



# Lecture 6a

*Searching Algorithm - A\**

**FIT3094** AI for Gaming, Alan Dorin

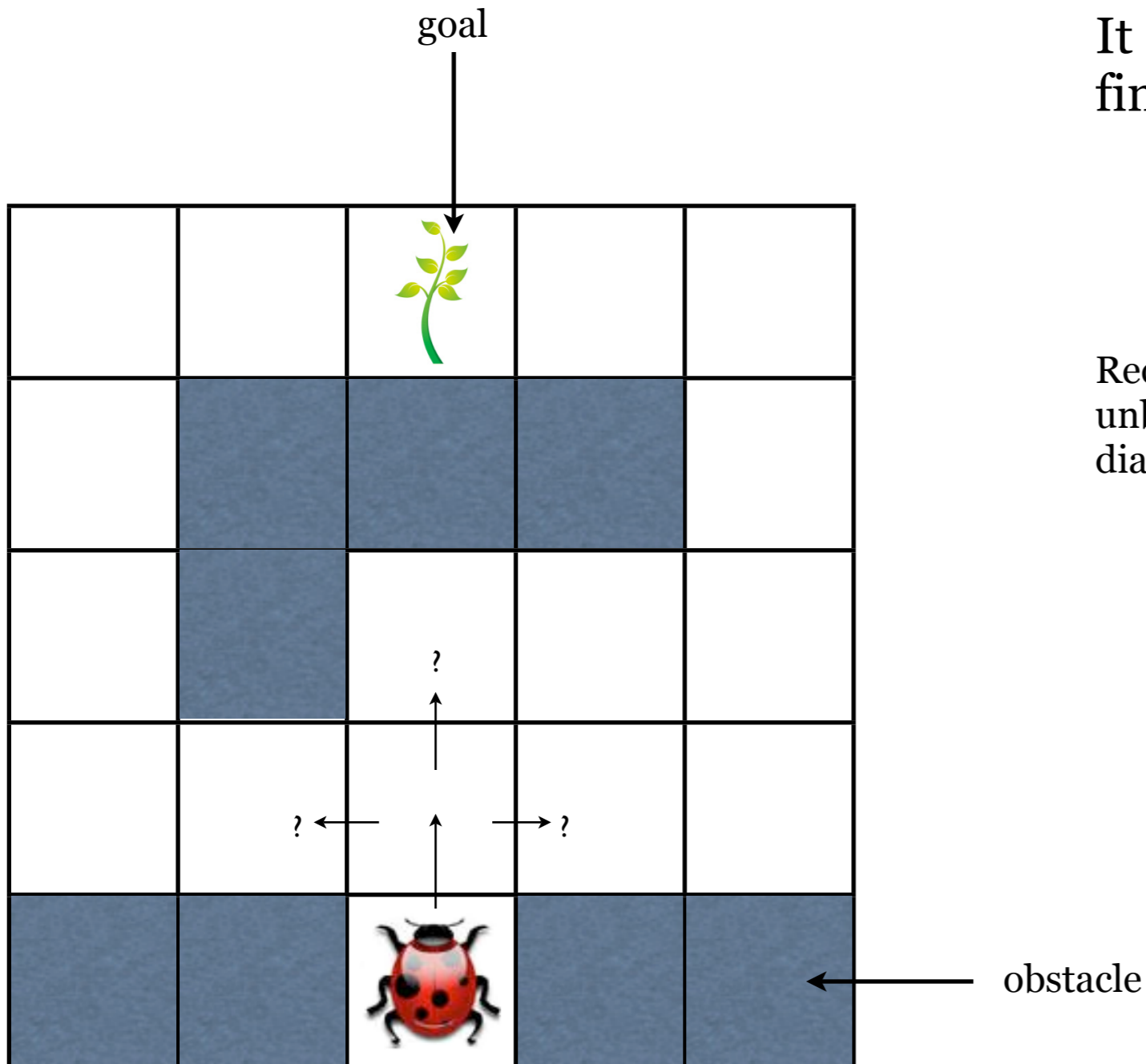
## **Learning Objectives**

To know the advantages of using the A\* search algorithm.

