UNIVERSITÀ DEGLI STUDI
DI MILANO

A*

A.I. for Video Games

Dario Maggiorini (dario@di.unimi.it)

A* (Pronounced A-Star)

- It is “just” an evolution of Dijkstra’s algorithm
- Actually, it is the same “but” for a small detail

A* (Differences from Dijkstra)

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* value (from the start) 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 estimated total cost of the path to the goal
4. Process the current node
   - Assign tentative distance, predecessor and estimate distance to the goal for all its neighbors only where the new tentative distance 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 goal was not reached and the unvisited list is not empty go back to step 3
7. Devise the minimal path by backtracking from the goal node

A* vs Dijkstra

- There are two differences:

  – Instead of selecting as next current node the one closest to the starting point we select ”the most promising one”.
  I.e., we select the node that, based on our estimation, is on the shortest path from source to destination
    - This will will keep ”pushing” the exploration toward the destination and consider alternate routes only if no direct way is found

  – We drop the exploration as soon as we reach the destination
    - This will avoid exploring the whole graph and save time

Estimation?

- We must define a heuristic way to estimate distance to the goal
  - Good heuristic ? → great results
  - Bad heuristic ? → results suck

- NOTE: the worst heuristic is ... Dijkstra

- Our problem just changed to "finding a good estimation"

Beware of the Loops

- Unlike Dijkstra, we may find shortest paths to already visited nodes
- We are following a “lead”: we may just get it wrong and discover a better way later

- In such cases we must re-compute the distance of the node from the start and put it back in the unvisited list
  - That node will be re-evaluated, and nodes on its outgoing edges might be as well
- Just be aware, looping is NOT going to happen all the times: it depends on the estimation we use
  - More on this later

Really Important!

- The "found a better loop" issue applies also to the destination

- Even if we reached the destination there is no proof (without an exhaustive search) that we did it using the best possible path
  - And "exhaustive search" implies "falling back to Dijkstra"

- We can use two approaches:
  1. Wait for the goal node to have the shortest distance from the start when compared to all unvisited nodes and be more accurate
  2. Terminate immediately and save time

Implementing A*

Underlines are differences from Dijkstra

Source: AStarSolver
Folder: Pathfinding/AStar

This delegate allows us to set the estimation policy from the outside and avoid touching this code when changing heuristic

```
public delegate float HeuristicFunction(Node from, Node to);

public static class AStarSolver {

    public static bool immediateStop = false;

    // two set of nodes (1)

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

    // data structures to extend nodes (2)

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

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

A flag to stop as soon as we reach the destination or to keep exploring until we are satisfied (see previous slide)

We extend the NodeExtension struct to accommodate the estimation of the path traversing the node

Implementing A*

Thanks to the delegate, we can pass the estimation function as a parameter to the solver

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

    // 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
        ne.estimate = ( n == goal ? 0f : float.MaxValue );
        status [n] = ne;
    }
```

Since the distance is infinite, the estimation is also infinite when starting the solver

Implementing A*

```
// iterate until goal is reached with an optimal path (6)
while (!CheckSearchComplete(goal, unvisited)) {
    // 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.estimate = ne.distance + heuristic(e.to, goal);
            ne.predecessor = e;
            status[e.to] = ne;
            // unlike Dijkstra's, we can now discover better paths
            if (visited.Contains(e.to)) {
                unvisited.Add(e.to);
                visited.Remove(e.to);
            }
        }
    }
    // mark current node as visited (5)
    visited.Add(current);
    unvisited.Remove(current);
}
```

We need to perform a more complex evaluation this time

The estimate is calculated as the current distance from the start and the value of the heuristic calculated on the node

If we have outgoing edges leading to visited nodes, then we found a shortest loop and all outgoing nodes must be put in the unvisited set

Implementing A*

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

The selection of the best candidate is now based on the estimate field of the NodeExtension struct

Implementing A*

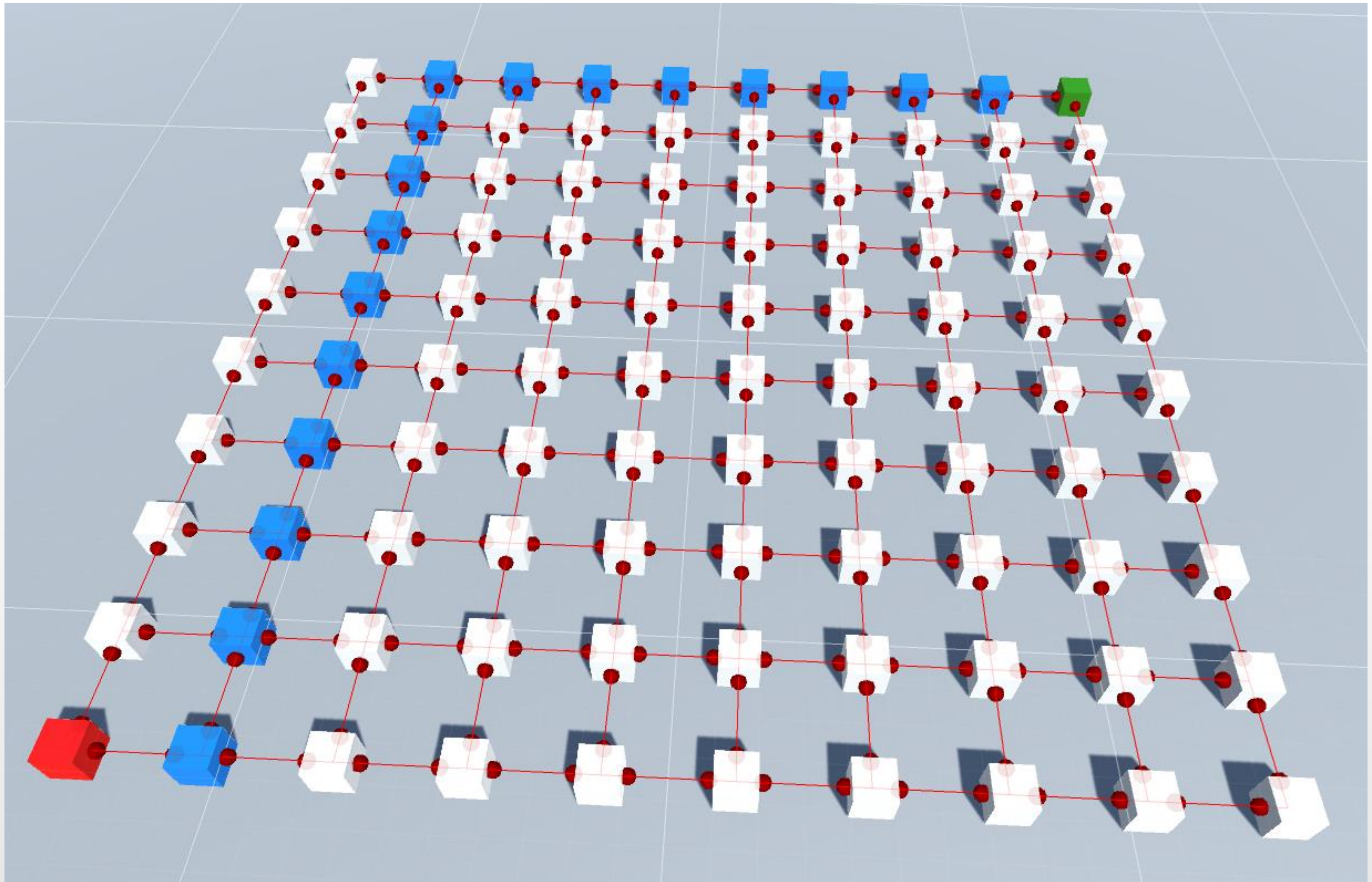
This method is new in A*

```
// chek if the goal has been reached in a satisfactory way
protected static bool CheckSearchComplete(Node goal, List<Node> nodeList) {
    // check if we reached the goal
    if (status [goal].distance == float.MaxValue) return false;
    // check if the first hit is ok
    if (immediateStop) return true;
    // check if all nodes in list have loger or same paths
    foreach (Node n in nodeList) {
        if (status[n].distance < status[goal].distance) return false;
    }
    return true;
}
```

