Unit 1
SEARCH

Context:

An ***intelligent agent*** – this is an entity (could be a robot, could be an automated system or a component of an automated system, could be a program that implements some A.I. algorithm) that:

- ***Senses*** the environment – either by using its own sensors, or by being provided with information about the environment as part of a set of input data provided to the agent and updated as needed.

- ***Takes action*** in order to achieve a pre-defined ***goal*** - Action here doesn't necessarily mean a physical action, it more generally indicates the agent in some way affect the state of the environment, for example, by changing the value of parameters that control what the agent is doing, or by making a choice between possible actions it could take in the future, or by making decisions regarding what is happening in the environment.

- The ***goal*** - defines what the agent is intended to achieve, and is usually encoded in the form of a ***utility function*** that the agent has to maximize (alternately, and often seen in machine learning, the agent has a ***loss function*** it's trying to minimize).

(Very simplified) ***Examples of agents:***

- A ***smart*** thermostat: goal is to keep the temperature inside a room at a comfortable level
  - senses the current room temperature
  - actions include turning the heating or A/C on/off as needed

- A car's simplest's ***autopilot***: goal is to keep the car driving at a constant speed within its lane
  - senses the car's velocity and location w.r.t. the lane and other cars
  - actions include accelerating/braking, and steering as needed

- A smartphone's ***assistant***: goal is to provide a suitable response to user queries
  - senses the user's voice to detect requests for assistance
  - actions include carrying out on-line searching, performing tasks like scheduling, saving common queries, providing voice feedback

(Very simplified) ***Examples of non-agents:***

- A light switch: does not have a goal, it simply provides a way for a user to toggle lights on/off
- A traffic light: produces a time-dependent, pre-programmed pattern, does not respond to the environment (caveat: there are smarter versions, which may qualify as agents)

We will look at many types of agents in this course, so the above description is necessarily very general. Details will fill-in as we look into specific A.I. problems and techniques for solving them.

Agents (continued).

Simplest type of intelligent agents: Reactive agents (like the thermostat example above). They sense the environment and their action is decided entirely as a response to the current conditions in the environment.

Control systems (e.g. for industrial processes, vehicles, and so on) are a good example of this kind of agent. They are incredibly useful for automating things, but not very smart.

More interesting: Agents that rely on ***a model*** of the environment to ***make a prediction*** regarding what will happen in the near future, and what the effect of the agent's actions will be, and make decisions/take actions accordingly. These are the type of agents we usually think of when we speak of *intelligent* agents.

Search Problems

The first general technique we will discuss for solving problems is ***search***. This is a completely general technique that can solve any problem you have (if you are willing to wait long enough). How? it's very simple: Given a problem, and a concrete definition of how a solution to this problem is specified, *try out every possible solution until we find the best (or the correct) one.*

Of course that turns out to be not a very practical way of solving any real problems, so we have to be a bit more concrete regarding what problems we can solve realistically using search, how we specify these problems, and how we go about finding a solution.

Components of Search Problems:

- A definition of what a ***configuration*** for the problem we are solving looks like.
  In order to solve a problem using search, we need to determine what are the relevant variables that define the state of the environment – any variables or quantities whose values affect the outcome of our problem need to be included, we call these variables ***state variables.*** A particular assignment of values to these variables is a ***state*** or ***configuration*** for the problem.

- A definition of how ***actions*** taken by our ***agent***, or by any other ***agents*** that are involved in this problem can change the configuration of the problem. We need to know how actions affect the values of variables, which allows us to determine given the ***current configuration*** what are possible ***future configurations*** depending on what actions are taken by the agents involved in the problem.

