7 Reason

Reasoning is the process of applying the knowledge we've gathered to find solutions to problems. In many reasoning problems we begin in a certain configuration, and then apply various **operations** to change the configuration until we have reached a desired **goal** configuration. There may be more than one valid goal: perhaps we wish to find the one which is optimal in some fashion.

In some cases, the goal is itself what we seek. For example, in Sudoku, or a crossword puzzle, we start with an empty puzzle and change the configuration by filling in the blanks: ultimately we seek the goal state, or **solution** to the problem. Exactly *how* we arrived at the solution is relatively unimportant.

In other cases, the goal itself is completely uninteresting. What is interesting is *how* we achieved the goal. For example, it's obvious what the solution is to a Rubik's Cube. More interesting the sequence of operations required to solve the cube. This sequence is known as a **proof**.

In still other cases, both the goal and the proof are interesting to us, particularly if we're tasked to explain how we arrived at our answer. For example, when solving an integration problem, both the answer (the integration) and its proof are useful to us.

Not all reasoning problems fit into this mold: indeed one of the hottest areas in artificial intelligence, **probabilistic reasoning**, has solves problems by performing probabilistic queries to a distribution.

7.1 Search Spaces

Many of the above reasoning examples (excepting probabilistic reasoning) can be described as wandering through a graph of **states**—known as the **state space** or **search space**—starting at an initial state, until we have reached a state which matches some **goal predicate** (a function). We call this a **goal state**, and there may be more than one goal state. Our objective is often to find the *best* such goal state, or perhaps the one which is *closest* to our starting state in some fashion. The graph structure contains edges when we may perform an **operation** to transition from one state to another (the edges are commonly known as **transitions**. These edges may be weighted with the **cost** of performing the transition, and so in some cases our objective is to find the minimum-cost path from the start state to some goal state. If there is no such cost, we often assume that the weight of each edge is 1.0.

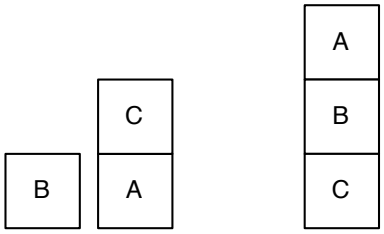


Figure 33 Typical states in the 3-block "Blocks World" problem search space. The left state is the so-called "Sussman Anomaly". We'll use that as our start state. The right state is the goal state.

One classic example of a reasoning problem of this kind is the 3-block "Blocks World" problem.[55] In this problem we have a table with three blocks, labelled A, B, and C. These blocks may be stacked on one another, or may rest on the table. The states in this problem are different stacking configurations. Two are shown at right, including the goal state (A stacked on B, which is then stacked on C). You can start in any start appropriate: we have chosen the so-called *Sussman Anomaly*[56] as our starting state.

[55] Believe it or not, this trivial problem used to be pretty important! Nowadays, it's the canonical example of a so-called **toy problem**.

[56] A famous configuration named after Gerald Sussman, who noted that this problem stymied early planning systems which made certain assumptions. See http://en.wikipedia.org/wiki/Sussman_Anomaly

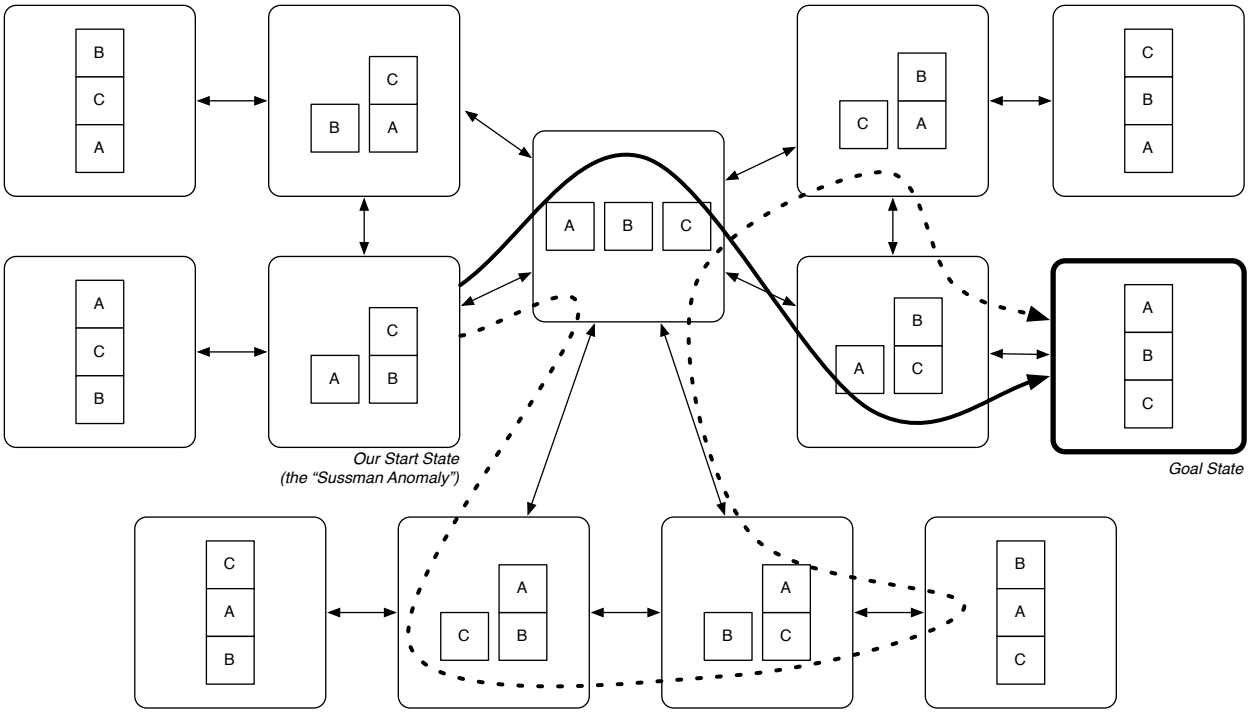


Figure 34 "Blocks World" search space. Shown are all 13 states, with the goal state specially marked. Valid transitions between the states are shown as edges. The solid thick path shows the optimal route (proof) from our start state to the goal state. A dotted thick path shows a suboptimal route.

We can perform three kinds of operations. First, if some block x is stacked on block y, and no block is stacked on x, then we may pick up x and **move it to the table**. Second, if a block x is sitting on the table, with no other block stacked on it, and there exists some other block y with no block stacked on it, we may pick up x and **stack it** on y. Third, if a block x is sitting on another block y, with no other block stacked on x, and there is another block z with no block stacked on it, then we may **move** x from on top of y to on top of z.

So there you have it. A search space of this type is defined by a graph with:

- A set of states.
- A state of edges describing the transitions among the states due to performing operations when in those states, possibly weighted with a cost.
- One or more goal states.
- An initial state.

The search space of the three-blocks Blocks World problem is shown in Figure 34, including optimal and suboptimal routes to the goal from our notional start state.