To understand how the A\* algorithm explores a map from a starting location to a goal.

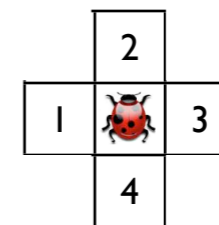
**We return to this frequently encountered problem:**

**How does an agent *choose* between multiple paths?**



It may *search* the possible options to find the most satisfactory solution.

Recall: The lady-bird (agent) can move into any unblocked, neighbouring square, *not* including diagonals. This is its *von Neumann neighbourhood*.



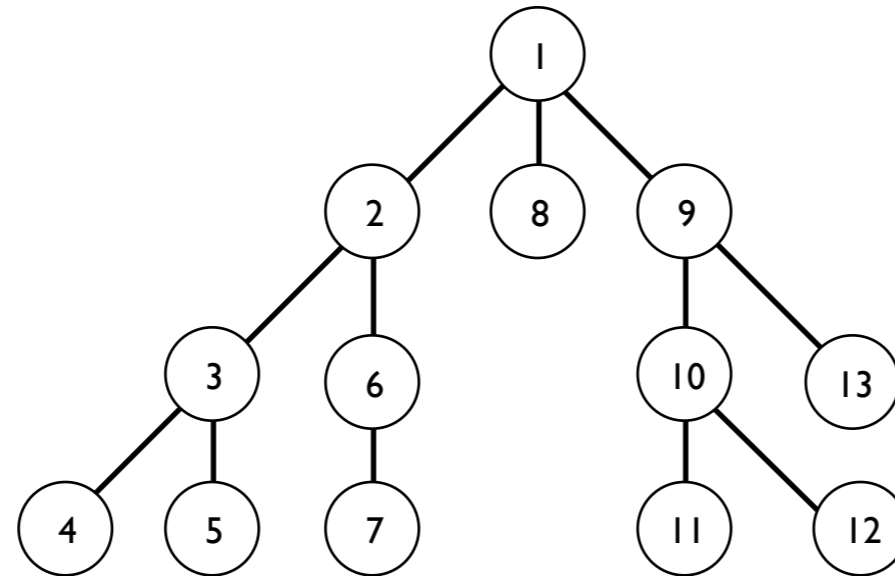
*von Neumann neighbourhood*

## Basic algorithms : Depth-first search (revision)

The algorithm expands the first child node of the search tree that appears at each stage.

Hence, it explores deeply into the space until the goal is reached, or it finds a dead end.

At a dead end, the algorithm *backtracks* to the most recent node that contains unexplored branches and continues taking the first child node that appears at each stage.



In a very deep search space, the algorithm can be bogged down searching deeper and deeper into the space (a recursive algorithm can recurse so deeply the computer runs out of memory).

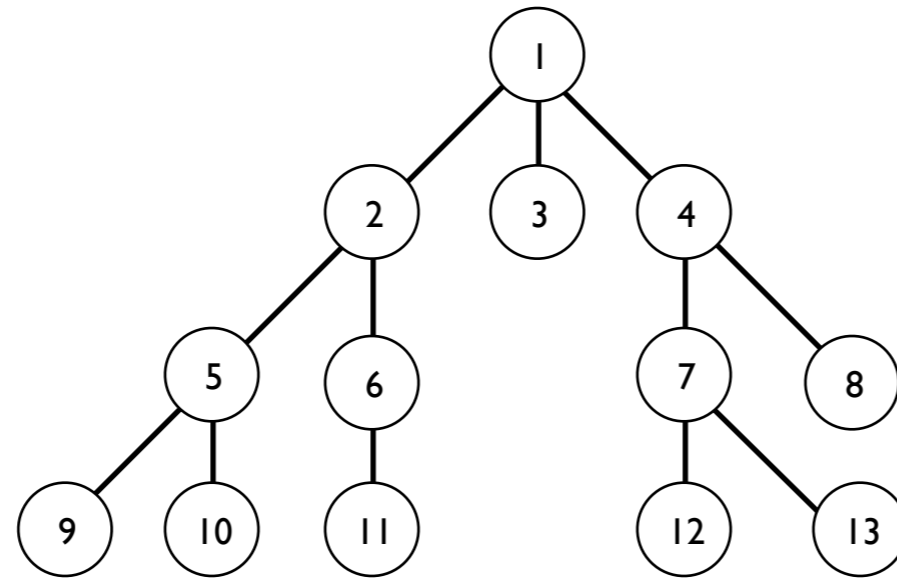
To avoid this trap, a depth limit is imposed to cause the algorithm to backtrack without exploring beyond a certain level. This variation of the algorithm is called a *Depth-limited* search.

