# Moving Intelligently (2): Pathfinding

## CE811 Game Artificial Intelligence

Dr Michael Fairbank

1st November 2021

University of Essex

# Outline

- Graph Algorithms
  - Dijkstra's Algorithm
  - A* Algorithm
- Path Following
- World Representation
  - Tile Graphs
  - Dirichlet Domains
  - Points of Visibility
  - Navigation Meshes
- Hierarchical Pathfinding
- Wrap-up

# Getting about using Pathfinding

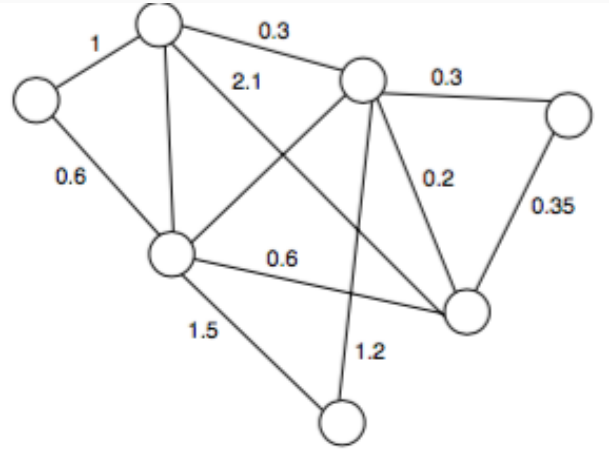
- Pathfinding is one of the most fundamental parts of game AI
- Pathfinding can’t work directly on complex geometry that is used to make up the level (ie, collision meshes)
- Instead, we need to build a *Graph* that describes the level

# Graph Algorithms

# Pathfinding in Graphs

We need to construct a **graph** that we can search:

- Nodes represent points in 2D or 3D space
- Nodes have neighbours (adjacent nodes)
- Neighbouring nodes may have different (positive) distances / costs
- Nodes (i.e., their locations) can be constructed manually or automatically



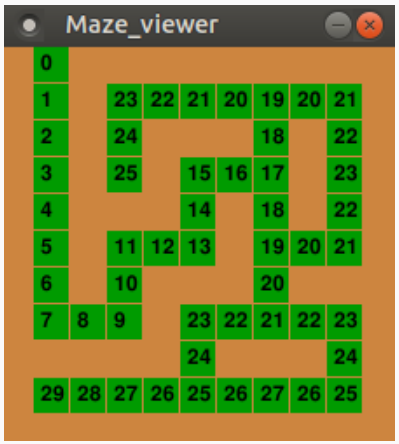
# Outline

- Graph Algorithms
  - Dijkstra's Algorithm
  - A* Algorithm
- Path Following
- World Representation
  - Tile Graphs
  - Dirichlet Domains
  - Points of Visibility
  - Navigation Meshes
- Hierarchical Pathfinding
- Wrap-up

# Dijkstra

Dijkstra's algorithm is a classical pathfinding algorithm that is very efficient (polynomial runtime). It finds the shortest path from a node to any other node in the network. It is a variation of breadth first search.

The algorithm assigns a "gScore" value to each node in the graph. The gScore of node *n* is the shortest distance of *n* to the start node.




- In this simple maze, green squares denote graph "nodes".
- Adjacent green squares have a graph "edge" connecting them of length=1
- 0 is the start node.
- The numbers denote the gScores.

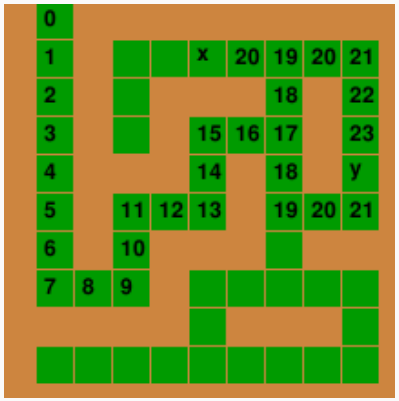
Instead of recalculating each gScore afresh (i.e. by measuring the full path length to the start node), Dijkstra uses

$$gScore = min(\text{neighbour's gScore} + \text{distance to neighbour})$$

over all immediate neighbours. This **reuse of neighbouring numbers** it what makes the algorithm **fast**.

# Dijkstra

- Remember, the gScores represent the total distance of the node to the start node.
- They are calculated by looking at the value of their neighbours.
- Q: What gScore should node *x* have?




- Q2: Consider Node *y*. Does it get its gScore by looking at the node above it or the node below it? Why?

# Dijkstra Pseudocode i

```
function Dijkstra(startNode, goalNode, h)
    openSet := [startNode] # list of nodes currently being considered
    closedSet := [] # list of nodes that are finalised
    parentNode := {} # Records chain of which node comes before in pathway
    gScore := {} # cost of the cheapest path from start to n currently known.
    gScore[startNode] := 0

    while openSet is not empty
        # This operation can occur in O(1) time if openSet is a priority queue
        currentNode := the node in openSet having the lowest gScore[] value
        if currentNode = goalNode
            # Success!  Best path found
            return reconstruct_path(parentNode, currentNode)

        openSet.Remove(currentNode)
        closedSet.Add(currentNode)
        for each neighbour of currentNode which is not in closedSet:
            # d(current,neighbour) is the weight of the edge from current to neighbour
            # tentative_gScore is the distance from start to the neighbour through current
            tentative_gScore := gScore[currentNode] + d(currentNode, neighbour)
            if neighbour not in openSet or tentative_gScore < gScore[neighbour]
                # This path to neighbour is better than any previous one. Record it!
                parentNode[neighbour] := currentNode
                gScore[neighbour] := tentative_gScore
                if neighbour not in openSet
                    openSet.add(neighbour)
    # Open set is empty but goal was never reached
    return failure
```

# Dijkstra

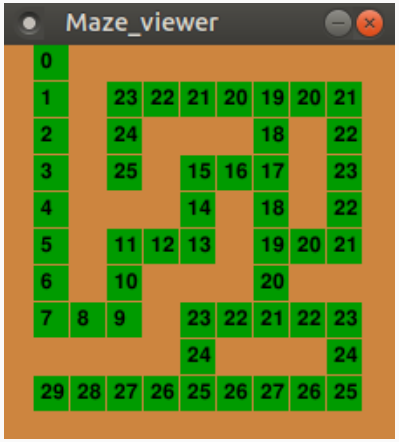
Dijkstra's algorithms makes use of a **priority queue**: a list that holds items with an associated priority. The list if always ordered such that the highest priority items are on top. The algorithm's efficiency depends crucially on the implementation of the priority queue used.

Both Dijkstra and A* search a graph $G = (N, E)$. We thus need to represent our world as a weighted undirected graph.