Search Trees Rather than view the state space as a graph, it's sometimes useful to view it as a **search tree** of nodes. Each node is labeled with a state (a state may appear in a variety of nodes throughout the tree). The children to a given node are all the states which are reachable from its

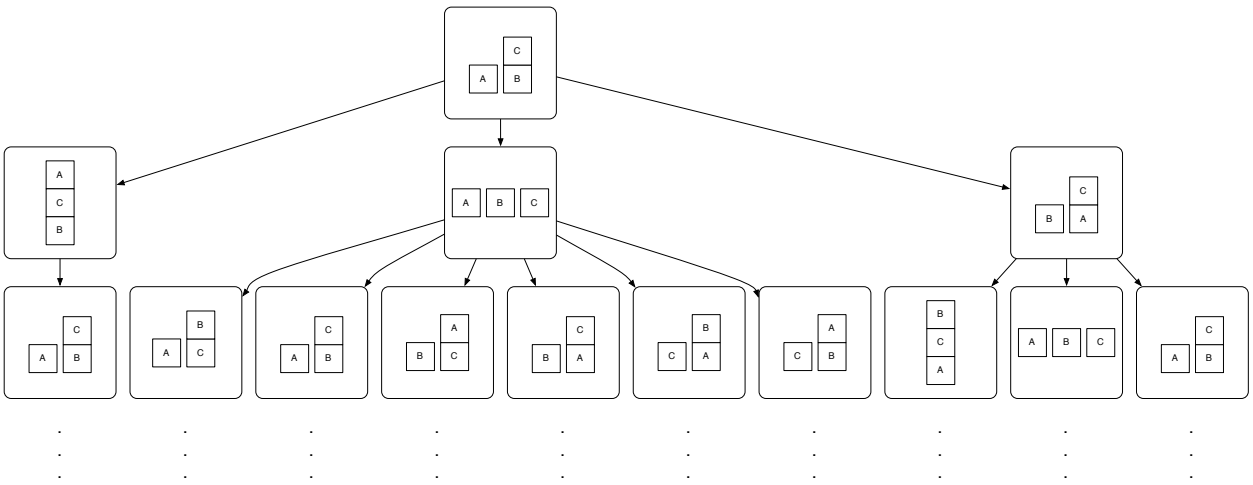


Figure 35 First three plies of the 3-block "Blocks World" search tree, rooted by assuming we're starting at the Sussman Anomaly. Further plies omitted for brevity (and sanity). Note that, as is the case here, states may appear more than once in the tree if the corresponding space has cycles. Also note that the branching factor is not consistent from state to state: one state has six children, while another has only one child.

state by performing a single operation. The root node is the start state. If the graph is cyclic, this tree may be infinite in size. Each layer in the search tree is sometimes called a **ply** (a term which has more relevance when we get to adversarial search). Figure 35 shows the first three plies for the 3-block Blocks World problem, starting at the Sussman Anomaly.

Completeness and Correctness Some, but not all, search problems have goals which may be viewed as being both *complete*, and *correct*. By *correct* we mean that none of the parts of the configuration are violating any rules. By *complete* we mean that all necessary parts of the configuration are present.

For example, consider the **N-Queens** problem, where we have an $N \times N$ chessboard filled with up to N queens. The objective is to place the queens on the board in such a way that no queen is attacking any other queen. If any queen is attacking another, the current queen configuration is incorrect. If we have less than N queens on the board, the configuration is incomplete. A goal configuration, with N queens, none of which is attacking any other, is both complete and correct.

In such problems, sometimes you can perform search in two ways. First, you could define a state space whose states describe correct but usually **incomplete** configurations, and transition operations add more and more parts to the configurations until we have reached a goal state. This kind of state space is usually a directed acyclic graph: there are no cycles because you cannot delete a queen. Figure 36 shows the N-Queens search process in such a space.

Second, you could define a state space whose states describe complete but usually **incorrect** configurations, and transition operations rearrange these configurations until we have a complete and correct solution. This kind of state space can have many cycles: you can easily make changes which wind up putting you back in your original configuration. Figure 37 shows the N-Queens search process in a space like this.

When a problem (such as the N-Queens problem) may be tackled in either of these search procedures, it's often best to configure it as the first: a DAG is often a less complex graph than its

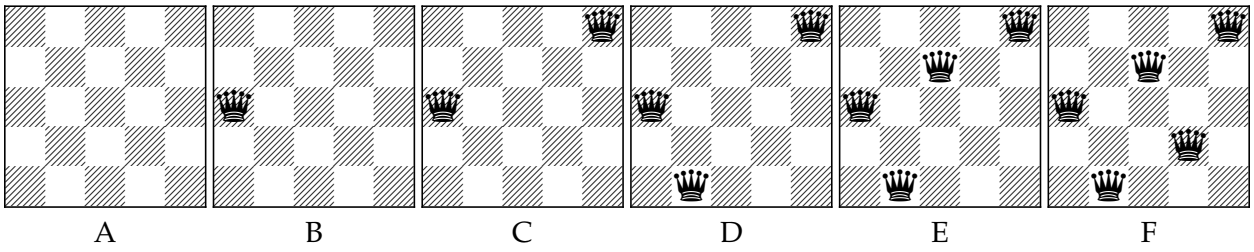


Figure 36 Searching through the space of partially correct solutions until a correct and complete solution is found (a goal). At no time is any queen threatening any other (each state is a *correct* solution). As is the case here, often such state spaces present themselves as directed acylic graphs (DAGs): because a queen is always added, you cannot repeat a previous state by progressing forward through the space.

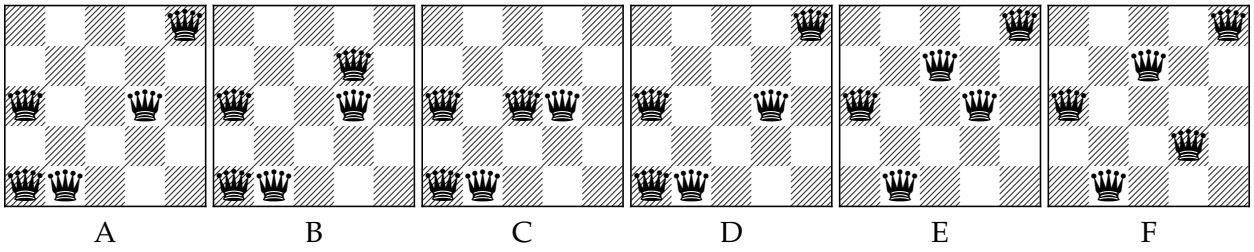


Figure 37 Searching through the space of complete solutions until a correct and complete solution is found (a goal). Queens are often threatening each other: but all five of them are on the board at all times (each state is a *complete* solution). As is the case here, often such state spaces can have cycles. For example, the moves $A \rightarrow B \rightarrow C \rightarrow D$ is a cycle. Note that just because the board has queens on it doesn't mean queens have to move like queens: in the above state space you just have to change the location of a queen (for example, $D \rightarrow E$ is an invalid queen move). The particular constraints on permissible operations depends on your problem.

corresponding cyclical graph. Many **constraint satisfaction problems** (where the route to the goal is less interesting than the goal itself) fall into this category.