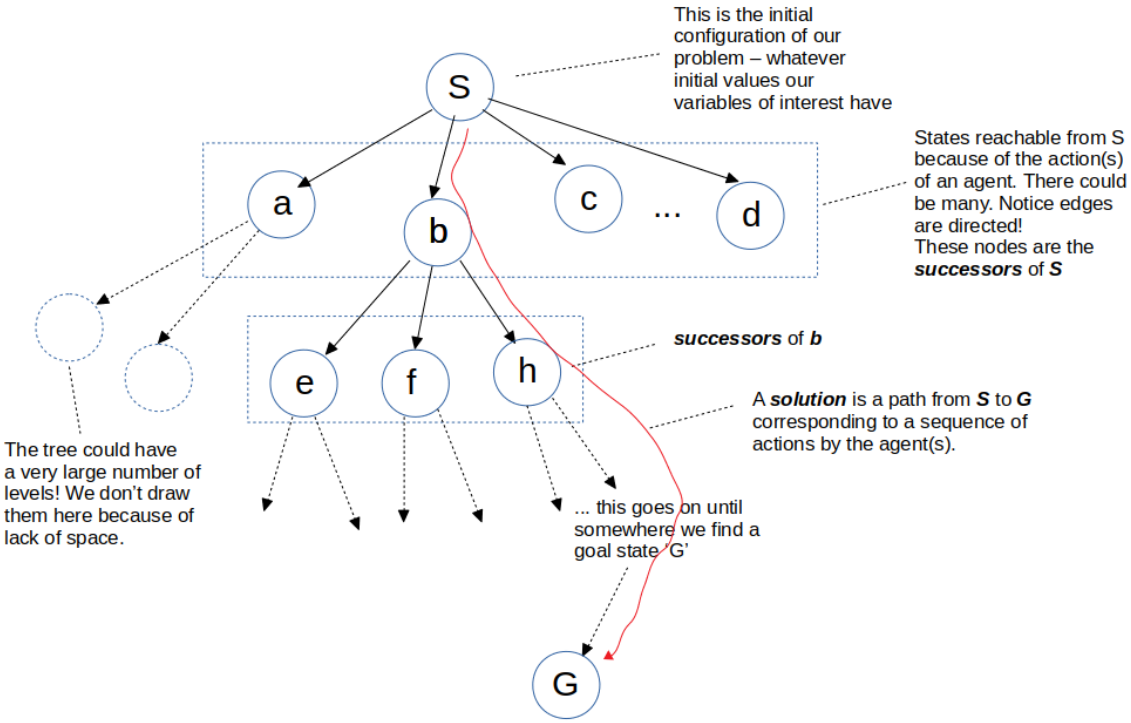
- A definition of a ***goal state***. This is simply the desired outcome for the problem we're solving, characterized in a way that can be checked by looking at the values of variables in our ***state***. There can be ***multiple goal states*** for any given problem, and they can have different value (i.e. some may be better than others).

The natural representation for a search problem is a ***graph***. The graph contains a set of ***vertices (also called nodes)*** that represent the different ***possible configurations*** for the problem, and a set of ***edges*** that connect nodes that are reachable from one another as a result of agent actions.

In typical search problems, the graph takes the form of a ***tree*** (there is no reason to evaluate a given configuration more than once – if it is the goal, we're done, if it's not, then we move along – there are no loops in the graph because of this).

A ***solution*** is a sequence of ***actions*** that leads from the ***initial state*** to ***(the/a) goal state***. It is therefore also a ***traversal in the tree*** that represents our problem.

The setup of a general search problem is shown below:



Search Tree:

- Contains a ***root*** node corresponding to the ***initial configuration*** for our problem
- Nodes correspond to ***states (configurations)***
- Each level contains ***states*** reachable from nodes in the ***previous level*** that result from possible actions the agents can take
- A ***path*** in this search tree corresponds to a ***plan***. A sequence of actions that takes the agent from the initial state to some other state in the tree. If the final node is a ***goal node***, the ***plan*** is a ***solution*** for the problem.
- For most of the actually interesting problems we may care to solve, the search tree is ***too large to explore exhaustively***.

Concrete Example

To make things clear, let's have a look at how a specific problem gets translated into a search problem we can handle using the ideas set above.