If we reached the destination and did not stop immediately, then we must check all the nodes in the graph.

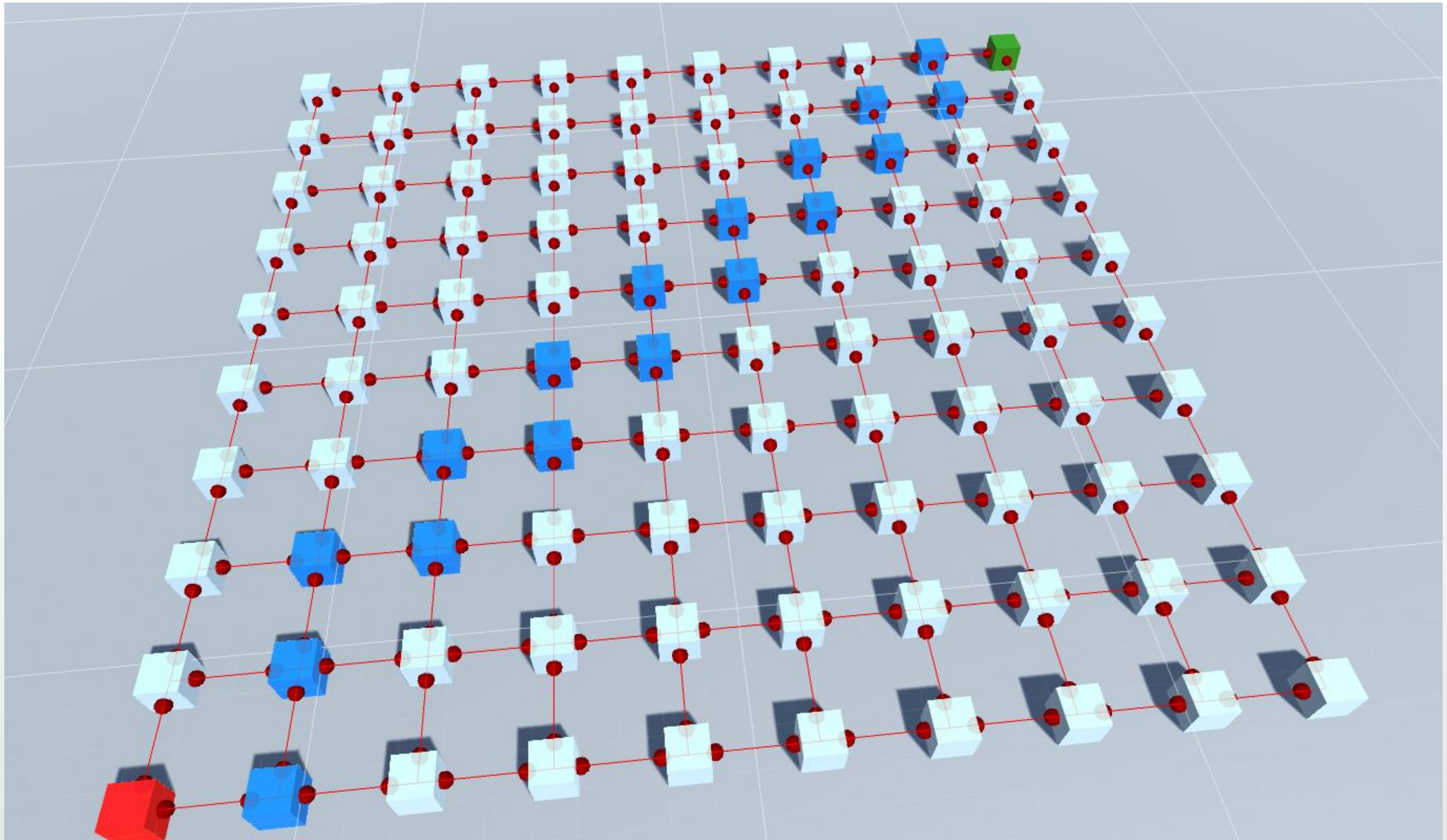
If one or more nodes have a distance from the start shorter than the path we found, we must go on exploring