On the other hand, problems where the route (proof) is more interesting usually are more straightforwardly tackled using the second of these search procedures. And then there are problems where "incomplete" configurations makes no sense at all. For example, the Rubik's Cube search space be configured as (1) add a block to the top back left, then (2) add a block to the top back right, and so on. But this is nonsense — we're not interested in breaking our Rubik's Cube down and reassembling it to reach the goal. Problems like these often are tackled using **state-space search**, which we get to next. Note that state-space search works fine for both search processes above: but it may not be as efficient as constraint satisfaction search for certain problems.[57]

[57] In fact, you can usually convert complete-solution searches into correct-solution searches. Here's how you do that for N-Queens. Instead of having a board configuration be your state, let your state be the *sequence of operations* which led to that configuration. To go from one state to another, you just perform another operation. Obviously this is a DAG because you always have to add more operations to your string, even if they're "revert" operations per se. This kind of conversion results in a state space commonly known as **plan space**, and it's the standard space used by partial-order planning systems.

7.2 State-Space Search

In State-Space Search, we're looking for a **path** from our start state to a goal state which has a **minimum cost**. If the edges are unweighted (so we've assigned them all 1.0) then the objective is of course to find the **shortest** path.

There are two common kinds of procedures used to perform state-space search: **uninformed** (or **brute force**) and **informed** (or **heuristic**) procedures. You've probably learned all about the first category. Before we get to the second category, we'll review some algorithms in the first-category: **Breadth-First Search** (or **BFS**), **Depth-First Search** (or **DFS**), and an important algorithm you've probably not seen before: **Iterative Deepening Search** or **IDS**.

7.2.1 BFS and DFS

BFS and DFS are two sides of the same coin. Consider this general-purpose search algorithm:

Algorithm 13 *State-Space Search*

```
1:  S ← initial state
2:  children(...) ← children function    ▷ Given a state, returns all its children states in the search tree
3:  Q ← queue                                                    ▷ Returns the next state to consider
4:  maxdepth ← maximum desired depth
5:  goal(...) ← goal predicate function                          ▷ Returns true if a given state is a goal state
6:  process(...) ← processing function ▷ Given a child state and its parent, update info about the child

7:  Q ← {S}
8:  loop
9:      if Q is empty then
10:         return FAILURE                                       ▷ We searched the whole space and found no goal
11:     S ← remove next item from Q
12:     if goal(S) then
13:         return S
14:     else if depth of S < maxdepth then
15:         for each child ∈ children(S) do
16:             process(child, S)
17:             if child ∉ Q then
18:                 Q ← Q ∪ {child}
```

In this algorithm, we repeatedly take an item out of our queue, check if it's the goal state, and if it is we return it. Otherwise if it's not too deep a node, we open it up (to get its children), and put the children into the queue if they're not already waiting there. And we repeat this process. If we run out of things in the queue, we give up.

We include a special function called process(...). This function is a hook which allows us to (later on) update information about children. For now, it's not going to do anything at all.

Breadth-first Search and Depth-first Search are both versions of exactly this algorithm. In depth-first search our priority queue is a simple (LIFO) *stack*. In breadth-first search our priority queue is a simple (FIFO) *queue*. That's the only difference between them! Typically BFS and DFS don't have a maximum depth (so we set it to ∞). As in:

Algorithm 14 *Depth-First Search*

1: $S \leftarrow$ initial state
2: $children(...) \leftarrow$ children function
3: $goal(...) \leftarrow$ goal predicate function
4: $maxdepth \leftarrow$ maximum desired depth ▷ Commonly set to ∞

5: $process(...) \leftarrow$ processing function (which does nothing at all)
6: $Q \leftarrow$ new (LIFO) stack
7: **return** State-Space Search with S, *children*, Q, *maxdepth*, *goal*, and *process*

Algorithm 15 *Breadth-First Search*

1: $S \leftarrow$ initial state
2: $children(...) \leftarrow$ children function
3: $goal(...) \leftarrow$ goal predicate function
4: $maxdepth \leftarrow$ maximum desired depth ▷ Commonly set to ∞

5: $process(...) \leftarrow$ processing function (which does nothing at all)
6: $Q \leftarrow$ new (FIFO) queue
7: **return** State-Space Search with S, *children*, Q, *maxdepth*, and *goal*, and *process*

So why pick one over the other? It depends on the nature of your problem.

7.2.2 Advantage of DFS over BFS

Consider the search tree at right, where G is the goal state. The tree is finite, implying that the equivalent graph has no cycles. The tree has a branching factor b and a depth of d.[58]

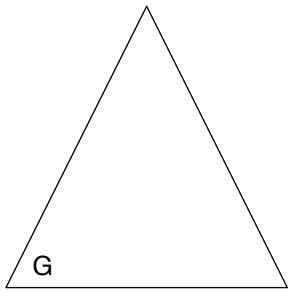


Figure 38 Finite search tree with a goal.

Our search algorithm has *just discovered* the goal state. At this point, DFS has in its stack a bunch of children. For each depth from 0 to $d-2$, DFS has $b-1$ children in the stack waiting to be visited. This is the worst-case scenario for this tree: DFS's stack holds at most $(b-1)(d-1)$ children.

Now consider breadth-first search in this situation. BFS's queue is holding every single node on the bottom ply of the tree. How many nodes are on a ply at depth (in this case) $d-1$? The answer is: b^{d-1}. So for a finite tree, DFS's stack holds at most $O(bd)$ nodes, but BFS's queue holds at most $O(b^{d-1})$ nodes. Quite a difference!

For the same reason, DFS has memory advantages over DFS if your goal is to **fully traverse a finite acyclic graph** rather than perform search. That is, you want to visit every node at least once. Here again, DFS would, at worse, require $O(bd)$ memory while BFS would require $O(b^{d-1})$.

[58] I am defining the **depth of a tree** as the number of nodes in the longest path from the root to a leaf. The **depth of a node** in the tree is the number of nodes in the path from the root to the node, not including the root itself. Thus the root is at depth 0. A tree of of depth 3 has nodes at depth 0, depth 1, and depth 2. It's sort of like an array, of length n which has elements at positions 0, 1, ..., $n-1$.

7.2.3 Advantage of BFS over DFS, and Histories

Now imagine our search space has cycles in it, resulting in an infinite search tree. The goal state is located in the position on the right side of the search tree marked G, as shown in the Figure at right. Let us presume that the order in which we select the children is left-to-right. Depth-first search will search the left side of this infinite tree deeper and deeper. If the tree is infinite, depth-first search will *never find* G. It'll just keep on going down the left edge of the tree.

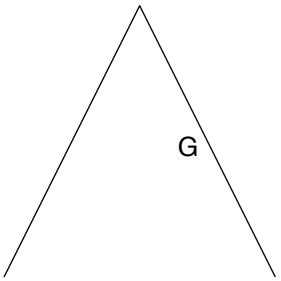


Figure 39 Infinite search tree with a goal.