Problem: ***Finding a path between two specific locations***

*Example situations w*here this comes up: ***Maps! (Waze, Google maps), Network routing.***

Definition of states (configurations) for the problem: ***Possible locations the agent can be at***.
in the maps example, we can divide our map into a grid, each grid location is a possible location where our car could be, so each location is a configuration/state.

Successor states: ***Locations that can be reached from the current one by driving (safely! And legally!)***

Initial state: ***Starting point for our path finding problem***

Goal state: ***Destination location***

Important note: Especially in mapping applications, the map is normally stored as a graph. This graph ***should never be confused*** with the search tree we will use to find a path in the map!

Solving Search Problems

The process for solving a search problem is fairly straightforward – it is simply a graph search that starts at the initial node, and proceeds in some order to explore nodes that can be reached by different sequences of actions, until a goal node is found.

In pseudocode:

```
Initialize starting node (i.e. determine the root of the search tree)
Create an initially empty list of expanded nodes
Add starting node to the list

while the list is not empty:

    get the next node(*) from the list (and remove it from the list)
    if this node is a goal node
           success! : return path from start node to this goal node
    else
           add successor nodes for this node to the list in some order(**)

list is empty and didn't find a goal : failure! (return an empty path)
```

(*) The order in which we ***get the next node*** is important – see below
(**) The order in which we ***add successor nodes to the list*** is also important – see below

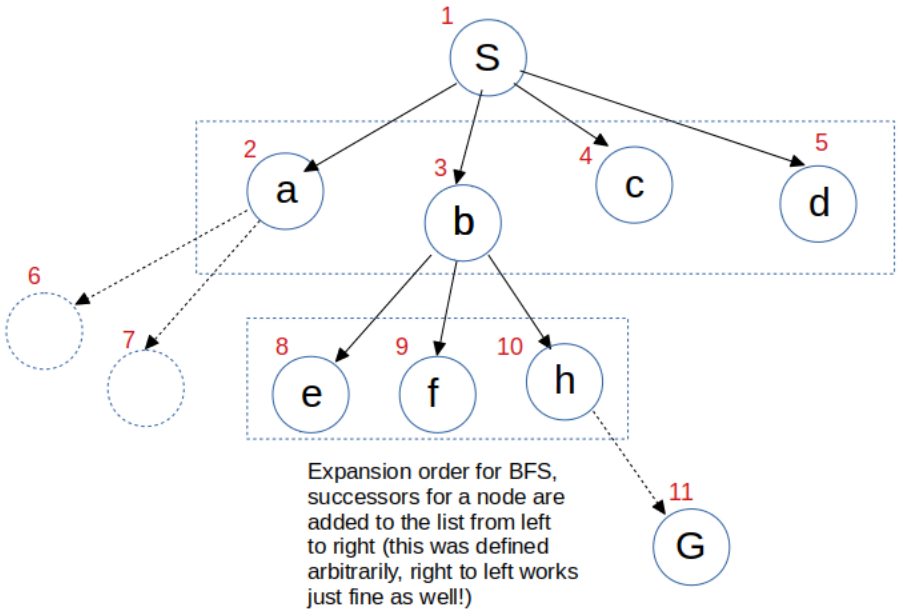
As you see, the process is quite simple – there's some book-keeping involved (you have to keep track of paths between sequences of nodes so you can return a solution path if found), but overall, it is straightforward. The interesting bits are the details of specifying ***in which order*** we get the next node from the list, and ***in what order*** we add successor nodes for a given state to the list.

The two most basic search algorithms, which you covered in detail in your data structures and algorithms course (B63) are Breadth First Search (BFS) and Depth First Search (DFS).

BFS adds successor nodes to ***a queue*** – each node added goes to the end of the queue. The ***get the next node from the list*** operation then gets the node at the ***front*** of the queue. So nodes are expanded (explored, checked to see if they are the goal node) in the order in which they arrived at the queue.

This produces a search ordering in which all nodes at level 1 (just below root) are expanded before any nodes at level 2 (two down from root). All nodes at level 2 are expanded before any node at level 3 is expanded, and so on. The search explores the tree level by level, and within a given level the search ordering is either arbitrary (doesn't matter as long as there is a consistent rule to add successor notes to the list), or can be dependent on the problem we're solving.