# Reconstructing the Route

- Once all the gScores are found, it's possible to reconstruct a route from the start node to *any* node in the graph
- Q: If the "end node" is the bottom-right corner node (labelled "25"), which way must the optimal path have come from to get to that node?

- From the north, or from the west?
- How do we know?




- A: It can't be from the west, because $26 + 1 \neq 25$.
- The "parent" node of the 25 node is the 24 node to its north.

# Reconstructing the Route

The route can be extracted easily by following the chain of parents from the ”end Node” to the start.

```
function reconstruct_path(parentNodes, current)
    total_path=[]
    while current!=None
        total_path=current+total_path
        current=parentNodes[current]
    return total_path
}
```

Notice how the loop above follows the chain of parent nodes backwards, adding them in reverse order to the result path.

# Further tutorials on Dijkstra

A demonstration by YouTube's PatrickJMT (Very good and clear):

- Dijkstra's Algorithm : A Quick Intro on How it Works. (8 mins).

Also interesting, Computerphile channel

- Dijkstra's Algorithm - Computerphile. (11 mins).
- This video gives very nice intro and motivation. Nice demonstration of the openSet and closedSets being filled. But not as clear explanation of the algorithm itself.[1]

[1] Plus it has a bug in description of algorithm's termination condition

# Outline

**Dijkstra** is complete and always finds the shortest path. Also, it is efficient. However, we can often do better.

**Question**
How could we improve Dijkstra?

- Dijkstra uses the distance travelled so far (”gScore”) and utilises a **priority queue** to consider nodes in the **openSet** with the **smallest gScore**.
- But that means it doesn’t know how far away the goal is?
- What happens if we give it a ‘best guess’ of how far away it is?

# A*

- A* is the same as Dijkstra, but its priory queue sorts the openSet of nodes in the order of smallest **fScore**, where

$$fScore = gScore + h(node, endNode)$$

- Here "$h(A, B)$" is a heuristic function which estimates the shortest path distance from $A$ to $B$
- "EndNode" is the final destination point our pathfinding algorithm is aiming for.

# A* pseudocode i

The only lines that have changed (compared to Dijkstra's code) are in red.

```
# A* finds a path from start to goal. h is the heuristic function.
# h(n) estimates the cost to reach goal from node n.
function A_Star(startNode, goalNode, h)
    openSet := [startNode] # list of nodes currently being considered
    closedSet := [] # list of nodes that are finalised
    parentNode := {} # Records chain of which node comes before in pathway
    gScore := {} # cost of the cheapest path from start to n currently known.
    gScore[startNode] := 0

    # For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our
    # current best guess how short a path from start to finish can be if, it
    # goes through n.
    fScore := {}
    fScore[startNode] := h(startNode, endNode)

    while openSet is not empty
        # This operation can occur in O(1) time if openSet is a priority queue
        currentNode := the node in openSet having lowest fScore
        if currentNode = goalNode
            # Success! Best path found
            return reconstruct_path(parentNode, currentNode)

        openSet.Remove(currentNode)
        closedSet.Add(currentNode)
```

# A* pseudocode ii

```
    for each neighbour of currentNode which is not in closedSet:
        # d(current,neighbour) is the weight of the edge from currentNode
        # to neighbour
        tentative_gScore := gScore[currentNode] + d(currentNode, neighbour)
        if neighbour not in openSet or tentative_gScore < gScore[neighbour]
            # This path to neighbour is better than any previous one.
            parentNode[neighbour] := currentNode
            gScore[neighbour] := tentative_gScore
            fScore[neighbour] := gScore[neighbour] + h(neighbour, endNode)
            if neighbour not in openSet
                openSet.add(neighbour)

# Open set is empty but goal was never reached
return failure
```

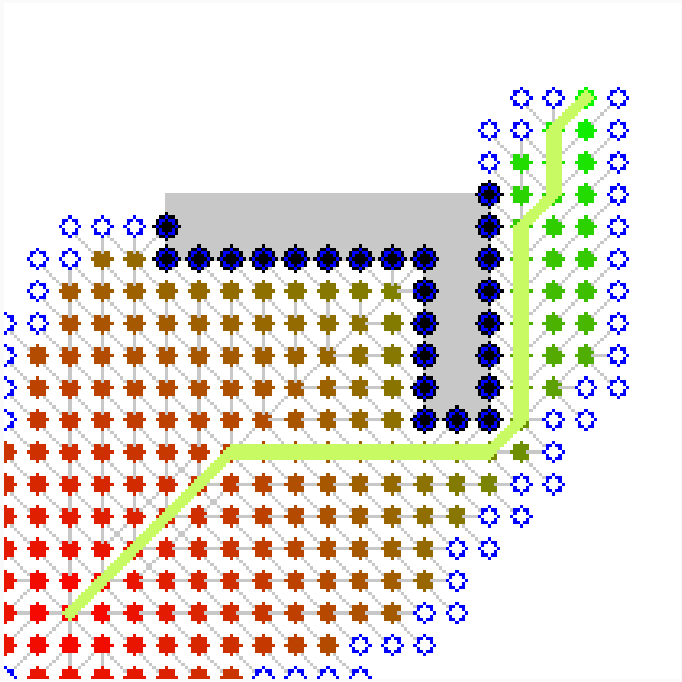
# Heuristics

- The A* *heuristic* function $h(node, endNode)$ must estimate the *distance from node to the end node*.
- If we have the (x,y) coordinates of the current location, then we could just define the *h* function to give the distance *as the crow flies*[2] to the end node.
- As long as our estimate *h* is not an over-estimate, then A* is guaranteed to find the optimal path.
- Q: Can the ”distance as the crow-flies” ever be an over-estimate?

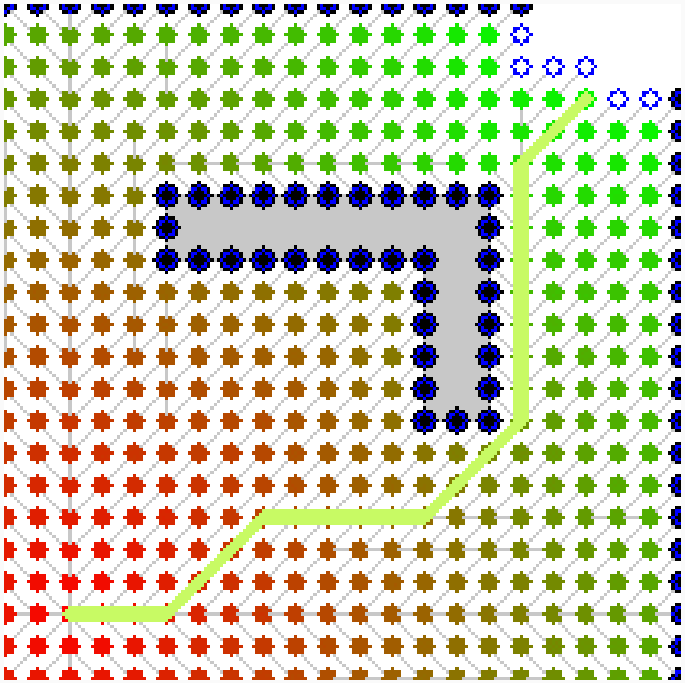
[2] ”As the crow flies” means ”in a straight-line path”, or equivalently the ”Euclidean distance from (x,y) to the end node”.