## Basic algorithms : Breadth-first search (revision)

The algorithm expands every available node at a single level of the search tree.

Then it moves on to explore the next deeper available level from the nodes it has already explored.

Hence, it explores broadly across the space until the goal is reached.



In a very broad search space, this algorithm can get bogged down storing all of the nodes in a level before moving on to the next level... the number of nodes per level increases exponentially.

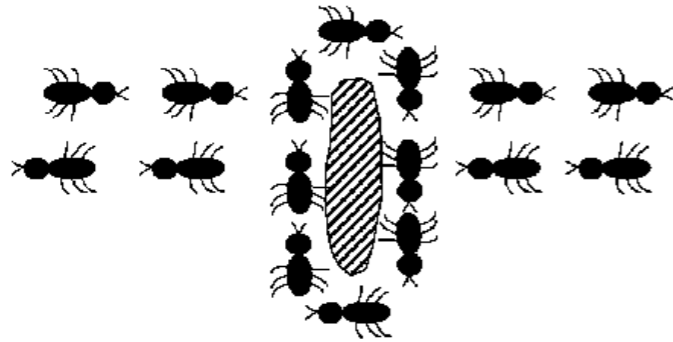
E.g. Consider the simplest tree, a *binary* tree... how many nodes are there in its 20th level? What if each node of a tree has *three* children instead?

# Basic algorithms

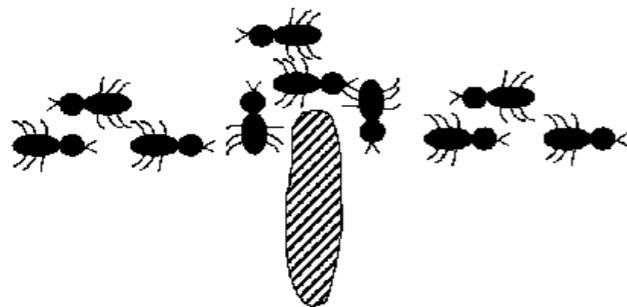
Basic depth-first and breadth-first algorithms perform *uninformed* searches.

They exhaustively search the nodes in a tree or graph.  
They do *not* estimate the cost of a particular route to the goal.

They halt when they *first* find the goal.  
They do not necessarily find the *shortest* path to the goal.



Can we do better than this?

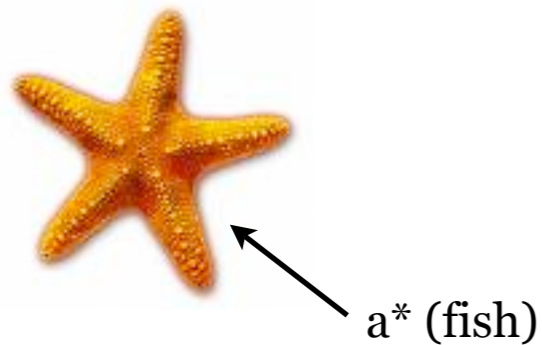


# A\* search

Is a graph or tree searching algorithm.  
Finds a path from a starting position to a goal if one exists.  
Finds the *shortest* path to the goal.

And best of all, it is relatively quick because it...

Assesses the *best* direction to explore at each stage of the search, rather than blindly searching every available path in a predetermined order.