BFS is easily implemented with the pseudocode above by using a ***queue*** to store the list of nodes.

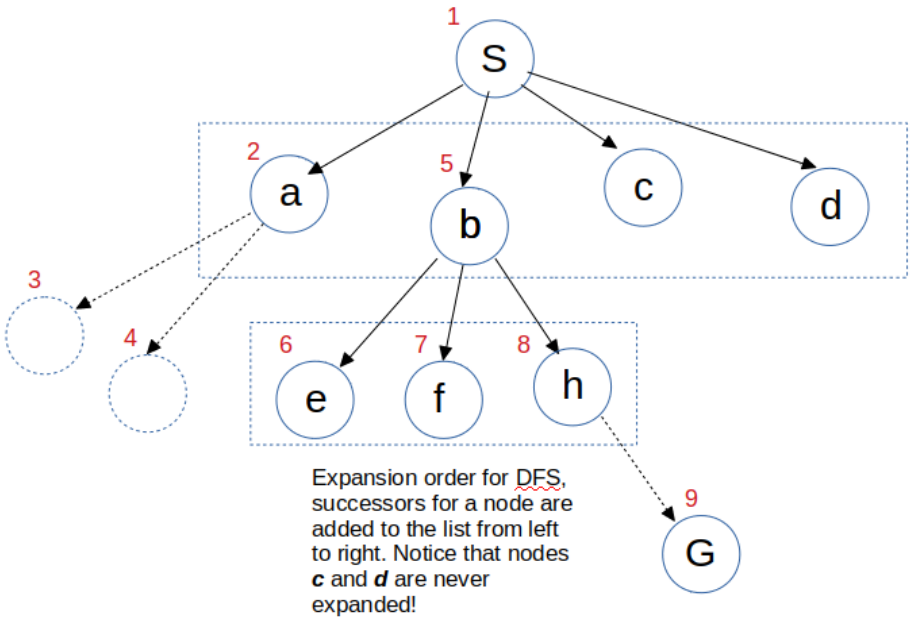


You should replicate the BFS process on paper, and see how the queue works to produce the search ordering shown above.

In contrast to BFS, Depth First Search (DFS) expands nodes in the order that leads to travelling quickly away from the starting node. This is achieved by using a ***stack*** to keep the list of nodes.

Successor nodes are added to this stack in some order, and the ***get the next node from the list*** operation gets whatever node is at the top of the stack at any given point.

DFS is more naturally implemented using a recursive function, and if you do this carefully, the recursion keeps track of the path from the start node to any goal discovered during search.



You should verify on paper that the stack produces the search ordering shown above. Both BFS and DFS should be very familiar to you, and you should be able to implement them without much difficulty – otherwise it's time to review your B63 material!

Generalizing Search

The naive search formulation described above needs to be extended a bit in order to handle more interesting problems.

First off – different actions are typically associated with different ***cost***. The idea here is that from the point of view of our agent, or in the context of the problem we're trying to solve, some actions are considered better than others.

For example, suppose there are two roads that have the same length, and lead to the same location. However, one is completely congested with traffic while the other is free. Both of these roads can be part of a plan that reaches the same destination, but one is going to be much worse than the other for the agent that chooses it.

In general, between multiple choices the agent can make at any given time, some will turn out better or result in a solution that is preferable for the agent in some concrete way.

In order to encode the cost of different actions, we allow edges in our search tree to have different weights. The weight of the edge leading from some node A to some node B represents the cost of the action that changes the state from A to B.

Given a graph with weighted edges, ***we no longer want just any solution*** to the search problem. We are interested in finding ***an optimal solution***. This normally means the solution with the ***lowest cost*** for the agent (e.g. in the maps example, the path between two locations that takes us there in the smallest amount of time). Note that there may be multiple solutions with the same optimal cost, we don't have a preference between them, any of those will do!

How do we search for an optimal path?