# A*: the difference

A Star:



Dijkstra:



A* only explores nodes which look promising - allowing us to limit our search.

# Heuristics

- A heuristic which never over-estimates the distance from *A* to *B* is called **admissible**.
- So for A* to always give an optimal path, the heuristic used must be **admissible**.
- The closer the heuristic to the actual distance, the better A* performs.

# Heuristic vs Real Value

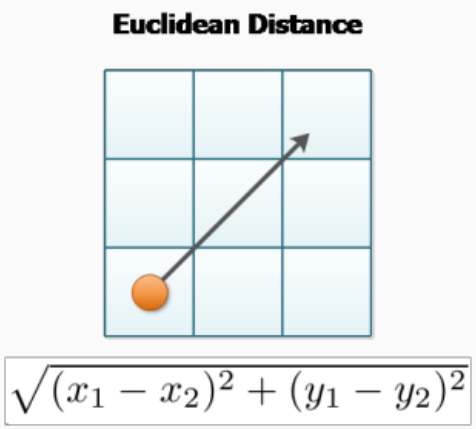
The more accurate the heuristic, the less nodes A* will visit, and the faster A* will run. If the heuristic is **perfect** (provides an exact estimate to the final goal), A* will only explore the correct answer.

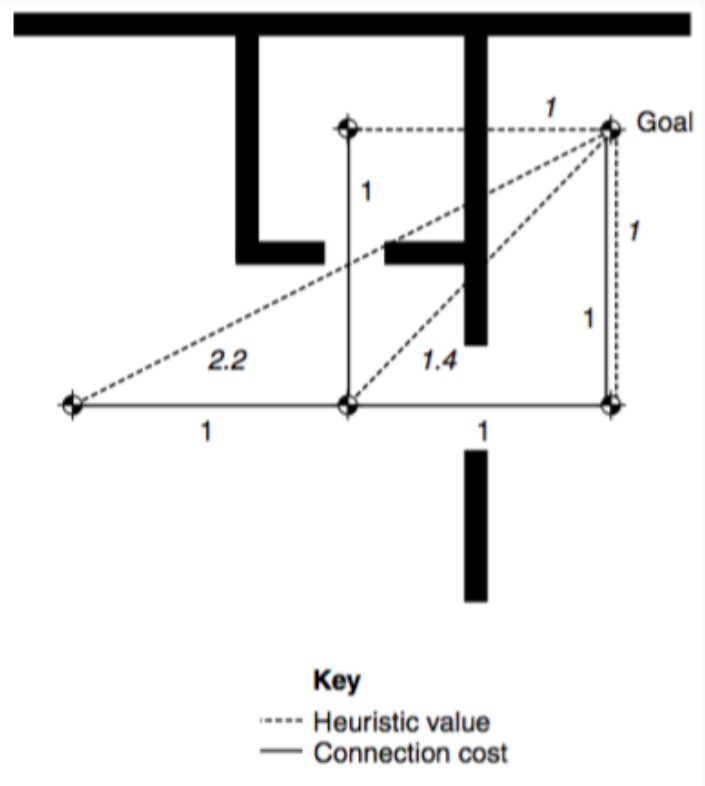
An **Underestimating heuristic** makes A* slower. The algorithm will explore more nodes closer to the explored part than closer to the goal (the heuristic value is smaller than reality), increasing the time needed to find the route.

An **Overestimating heuristic** may make A* **not** return the best path. The algorithm will try to select nodes that are closer to the goal (the heuristic value is higher than reality), leaving some portions of the graph unexplored. The more the heuristic overestimates the distance to the real target, the worse the path returned will be.

# Euclidean Distance

Euclidean distance is the distance between two points without considering any walls and obstructions. It's defined by the Pythagoras' theorem:




The Euclidean distance is always either (perfectly) accurate (no obstacles at all) or an underestimate (with obstacles). Therefore, A* with an Euclidean distance will **always** find the shortest path.

In environments with multiple obstacles (i.e. walls, doors), the performance will be poorer.

# Heuristic Properties

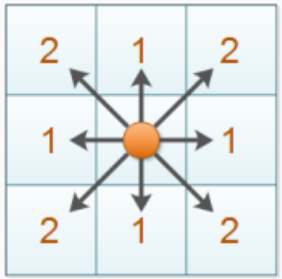
**Question**
As well as admissibility, what else do we need to consider?

**Speed** - we want an accurate but *fast* Heuristic.

# Other Heuristics

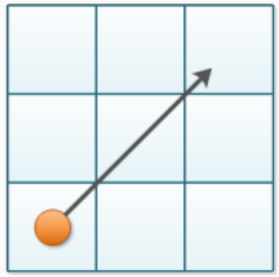
Some types of heuristic:

**Manhattan Distance**



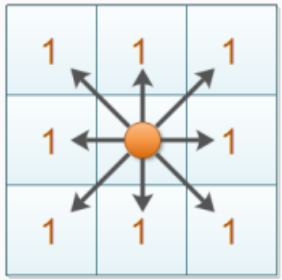

$$|x_1 - x_2| + |y_1 - y_2|$$

**Euclidean Distance**



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

**Chebyshev Distance**




$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

(lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/)

- Rectangular grid, 4-way connectivity: Euclidean or Manhattan.
- Rectangular grid, 8-way connectivity: Euclidean or Chebyshev (Diagonal).
- In some cases we can also use a pre-computed exact heuristic.

# A* heuristic functions

Our helper functions / heuristics are defined as follows:

```
function EuclideanDistance(from, to)
  return Sqrt(Pow((from.x - to.x), 2) + Pow((from.y - to.y), 2))

function ManhattanDistance(from, to)
  return Abs(from.x - to.x) + Abs(from.y - to.y)
```

Here you can add additional heuristics including those for other grids etc.

# Caching Heuristics

Even though the above heuristics are fast to calculate, they cannot calculate the distance round corners and walls.

It is sometimes possible to get round this limitation by pre-calculating true distances between pairs of nodes of the map.

If level is static (and pathfinding network is small), can use a lookup table of precomputed shortest distances:

> From A to E: lookup A-E, find node C, lookup C-E, find node K ...