But BFS will find G. This is because BFS searches ply by ply (layer by layer). If there's a goal, obviously it must exist at some ply l_G. When BFS reaches l_G, it'll find G. Thus in general, BFS is guaranteed to find a goal if it exists. DFS is not: it could go on searching forever.

You can fix DFS's problem here, by modifying the State-Space Search function to only add children to Q that it has *never encountered before*. We do this by adding a **history** to the search algorithm which keeps track of every state we've visited so far. The history will also indicate, for each such state, the parent of that state (that is, the previous state which led us there). We'll store this mapping as $\langle state \rightarrow parent$. A history is usually implemented as a hash table, with *state* being the key and *parent* being the value. The parent of the start state is an arbitrary object we'll define as "$\square$". Here is a revised version of the algorithm. Notice the four modified lines, each marked with $\star$.

Algorithm 16 *State-Space Search with History*

```
 1: S ← initial state
 2: children(...) ← children function   ▷ Given a state, returns all its children states in the search tree
 3: Q ← queue                                                  ▷ Returns the next state to consider
 4: maxdepth ← maximum desired depth
 5: goal(...) ← goal predicate function                        ▷ Returns true if a given state is a goal state
 6: process(...) ← processing function ▷ Given a child state and its parent, update info about the child

 7: H ← {⟨S → □⟩} ▷ History, as a hash table. ⟨a → b⟩ means a is the key and b is the value.        ⋆
 8: Q ← {S}
 9: loop
10:     if Q is empty then
11:         return FAILURE                  ▷ We searched the whole space and found no goal
12:     S ← remove next item from Q
13:     if goal(S) then
14:         return S and H                                                                  ▷ ⋆
15:     else if depth of S < maxdepth then
16:         for each child ∈ children(S) do
17:             process(child, S)
18:             if child is not a key in H then                                             ▷ ⋆
19:                 Q ← Q ∪ {child}
20:                 H ← H ∪ {⟨child → S⟩}                                                   ▷ ⋆
```

And our revised DFS is now:

Algorithm 17 *Depth-First Search with a History*

1: $S \leftarrow$ initial state
2: $children(...) \leftarrow$ children function
3: $goal(...) \leftarrow$ goal predicate function
4: $maxdepth \leftarrow$ maximum desired depth ▷ Commonly set to ∞

5: $process(...) \leftarrow$ processing function (which does nothing at all)
6: $Q \leftarrow$ new (LIFO) stack
7: **return** State-Space Search with a History, with S, *children*, Q, *maxdepth*, *goal*, and *process*

Now even if we have a graph with cycles in it, DFS is guaranteed to terminate because it won't keep traversing those cycles over and over again, and if G exists it'll find it. However, DFS will still fail if we have an infinite search space: DFS may search forever deeper in the space even if G is right next to the root, simply because DFS chose the wrong initial child to evaluate. On the other hand, BFS will find G in an infinite search space as long as G is reachable.

Note that a history benefits BFS as well: if you have a graph with a lot of cycles, a history can prevent BFS from including those cycles over and over again in its search.

Other Failings We now have algorithms which are **sound**: if a solution goal exists, they'll find it. But If goal does *not exist*, and the graph is infinite, BFS, like DFS, is not guaranteed to tell you this fact. They'll both get stuck in infinite loops. This is closely related to the notion of **semidecidability** in computability theory: procedures which find answers if they exist, but cannot tell you if they do *not* exist.

Another place where these algorithms may fail is if *children(...)* returns infinite numbers of nodes. For example, consider if we're doing mathematical integration. A state, in this context, would be a mathematical equation. Our objective is to perform operations on an equation until it's in the form which allows us to compute the integral. What kinds of mathematical operations could we do? We could:

- Multiply both sides by 2
- Multiply both sides by 2.5
- Multiply both sides by 3
- Take the sine of both sides
- Take the cosine of both sides
- Add 72 to both sides
- Add 72.1 to both sides
- Subtract both sides from x
- Subtract both sides from x^2
- ...

It should be clear that the number of possible operations is huge, likely infinite if we're not careful. So BFS and DFS may both fail in this case.

Proofs So why is our history a mapping, rather than just a set of nodes? After all, so far we're just checking to see if a given *child* is in the history or not. And why are we returning the history along with the solution?

Answer: we're doing it this way because we can use the history to extract the proof. Each node is pointing to the node which led to it. This allows us to work our way from the solution all the way back to the initial state. For example we could use this information to extract a list of all states starting at the initial state and leading to the goal state, along these lines:

Algorithm 18 *Extract a Proof*

1: $H \leftarrow$ history
2: $S \leftarrow$ discovered goal state

3: $P \leftarrow \langle S \rangle$ proof, starting with S
4: **while** $S \neq \square$ **do**
5: $\quad X \leftarrow$ the value associated with the key S in H ▷ That is, $\langle S \rightarrow X \rangle \in H$
6: $\quad$ Append X to the end of P
7: $\quad S \leftarrow X$
8: $P \leftarrow$ Reverse(P)
9: **return** P

7.2.4 Computational Complexity of BFS and DFS with Histories

Let's consider the worst-case computational complexity of BFS and DFS on a finite, full tree. We'll put the goal in its worst possible location, at the bottom right, as shown in the Figure at right. This tree has depth d and branching factor b.

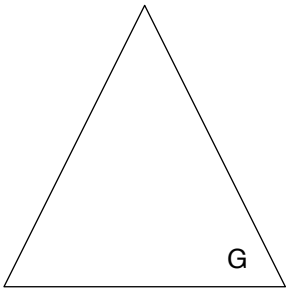


Figure 40 Finite search tree with a goal.

So we have just discovered the goal state G after traversing the tree. At this point in time, both BFS and DFS (augmented with histories) have visited every node in the tree exactly once. How many nodes are there in a full tree with branching factor b and depth d? The answer is: it's the sum, from 0 to $d-1$ (each ply in the tree), of the number of nodes in that ply. For any given ply of depth i, the number of nodes is b^i. So the total number of nodes is $\sum_{i=0}^{d-1} b^i$. This is the Geometric Series,[59] and it's equal to $\frac{b^d-1}{b-1}$, which is $O(b^{d-1})$. Not convinced? Here's the proof:

Proof

$$\sum_{i=0}^{d-1} b^i = b^0 + b^1 + ... + b^{d-1}$$

$$b \sum_{i=0}^{d-1} b^i = \sum_{i=0}^{d-1} b^{i+1} = b^1 + b^2 + ... + b^d$$

$$(b-1) \sum_{i=0}^{d-1} b^i = b \sum_{i=0}^{d-1} b^i - \sum_{i=0}^{d-1} b^{i+1} = (b^1 + b^2 + ... + b^d) - (b^0 + b^1 + ... + b^{d-1}) = b^d - b^0 = b^d - 1$$

$$\sum_{i=0}^{d-1} b^i = \frac{b^d - 1}{b-1} \simeq O(b^{d-1}) \qquad \square$$

[59] See http://en.wikipedia.org/wiki/Geometric_progression

7.3 Iterative Deepening Search

A more recent search algorithm, known as **Iterative Deepening Search** (or **IDS**) has the same computational complexity as BFS and DFS for trees with a branching factor $b \geq 2$. Like BFS it's guaranteed to find the goal if it can be reached even in an infinite graph, and it has the memory requirements of DFS *without* a history! That is, for $b \geq 2$, it's as good as or superior to BFS and DFS in every way. It's also dead simple.[60]