# The elements of an A\* search



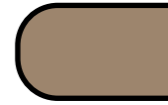
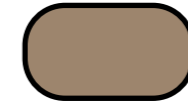
## Map

A *map* is a space (usually) containing:

A goal position

A start position

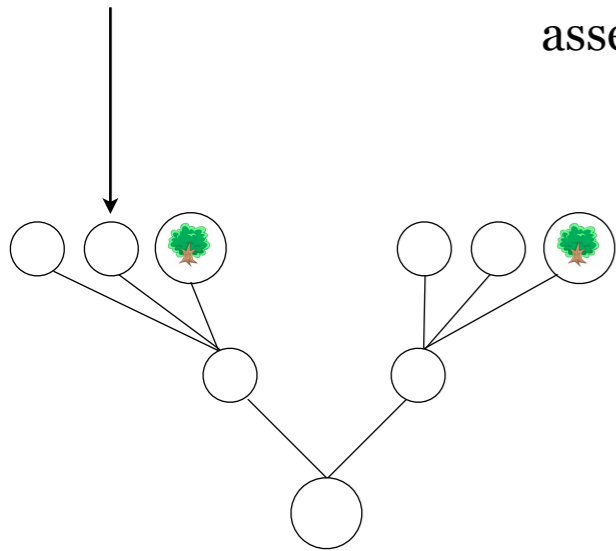
Routes from the start (hopefully!) to the goal.



## Node

A *node* is a data structure representing a position  $p$  on the map.

Nodes are not the map, they are kept in an independent data-structure that is assembled into a tree to represent the map.



In the A\* algorithm, a node stores:

- The distance from the start, via a specific path, *to*  $p$ .
- An estimate of the distance *from*  $p$  to the goal.

There may be several paths to a single location  $p$ !

# More elements of an A\* search

## Heuristic

A *heuristic* is a rule that helps to find a solution.

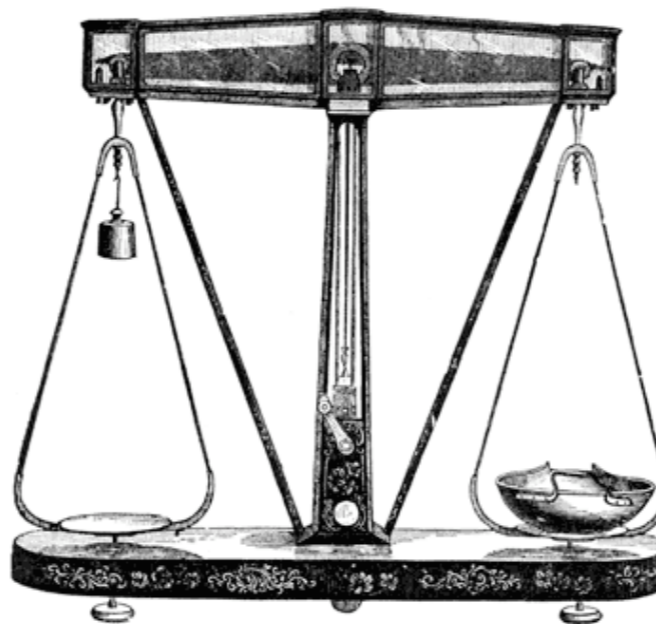
In this case, the heuristic is an *estimate* of the distance from the current position on the map to the goal.

## Cost

The cost of a path is the result of weighing up its benefits and drawbacks.

The A\* algorithm attempts to *minimise* a cost that is defined for each specific problem it is applied to...

points earned  
experience gained  
areas explored  
objects found  
puzzles solved



distance travelled  
energy consumed  
injury sustained  
time taken

## The A\* node contents

**g**

The cost from the start node to this node via the specific path used to get here.

**h**

The *heuristic* (estimated) cheapest cost of the path from here onwards to the goal.  
This must not over-estimate the distance to the goal (under-estimation is OK).

**f**

The *fitness* = **g** + **h**.

This is an estimate for the complete cost of travelling from the starting point, via here, onwards to the goal.

The lower **f** is, the better we *think* this path is likely to be.