Table can be generated using **Dijkstra's algorithm**.

Obivously, memory will be a problem for any moderate to large sized map.

# More Facts about A*

- If $h(n)$ is 0, only $g(n)$ matters, and A* equals Dijkstra's algorithm.
- If $h(n)$ is exactly equal to the actual distances, then A* will only follow the shortest path and never expand anything else.
- If $h(n)$ is very high relative to $g(n)$, then only $h(n)$ plays a role, and A* turns into Best-First-Search.

# Further tutorials on A*

Computerphile channel

- A* (A Star) Search Algorithm - Computerphile. (14 mins).
- This video shows A* in action and gives motivations for the heuristic function.[3]

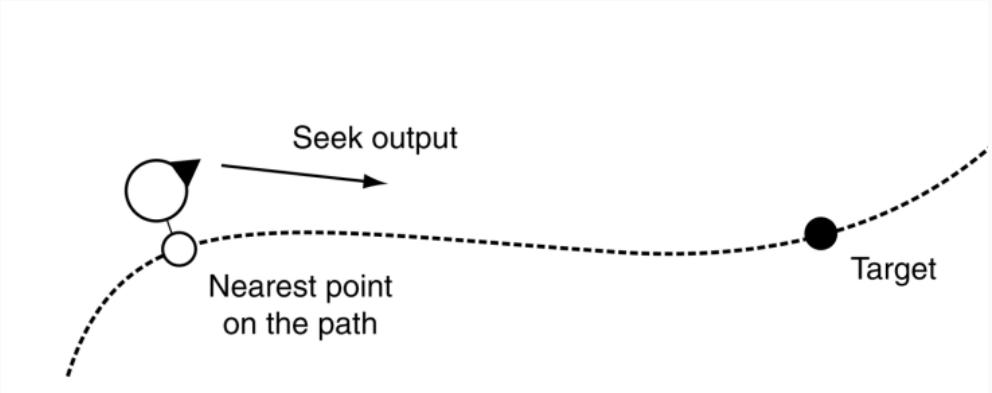
[3] Plus he explains the bug in his previous Dijkstra video!

# Path Following

# Path Following: basic idea

Once an optimal path from *A* to *B* has been found, the agent must follow it.

- This could be implemented by making successive seek(node) calls, going from node to node along the path:



# Path Following: possible problems

But with path following, there could be complications:

- Other agents or moving objects might occupy nodes in the graph
- The level might be fully dynamic
- Fog of War

# Path following: possible solutions

Numerous solutions:

- Re-compute (parts of) the path
- Use large nodes that can be occupied by multiple agents
- Divert to closest available node
- Step through or side-step
- Plan all paths centrally, block nodes before they are occupied
- Minimise crossings of agents (use shortest distance)
- Also need to bear in mind visual perception:
  - Need to turn into the right direction before moving
  - Movement should look natural and proportional to distance

# World Representation

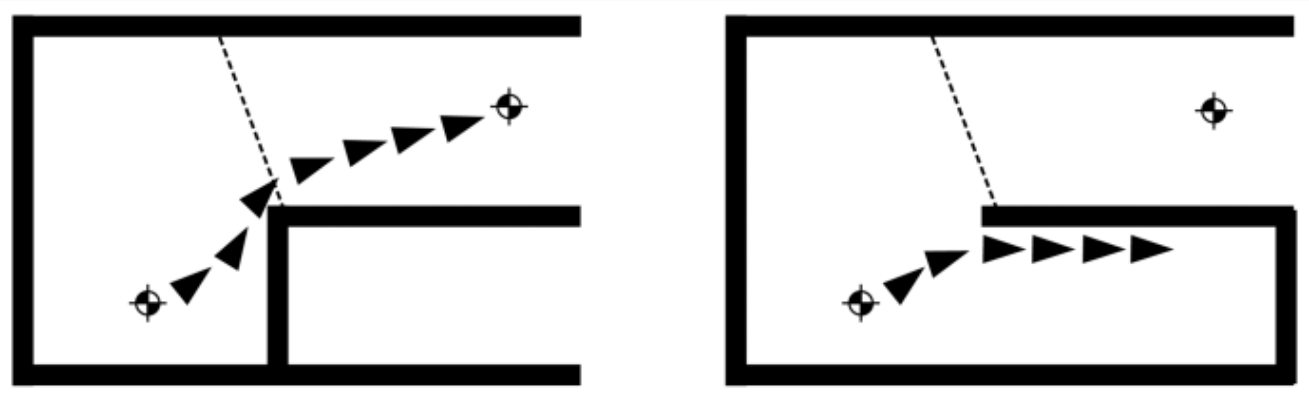
# World Representation

There must be a **division scheme** to convert the open areas of the level/map where the action takes place, intto a graph (nodes, connections and weights) that we use to do pathfinding. There are several ways this can be done:

1. Tile Graphs
2. Dirichlet Domains
3. Points of Visibility
4. Navigational Meshes

# World Representation

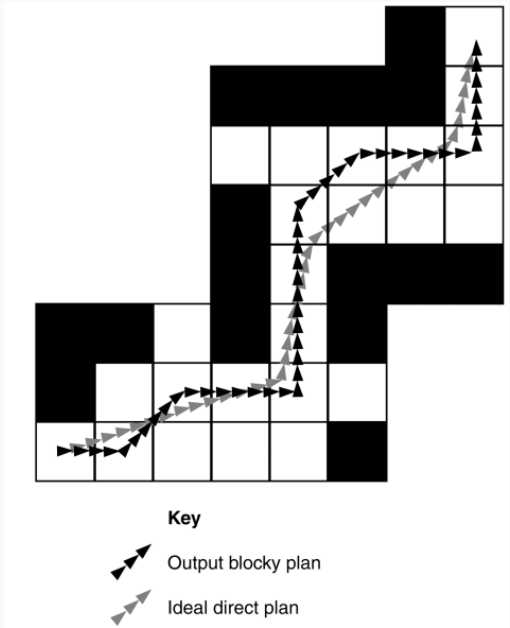
It is important to get the division scheme for the world right, otherwise it can lead to problems to character movement:



# Outline

# Tile Graphs

The graph structure is a 2D grid, like in a top-down view. The nodes in the graph represent tiles in the real world:




Easy to generate:

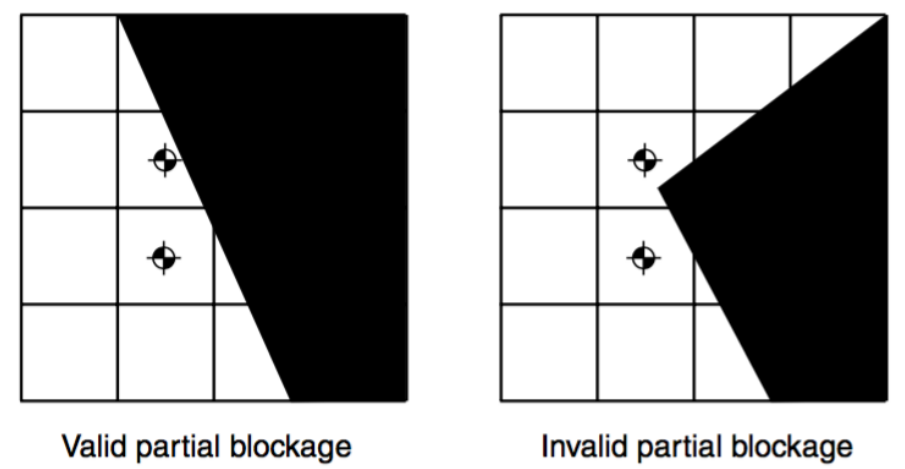
- $tileX = floor(x / tileSize)$
- $tileY = floor(y / tileSize)$

Can lead to Blocky and irregular movement[4]

[4](this happens in all schemes, but it's especially noticeable in this one).

# Tile Graphs: Validity

Other drawbacks:




- Not every point in one tile can automatically connect to every point in another (as shown on left).
- Usually too many tiles in the level (A* has to work hard).

Tile Graphs are easy and convenient, but there are usually far too many tiles in a level. This makes A* searches long to find sensible paths.

# Tile Graphs

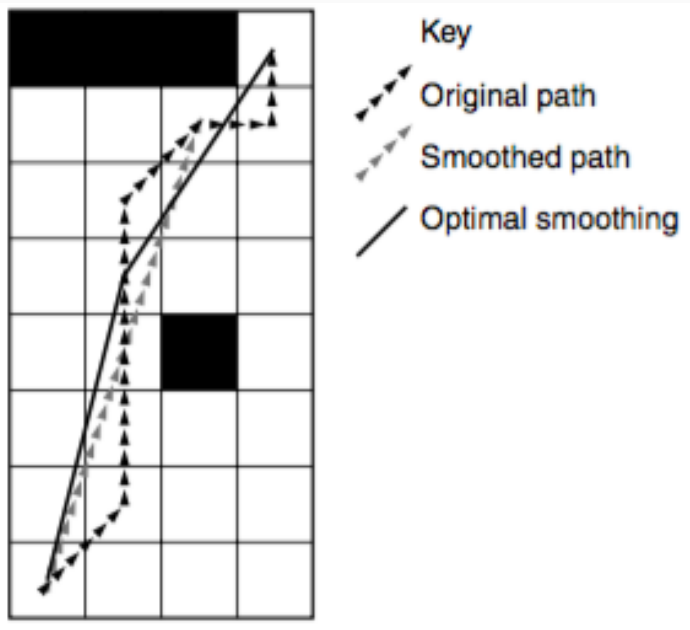
Once we've spilt our world into a rectangular grid of nodes, how should those nodes be connected?

Could be 4-way (up, down, left and right cells are adjacent) or 8-way (includes diagonals).

The euclidean heuristic is always feasible (with A*) for these two types of connectivities.

# Path Smoothing

A path that travels from node to node through a graph can appear erratic. In some cases, the world representation produces very unnatural paths:




Smoothing a path can be achieved in different ways (by means of ray-tracing):

- Offline: Iterate through the calculated path finding visibility points. - It increases path finding time, but it provides a stable, re-usable solution.
- Online: At each step, calculate which is the furthest point in the path visible, and chase it - It takes more time during navigation, but it’s reactive to obstacles.

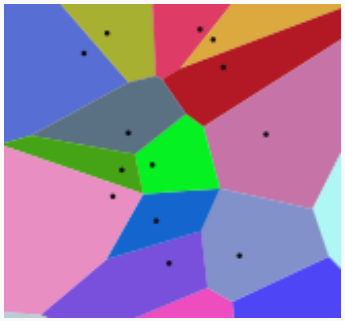
# Outline

# Dirichlet Domains

In this method of quantising the game world into a graph, we create some carefully positioned **characteristic points** within the game world. These become the nodes of our graph. Then, each point in the world gets grouped into the region which encloses its closest characteristic point.

This means areas of the world are partitioned into Dirichlet Domains:

A **Dirichlet** domain, also called a *Voronoi polygon in two dimensions*, is a region around one of a finite set of source points (*characteristic points*) whose interior consists of everywhere that is closer to that source point than any other.



# Dirichlet Domains

Connections are placed between the borders of the domains. These connections can be found by computing the *Delaunay* triangulation, although it's possible to establish them manually as a part of the level design.

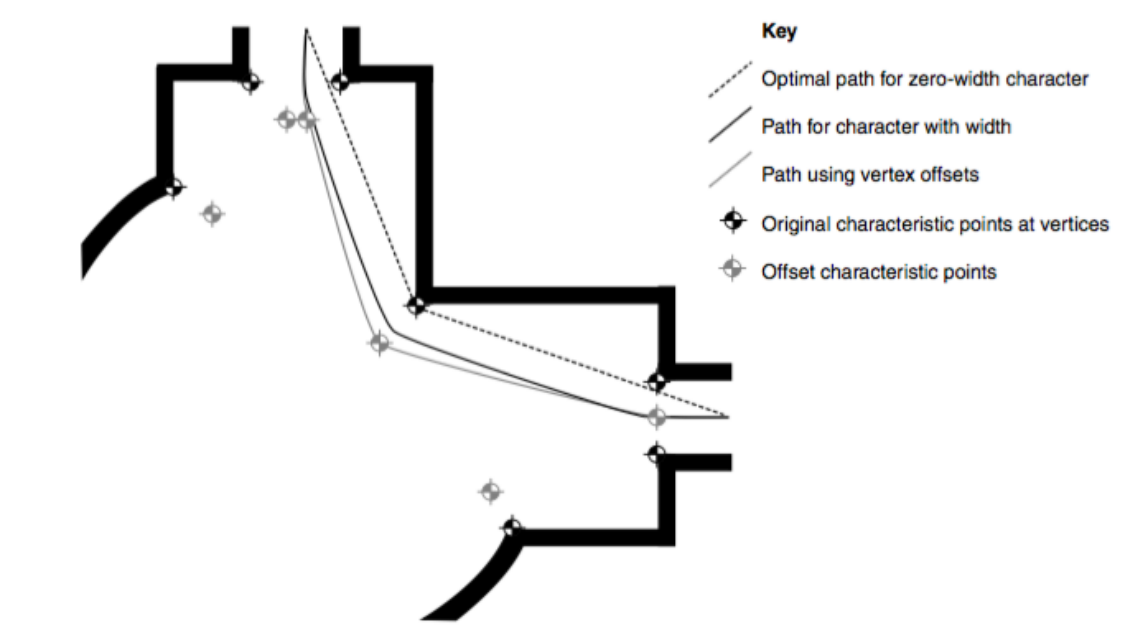
- Advantages:
  - Easy to program and change.
  - Possible to be generated automatically.
- Drawbacks:
  - Strong dependence between location of the characteristic points and obstacles.