Remember Dijkstra's Average Result



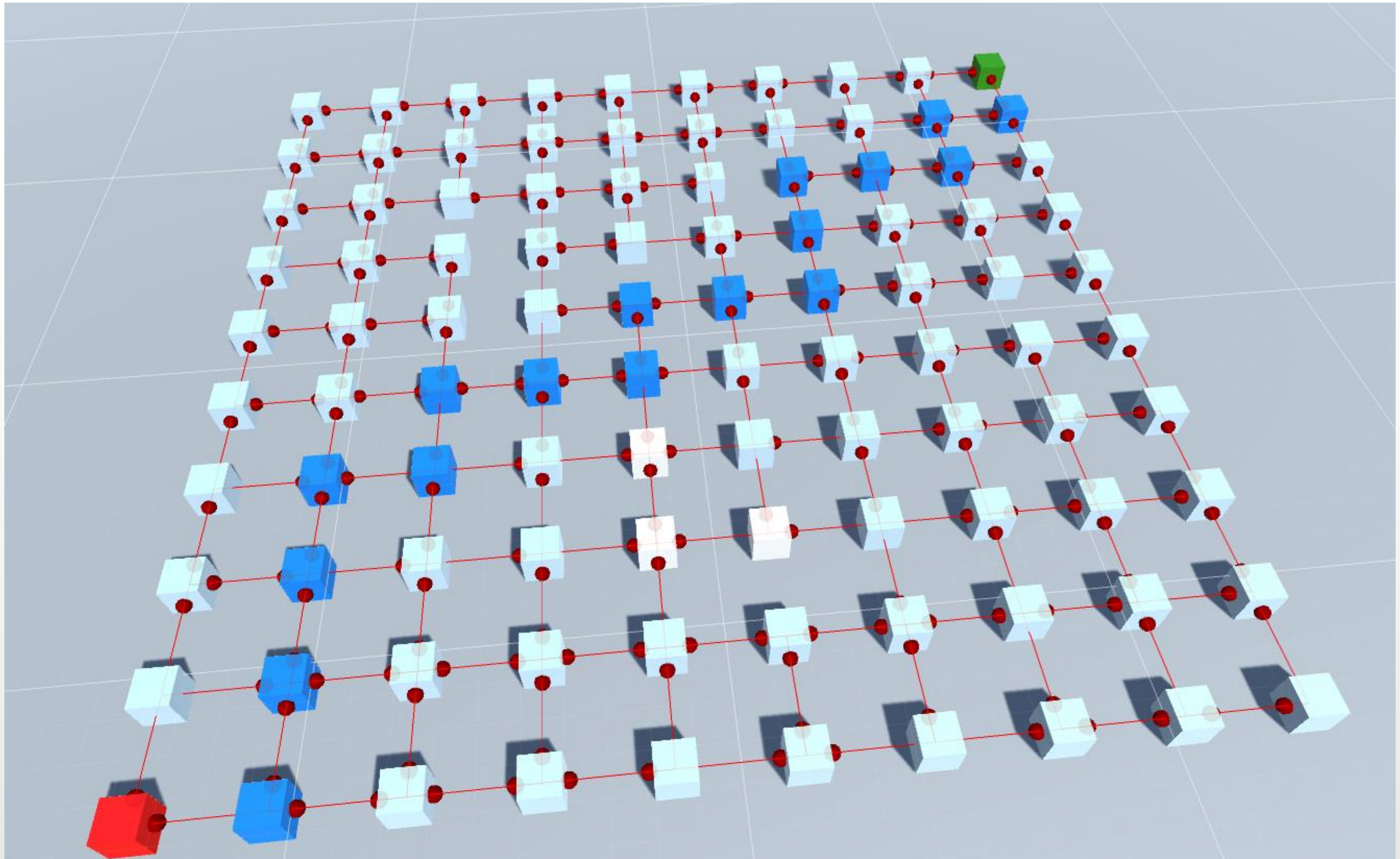
Using a full mesh

Using A*


Using a full mesh

Putting A* Into Unity

Unvisited nodes here are marked white



With connection probability 0.75

How to Get There (Basic Version)

Source: AStarSquareBasic
Folder: Pathfinding/AStar

```
public class AStarSquareBasic : DijkstraSquare {

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

            // create a x * y matrix of nodes (and scene objects)
            // edge weight is now the geometric distance (gap)
            matrix = CreateGrid(sceneObject, x, y, gap);

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

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

            // 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);
            }
        }
    }

    private float EuclideanEstimator(Node from, Node to) {
        return (from.sceneObject.transform.position - to.sceneObject.transform.position).magnitude;
    }
}
```

This is the only change we need from DijkstraSquare

Choosing a Heuristic

- The best choice is a heuristic which can predict the exact minimum path between each node and the destination
  - This is not possible. This is implying a previous knowledge of the result. If we already know the result ... why are we running this algorithm?

- Underestimation
  - A* will be biased toward exploring nodes close to the start
  - If we ALWAYS underestimate, A* will converge di Dijkstra
    - Setting the estimation to 0 is replicating Dijkstra
  - Underestimating is good for accuracy but bad for time

- Overestimation
  - A* will be biased toward the heuristic
  - Will create sub-optimal paths with fewer nodes and higher total cost
  - If the heuristic overestimate is bounded by *X*, so it is the sub-optimality of the path

Back in the Loops

- Loops cannot occur in those cases where we can demonstrate A* is optimal
  - I.e., A* can always find the optimal solution

- It can be demonstrated that A* is optimal if the heuristic we are using is admissible
  - An admissible heuristic is **ALWAYS underestimating** the shortest possible path to the destination
    - An admissible heuristic is optimistic about the distance to cover

- While this is fine from a mathematical standpoint, we cannot make assumptions about the function our developer is going to implement

- From a software engineering standpoint, we must assume the heuristic is not admissible unless it is hardcoded (and verified) inside the A* implementation

Heuristic Pre-Calculation

- Performance question: is it ok to perform the estimation all the time for every node?
  - Could it be better for us to compute the estimation on startup for each node and then consider it as a feature of the node?

- Yes, this can be done, but there are a number of reasons not to do so:
  - Estimation might change dynamically based on external parameters (e.g., the amount of fuel the NPC has)
  - If the map is HUGE, it is not going to improve performances (especially if the NPC starts mid-way from the border and the destination)
  - We must do a round of computation for every possible destination (for e free-roaming map that is going to be a big CPU and memory hog)

- What we can do, if the heuristic is computationally intensive, is keeping a cache of the last N estimations
  - Locality will help you a lot, unless your NPC is teleporting around

How to Get There (Less Basic)

Source: AStarSquare
Folder: Pathfinding/AStar

```
public class AStarSquare : DijkstraSquare {

    public bool stopAtFirstHit = false;
    public Material visitedMaterial = null;

    public enum Heuristics { Euclidean, Manhattan, Bisector, FullBisector, Zero };
    public HeuristicFunction [] myHeuristics = { EuclideanEstimator, ManhattanEstimator, BisectorEstimator,
                                                 FullBisectorEstimator, ZeroEstimator };
    public Heuristics heuristicToUse = Heuristics.Euclidean;

    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)
            // edge weight is now the geometric distance (gap)
            matrix = CreateGrid(sceneObject, x, y, gap);

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

            // ask A* to solve the problem
            AStarSolver.immediateStop = stopAtFirstHit;
            Edge [] path = AStarSolver.Solve (g, matrix [0, 0], matrix [x - 1, y - 1], myHeuristics [(int) heuristicToUse]);

            // Outline visited nodes
            OutlineSet(AStarSolver.visited, visitedMaterial);

            // 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);
            }
        }
    }
```