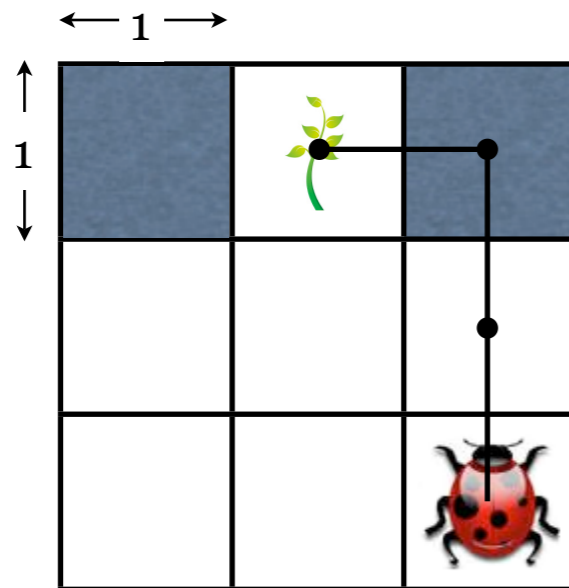
At each stage, the A\* algorithm chooses which node to explore by selecting the one with the lowest *f*.

## Choosing a heuristic for A\*

# h

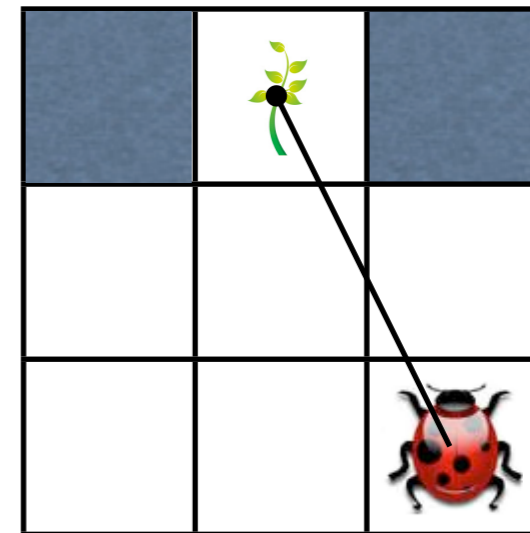
A heuristic that estimates a value that is guaranteed to be less than or equal to the actual value it is estimating is called an *admissible* heuristic.

For A\* to be optimal the heuristic it uses must be admissible.  
Possibilities for our ladybird problem include:



*City-block distance* between two points = 3

This is a very accurate admissible estimate for the ladybird problem.



*Straight-line distance* between two points = 2.236

This is an admissible heuristic that often under-estimates the distance between two points in our ladybird problem.

# The A\* algorithm's utility ingredients

**the Open List**      A list of all the nodes visited but not yet explored.

**the Closed List**      A list of all the nodes visited and completely explored.

A node is *explored* if A\* algorithm has:

Looked at every node that leads *from* it; *and*

Calculated their f, g and h values; *and*

Placed these nodes in the Open List for later exploration.

```

Clear OpenList
Clear ClosedList
startNode.g = 0
startNode.h = EstimateCost (startNode, goalNode)
startNode.f = startNode.g + startNode.h
Add start Node to OpenList

While (OpenList is not empty)
{
    currentNode = RemoveNodeWithSmallest_f from OpenList
    Add currentNode to ClosedList

    If (currentNode is goalNode)
        Return path from startNode to goalNode --- success!

    For Each (nextNode accessible from currentNode)
    {
        If (nextNode is in ClosedList)
            Skip nextNode

        Else
            possible_g = currentNode.g + DistanceBetween(currentNode, nextNode)
            possible_g_isBetter = false

            If (nextNode is not in OpenList)
                Add nextNode to OpenList
                nextNode.h = EstimateCost (nextNode, goalNode)
                possible_g_isBetter = true

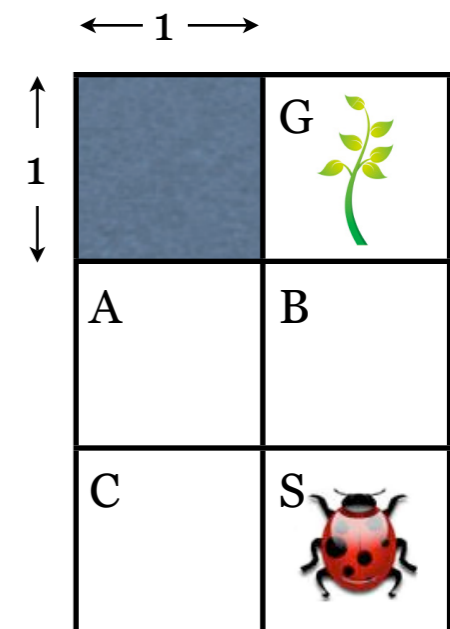
            Else If (possible_g < nextNode.g)
                possible_g_isBetter = true

            If (possible_g_isBetter is true)
                nextNode.cameFrom = currentNode
                nextNode.g = possible_g
                nextNode.f = nextNode.g + nextNode.h
    }
}
Return null --- failed to find a path!

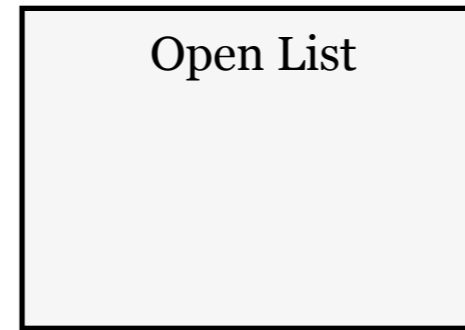
```



## The A\* algorithm

Using the simple example below, carefully trace this algorithm through from the ladybird's S position to the goal, G.