Iterative Deepening works like this: for each depth from 1 on up, do DFS up to that depth. Repeat until you find the goal. Here it is using our general-purpose search algorithm:

Algorithm 19 *Iterative Deepening Search*

```
1: S ← initial state
2: children(...) ← children function
3: goal(...) ← goal predicate function
4: maxdepth ← maximum desired depth                    ▷ Commonly set to ∞

5: process(...) ← processing function (which does nothing at all)
6: for depth from 1 to maxdepth do
7:     Q ← new (LIFO) stack
8:     result ← State-Space Search with a History with S, children, Q, depth, goal, and process
9:     if result ≠ FAILURE then
10:        return result
11: return FAILURE
```

Notice that IDS is passing in *depth* as the maximum depth parameter in State-Space Search. Essentially IDS is doing a depth-limited DFS to nodes of depth 0, then repeats the DFS for nodes of up to depth 1, then repeats it for nodes of up to depth 2, and so on. Because IDS uses DFS (see the stack?) as its search algorithm, its worst-case memory usage is as good as DFS. Because IDS increases its search ply-by-ply, it's also guaranteed to find the goal at some point, just like BFS does.

Just like BFS, IDS **doesn't need a history** to guarantee that it'll find a goal, but it sure can use it to improve efficiency by cutting off repeated cycles. So we're using it here.

Complexity But IDS repeats the same DFS search over and over and over again! So IDS revisits nodes high-up in the tree over and over again. That's *got* to be wasteful, right?

Nope. Let's presume we're using a history. Now consider our worst-case tree again, shown at right. This tree has depth d and branching factor b. Let's see how bad Iterative Deepening is on this tree. Recall that the cost of doing DFS (without a history) for a tree of depth d is $\frac{b^d-1}{b-1}$.

Since Iterative Deepening repeatedly calls DFS over and over again for increasing tree depths, thus the total number of nodes it visits before it finds G is the sum, from ply 1 to d, of the number of nodes of trees up to each depth. This is $\sum_{i=1}^{d} \frac{b^i-1}{b-1}$, and it is also $O(b^{d-1})$. Here's the proof:

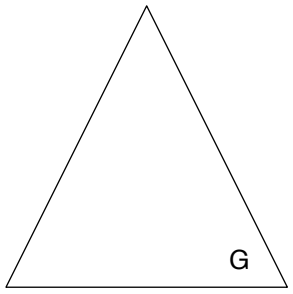


Figure 41 Finite search tree with a goal.

[60] Ask yourself: why didn't you learn this algorithm in algorithms class?

Proof First we derive the value of $\sum_{i=1}^{d} \frac{b^i-1}{b-1}$.

$$\sum_{i=1}^{d} \frac{b^i-1}{b-1} = \frac{1}{b-1}\sum_{i=1}^{d} b^i - 1 = \frac{\left(\sum_{i=1}^{d} b^i\right) - \sum_{i=1}^{d} 1}{b-1} = \frac{\left(\sum_{i=1}^{d} b^i\right) - d}{b-1}$$

Now we derive the first sum, $\sum_{i=1}^{d} b^i$.

$$\sum_{i=1}^{d} b^i = b^1 + b^2 + ... + b^d$$

$$b\sum_{i=1}^{d} b^i = \sum_{i=1}^{d} b^{i+1} = b^2 + b^3 + ... + b^{d+1}$$

$$(b-1)\sum_{i=1}^{d} b^i = \left(b\sum_{i=1}^{d} b^i\right) - \sum_{i=1}^{d} b^i = (b^2 + b^3 + ... + b^{d+1}) - (b^1 + b^2 + ... + b^d) = b^{d+1} - b^1 = b^{d+1} - b$$

$$\sum_{i=1}^{d} b^i = \frac{b^{d+1}-b}{b-1} = \frac{b(b^d-1)}{b-1}$$

Putting it together:

$$\sum_{i=1}^{d} \frac{b^i-1}{b-1} = \frac{\frac{b(b^d-1)}{b-1} - d}{b-1} = \frac{\frac{b(b^d-1)}{b-1} - \frac{(b-1)d}{b-1}}{b-1} = \frac{\frac{b(b^d-1)-(b-1)d}{b-1}}{b-1}$$

$$= \frac{b(b^d-1)-(b-1)d}{(b-1)^2} = \frac{b^{d+1}-b-bd-1}{b^2-2b+1} \simeq O(b^{d-1}) \qquad \square$$

Exactly the same as DFS and BFS! Assuming that you're searching trees with a branching factor of 2 (binary) or larger, of course. If your branching factor is less than 2, then the other factors can't be removed from the fraction and Iterative Deepening becomes more expensive than DFS or BFS. But in Artificial Intelligence we're almost always interested in search spaces with big, nasty branching factors.

So in short: if the branching factor $b \geq 2$:

- Like BFS, IDS is guaranteed to find a goal even without a history.
- IDS, like DFS, is $O(bd)$ memory usage without a history.
- IDS, like DFS and BFS, is $O(b^{d-1})$ memory usage with a history.

7.4 Informed Search

The problem with the methods above is that they don't scale with large spaces or (even more importantly) large branching factors, because they essentially search the *entire* space until they find the solution, without considering *which regions of the space* they should examine first.

It turns out that you can inject knowledge about your problem to give hints to these search methods to guide them to try certain regions of the space which are likely to be more promising.

This knowledge may or may not be correct: the only promise is that it is *correct enough* to be useful, generally, in reducing the scope of the search.

Before we get into that, let's consider an extension of the BFS algorithm which tries to find the minimum-cost path to a goal when the edges are weighted. We will assume that the edges all have weights ≥ 0.

In this algorithm, instead of a queue or stack, we will use a priority queue (perhaps a heap). When states are added to the queue, they are assigned the **total cost** of the lowest-cost path known to reach them from the start state. We pull out of the queue the states with the lowest such cost. How do we compute this cost? Simple: if we are adding a child c to the queue, its cost is the cost of its parent, plus the weight of the edge from the parent to c.

The idea is to try expanding the lowest-cost nodes first, in the hopes that they will lead us to routes the goal which are lower in total cost.

This is basically Dijkstra's algorithm. Because Dijkstra's algorithm is normally used to find the shortest paths to all nodes in the graph, it runs until the graph is exhausted. We won't do that: we'll just run until we find a goal node. Thus our algorithm is a **truncated variant of Dijkstra's algorithm**. It's possible that we may need to update the cost (g value) of a node in the queue if we discover a lower-cost route (what Dijkstra's algorithm calls **relaxation**). When we do that, we'll also need to update the history so that the parent of the node is its new lower-cost parent. We will do this during the process(...) function.[61]

You may ask yourself: what if have taken a node n out of the queue, then processed it, and only later on discover a lower-cost route to n? How do we update it, and all of its descendants, with this new information? The answer is: this will never happen. Let's say that we have chosen to extract n from the queue. We did so because it's the lowest cost node at present. The only way we can discover a lower-cost route to it is by later opening some *other* node m which leads to n. But m already has a higher cost than n, which is why we chose to extract n rather than m in the first place. Since edge weights are ≥ 0, it's not possible for m to lead to a lower-cost route to n than we'd already discovered.