We are in a subclass.
These fields will add elements to the UI

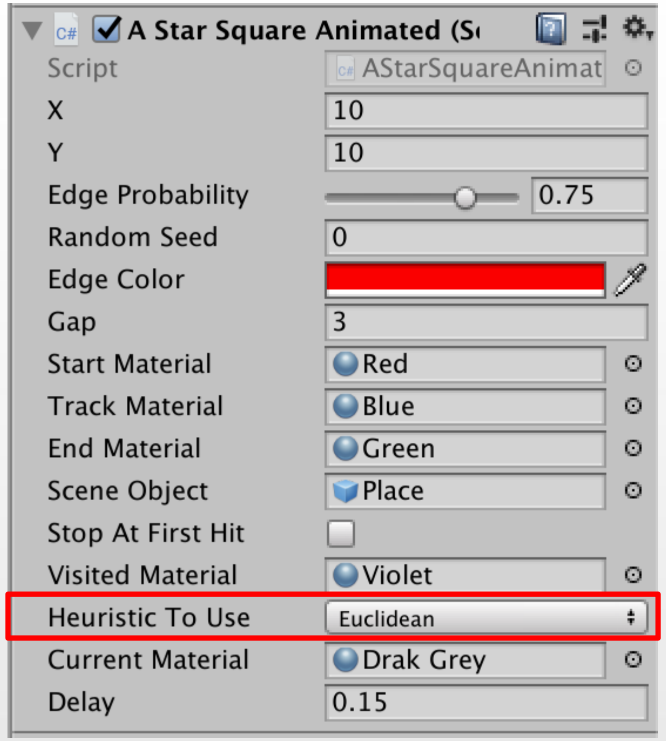
This will allow us to select a heuristic from the UI

The stop at first hit is set here because I was not willing to change the API interface of the solver

Casting enum to int will convert it to an index.
You must be sure to use the same order in the Heuristics and myHeuristics arrays

Let’s Try Different Estimators

- In the following slides, we will test some basic heuristics
  - You can reproduce the same results in unity setting the StarSquare component as follows:
    - X = 20
    - Y = 20
    - Edge Probability = 0.5
    - Random Seed = 18138



- The heuristic we will present are neither *standard* nor *universal*. They are just easily implementable given the context

- Every heuristic must be modeled based on the intended behavior of the NPC.
  - The way you perform the estimation must be in line with the way the NPC should “think about reaching the destination”
  - This is will add **a lot** of realism to your NPCs

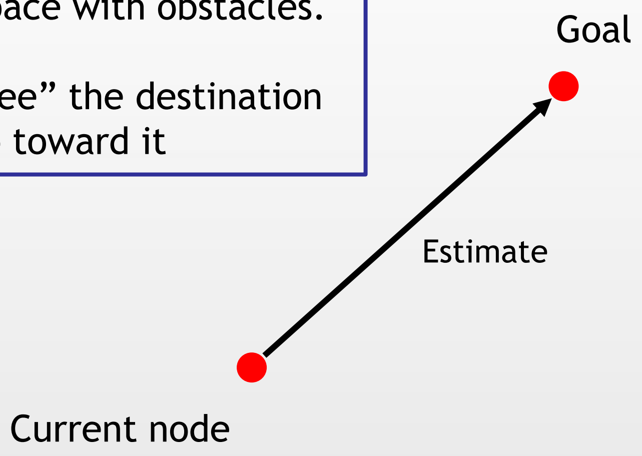
Euclidean Estimation

- The geometric distance to the destination

```
protected float EuclideanEstimator(Node from, Node to) {
    return (from.sceneObject.transform.position - to.sceneObject.transform.position).magnitude;
}
```

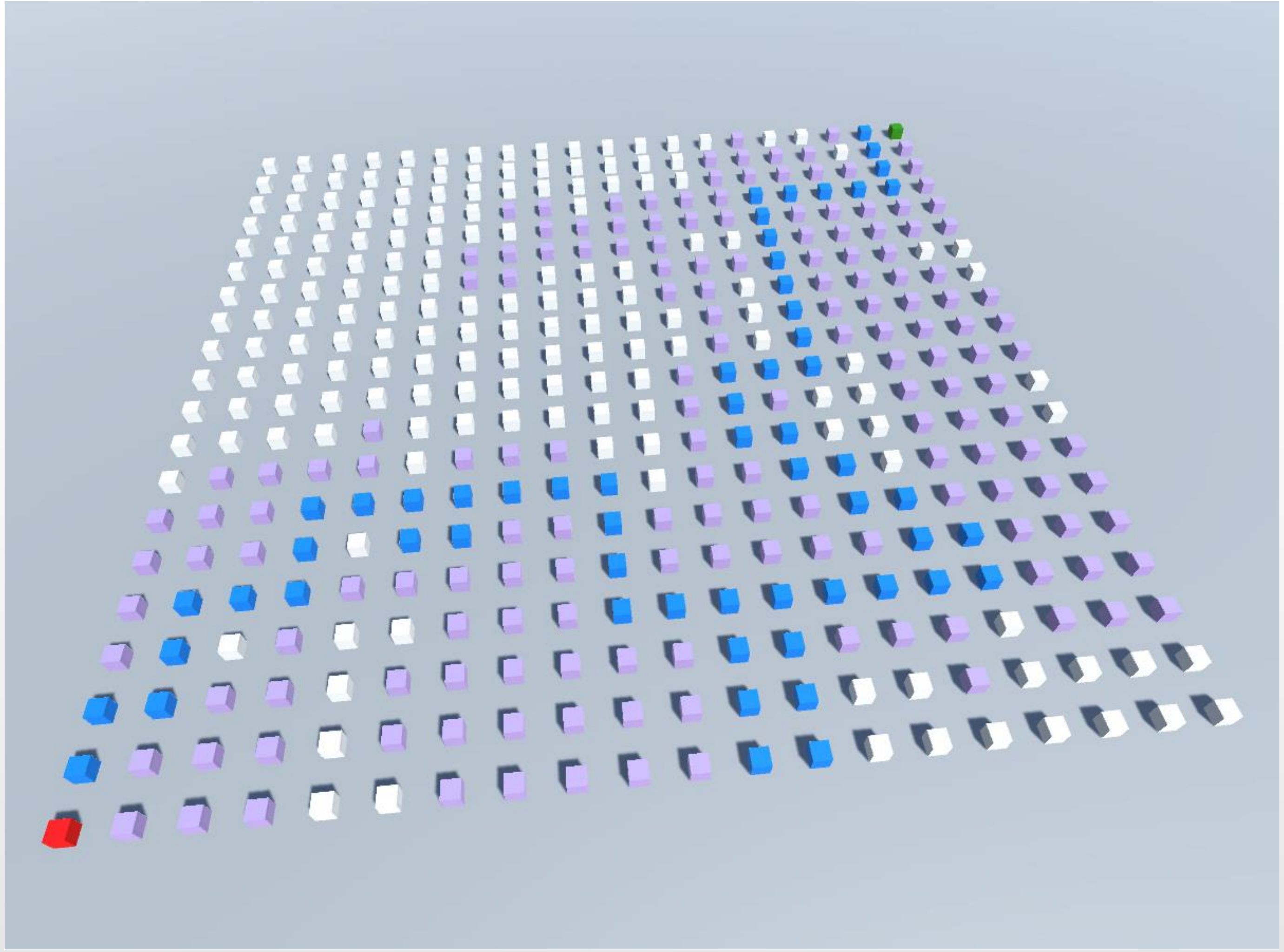
This can me meaningful when moving inside an open space with obstacles.