# Suggested layout for your trace of the A\* algorithm

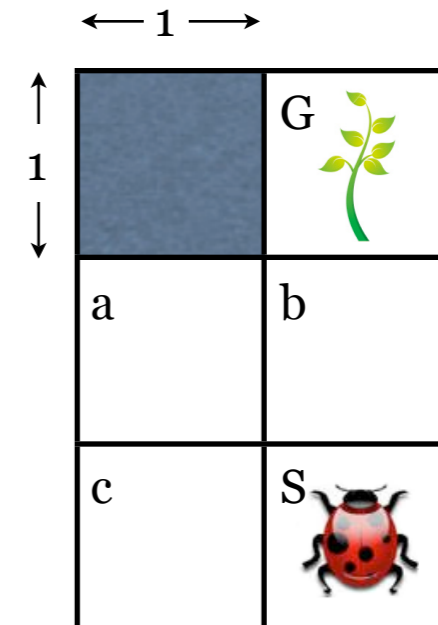


	<i>g</i>	<i>h</i>	<i>f</i>	<i>came from</i>
 S				
 G				
a				
b				
c				

← current

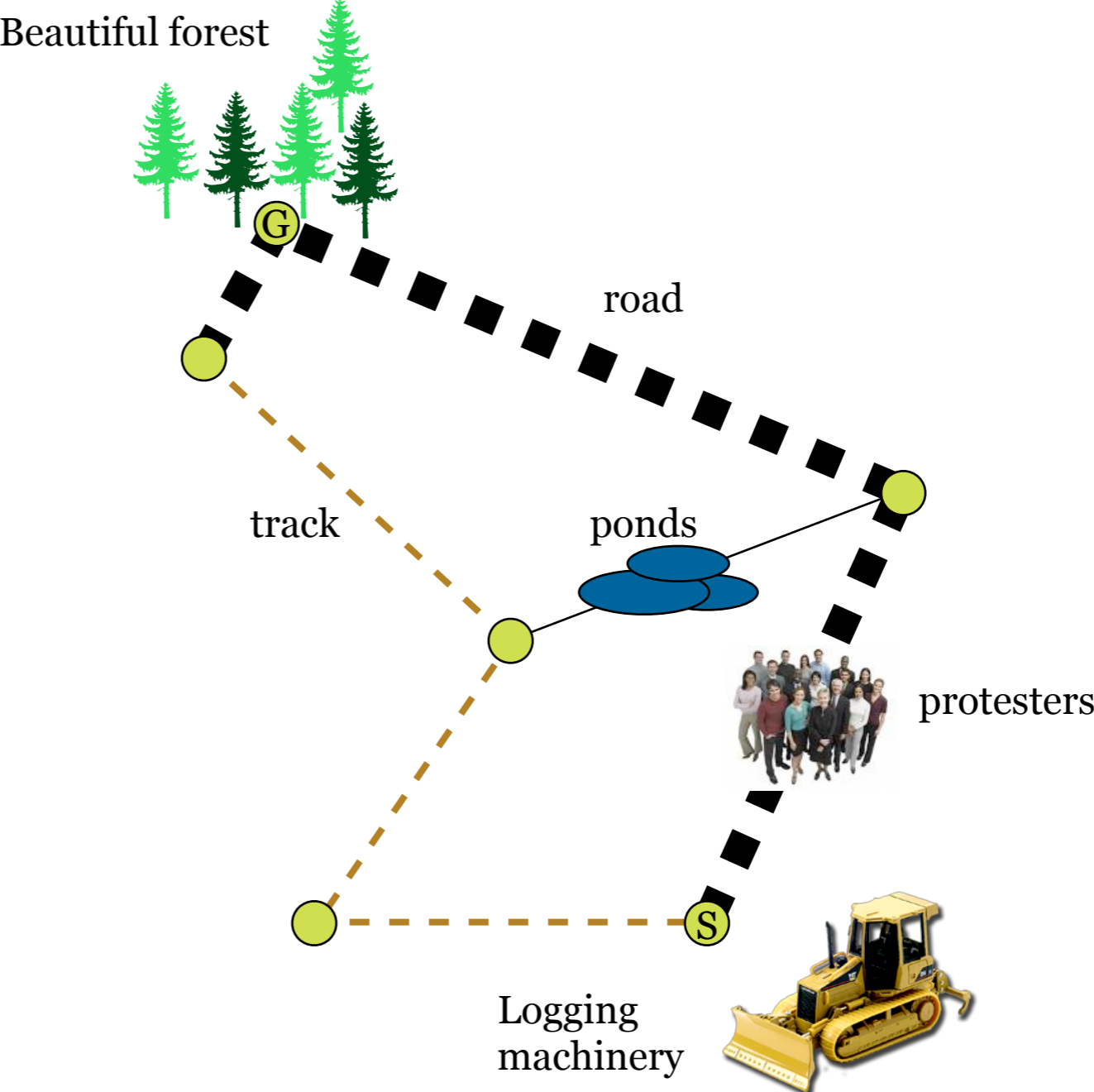
← next

Possible\_g =



# Example search problem

How might you evaluate the cost of a particular route in this example?  
How would you compute the heuristic?



## Example search problem

How could you search this puzzle for a solution from a random starting condition using A\*?



Goal state

Hints...

Start state: 3, 2, 4, 1, <SPACE>, 8, 6, 7, 5

Goal state: <SPACE>, 1, 2, 3, 4, 5, 6, 7, 8

What could you use for **g** and **h**?  
Do a little research to find out.

# Dijkstra's Search Algorithm

In A\* recall that the *fitness*  $\mathbf{f} = \mathbf{g} + \mathbf{h}$ .

This is an estimate for the complete cost of travelling from the starting point, via a specific node, onwards to the goal.

The fitness is used to decide which node we should take from the Open List to explore next.

Dijkstra's algorithm does without  $\mathbf{h}$  — the fitness of a node is just assumed to be  $\mathbf{g}$ .

Dijkstra's algorithm still finds the shortest path to the goal, only it needs to explore more widely than does A\*, so it takes longer.