Keeping Track of Who's Best Recall that we need to maintain the lowest cost route from the initial state to n. Here's how we do that. We define the minimum cost of the path from the start state to a given node n to be $g^\star(n)$, and the best known cost of a path to n to be $g(n)$. Clearly $g(n) \geq g^\star(n)$. Furthermore, $g(\textit{initial state}) = g^\star(\textit{initial state}) = 0$. We don't use $g^\star(n)$ in the algorithm, but we *do* use $g(n)$, and more importantly, we need to keep track of it and update it.

We'll use the process(...) function to do this. process(...) takes two arguments: a child state n and its parent p. In the process(...) function we'll do the following. If n is brand new, its $g(n)$ value is set to $g(p) + \textit{cost}(p \to n)$, where $\textit{cost}(p \to n)$ is the cost of the edge $p \longrightarrow n$. If we have seen n before but we're processing it again, then $g(n) \leftarrow \min(g(n), g(p) + \textit{cost}(p \to n))$, and furthermore update the history H so it contains $\langle n \to p \rangle$. In other words, we store in $g(n)$ the cost of the lowest-cost route from the root to n that we've ever seen, and record p as the new best parent for n.

Anyway, let's assume that $g(n)$ is stored away in, say, the History, or it's part of the queue. Thus the variant of Dijkstra's algorithm may be described as follows:

[61] Oh, so *that's* what it's for.

Algorithm 20 *Dijkstra's Algorithm (Truncated Variant)*

1: $S \leftarrow$ initial state
2: $children(...) \leftarrow$ children function
3: $goal(...) \leftarrow$ goal predicate function
4: $maxdepth \leftarrow$ maximum desired depth ▷ Commonly set to ∞

5: $process(...) \leftarrow$ processing function ▷ Given a child and its parent, updates $g(child)$ as described above
6: $g(S) \leftarrow 0$
7: $Q \leftarrow$ new priority queue based on $g(n)$ (lower is better) ▷ $g(n)$ sometimes revised for some $n \in Q$
8: **return** State-Space Search with a History with S, *children*, Q, *maxdepth*, *goal*, and *process*

What happens if all the edges have a cost of 1? Then this algorithm is just a form of BFS.

7.4.1 Heuristic Search: A*

Dijkstra's Algorithm helps us consider routes which are potentially less expensive than others. But we can easily be fooled by it: for example, perhaps a route is very inexpensive early on, then becomes quite expensive later. We'll have wasted a lot of time exploring that region of space only to find it not panning out.

In addition to an estimate of how costly it's been to get to a given node, what we need is a guide which suggests to us how far we have to go to reach the goal from there on. This is an example of a **heuristic**.

A heuristic is a rule of thumb. It's a guide which often or usually helps us zoom in on the goal, but is not guaranteed to work. In AI we use heuristics to help us narrow our search on average, though sometimes it doesn't work.

For example, consider the Figure at right. We are searching through the space of robot locations to find the shortest route (the proof) from the start position to a goal position. At present we have three possible positions. Which should we try extending first? We can use a heuristic to help us here. For example, we could use the as-the-crow-flies distance from a given position to the goal as our heuristic. We add together the length of the path from the start to a position n (A, B, or C in this example), plus this heuristic distance from n to the goal, and use that as our estimate of the full cost of the corresponding path. We extend whichever position has the lowest such estimate.

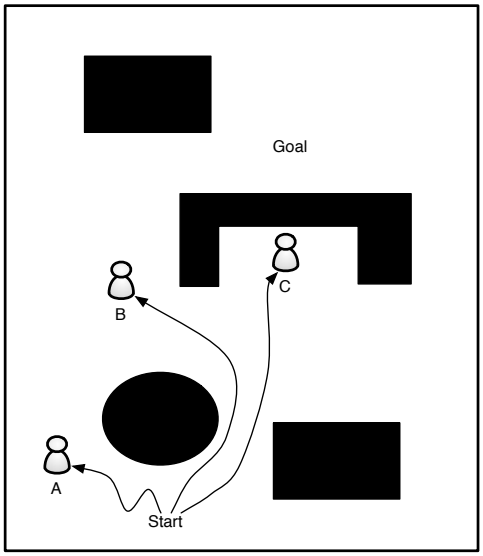


Figure 42 A heuristics example. The three points A, B, and C reflect the end-points of three routes we have discovered in our search so far. We want to extend one of these routes to reach the goal. Which should we select first?

This actually isn't a bad heuristic. For example, it'll help us see that A is a worse choice than B. But it sometimes misleads us, as in the case of C, which has the best heuristic, and thus the best estimate, but is obviously inferior to B.

When we modify Dijkstra's to include this heuristic, under certain constraints about the heuristic we choose, we arrive at a form of search called **A* Search**.

The A* Algorithm The objective of the A* Search algorithm is to guarantee that we will find a route to a goal if one exists, and if there is more than one route to a goal, we will find the minimum-cost route to the goal before we discover any other routes. In order to understand the algorithm, recall that the *true* minimum cost from the start state to a given state n is called $g^\star(n)$, and that the cost of the best known route (so far) from the start state to n is called $g(n)$. Further recall that it's always the case that $g^\star(n) \leq g(n)$.

We will make similar guarantees about our heuristic. Imagine if we had a magic oracle which told us the minimum remaining cost from n to a goal node. If there is no possible path from n to a goal node, then the cost will be ∞. We call the value the oracle tells us $h^\star(n)$. Using this magic oracle, we could ascertain the cost of the minimum cost path $f^\star(n)$ which runs from the start state to the goal state and *passes through n*. $f^\star(n)$ is simply defined as $g^\star(n) + h^\star(n)$. If we sorted our priority queue by $f^\star(n)$, we'd zoom right in on the optimal route to the goal.

Sadly, $h^\star(n)$ is not available to us. But maybe we might be able to use a **heuristic function** to help us out. We define the function as $h(n)$, and it gives us a rough estimate of $h^\star(n)$. Instead of $f^\star(n)$, we'll use $f(n) = g(n) + h(n)$ to sort our priority queue.

Admissibility In order for us to make the guarantees A* makes (always finding the *minimum-cost* route to a goal) we must place a constraint on $h(n)$. Specifically, it must be the case that:

$$\forall n : 0 \leq h(n) \leq h^\star(n)$$

That is, $h(n)$ must always **underestimate** the true remaining cost to the goal. This is called the **admissability constraint** on $h(n)$. Note that this also implies that $h(goal)$ must be equal to 0, since $h^\star(goal)$ is obviously 0.

Without admissibility, it's not A*.

Monotonicity There's another constraint on your heuristic which isn't mandatory, but you'll regret not having it. If you can make the **monotonicity** guarantee about $h(n)$, A* becomes a lot easier to compute.