The agent can “see” the destination and will try to go toward it



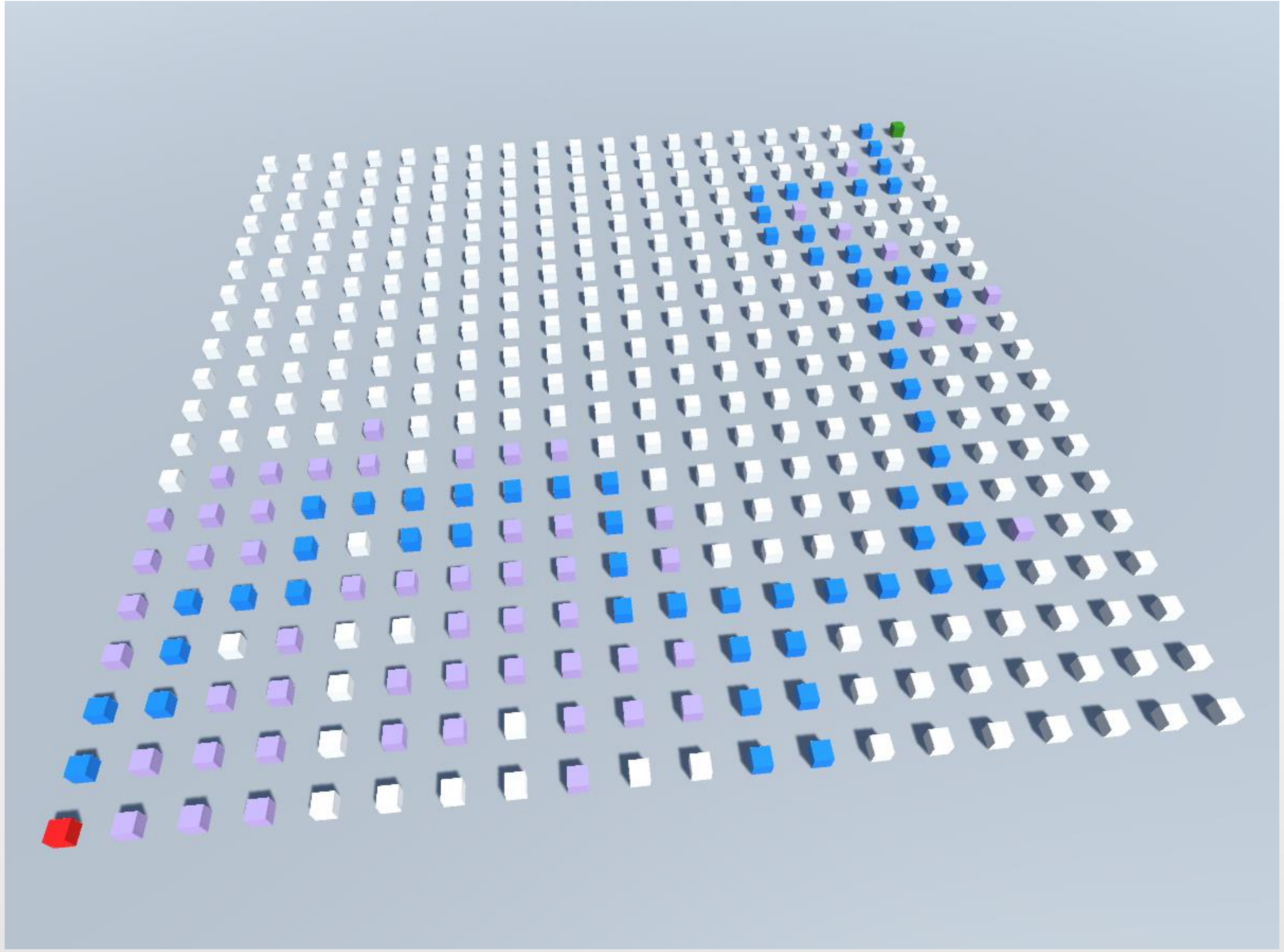
A* with Euclidean Estimation

Stop Based on Distance of Unvisited Nodes



A* with Euclidean Estimation

Stop at first hit



Not only we explore less nodes, but the resulting path is different (less optimal)

UNIVERSITÀ DEGLI STUDI DI MILANO

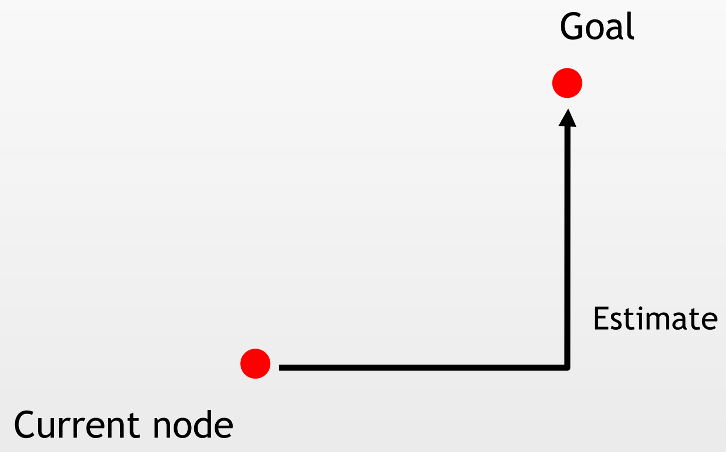
Manhattan Estimation

- It is the distance along the axes

```
protected float ManhattanEstimator(Node from, Node to) {
    return (
            Mathf.Abs(from.sceneObject.transform.position.x - to.sceneObject.transform.position.x) +
            Mathf.Abs(from.sceneObject.transform.position.z - to.sceneObject.transform.position.z)
        );
}
```

