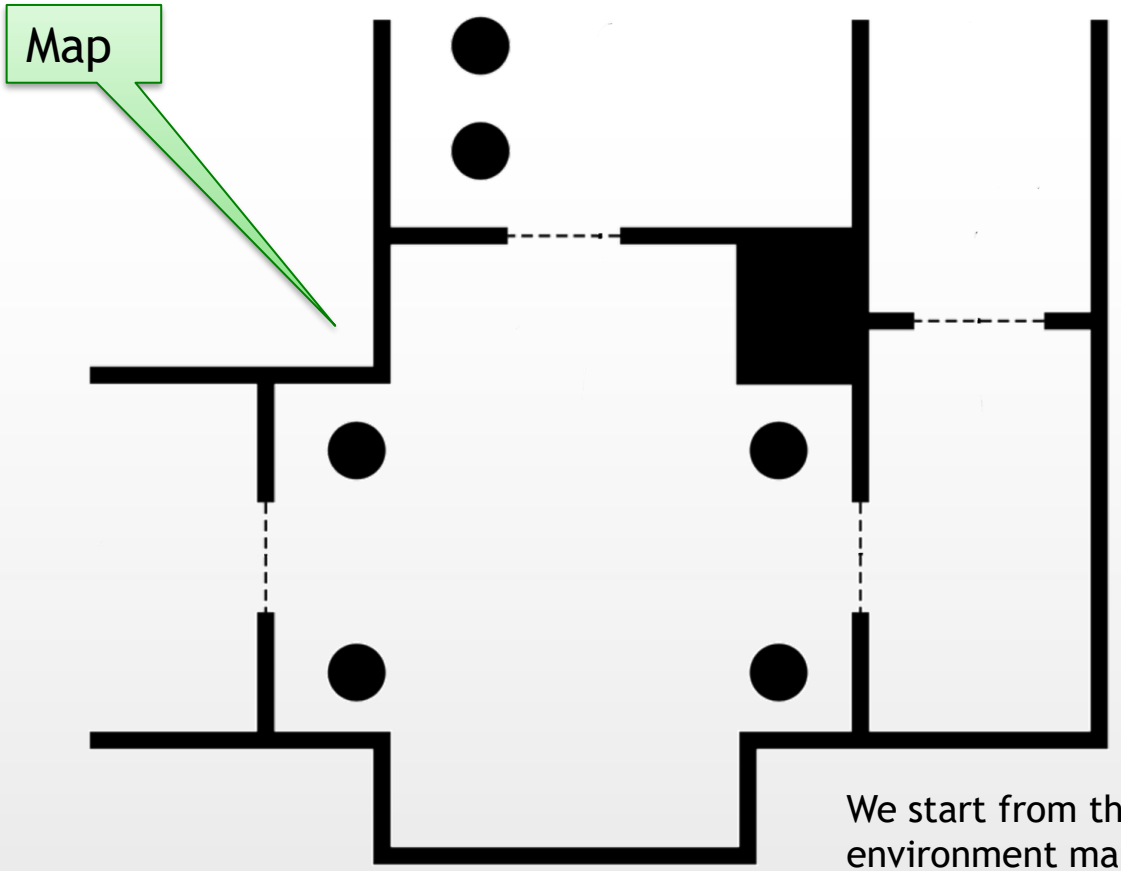
- Directed
  - Connections may be two-ways or one-way, like driving a car in a city

- Oriented
  - Connections are only one way, like in a rail shooter or an infinite runner

Many Kinds of Graphs … for Many Games

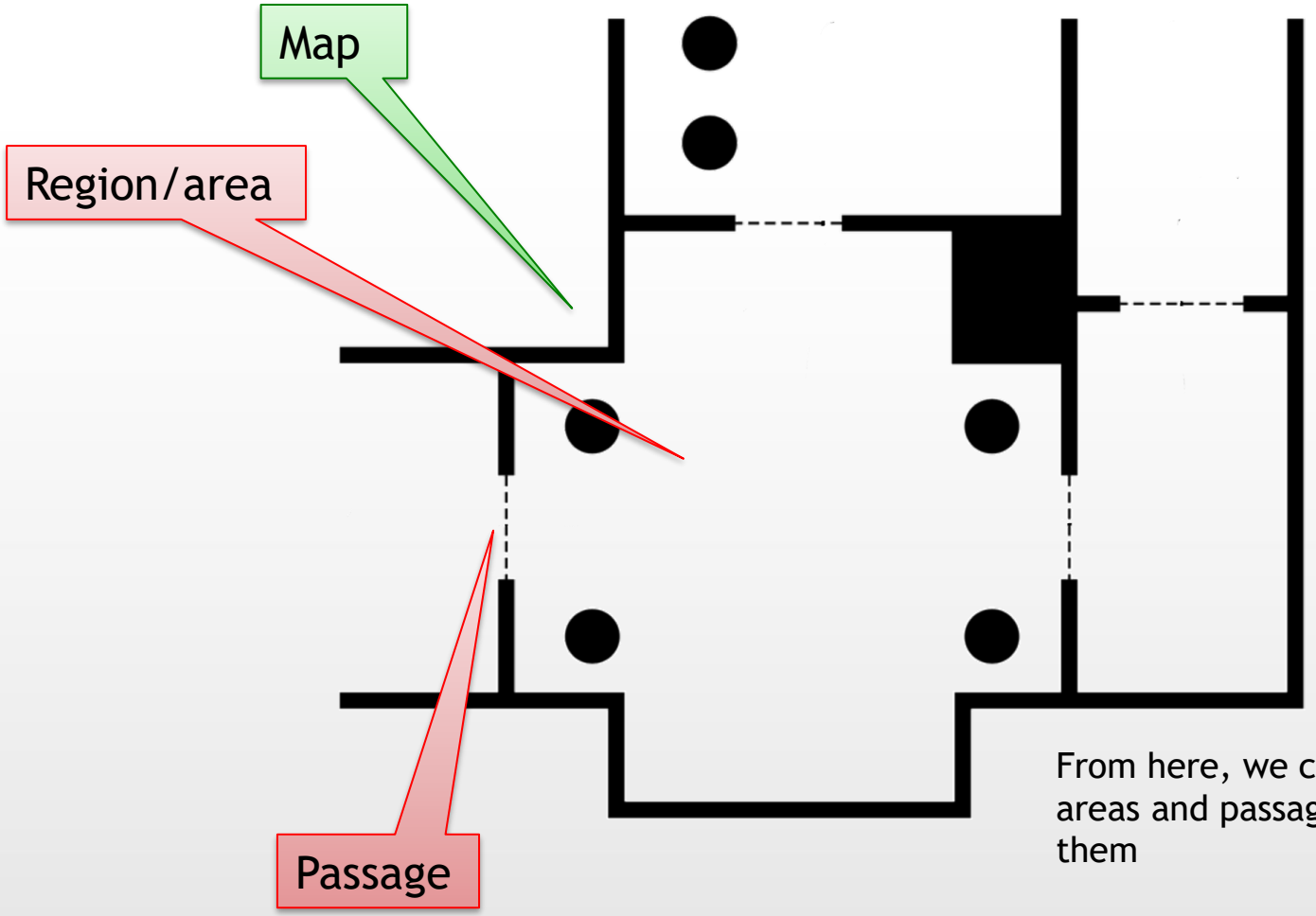
- Mixed
  - When you move freely in a level but you have a one-way to the next one, like in many dungeon crawler
- Multigraph
  - Many way to cover a path, like when you can choose between stealth and assault approach
- Weighted
  - Not all connections have the same importance/cost, like having different probabilities to get spotted depending on the path you take