Recall that as Dijkstra's algorithm progressed, it might need to update the $g(n)$ values in nodes in the queue, but not any nodes which have already been processed and are no longer in the queue. This was because by the time a node n was processed, we'd found the minimum $g(n)$ value (that is, $g^\star(n)$), because we'd have already processed all possible nodes m which might lead to a better $g(n)$.

This is no longer the case for A*. We don't just sort the nodes in the queue on $g(n)$, but on $f(n) = g(n) + h(n)$. Given a crummy heuristic $h(n)$, we might process n before we can guarantee that $g^\star(n)$ has been finalized. Thus we discover new, better $g(n)$ values, we'll have to revise $g(n)$ plus the $g(...)$ value of any descendant of n we've discovered since then. This could lead to a *lot* of bookkeeping, and a much more expensive algorithm, in terms of computational complexity.

Thankfully we can eliminate all this with a simple monotonicity constraint placed on our $h(n)$. Let m be an immediate child reachable from n by one edge. Then:

$$\forall n, m : h(n) - h(m) \leq cost(n \to m)$$

and

$$\forall goal : h(goal) = 0$$

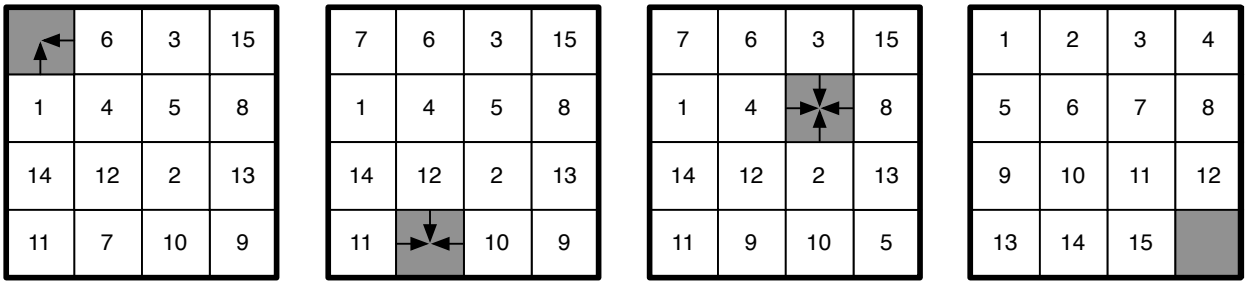


Figure 43 Configurations of the 15-Puzzle. The first three configurations show which tiles may slide into the empty space. The final configuration is the goal configuration.

Another way of putting it is: monotonicity states whether $h(...)$ underestimates the change in cost from node to immediate node, that is, it is the admissibility requirement in microcosm. From this it's easy to show that any monotonic heuristic is automatically admissible.

Armed with the monotonicity guarantee, we can describe A* easily in our search framework:

Algorithm 21 *A* Search (Assuming Monotonicity)*

1: $S \leftarrow$ initial state
2: $children(...) \leftarrow$ children function
3: $goal(...) \leftarrow$ goal predicate function
4: $maxdepth \leftarrow$ maximum desired depth ▷ Commonly set to ∞
5: $h(...) \leftarrow$ monotonic and admissible heuristic function

6: $process(...) \leftarrow$ processing function ▷ The same function as used in Dijkstra's Algorithm
7: $g(S) \leftarrow 0$
8: $f(...) \leftarrow$ defined as $f(n) = g(n) + h(n)$
9: $Q \leftarrow$ new priority queue based on $f(n)$ (lower is better) ▷ $g(n)$ sometimes revised for some $n \in Q$
10: **return** State-Space Search with a History with S, *children*, Q, *maxdepth*, *goal*, and *process*

If our heuristic were admissible but not monotonic, we'd have to modify the search algorithm itself to update already-processed states, which would be more than a bit of a mess. So stick with monotonicity.

Admissability isn't a panacea. There are lots of smart heuristics which aren't admissible. There are lots of poor heuristics which *are* admissible. For example, what if we set $h(n) = 0$ for all n? This is an admissible heuristic — and a monotone heuristic — but it's not very good: it's just Dijkstra's algorithm.

So why do we care about admissible heuristics? Because if a heuristic is admissible, then A* is *guaranteed* to find the lowest-cost path to the goal.

We want an admissible heuristic $h(n)$ which is as close to $h^\star(n)$ as possible, that is, $h^\star(n) - h(n)$ is minimized. How do we do this? Let's consider an example. The **15-puzzle** is a classic sliding-block puzzle consisting of 15 tiles arranged in a $4x4$ grid, with one empty space. You can slide a tile into that space from an adjoining space. The objective is to reach a configuration resembling the far-right configuration in Figure 43.[62]

[62] There's also a simpler $3x3$ version called, unsurprisingly, the **8-puzzle**.

In the search space of the 15-puzzle, a state is a puzzle configuration, and a transition operation is sliding one tile into the empty space. All transition edges will be weighted with a cost of 1 (no tile is "costlier" to slide than any other). The goal state is the far-right configuration in Figure 43.[63]

So what would constitute an admissible heuristic for the 15-puzzle? Consider using the **number of tiles out of place**, not including the empty space. For the goal, this is 0 of course. For the leftmost figure in Figure 43, the value is 13, since tiles 3 and 8 are in their right places (and no one else is). It's easy to see that this heuristic is admissible. $h^\star(n)$ for the 15-puzzle is the number of moves necessary to solve the puzzle. Obviously if t tiles are out of place, then *at least* t moves are necessary to achieve the goal configuration. Since $h(n) = t$, it's clear that $0 \leq h(n) \leq h^\star(n)$.

Is this heuristic monotonic? Well, consider the leftmost state in Figure 43 again. We will call it state n. There are two children states from this state: one where 6 has been slid to the left, and one where 1 has been slid up. Let's consider one of them, which we call state m. The cost to go from n to m is 1 (one tile slide). And at *best*, we have improved our heuristic by 1: for example, by sliding the 1-tile up, our heuristic improved. But by sliding the 6-tile to the left, our heuristic didn't improve. Thus $h(m) - h(n) \leq cost(n \to m)$ for any n and m. Furthermore $h(goal) = 0$ (no tiles are out of place. Our heuristic is monotonic.

But is it a very *good* heuristic? Can we make a better one? How about this: the sum, over every tile, of the **Manhattan distance** between where the tile is presently and where it should be in the goal configuration. The Manhattan distance is the number of horizontal moves, plus the number of vertical moves, to get the tile in the right position, as shown in Figure 44 at right.

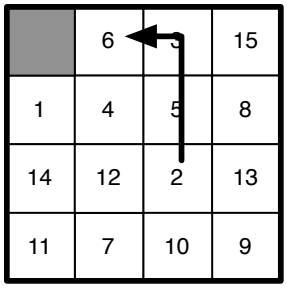


Figure 44 Manhattan Distance. To move the 2-tile to its proper place requires two moves up and one move to the left, for a Manhattan distance of 3.