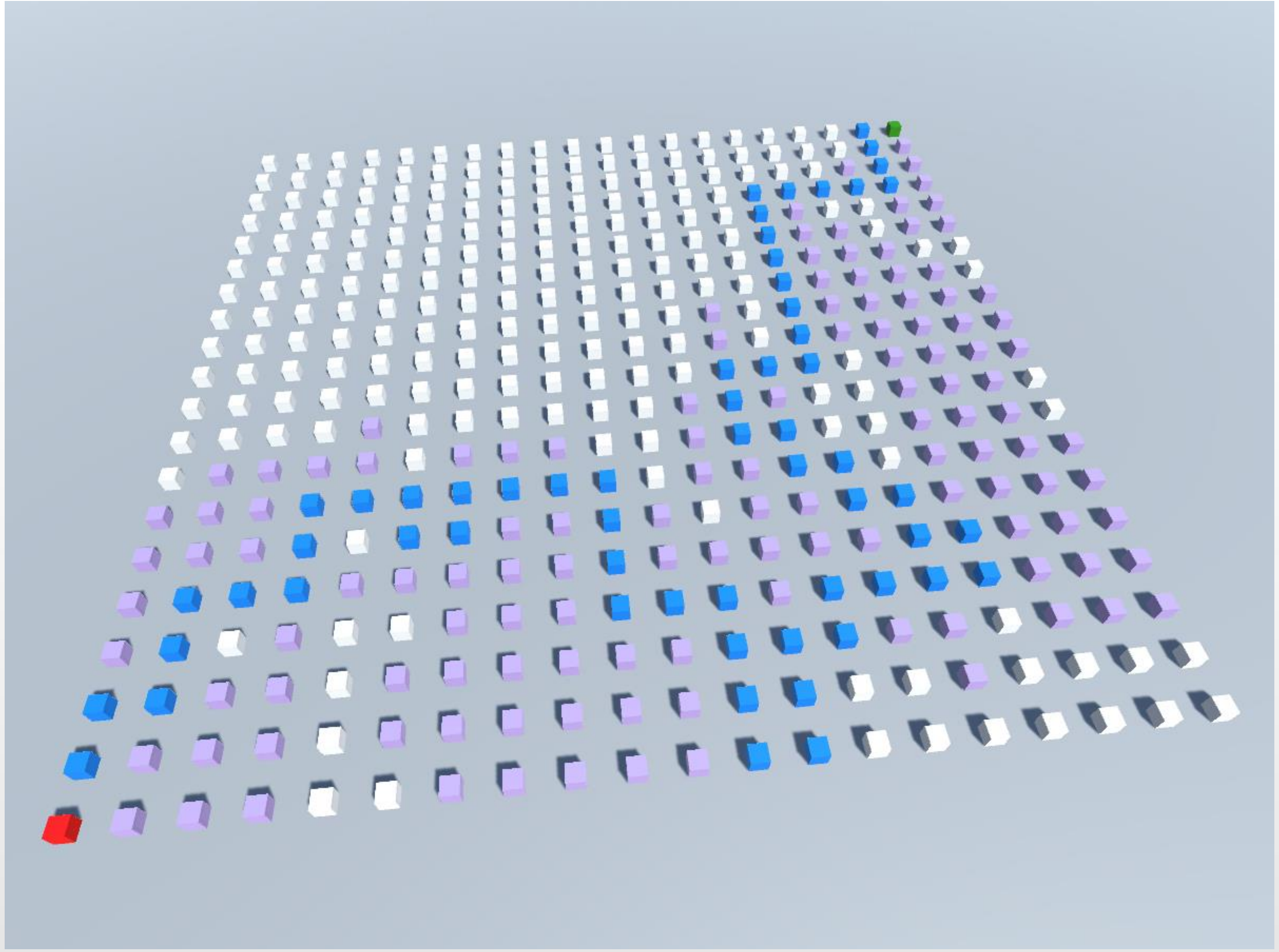
This can me meaningful when driving a car inside a city made of square blocks (hence, the name)

The agent know the general position of the destination but can only move along one axe at a time



A* with Manhattan Estimation

Stop at first hit



In this case, we have a slightly better solution than using the Euclidean estimator, but we visited more nodes

UNIVERSITÀ DEGLI STUDI DI MILANO

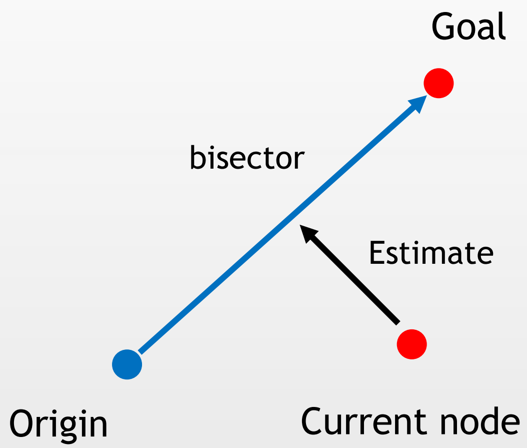
Bisector Estimator

- It's the distance to the line connecting the origin and the goal

```
private float BisectorEstimator(Node from, Node to) {
    Ray r = new Ray (Vector3.zero, to.sceneObject.transform.position);
    return Vector3.Cross(r.direction, from.sceneObject.transform.position - r.origin).magnitude;
}
```

This can be meaningful when there is an established straight road from start to finish and the agent starts from an arbitrary point.