A Graph for a Path



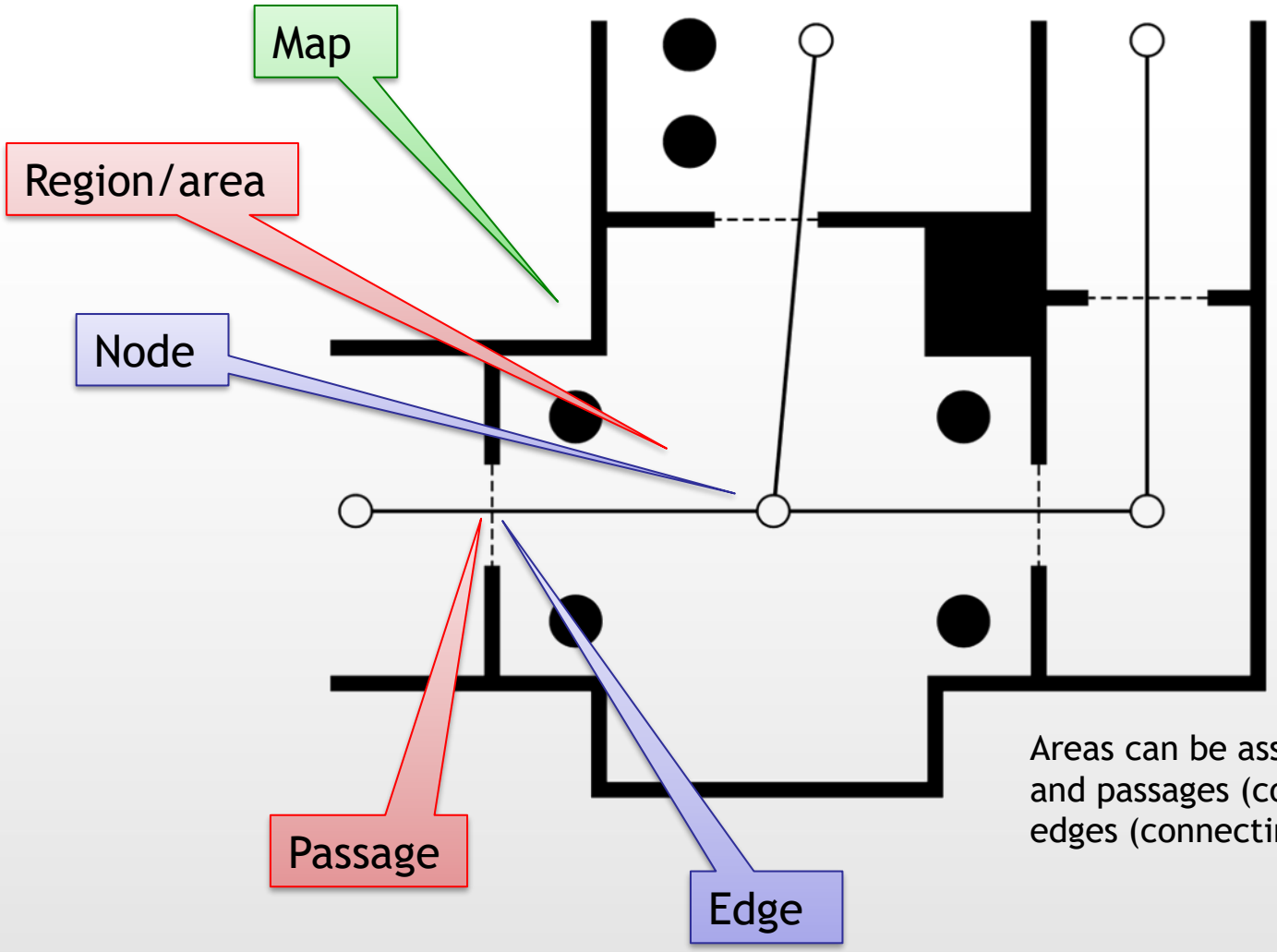
We start from the environment map to understand where an NPC can be interested to move

A Graph for a Path



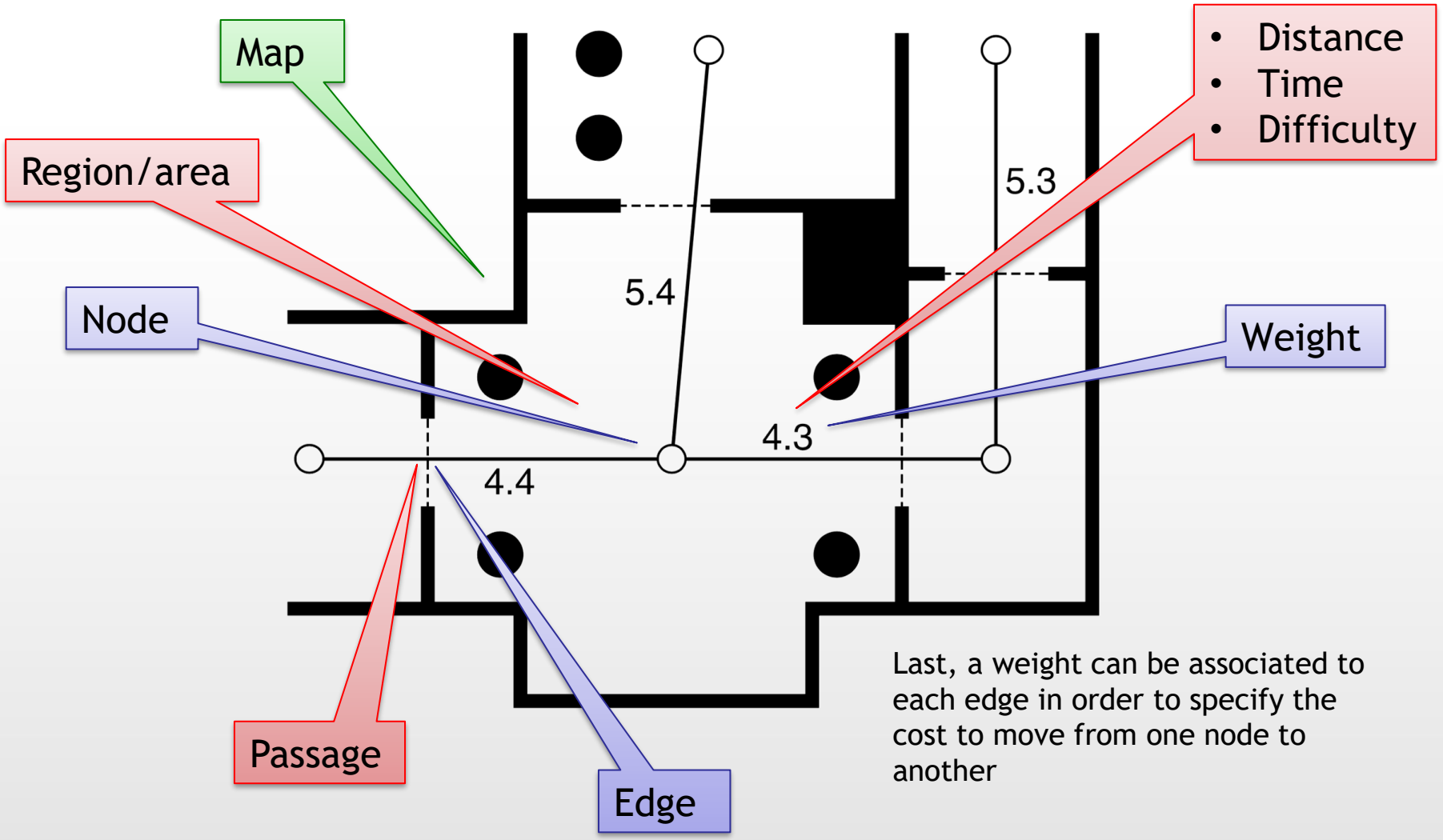
From here, we can recognize areas and passages between them