Uniform Cost Search (UCS)

UCS is a generalization of BFS that works as follows:

* Use a priority queue to keep the list of nodes to expand. Nodes in this queue are ordered by increasing cost to get there from root
* When we expand a node ***B*** that is not the goal node:
  - Add each of the neighbours of ***B*** to the queue with the cost to get to each of them from the root node
  - If a neighbour is ***already*** in the queue, check if it can be reached with a lower cost by going through ***B***. If so, update its cost in the priority queue
* The ***get next node to expand from the list*** operation gets the node from the priority queue with the lowest cost to reach from the root

UCS will find the ***shortest path,*** or equivalently, the ***lowest cost path, optimal path*** between the start node and a goal node. If all edge weights are set to the same constant value, it is equivalent to BFS.

You may already be familiar with Dyjkstra's Shortest Path algorithm, UCS is similar and can be implemented in much the same way – the main differences being that UCS works on a search tree, and stops as soon as a goal node has been discovered, whereas Dijkstra's method will determine the shortest path to every node in a graph.

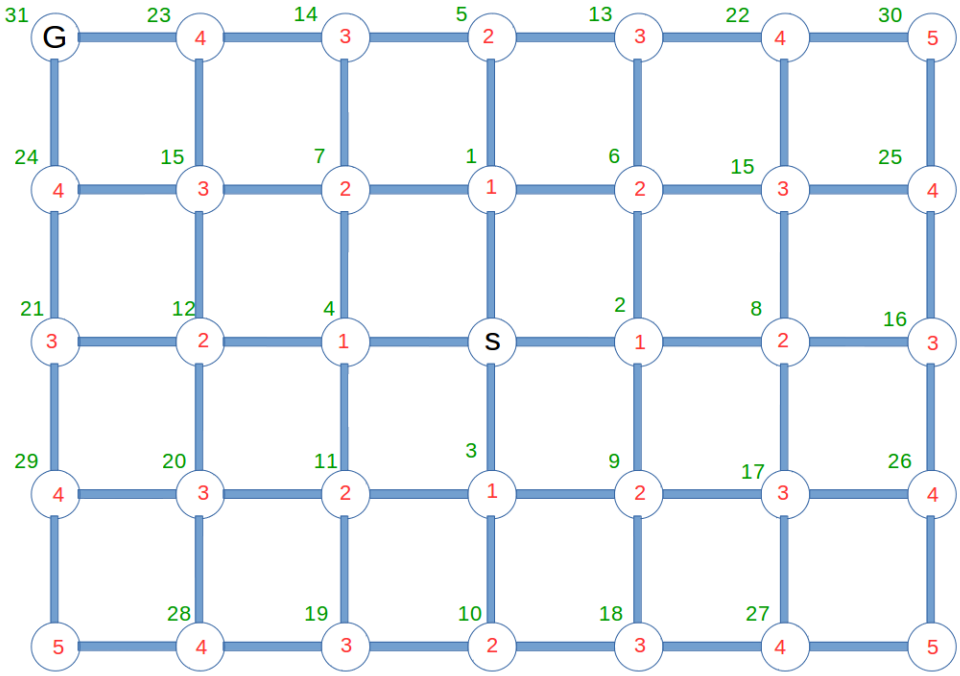
Importantly, ***UCS is guaranteed to find the optimal path to a goal node*** if any such path exists under the condition that all edge weights are positive and greater than zero.

Heuristic Search

There's nothing particularly new or 'intelligent' about the search process described above. It is *brute force* in that it relies on just doing computation to find a plan for the agent to solve the given problem. However, such an approach may eventually become unwieldy or impractical. Let's think about how to make search a bit smarter.

The goal is to reduce (if possible) the computational burden of finding the ***optimal path to the goal***. We do not want to sacrifice optimality in order to achieve a lower computational cost.

In order to see how this may work, let's consider once more the maps application, and the problem of finding an optimal path between two destinations:



In the above diagram, the map is a very simple grid of streets. Intersections are circles, streets are blue edges connecting neighbouring intersections. The start and goal nodes are labeled. To make things simpler for the time being, we assume all streets have the same ***cost***, and let's make that cost equal to 1.