# Outline

# Points of Visibility

Idea: use the convex vertices of the environment (inflection points) as characteristic points for navigation.

Since all interesting routes will pass through these points, these points become the nodes of our path-finding graph!

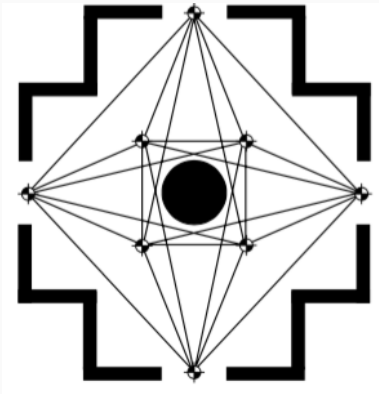



Different radius of the agents will give place to different locations for these characteristic points. (Larger characters will need to give more corner clearance, to avoid bumping)

# Points of Visibility

Once the points of visibility are found, they become the characteristic points of Dirichlet domains

To work out how the points are connected, rays are cast between them and a connection is made if the ray doesn't collide with any other geometry:



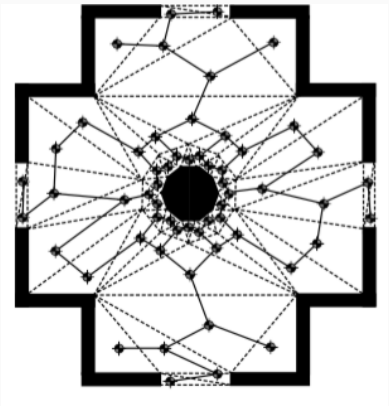
- Advantages: Paths that follow these points are particularly believable.
- Drawbacks: Can get huge number of inflection points.

# Outline

# Navigation Meshes

A **Navigation Mesh** (or **NavMesh**) is a data structure which is used to model the traversable areas of a virtual map.

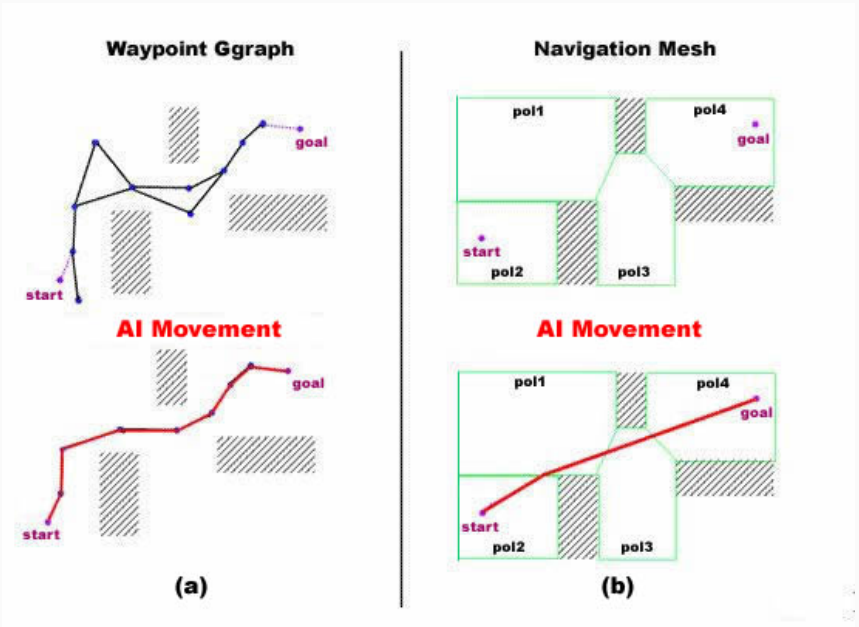
- It is a collection of two-dimensional convex polygons (usually triangles) that define which areas of a virtual map are traversable by agents.
- Each polygon acts as a single node that links to other adjacent nodes.
- Once a path is found through the nodes, the actual walking path can be **smoothed** by a subsequent algorithm.
- NavMesh is the most popular way of representing the world for pathfinding.



# Navigation Meshes

The polygons in the navmesh are **convex**: This means after the path through the graph of nodes is found, further path smoothing can be applied...

From docs.unrealengine.com: *"With a navigation mesh you have a path representing a series of polygons you need to walk through in order to reach your goal, but you also know exactly what the walk-able space looks like along the way. Instead of having to hit each point exactly along a waypoint-graph generated path an AI now has all the information associated with the interface between nodes of the navigation mesh. This allows for accurate and practically free cutting of corners, and in general much more natural looking movement."*



# Navigation Meshes

- Advantages:
  - Creates shorter and more natural paths than traditional grid and waypoint-based pathfinding systems.
  - Allows pathfinding algorithms to ignore static and dynamic obstacles.
  - A wide variety of pathfinding algorithms can be modified and optimized for using navigation meshes.
- Drawbacks:
  - Not as efficient as grid and waypoint-based pathfinding systems.
  - Procedural generation of a NavMesh can yield sub-optimal solutions.
  - Their manual creation can be time-consuming and yield flawed solutions.

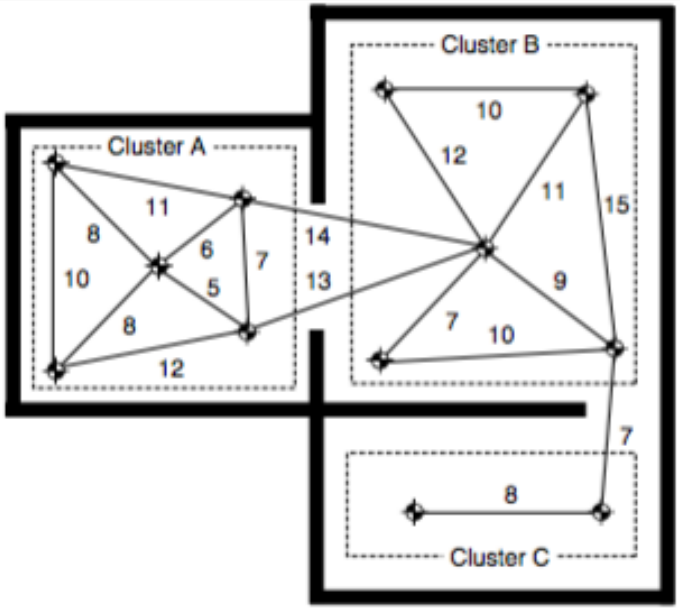
# Hierarchical Pathfinding

# Hierarchical Pathfinding

- Both Dijkstra and A*-search become **too slow** if there are too many nodes in the map.
- **Hierarchical maps** can be used which vastly improve the efficiency of path finding.