A Graph for a Path



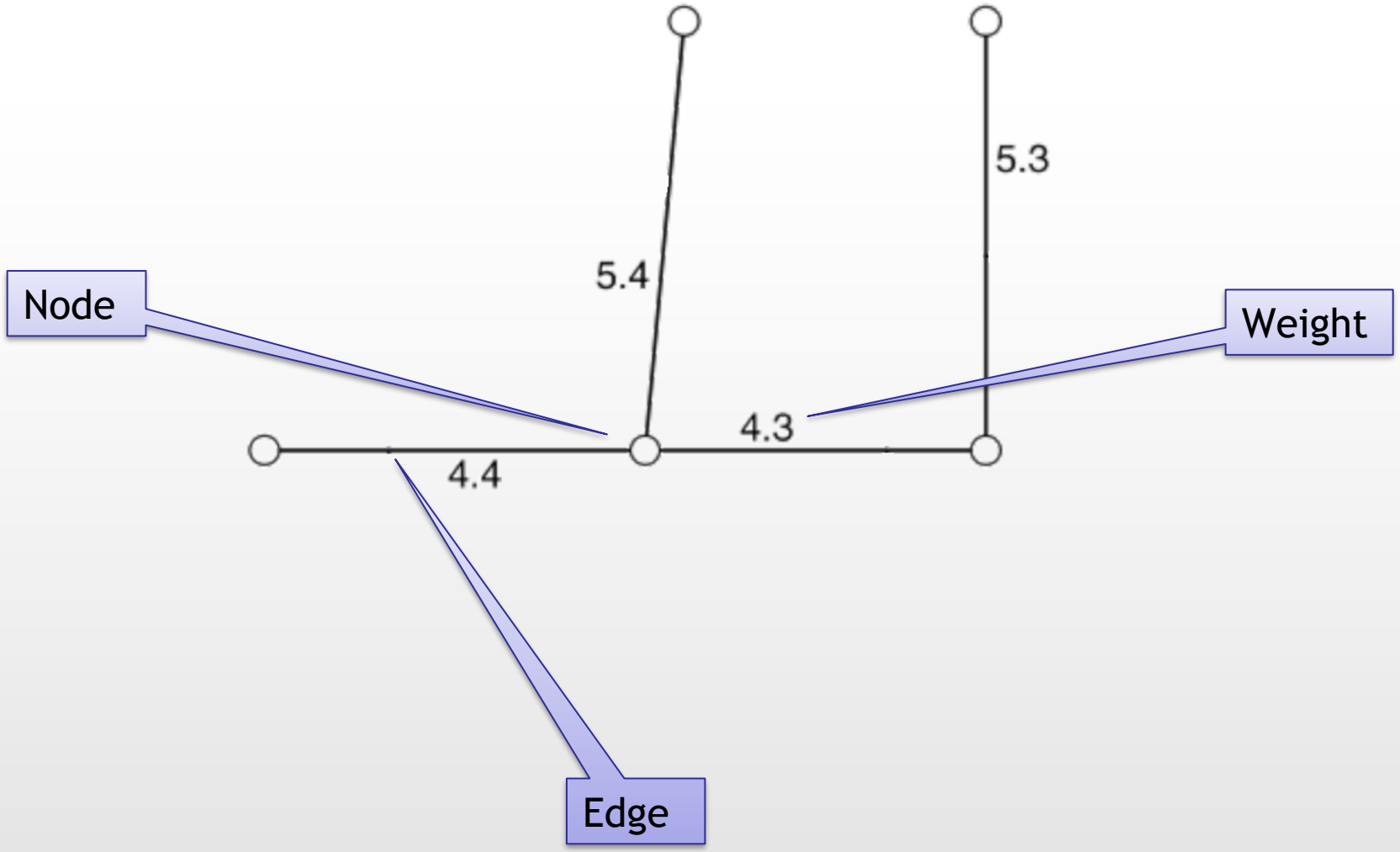
Areas can be associated to nodes and passages (connecting areas) to edges (connecting nodes)

A Graph for a Path



Last, a weight can be associated to each edge in order to specify the cost to move from one node to another

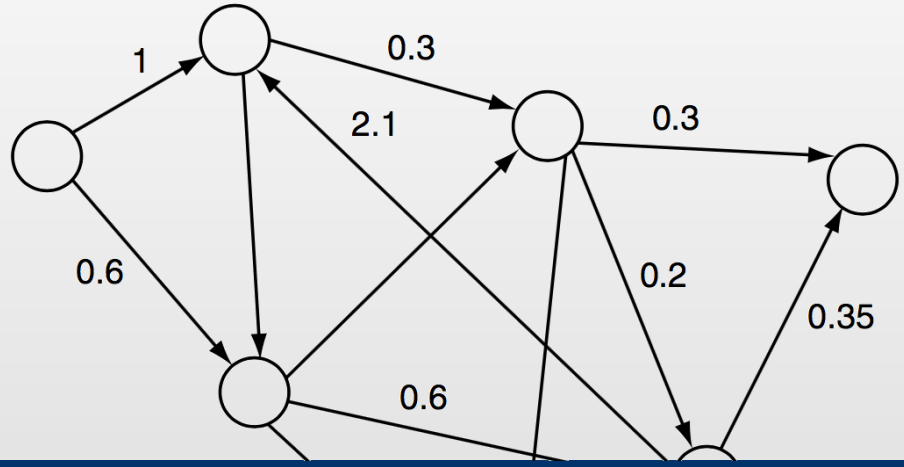
But for the AI it is Going to be Just This!