Inside each node, a number in red shows the cost of getting to that node from the start location. This is simply equal to the number of streets we have to traverse to get there. This also corresponds to a commonly used distance measure called the ***Manhattan Distance*** (the L1 norm) between two locations.

Next to each node, in green, is the order in which UCS expands the node to check if it's the goal. When expanding a node, its neighbours are added to the priority queue in clockwise order, starting at the top. *You should verify that the expansion order makes sense given how UCS works and the order in which the neighbours were added to the priority queue*. But you can see the pattern – all nodes at distance 1 were expanded before any neighbours at distance 2, and so on.

The end result is that the goal node was reached after most of the graph was explored. The problem here is that UCS uses no information other than distance from the start node. However, ***for many problems, we have additional information regarding the problem that we can use to make smarter decisions in terms of the order in which nodes are expanded***.

In the problem above, it should be easy to see that expanding nodes that are below and/or to the right of the start node, actually moves us ***away*** from the goal, whereas exploring nodes upward and/or the left moves us ***closer*** to the goal. We want our search process to ***explore more promising paths earlier*** compared to other possible paths.

Heuristic Search, also called ***A* (A-star) search*** is a powerful and extensively used search algorithm that makes use of a ***heuristic function*** to ***guess*** which nodes in the search tree are likely to be closer to a goal state, so those nodes can be expanded first.

Because it is a ***guess,*** it can not be used exclusively to guide search expansion order (if you think about it, in order to know exactly how far each node is from the goal, we'd have to have completed the search process on the entire tree!). Instead, A* search uses the heuristic function ***in combination with*** the actual cost of getting to a node from the initial state (this value is updated as nodes are expanded, just like with UCS).

The ***heuristic cost*** used by A* search to determine expansion order is:

$$f(n)=g(n)+h(n)$$

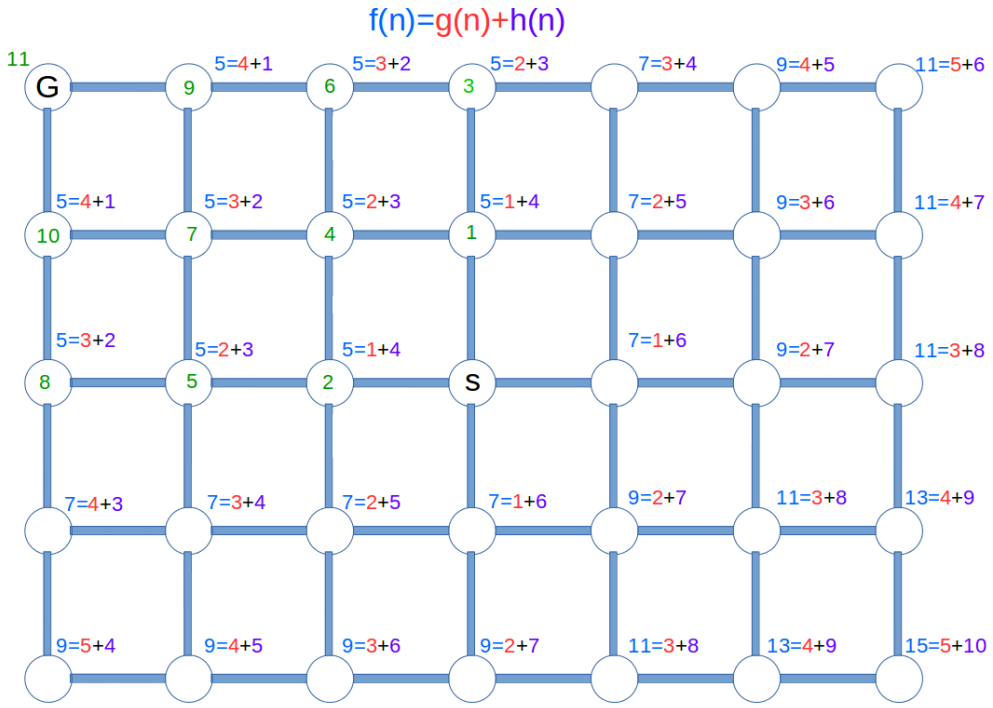
Where f(n) is the ***heuristic cost*** for some node ***n,*** and it consists of ***g(n)*** – the actual cost of reaching node ***n*** from the initial node, and ***h(n)*** a ***guess of the cost to get to [the/a] goal state from n.***