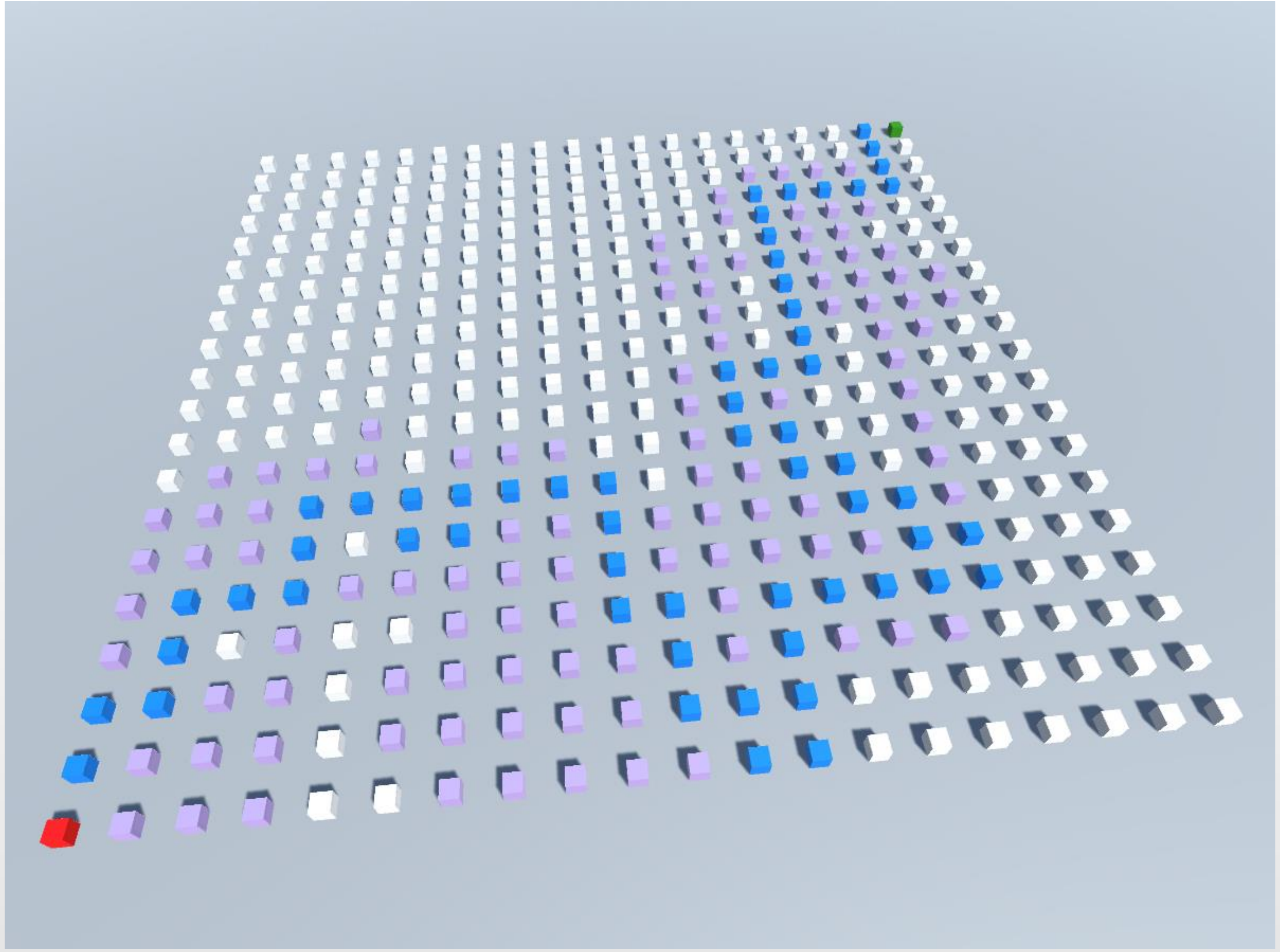
The agent will try first to reach the road and then use it to go to destination



To make the code easier, I am assuming the starting point is always in the origin

A* with Bisector Estimation

Stop at first hit



As we can see, compared to the Manhattan estimation, we push more toward the center of the field

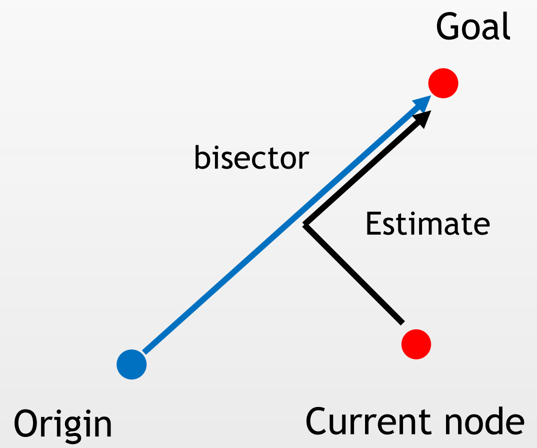
Full Bisector Estimator

- It's the distance to reach the line connecting the origin and the goal and then the goal

```
private float FullBisectorEstimator(Node from, Node to) {
    Ray r = new Ray (Vector3.zero, to.sceneObject.transform.position);
    Vector3 toBisector = Vector3.Cross (r.direction, from.sceneObject.transform.position - r.origin);
    return toBisector.magnitude + (to.sceneObject.transform.position - ( from.sceneObject.transform.position + toBisector  ) ).magnitude ;
}
```