What We Are Going to Use

- To be ruled out from our implementation
  - Negative weight
    - Algorithms are not working so well or are too complicated to be implemented with good performances
  - Undirected edges
    - Because we do have one-ways

- We are left with positive-weighted oriented graphs
  - Need undirected ? → just use two edges
  - No use for weight ? → Just set everything to 0



Unity Implementation

- Let’s start by defining the data structure for the graph

- This code will be “generic” C#
  - There is no need to fiddle with the internal data structures of unity

Node

Source: Node
Folder: Pathfinding/Graph

```
public class Node {

    public string description;

    public Node(string description) {
        this.description = description;
    }
}
```

All we care about a node right now is to give it a name in order to let the user tell it apart from the others

Edge

Source: Edge
Folder: Pathfinding/Graph

```
// Transition between two nodes
public class Edge {

    public Node from;
    public Node to;
    public float weight;

    public Edge(Node from, Node to, float weight = 1f) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }

}
```

An edge goes from a node to another node with an optional weight (defaults to 1)

Graph

Source: Graph
Folder: Pathfinding/Graph

A graph is a collection (a Dictionary) of nodes linking to each node the list of outgoing edges.

```
public class Graph {

    // holds all edgeds going out from each node
    private Dictionary<Node, List<Edge>> data;

    public Graph() {
        data = new Dictionary<Node, List<Edge>>();
    }

    public void AddEdge(Edge e) {
        AddNode (e.from);
        AddNode (e.to);
        if (!data[e.from].Contains(e))
            data [e.from].Add (e);
    }
```

Adding an edge means adding the the linked nodes and then assigning the link inside the dictionary (if not already there)

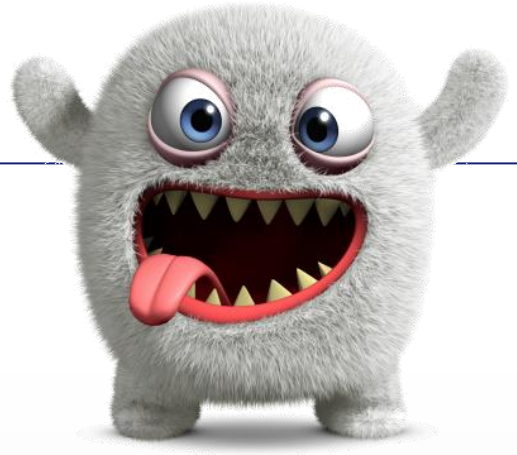
Graph

```
// used only by AddEdge
private void AddNode(Node n) {
    if (!data.ContainsKey (n))
        data.Add (n, new List<Edge> ());
}

// returns the list of edged exiting from a node
public Edge[] getConnections(Node n) {
    if (!data.ContainsKey (n)) return new Edge[0];
    return data [n].ToArray ();
}

public Node[] getNodes() {
    return data.Keys.ToArray ();
}
```

Adding a node will just create a new entry in the dictionary

To explore the graph it is useful to get all outgoing edges from a given node

DIJKSTRA



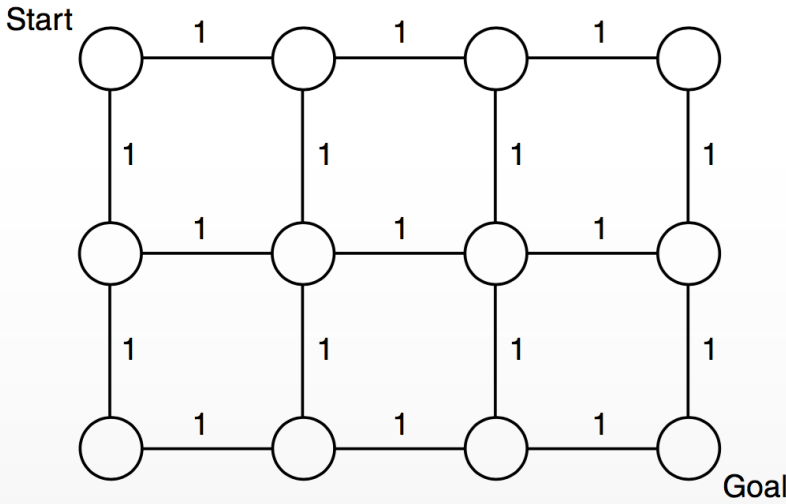
- NOT a pathfinder, just a shortest path inside a graph!
  - But, if we associated the map to a graph .. it is exactly the same thing!

- Actually, Dijkstra's algorithm calculates the shortest path from a starting point to anywhere in the graph, so we have a superset
  - Resources will be wasted
  - Kind of inefficient

- Spoiler alert: A* (next lesson) is an improvement on Dijkstra's

Running Dijkstra

- Input
  - A graph
  - A start (node)
  - A goal (node)



- Output
  - A list of edges to go from start to goal with minimal weight
    Considering the total weight as the sum of the weight of all edges

- Important!
  - Result may not be unique because there might be many ways with the same weight to go from start to goal (see the picture above)

Classical Algorithm

1. We define two set of nodes: visited and unvisited
   - Set the unvisited set as the complete list of nodes and the visited set as empty
2. For every node we define a *tentative distance* (from the start) value and a predecessor node
   - Set 0 to the start node and infinite for all other nodes
3. Select a current visited node as the one in the unvisited set with the smallest tentative distance
4. Process the current node
   - Assign tentative distance and predecessor to all its neighbors only where the new tentative weight is lower than the current
   - Neighbors' value will be the sum of local tentative distance and the weight of the edge leading to the neighbor
5. Mark the current node as visited
6. If the unvisited list is not empty go back to step 3
7. Return the minimal path by backtracking from the goal node