# Cluster Heuristic

One simple **Hierarchical Pathfinding** method is to cluster groups of nodes together into **clusters**. Each cluster groups nodes in a region of the space that are highly interconnected. Clustering can be done manual or automatically.




| | A | B | C |
|---|---|---|---|
| A | x | 13 | 29 |
| B | 13 | x | 7 |
| C | 29 | 7 | x |

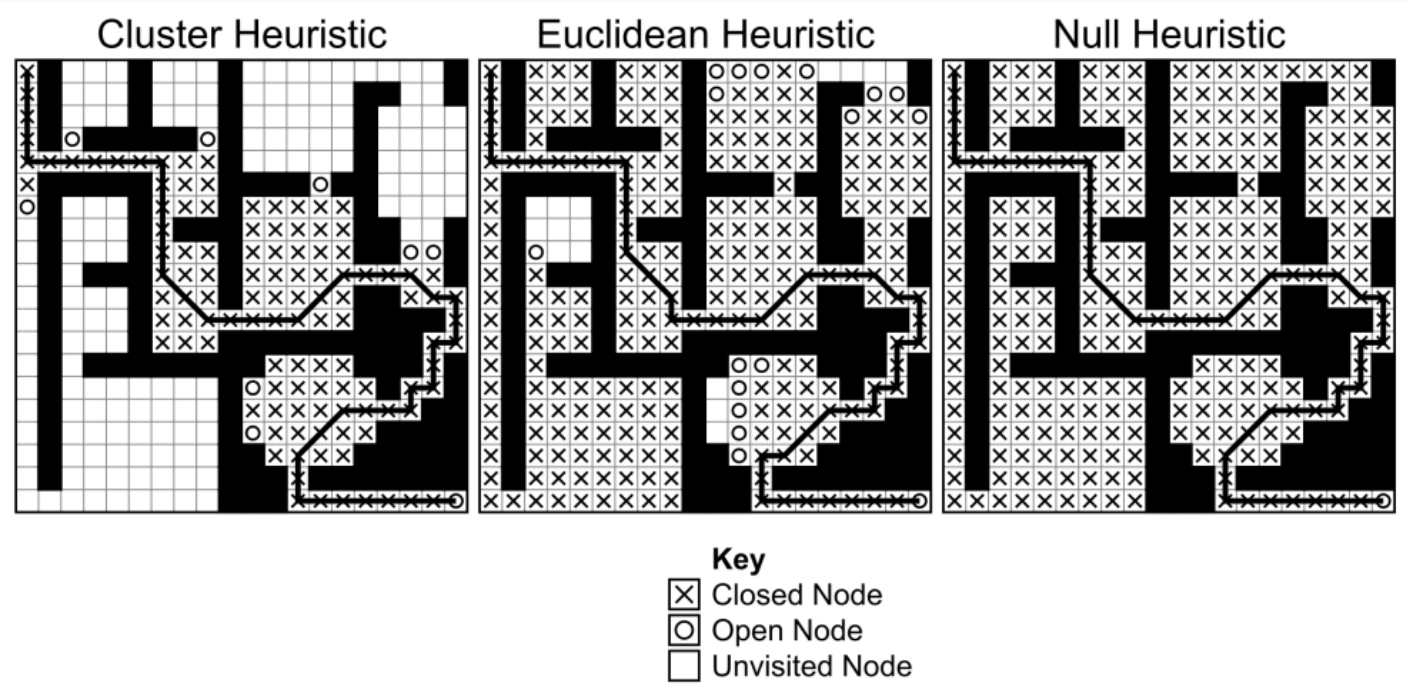
Lookup table

A lookup table (precalculated offline) gives the smallest path length between each pair of clusters. The cluster heuristic often dramatically improves pathfinding performance in indoor areas over Euclidean distance.

# Cluster Heuristic

The cluster heuristic can vastly improve the solution time compared to Euclidean heuristic or Null heuristic (Dijkstra):

The *h*() function scores all nodes in one cluster as the same.



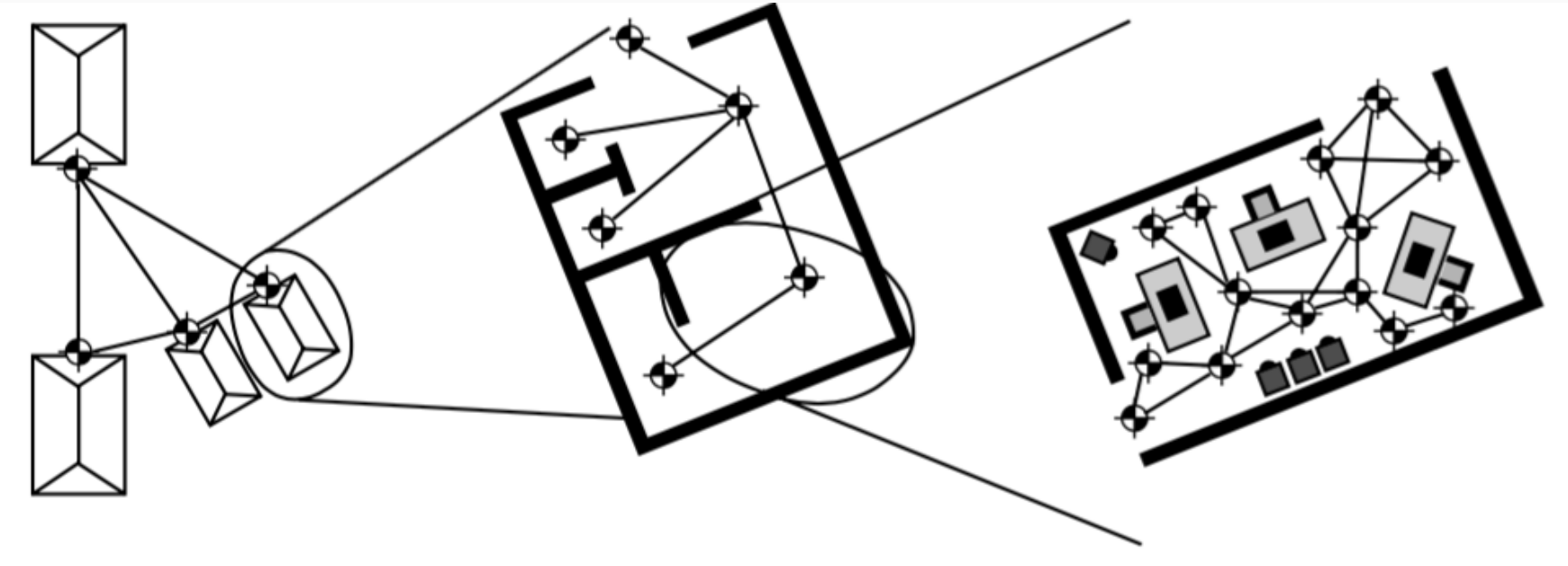

Notice how even with the cluster heuristic, A* will explore most of the nodes in the cluster, before moving ahead to the next one.

