One definition of a plan is: a sequence of actions that results in a specified goal state. And to determine such a plan, an agent can consider the anticipated results of applying various actions in various orders, over and over, from a given start-state, until a goal is discovered; and then the path (sequence of actions) leading to that goal state (from a given start state) can be stored and/or put into use.

This conception works well for certain situations, and in effect converts the problem of determining a plan as a search problem through the space of possible states of the world. As such, the notion of an agent is often left aside. In fact, some of the earliest work in AI was focused on such situations, and this is often where an AI course begins. We will look at it now. It is not outmoded at all, but I left it until this point so that we could develop the idea of an agent in more depth. But it is time to take note of the fact that a lot of AI can be done with hardly any notion of agent at all.

We need some terminology: a *state* is a state-of-the-world (at a presumed time); think of it as similar to a situation in the sit-calc. It changes as the result of an action being performed. A *node* is a data structure that holds not only information about a presumed current state, but also information about what led to it – e.g., what action changed the world into that state from that of a previous node. Thus it is possible for distinct nodes to contain the same state s, if there are distinct ways to achieve s (i.e., by means of different sequences of action. For instance, if the world is a simple place that has only a color associated with it, the "paint-it-green" action will result in the green world state, no matter when it is applied; and similarly for "paint-it-red". So one could for instance make the world green, then red, then green, then red, etc. Each such painting action would result in a new node, but would have the same state as two nodes earlier. In effect, nodes indirectly keep track of history or the passage of time, whereas states do not. (This is a serious defect, but despite which a great deal of important AI can be done.)

We also assume each action (or each parent-child pair of nodes) has an associated cost – a number that represents the work or effort to go from parent to child. If not otherwise specified, we assume this cost is simply 1; and in that case the total cost c(n) from start-node to any node n just encodes the level n is on, i.e., how far below the start.

Keeping track of nodes – as opposed to simply states – produces a search tree: each change takes one down to a deeper level in the tree, even if it reproduces a world state already seen higher up.

There is a broad general format that covers a wide variety of search algorithms:

While frontier is not empty do
    Remove one node from frontier
    If node = goal then stop
    Else put node's children on frontier

One gets quite a variety of specific search algorithms using this format, by putting in more detail about *which* node is removed from the frontier (which is a collection of nodes, initially set to contain only the start-node). In many applications of search algorithms, one inputs an entire graph or tree as data, through which the algorithm then searches.

However in AI, often the tree or graph is infinite, or not known in advance. Instead what one knows (what the system or agent is given) consists of possible actions and how they affect the world, and a start-state. Then the search-tree is *created* during execution of the algorithm: each time a node is "expanded" (actions applied to it to form its children), the tree grows to include those children.

But the same algorithm format above applies equally to both settings. Often there is also a *cost* value associated with each parent-child pair (or with whatever action changes the world from parent to child).

Some simple choices of *removal-from-frontier* lead to standard and familiar search algorithms:

Remove the newest node (the one most recently placed on the frontier) – this leads to **depth-first search.**

Remove the oldest node – this is **breadth-first search.**

[[Side-notes: BFS is **complete**: if there is a goal, it will be found using BFS (assuming finite branching).
BFS is **optimal**: it will always find the goal closest to the start (and cheapest, assuming equal step-costs for all actions).

There is a subtlety that may have occurred to you. Why do we bother (in the above algorithms) with nodes that contain world states that have already been looked at in earlier nodes? This seems wasteful, and easy to correct as follows (where the change is underlined and in italics):

- Choose one frontier node and remove it from the frontier (it is now being explored).
- If its state is a goal state then we are done; else expand it by generating its children and adding *any whose states are unseen* to the frontier *and marking those states as seen; if any more costly frontier node has a state matching one of the new ones, remove it.*

This sometimes called the *graph-version* of the basic search algorithm (as opposed to the tree-search version). But it still builds a search tree, not a general graph.]]

Remove the cheapest (lowest-cost) node n – this is **uniform-cost search.** Here cost refers to the *total* cost c(n) from the start-state node to any given node n, adding up the action-costs along the way. This version can be considered to be a generalization of breadth-first search: in the case where all actions have the same cost, c(n) essentially is the depth of n (how many levels below the start-node).

{Note that removing the cheapest node (to then examine its children) might *seem* like a good idea overall; but not necessarily so. True, we do usually want the cheapest overall path from start to goal. But doing the cheapest actions first may well not be the best way to to that. For instance, if you want to build a house, and need to use both a concrete foundation and plywood walls and ceilings, it might be cheaper to put up the walls and ceilings than to dig and pour a foundation. But it makes no sense in that order: the foundation, even though more costly, has to be done first.]

The three algorithms above are examples of what is called "blind" or "uninformed" search. One has no idea where the goal is likely to be, no special knowledge of the particular data or problem that might guide search one way or another.

But another variety of removal-from-frontier – so-called "informed" or "heuristic" search, can be seen as a further generalization of uniform-cost search. The most famous version works by forming a guess or estimate h(n) – for each node n that is encountered – of the cost from n to a goal. This is added to c(n) to give g(n) =c(n) + h(n); g thus gives an estimate for the total cost to get from start to n and then on to a goal node. Then the frontier node with the least g-value is the one removed. This is the so-called **A* search** algorithm. The "heuristic" function h takes advantage of special knowledge of the problem to guide the search in ways that can lead to a goal state much more quickly. Typically humans craft h themselves; but there has been some work on automating that as well.