Millington's Version

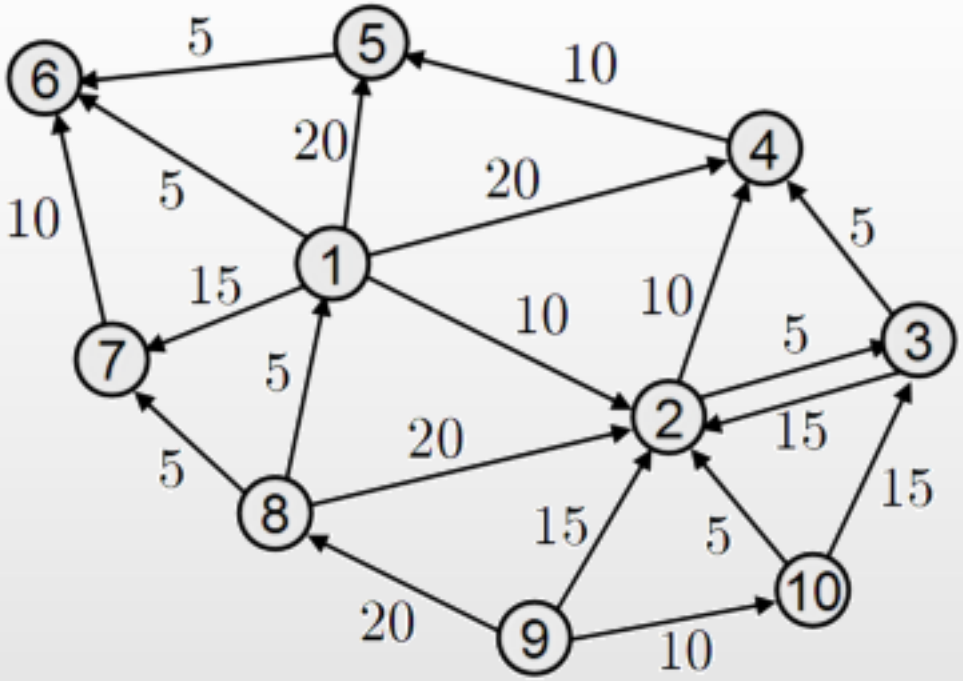
- Instead of using Visited and Unvisited sets, the author defines three sets

- Unvisited nodes
  - Unvisited nodes with infinite tentative distance
- Open Nodes
  - Unvisited nodes with a non infinite distance
  - These are the border of the visited nodes set
- Closed Nodes
  - Visited nodes

- Just the same but avoids to deal with the "*infinite*" value
  - There are good (performance) reasons to do this

What if …

- We loop back to a visited/closed node with a shortest path?
- Easy-peasy: Mr. Dijkstra proved that it is NOT going to happen



Implementation

Source: DijkstraSolver
Folder: Pathfinding/Dijkstra

```
public static class DijkstraSolver {

    // two set of nodes (1)

    static List<Node> visited;
    static List<Node> unvisited;

    // data structures to extend nodes (2)

    private struct NodeExtension {
        public float distance;
        public Edge predecessor;
    }

    static Dictionary<Node, NodeExtension> status;
```

This class is just a code container to run the Dijkstra algorithm inside unity.
Yet ... no unity code here

We use this struct to “enrich” each node with information about its predecessor and its distance from the start node

Association between a node and its extension is performed via a dictionary

Implementation

```
public static class DijkstraSolver {

    // two set of nodes (1)

    static List<Node> visited;
    static List<Node> unvisited;

    // data structures to extend nodes (2)

    private struct NodeExtension {
        public float distance;
        public Edge predecessor;
    }

    static Dictionary<Node, NodeExtension> status;
```

This is technically redundant

It is redundant space-wise because we could re-compute the path each time, but doing that will not be good performance-wise

Implementation

```
public static Edge[] Solve(Graph g, Node start, Node goal) {

    // setup sets (1)
    visited = new List<Node>();
    unvisited = new List<Node> (g.getNodes ());

    // set all node tentative distance (2)
    status = new Dictionary<Node, NodeExtension> ();
    foreach (Node n in unvisited) {
        NodeExtension ne = new NodeExtension();
        ne.distance = ( n == start ? 0f : float.MaxValue ); // infinite
        status [n] = ne;
    }
```

The Solve method is simply an implementation of the Dijkstra algorithm (numbers in parenthesis are the step numbers in the algorithm description)

Implementation

```
public static Edge[] Solve(Graph g, Node start, Node goal) {

    // setup sets (1)
    visited = new List<Node>();
    unvisited = new List<Node> (g.getNodes ());

    // set all node tentative distance (2)
    status = new Dictionary<Node, NodeExtension> ();
    foreach (Node n in unvisited) {
        NodeExtension ne = new NodeExtension();
        ne.distance = ( n == start ? 0f : float.MaxValue ); // infinite
        status [n] = ne;
    }
```

We are assuming that in no way there will be a possible path longer than float.MaxValue

(think about Red Dead Redemption 2)

Potential bug

Implementation

```
// iterate until all nodes are visited (6)
while (unvisited.Count > 0) {
    // select next current node (3)
    Node current = GetNextNode();
    // assign weight and predecessor to all neighbors (4)
    foreach (Edge e in g.getConnections(current)) {
        if (status[current].distance + e.weight < status[e.to].distance) {
            NodeExtension ne = new NodeExtension();
            ne.distance = status[current].distance + e.weight;
            ne.predecessor = e;
            status[e.to] = ne;
        }
    }
    // mark current node as visited (5)
    visited.Add(current);
    unvisited.Remove(current);
}
```

All nodes linked from node N will have tentative distance as the sum of the distance of N plus the weight of the linking edge.
They will stay in the unvisited set but will get a tentative distance lower than float.MaxValue, thus being eligible as a possible next node

Implementation

```
    // walk back and build the shortest path (7)
    List<Edge> result = new List<Edge> ();
    Node walker = goal;

    while (walker != start) {
        result.Add(status[walker].predecessor);
        walker = status[walker].predecessor.from;
    }
    result.Reverse();
    return result.ToArray();
}
```

To return the list of edges we walk back from the goal predecessor-by-predecessor and put everything in a list. The list is then reversed and converted into an array

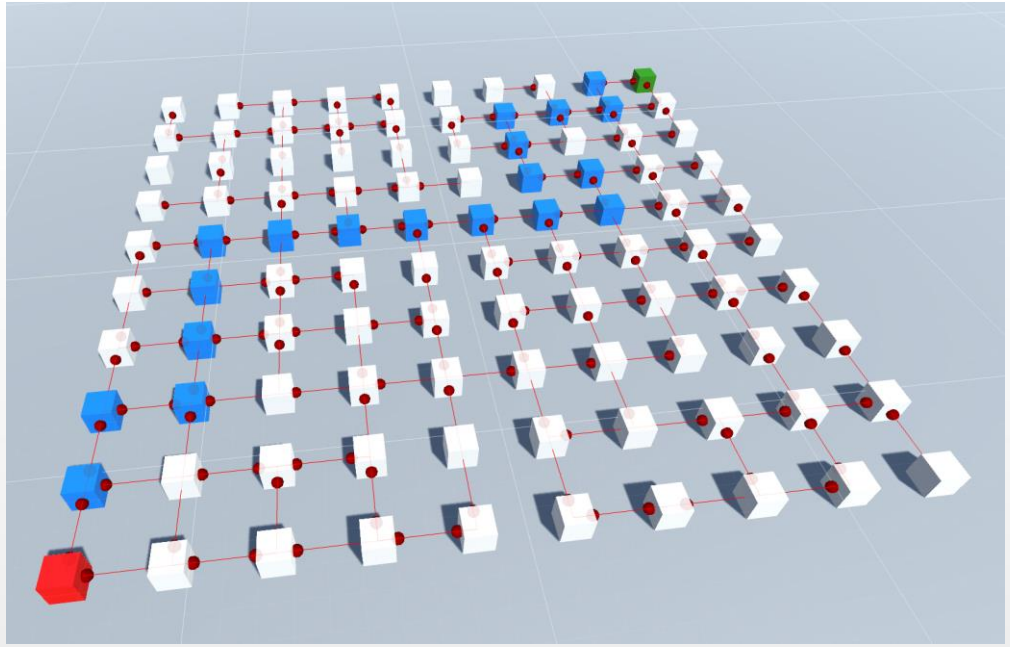
Implementation

```
// iterate on the unvisited set and get the lowest weight
private static Node GetNextNode() {
    Node candidate = null;
    float cDistance = float.MaxValue;
    foreach (Node n in unvisited) {
        if (candidate == null || cDistance > status[n].distance) {
            candidate = n;
            cDistance = status[n].distance;
        }
    }
    return candidate;
}
```