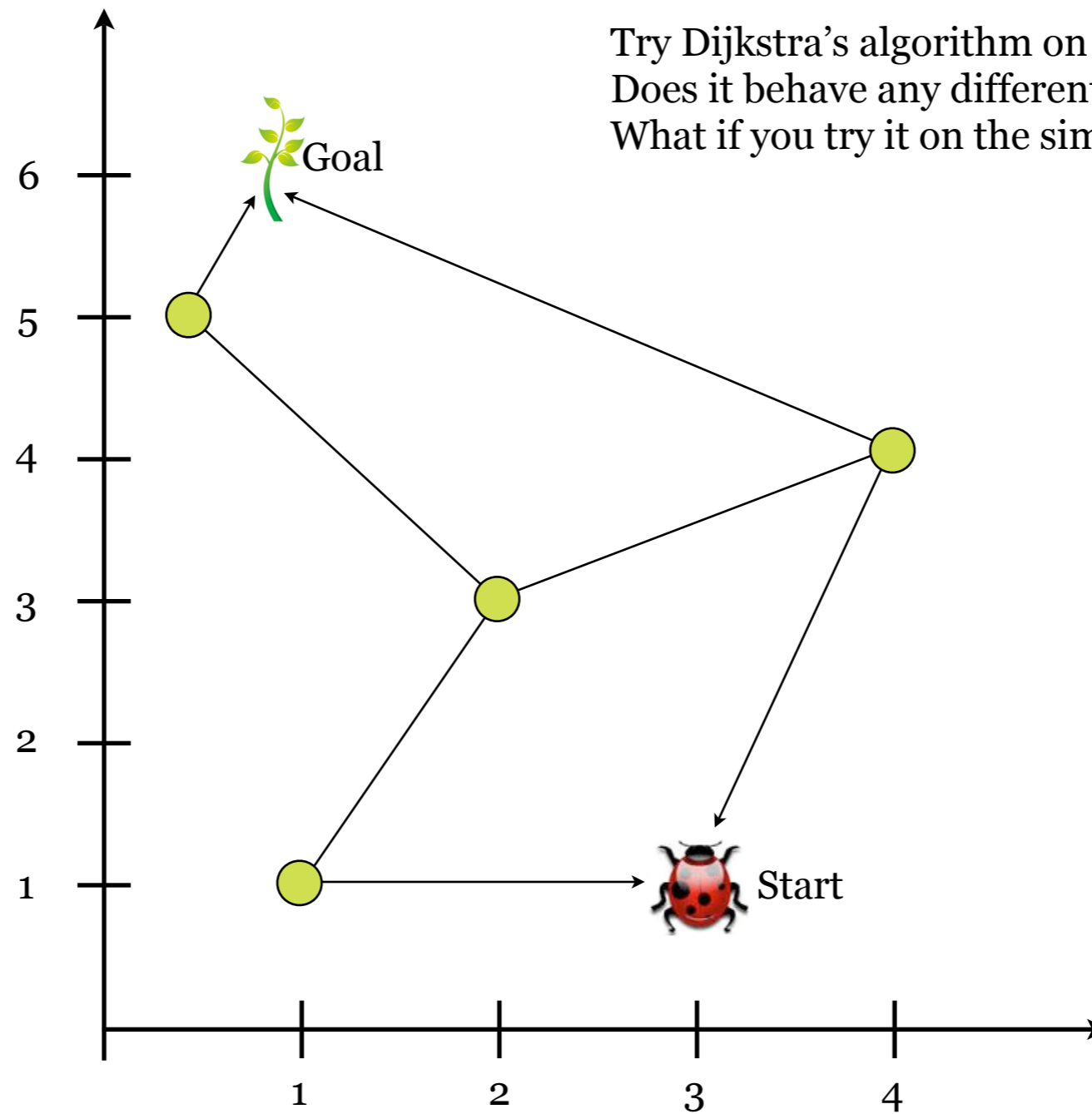
Edsger Wybe Dijkstra

If you don't know where the goal is you can't derive a heuristic  $\mathbf{h}$ ...

but you can still use my algorithm!

# In your own time...

Try the A\* algorithm on the following map.



Try Dijkstra's algorithm on the map also.  
Does it behave any differently?  
What if you try it on the simplified ladybird problem?

## Have you met the learning objectives?

What are the costs of employing the Depth-first or Breadth-first search?

Can you follow the A\* algorithm?

What components does it require to do its job?

Why is it a particularly useful search technique?

Try A\* on a simple grid-based map of your own invention, in which an agent can move from a cell to any location within its *Moore* neighbourhood.