But how can one possibly find a good h, or even a plausible one? This is actually often quite easy. Consider the 8-puzzle. If we start at a given state, say

1	2	3
4		6
7	5	8

and wish to get to the goal

1	2	3
4	5	6
7	8	

then there is an obvious two-action sequence that does it: move the 5 up and then the 8 to the left. But the available initial actions also include moving the 2 down, the 4 right, and the 6 left. An **uniformed** search algorithm would have these in some order determined in

advance by the algorithm's designer, without taking into account any special information about this particular problem. But now suppose we have an actual cost for each action, that combine (add) to give total costs of taking any sequence of actions. Assume in fact that the cost is just the number of individual tile-moves involved in getting to a goal state.

Moreover, we want h(n) to be something we can calculate only from information already available in the portion of the tree that has been generated so far – i.e., without having to look ahead to future moves and nodes. Of course, looking ahead is not forbidden, but that is more complicated, and not in the spirit of search based on growing the tree as we decide where to look.

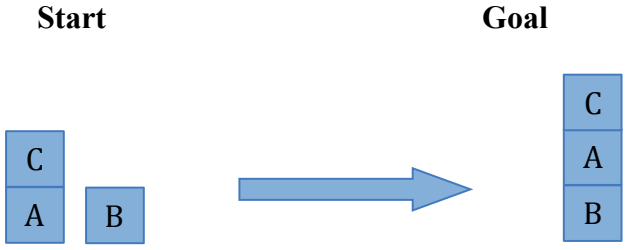
What general properties of the 8-puzzle can be exploited to make 5-up the best apparent choice, *without* our considering the following move (8-left)? Why is moving the 5 up a good idea in general, if it is where it is (at the bottom)? One answer is: because the goal has the 5 higher up than that. That is, we can reasonably suppose that it is in general plausible that *moving a tile closer to where it belongs in the goal* is a good thing to try. So one possible heuristic function – let's call it h1 – for this particular domain (the 8 puzzle) could be

h1(n) = the number of tiles that are out of place in state n (compared to the goal)

Another is

h2(n) = the total "manhattan distance" from n to the goal, where the *manhattan distance is the sum of the vertical and horizontal differences between where each tile is in n and where it is in the goal*. (This then is a lower bound on the amount of moving the tiles must do to get to a goal state. Indeed, h1 is also a – cruder – lower bound.)

[[Of course, this can backfire. Sometimes, to get to a goal, one has to move one thing far out of the way for awhile in order to first get something else where it belongs. We can see this easily in a blocks-world setting:



where C is already where it belongs (on A) but we have to move it off for a bit in order to get A on B, and then we can put C back on A. So in this case, an "obvious" heuristic function (number of blocks out of place) is not so good after all. This is why we refer to *heuristic* search: it is not guaranteed to do the right thing at all times, but at least it seems to stand a good chance of being better than blind (uninformed) search.]]

What we want to emphasize however is the following wonderful result, a bit of a rarity in AI:

Theorem *(A* optimality and completeness)*: Given a search problem with cost-function c(n). Suppose for every node n, the function h is such that h(n) <= the actual cost to go from n to a goal (that is, h never overestimates costs). Then A* is complete and optimal using that h (with c added in to use least-value of c(n)+h(n) for frontier-removal).

Such an h (satisfying the inequality in the theorem) is called **admissible**. We will not prove the theorem. But we can easily see that coming up with an admissible h need not be hard at all. For instance, both h1 and h2 above are so. Why is this? Consider h1, for instance: h1(n) is the number of tiles out of place in the state associated with node n. But clearly the required number of moves to the goal (true cost) is at least the number of tiles out of place. A similar argument works for h2.

The theorem is really good news! It says not only will A* work (get us to a goal, if there is a path to a goal at all) but it will get us the cheapest path to a goal (if there is more than one).

One can always of course just use h(n)=0 for all nodes n; but then A* is the same as uniform-cost search, and there is no real informed search going on.

Another kind of search: Constraint Satisfaction Problems

Some search problems arise not because we don't know how to get to a goal, but because the goal is not fully known except in terms of some general conditions ("constraints") it must satisfy. Then the "solution" is not a path taking us there but rather is simply the detailed specification of such a goal state. An example is that of *8 queens*: find a way to arrange 8 queens on a chessboard so that none is under attack from any other. Other examples are Sudoku and crossword puzzles. [[Why would the constraints in crossword puzzles be extremely hard to formalize, unlike 8 queens and Sudoku?]]

Such problems are called Constraint Satisfaction Problems (CSPs).

If we start with a *possible* goal state, but which turns out to have one or more constraint violations, we can try to adjust things so as to reduce the violations and get closer to a goal. Such a strategy is sometimes called "iterative improvement". It includes many particular methods; I will quickly comment on just one: min-conflicts.

Formally, a given CSP is described in terms of variables (parameters that are to be varied until they no longer are in conflict), and constraints (equations and other conditions involving the parameters that need to hold for a state if it is to be a goal). For 8 queens, these could be the numbers Q1,…,Q8, where each Qi specifies the row (Qi=r) where the queen in column i sits. Here r is one of 1,2,…,8.

Min-Conflicts is one such algorithm, that has proven to be of enormous practical value. For instance, scheduling the many experiments being applied for on the Hubble telescope used to be done by hand. It took 3 weeks to figure out a conflict-free schedule for just one week of actual use; but using Min-Conflicts the scheduling time was reduced to 10 minutes!

Roughly speaking, min-conflicts (i) simply selects at random one conflicted variable v; (ii) changes v to whichever of its alternative values has the fewest number of conflicts; and (iii) repeats (i) and (ii) over and over until no conflicts remain.

The idea sounds incredibly simplistic, so much so that we'd expect it not to work at all, or at best to run very slowly. But interestingly it tends to be extremely efficient, not only for 8 queens, or indeed n queens (for any n, even n=1,000,000) but also for many practical applications as well (such as the Hubble scheduling as noted above).