GetNextNode is a utility method to select the node with the lowest distance inside the unvisited set. In case of more nodes with the same distance, the first one is returned

From Theory to Practice

- Let’s create a random Manhattan-like maze and use Dijkstra’s algorithm to find the shortest path between two opposite corners



- To do this, we must first re-design the graph classes and make then aware they “live” inside unity

First Problem

- We must link each node to a place on the map or, at least, a GameObject

Red underlines are showing the differences

This field could be replaced by the Unity gameobject name

```
using UnityEngine;

public class Node {

    public string description;
    public GameObject sceneObject;

    public Node(string description, GameObject o = null) {
        this.description = description;
        this.sceneObject = o;
    }
}
```

sceneObject is the reference object for this node. It is not required to have a physical object: if the object in question has only the transform component, then it is the equivalent of a coordinate in world space and can easily be a rally point

Second Problem

- The graph is random, we cannot assume a solution exists

```
// iterate until all nodes are visited (6)
while (unvisited.Count > 0) {
    // select net current node (3)
    Node current = GetNextNode();

    if (status[current].distance == float.MaxValue) break; // graph is partitioned

    // assign weight and predecessor to all neighbors (4)
    foreach (Edge e in g.getConnections(current)) {
        if (status[current].distance + e.weight < status[e.to].distance) {
            NodeExtension ne = new NodeExtension();
            ne.distance = status[current].distance + e.weight;
            ne.predecessor = e;
            status[e.to] = ne;
        }
    }
    // mark current node as visited (5)
    visited.Add(current);
    unvisited.Remove(current);
}

if (status [goal].distance == float.MaxValue) return new Edge[0]; // goal is unreachable
```

If there is no way to use an edge to expand the visited set, then GetNextNode will return a node with distance float.MaxValue

Returning a zero-length array is a way to communicate that the goal is unreachable

Let’s go!

Source: DijkstraSquare
Folder: Pathfinding/Dijkstra

```
void Start () {
    if (sceneObject != null) {

        // initialize randomness, so experiments can be repeated
        if (RandomSeed == 0) RandomSeed = (int)System.DateTime.Now.Ticks;
        Random.InitState (RandomSeed);

        // create a x * y matrix of nodes (and scene objects)
        matrix = CreateGrid(sceneObject, x, y, gap);

        // create a graph and put random edges inside
        g = new Graph();
        CreateLabyrinth(g, matrix, edgeProbability);

        // ask dijkstra to solve the problem
        Edge[] path = DijkstraSolver.Solve(g, matrix[0, 0], matrix [x - 1, y - 1]);

        // check if there is a solution
        if (path.Length == 0) {
            UnityEditor.EditorUtility.DisplayDialog ("Sorry", "No solution", "OK");
        } else {
            // if yes, outline it
            OutlinePath(path, startMaterial, trackMaterial, endMaterial);
        }
    }
}
```

The Start method is run as soon as the object is in the scene.
In this case, when we hit “play”

Details

We will extend the battlefield later in a subclass

```
protected virtual Node[,] CreateGrid(GameObject o, int x, int y, float gap) {
    Node[,] matrix = new Node[x,y];
    for (int i = 0; i < x; i += 1) {
        for (int j = 0; j < y; j += 1) {
            matrix[i, j] = new Node("" + i + "," +j, Instantiate(o));
            matrix[i, j].sceneObject.name = o.name;
            matrix[i, j].sceneObject.transform.position =
                transform.position +
                transform.right * gap * (i - ((x - 1) / 2f)) +
                transform.forward * gap * (j - ((y - 1) / 2f));
            matrix[i, j].sceneObject.transform.rotation = transform.rotation;
        }
    }
    return matrix;
}
```

CreateGrid will deploy a grid of x by y similar gameobjects (first parameter) with a distance of gap (fourth parameter) in the scene.
Each object will be associated to a graph node (no edges right now)

Details

```
protected void CreateLabyrinth(Graph g, Node[,] crossings, float threshold) {
    for (int i = 0; i < crossings.GetLength(0); i += 1) {
        for (int j = 0; j < crossings.GetLength(1); j += 1) {
            g.AddNode(crossings[i, j]);
            foreach (Edge e in RandomEdges(crossings, i, j, threshold)) {
                g.AddEdge(e);
            }
        }
    }
}
```

CreateLabyrinth will populate a graph starting from the node array. Edges will be randomly generated between neighbors by method RandomEdges using a linear distribution with parameter threshold (fourth parameter)

```
protected void OutlinePath(Edge[] path, Material sm, Material tm, Material em) {
    if (path.Length == 0) return;
    foreach (Edge e in path) {
        e.to.sceneObject.GetComponent<MeshRenderer>().material = tm;
    }
    path[0].from.sceneObject.GetComponent<MeshRenderer>().material = sm;
    path[path.Length - 1].to.sceneObject.GetComponent<MeshRenderer>().material = em;
}
```

OutlinePath is just outlining the result by changing materials to the objects belonging to the path

Details

RandomEdges is taking care to generate random edges between neighbors in the grid. The reason why you want to keep this separate from CreateLabyrinth is to keep the logic generating your edges separated from the code converting them in a graph

```
protected Edge[] RandomEdges(Node[,] matrix, int x, int y, float threshold) {
    List<Edge> result = new List<Edge>();
    if (x != 0 && Random.Range(0f, 1f) <= threshold)
        result.Add(new Edge(matrix[x, y], matrix[x - 1, y], Distance(matrix[x, y], matrix[x - 1, y])));

    if (y != 0 && Random.Range(0f, 1f) <= threshold)
        result.Add(new Edge(matrix[x, y], matrix[x, y - 1], Distance(matrix[x, y], matrix[x, y - 1])));

    if (x != (matrix.GetLength(0) - 1) && Random.Range(0f, 1f) <= threshold)
        result.Add(new Edge(matrix[x, y], matrix[x + 1, y], Distance(matrix[x, y], matrix[x + 1, y])));

    if (y != (matrix.GetLength(1) - 1) && Random.Range(0f, 1f) <= threshold)
        result.Add(new Edge(matrix[x, y], matrix[x, y + 1], Distance(matrix[x, y], matrix[x, y + 1])));

    return result.ToArray();
}

protected virtual float Distance(Node from, Node to) {
    return 1f;
}
```