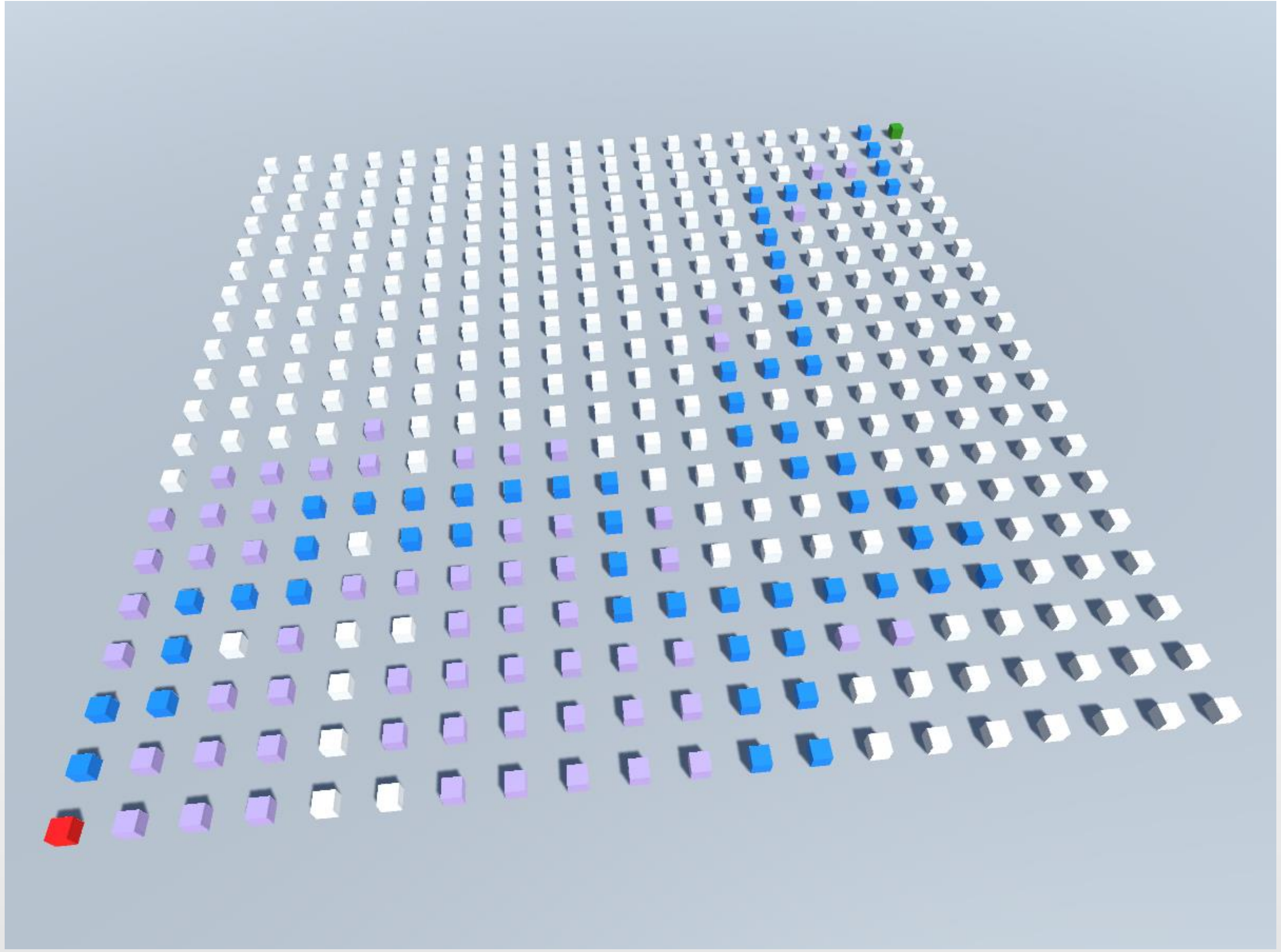
Same as the previous case, but with a more refined model.

You can also look at this as Manhattan distance rotated by 45 degrees



A* with Full Bisector Estimation

Stop at first hit



We get the same result as when using the bisector estimator, but surprisingly, we optimize a lot the exploration.

It looks like a more complex model might be of help (at least In this case)

ZeroEstimator

- Always returns zero

```
protected static float ZeroEstimator (Node from, Node to) { return 0f; }
```

- This is not really useful (as a heuristic) but can be used as a benchmark to compare the performances of other estimators with Dijkstra without changing application

Someone Might Ask …

"How comes when I do not stop at the first hit, I always end up exploring all the graph?"

- That is a problem of our setup:
  1. The nodes are usually well connected
     - Default edge probability is 0.75 and many nodes have 4 edges
  2. The map is small and regular
     - So, it is difficult to have a sensible variation between the estimation (any estimation) and the actual minimum path

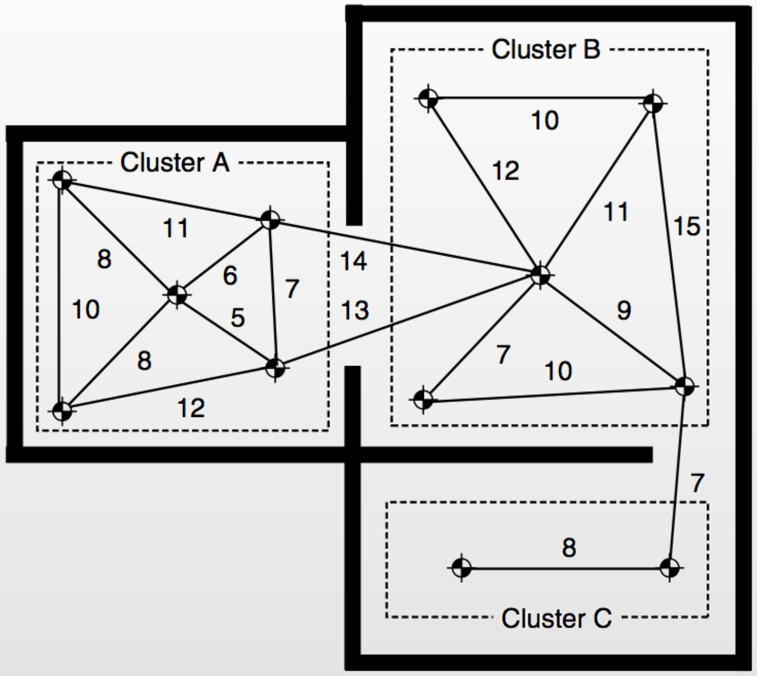
(Optional) Exercise

- Recreate the “Mountain” environment also for A*
  - HINT: you may want to define another heuristic

Clustering Heuristics

- In complex maps heuristics can be difficult to compute
  - A possible alternative is to divide nodes in groups (clusters) and create a table reporting the proximity between each group (lookup tables)
  - This approach is similar to the BGP protocol used in networking for inter-AS routing

- In a graph, we usually cluster nodes belonging to the same well-connected region



	A	B	C
A	x	13	29
B	13	x	7
C	29	7	x

Lookup table

Clustering Techniques

- Two ways to define clusters:
  - By Hand
    - The old school
  - Using graph algorithms
    - Easier, but must be supervised
      - I.e., it will create an output that an operator must check
- Then, we compute the lookup table
  - A table indicating the minimum distance between each pair of clusters
  - This activity is usually performed offline

Using Clusters

- Clusters are used as part of a heuristic strategy
  - Same cluster → Use Euclidean distance
  - Different clusters → Use the Lookup table
- NOTE: they are both underestimations!
- Keeping small clusters will dramatically improve performances, especially in indoor settings
  - See, e.g., the example in the textbook
- NOTE: this is NOT the same as hierarchical pathfinding!

References

- On the textbook
  - § 4.3.1
  - § 4.3.2
  - § 4.3.5
  - § 4.3.8