If cluster sizes are too large, then there will be marginal performance gain.

# Hierarchical Pathfinding

In more general **Hierarchical Pathfinding** methods: The overall path to follow is a *high level plan* divided into several phases. Each part of this plan is solved by finding a route that completes it.

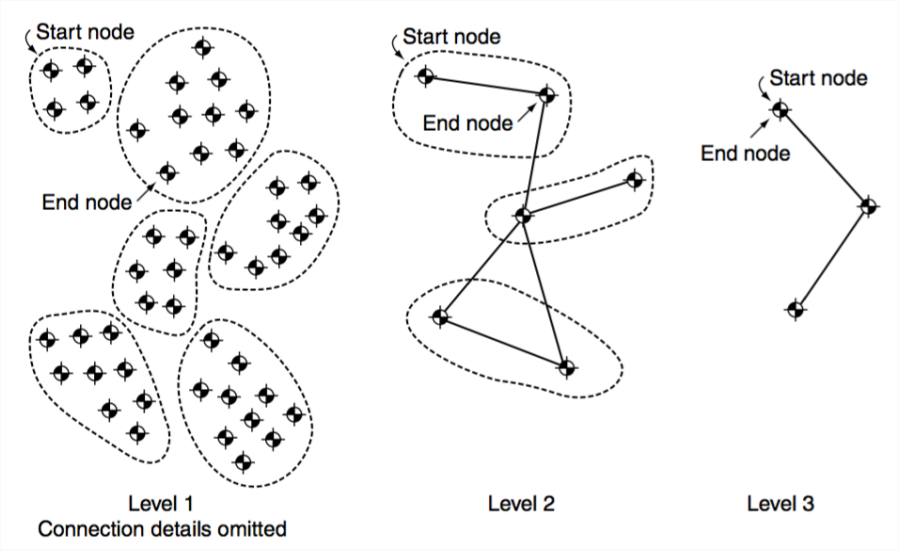
In order to achieve this kind of path planning, our data structure should be adapted accordingly:



# Hierarchical Pathfinding Structure

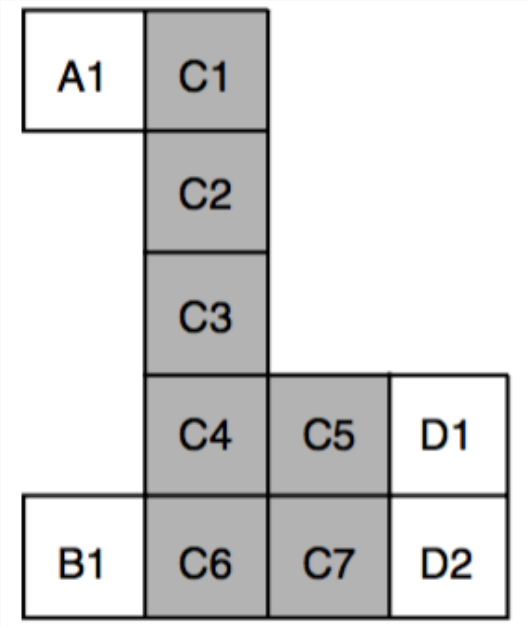
We can still use A* to plan a path at all levels in our hierarchy.

- Nodes:
  - Group closer locations in a cluster of positions.
  - This group can be treated as a single node in the pathfinder.
  - This process can be repeated as many times as needed.
  - There needs to be a mapping of which lower-level nodes belong to the immediate upper-level node.
- Connections: If a lower-level node from one cluster is connected to a node in a different cluster, then there must be a connection between the clusters at the upper level.



# Hierarchical Pathfinding Connection Costs

- Connection Costs:
  - The cost between two groups should be specified manually, or calculating the cost through the lower-level nodes.
  - This is not easy to calculate:




It's not the same cost between groups $C \rightarrow D$ if entering $C$ from $A$ or from $B$.

# Hierarchical Pathfinding Connection Costs

There are different ways to define the heuristics which calculate these connection costs between groups:[5]

- *Minimum Distance:* The cost of moving between two groups is the cost of the cheapest link between any nodes in those groups. This is an **optimistic** measurement.
- *Maximin Distance:* For each incoming link, the minimum distance to any suitable outgoing link is calculated ($C1 \rightarrow C5 =$**4**; $C6 \rightarrow C7 = 1$). The largest of these values is then added to the cost of the outgoing link and used as the cost between groups ($(C1 \rightarrow C5) + (C5 \rightarrow D1) = 4 + 1 = 5$). This is a **pessimistic** measurement.
- *Average Minimum Distance:* As in the previous one, but the distances of any suitable outgoing link is averaged (instead of selecting the largest). This is an **average** measurement, that works well in general, but fails in specific calls.

[5] Further reading: Millington+Funge section 4.6.1

# Heirarchical Pathfinding in the wild

This blog talks about how their game ”Factorio” used a form of heirarchical path finding to vastly speed up ordinary A*.

https://factorio.com/blog/post/fff-317

# Wrap-up

# Quizzes and This Week's Lab

- Finish the FIVE moodle quizzes on this unit:
  - Three on Part A of this lecture
  - Two on path-finding
- Next lab is on Dijkstra implementation and Steering Behaviours
- Finish the quizzes on Dijkstra and Vector2D (Programming) before then

# Next Lecture / Lab

- There is no lecture next week (Just the progress test and lab).
- There is a lab next week.

# Progress Test

The test:

- It contains 18 questions: some are multiple choice and some are numeric.
- You have 30 minutes.
- The test is to be taken under exam conditions.
- It is worth 18% of the module final mark.

Have a calculator, rough paper, and a pencil with you.

Do not miss or be late for this progress test[6]

[6](or you might have to retake the whole module this summer)

# Assignmnet 1 Reminder

- Assignment 1 deadline is due in Week 7; in just over 2 weeks.
- Read the assignment instructions, under CE812 Moodle Assessment Information.
- Do not miss Assignment deadline[7]
- Submit it early please (aim for a week early). You can resubmit improved versions later.

[7] (or you might have to retake the whole module this summer)

# Funded PhDs in Game AI

- Interested in a funded PhD in Game AI/Games research?
- See https://iggi.org.uk/apply
- Deadline 27th January 2022

# Questions?