The distance between two (connected) nodes is always 1. Technically correct even if it does not make much sense to you right now.
We will come back to change this later

A Little Bonus

```
void OnDrawGizmos() {
    if (matrix != null) {
        Gizmos.color = edgeColor;
        for (int i = 0; i < x; i += 1) {
            for (int j = 0; j < y; j += 1) {
                foreach (Edge e in g.getConnections(matrix[i, j])) {
                    Vector3 from = e.from.sceneObject.transform.position;
                    Vector3 to = e.to.sceneObject.transform.position;
                    Gizmos.DrawSphere(from + ((to - from) * .2f), .2f);
                    Gizmos.DrawLine (from, to);
                }
            }
        }
    }
}
```

Gizmos are drawn inside the editor to provide visual clues and debug information (you cannot see them in the camera).
This gizmo is drawing a (thin) line between connected nodes and a small sphere to emulate the head of the arrow (allowed movement).
This way, we can see the graph and check if there is no solution for Dijkstra.
See slide titled “From Theory to Practice”

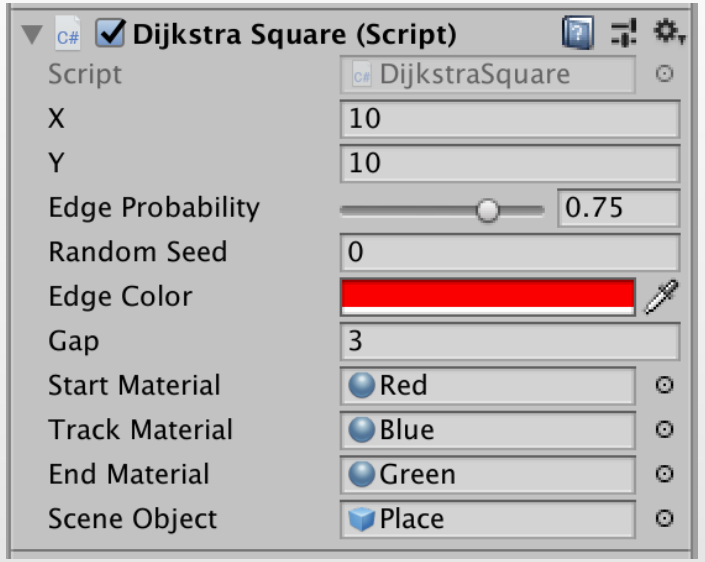
Inside Unity

Scene: Square
Folder: Pathfinding/Dijkstra

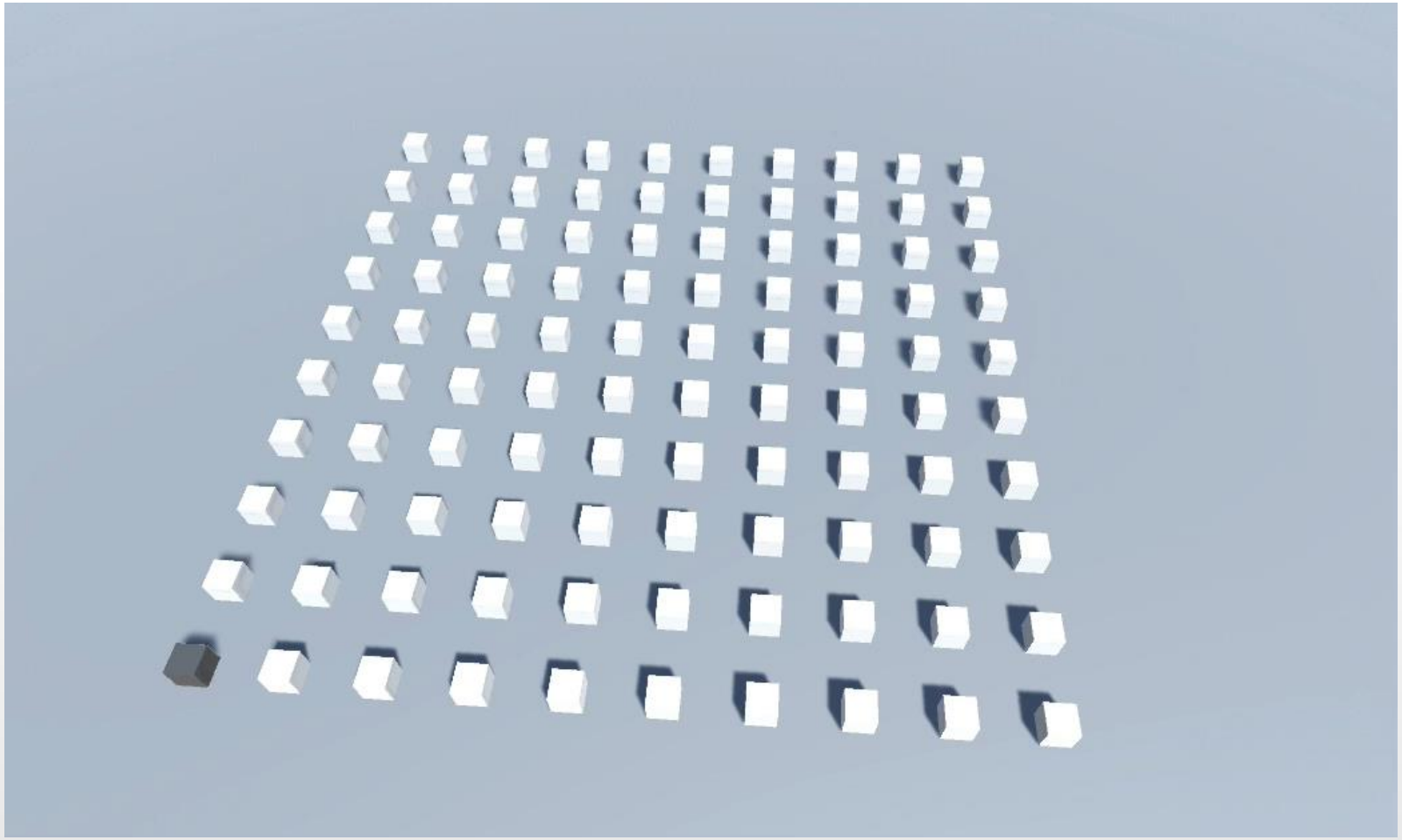
- In the scene, the FieldGenerator object has a DijkstraSquare component exposing the various working parameters and performing the calculation

- As it is, the scene is not really rewarding because the result is just popping out
- Open the “Animated Square” scene in the “Animated” subfolder to see Dijkstra running
  - Coroutines are used to achieve step-by-step progression

- In the Random Seed field you will see the number generating the labyrinth. Copy and paste that value to re-evaluate the same graph multiple times. Using 0, Unity will generate a seed for you.