The ***A**** search algorithm is ***identical*** to UCS, but we use the heuristic cost f(n) to order nodes in the priority queue. Otherwise there's no change! So if you have implemented UCS, implementing A* is straightforward, you just need to add a suitable heuristic function.

In the example above, a ***good guess*** would be to use the ***Manhattan distance between a node and the goal***. This happens to be a lucky guess for this problem because the Manhattan distance is the actual distance between two nodes – it is very often ***not*** the case that we can find such an appropriate guess, so we'll talk about what makes a good guess below. For now, let's see how things change once we introduce the ***heuristic cost*** for each node into the search process.

The image below shows the same map, but this time we have annotated each node with the heuristic cost – you can see the two parts of it, *g(n)* in red, and *h(n)* in purple, as well as the total cost *f(n)* in blue.

The expansion order is indicated inside the node, in green. You can immediately see that whereas UCS was exploring most of the map before reaching the goal, A* avoids expanding directions that move us farther away from the goal, and reaches the goal with significantly less exploration. In a large map, this could be a very significant reduction in the amount of search required to find the path to the goal!



Things to keep in mind:

- We don't actually know *g(n)* in advance. We know g(n) *only for nodes that have been added to the priority queue, and it is not actually fixed (it could change!) until the moment when we expand the node* (this is the same with UCS).

- For nodes with the same heuristic cost, expansion order is arbitrary – in the example above, neighbours of a node are added clockwise from top, and this is reflected in the expansion order above.

You should verify that the expansion order shown above is correct given the pseudocode for UCS/A and the values of f(n) shown in the figure.*

What makes a good heuristic?

The interesting and often challenging part of implementing A* is finding a good heuristic to use in our search process. We can not just make an arbitrary guess – ***in order to preserve the optimality guarantee of UCS, the heuristic function has to satisfy certain conditions.***

Admissible Heuristic: A heuristic h(n) is ***admissible*** if and only if

$$\forall n, 0 \leq h(n) \leq h^*(n)$$

where *h*(n)* is the ***true cost*** of getting to the goal node from node ***n*** – we do not know this cost, so proving that a heuristic is admissible is tricky! The Manhattan distance used above for the mapping example is admissible – convince yourself this must be the case!

Under the above conditions, *h(n)=0* is admissible (i.e. using ***no*** heuristic function preserves optimality of UCS – because ***it is UCS!***). Any non-zero admissible heuristic must have values that are ***less than, or at most equal*** to the true cost of getting from node ***n*** to the goal. Put in a different way, an ***admissible heuristic is optimistic*** – it does not over-estimate the cost of getting to the goal. Why is this important? What do you think would happen if the heuristic *h(n)* significantly over-estimated the cost of getting to the goal from some node in the graph?

Because the heuristic *h(n)* can be anywhere between 0 and *h*(n)*, there's ample space for different heuristic functions to be designed for the same problem – the closer *h(n)* is to *h*(n)* the better.

It should also be noted that ***how you define h(n) depends entirely on the problem you are trying to solve***. The heuristic function is attached to the definition of the ***cost*** used to determine optimality for a path. As an example, let's say we change the cost for traversing a street in the maps example above. Instead of being constant, let's make the cost proportional to 1/S where 'S' is the speed limit for the street. Faster streets have lower cost since we can travel through them in less time.

Under these conditions, the Manhattan distance is no longer appropriate (why?). Can you think of an admissible heuristic that we can use to carry out A* search when the cost of traversing a street is defined as inversely proportional to the maximum speed allowed down this street?