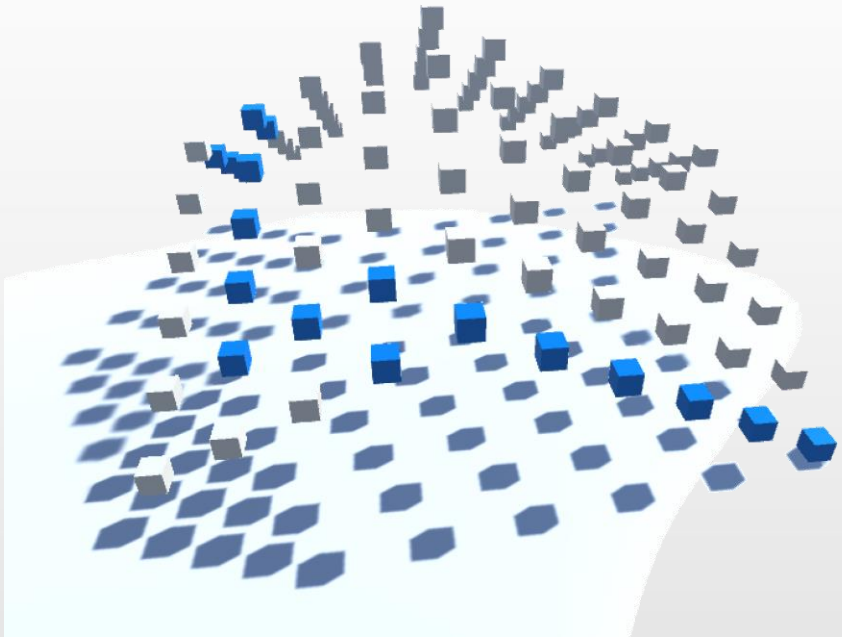


See it Running

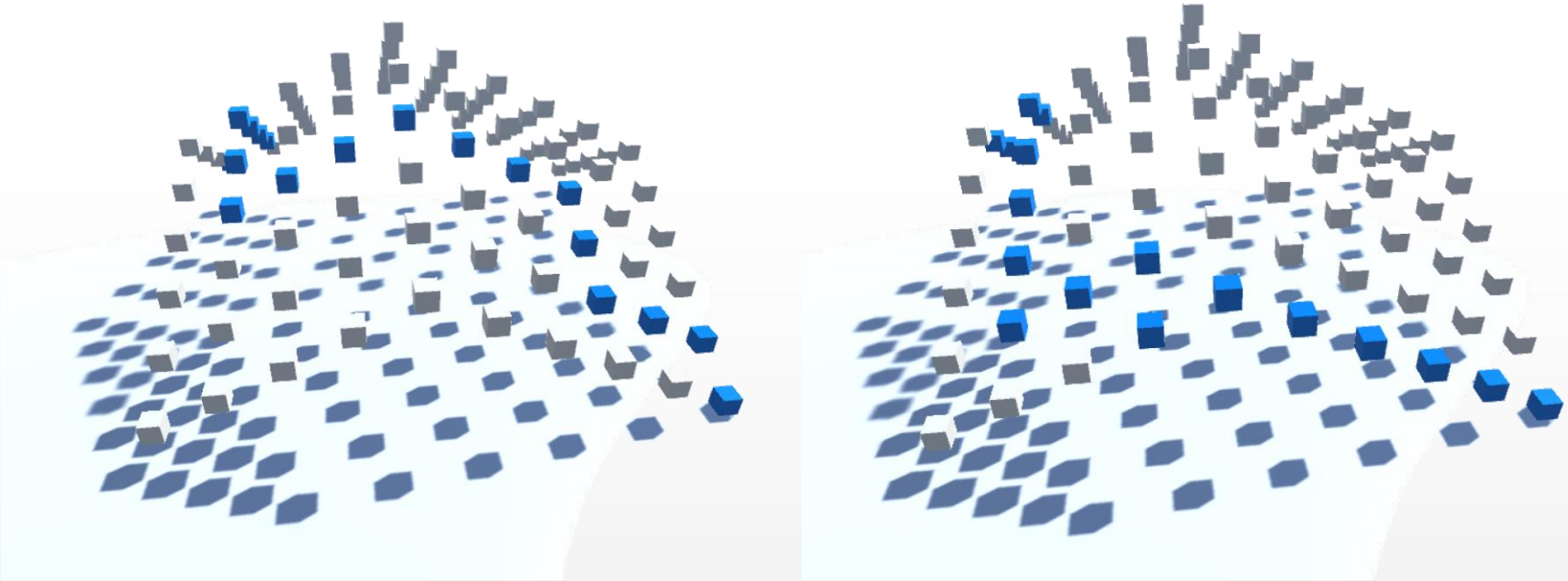


Uneven Terrain and Slopes

- Not all battlefields are flat as a pancake
  - Actually, it is very boring when they are
- If our NPC is lazy, hampered, or overloaded, it will generally avoid going upward and prefer the road going downward
  - When dealing with a mountain might you find acceptable a long path skirting it and will go to the top only when there is no other way around
- To achieve this, we must consider geometrical features of our objects



Mountains



0.7 0.9

Reducing the edge probability, a way up to the top is more likely to happen

Implementing Mountains

Tip: remember the virtual + override keywords

Scene: Mountain
Folder: Pathfinding/Dijkstra

```
protected override Node[,] CreateGrid(GameObject o, int x, int y, float gap) {
    Node[,] matrix = new Node[x,y];
    for (int i = 0; i < x; i += 1) {
        for (int j = 0; j < y; j += 1) {
            matrix[i, j] = new Node("" + i + "," +j, Instantiate(o));
            matrix[i, j].sceneObject.name = o.name;
            matrix[i, j].sceneObject.transform.position =
                transform.position +
                    Vector3.right * gap * (i - ((x - 1) / 2f)) +
                    Vector3.forward * gap * (j - ((y - 1) / 2f)) +
                    Vector3.up * gap * (((i <= x / 2f ? i : x - i) + (j <= y / 2f ? j : y - j) ) / 2f) ;
            matrix[i, j].sceneObject.transform.rotation = transform.rotation;
        }
    }
    return matrix;
}

protected override float Distance(Node from, Node to) {
    float h = to.sceneObject.transform.position.y - from.sceneObject.transform.position.y;
    if (h <= 0) return 1;
    return 3;
}
```

We move the object up by gap times a factor based on position in the matrix

Distance is 3 when going up and 1 otherwise.
Going up should really mean a shortcut for us

References

- On the textbook
  - § 4.1
  - § 4.2.1
  - § 4.2.2