This heuristic is clearly closer to $h^\star(n)$ than the sum of tiles out of place. In the sum-tiles-out-of-place heuristic, if a tile is out of place, it always counts for exactly 1 even if it's *really* out of place, requiring a large number of potential moves to get it in the right spot. But using the Manhattan distance heuristic, a tile contributes its Manhattan distance, which is the minimum number of possible moves required, to get it in the right place: it's ≥ 1. Remember, we want $h(n)$ to be as large as possible and still be $\leq h^\star(n)$. For all possible configurations n, the sum Manhattan distance on is at least as large, and nearly always larger, than the sum tiles out of place.

Okay, so sum-Manhattan-distance is **more informative** than sum-tiles-out-of-place. But is it admissible? Well, for each tile, the Manhattan distance of the tile is the *minimum* number of moves to *theoretically* get it to the right place. So Manhattan distance cannot possibly *overestimate* the total number of moves of tiles. Is it monotonic? Absolutely: but I'll leave that as practice. Hint: the argument is similar as the one for the sum-tiles-out-of-place.

7.4.2 Greedy Best-First Search

A*and Dijkstra's algorithm are both examples of **best-first search**, the family of algorithms where we use a priority queue which sorts according to some function $f(n)$. In Dijkstra's, $f(n) = g(n)$. In A*, $f(n) = g(n) + h(n)$.

[63]Note that if you consider *every possible rearrangement* of tiles, the 15-puzzle search space actually consists of disjoint subgraphs: that is, there are configurations from which it is impossible to reach the goal configuration. This is also the case with the Rubik's cube: try rotating a single cube sometime, and see if you can still solve the puzzle.

What if we just used $f(n) = h(n)$? That is, what happens if we don't care how long it took us to get to a node, but only how good a node looks *now*? This is commonly known as **greedy best-first search** (or sometimes just **greedy search**). Why would you want to use something like this? Here's an example. Suppose you're building a robot bloodhound. As it hunts around on a field, it gets more and more of a smell it's looking for. In this scenario, your bloodhound doesn't care about how long it got to the places it's now considering. It *only* cares about which ones look most promising. That's your $h(n)$.

Or suppose you are making a web spider. The spider's job is to hunt through web pages on the internet until it finds one which looks promising. It has a heuristic function which suggests to what degree a given page is likely to be in the vicinity (on the hyperlink graph of web pages) of promising pages. It doesn't matter how long it took to get to a given web page: all that matters is which page is the best to head off from *now*.

So depending on the task at hand, we might wish to have informed search routines which consider only how much it cost to get to a given state; or only how promising a state is *now*; or a combination of the two.

7.4.3 Proof of the Correctness of A*

Here we're going to prove that A* is guaranteed to discover the goal state with the least-cost distance to the start state, before it discovers any other goal state.

Here's our proof. Remember that for any state n, the following three things are true:

$$f^\star(n) = g^\star(n) + h^\star(n)$$

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

$$0 \leq h(n) \leq h^\star(n)$$

Furthermore, if n is a goal, then $h(n) = h^\star(n) = 0$.

We want to prove that A* search will always find the lowest-cost path from the start state to a goal state. Let's assume that this is not the case, and derive a contradiction.

Imagine that we have just extracted from the queue and discovered a suboptimal goal before finding the optimal goal. Let's call it the "bad goal" or b. At this point, at least one node along the optimal path must exist in our priority queue. To see this, consider: initially the start state was in the queue (it's on the optimal path). When we extracted it and processed it, we added a child which must be on the optimal path. If we extracted that child, we must have added *its* child which is on the optimal path, and so on, clear down to the "good goal".

Let's call this optimal node o. Why did we pick b instead of o? Because apparently

$$f(b) \leq f(o)$$

But we know that $g(o) = g^\star(o)$, because we've measured it exactly by successively picking nodes from the start, through o's parent, and finally to o (it's on the optimal path remember). Since we know that $\forall n : 0 \leq h(n) \leq h^\star(n)$ due to admissibility, therefore we also know that $f(o) = g(o) + h(o) \leq g^\star(o) + h^\star(o) = f^\star(o)$.

So therefore

$$f(b) \leq f(o) \leq f^\star(o)$$

Because b is a goal state, therefore we know that $h(b) = h^\star(b) = 0$. So therefore:

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

Furthermore,

$$f^\star(b) = g^\star(b) + h^\star(b) = g^\star(b)$$

Since $g^\star(b) \leq g(b)$, we thus conclude that

$$f^\star(b) \leq f(b) \leq f(o) \leq f^\star(o)$$

Whoa, wait a minute. This tells us that the actual cost $f^\star(b)$ from the start to the suboptimal goal b is *at least as good as* than the actual cost $f^\star(o)$from the root to the **optimal** goal passing through o (which is on the optimal path). That's impossible, because b is suboptimal. So we have a contradiction, and our proof is done. □

8 Constraint Satisfaction

In 2004 Google launched a campaign to recruit employees called the Google Labs Aptitude Test (or GLAT).[64] The test was patterned after standardized tests, and its very first question was:

> 1. Solve this cryptic equation, realizing of course that values for M and E could be interchanged. No leading zeros are allowed. WWWDOT − GOOGLE = DOTCOM

This is a **cryptarithmetic puzzle**. You are meant to replace each letter with a unique digit such that the resulting equation is correct. You may not use the same digit for two different kinds of letters. That is, if you substitute 8 for W, you cannot also substitute 8 for G. It's more helpful, for purposes of discussion here, to rearrange it as addition, that is, GOOGLE + DOTCOM = WWWDOT. Here's the puzzle and its two solutions.[65]

Puzzle

	G	O	O	G	L	E
+	D	O	T	C	O	M
	W	W	W	D	O	T

Solution 1

	1	8	8	1	0	3
+	5	8	9	4	8	6
	7	7	7	5	8	9

Solution 2 (M and E swapped)

	1	8	8	1	0	6
+	5	8	9	4	8	3
	7	7	7	5	8	9

Cryptarithmetic is an example of a **constraint satisfaction problem**, where the objective is to replace a set of **variables** with certain **values** (a process called **variable assignment**) such that those values meet certain **constraints** (in this case, the rules of mathematics). The set of legal values for each variable may vary from variable to variable and is known as the **domain** of the variable. Constraint satisfaction problems differ from, say, the 15-puzzle (or Rubik's Cube) in a fundamental way. In the 15-puzzle we're not interested in the final solution (we *know* what the final solution looks like). Instead, we're interested in how to solve the puzzle (the proof). But in a cryptarithmetic or other constraint satisfaction problem we're usually not interested in the proof at all: we just want to know what the final solution looks like. Although you can solve constraint satisfaction problems using state-space search, this fundamental difference generally leads to a set of approaches to solving the problem in a more efficient manner. I call these techniques **constraint-based search**.

To formulate the variables, values, and constraints for a cryptarithmetic problem, we must first realize that not all variables are spelled out. For example, in the Google puzzle, in addition to the variables G, O, L, E, D, T, C, M, W, and D, there are also **carry variables** $x_1, ..., x_5$ which are set to either 1 or 0, as shown at right. Given this configuration of the puzzle, we can formulate its components as:

	x_1	x_2	x_3	x_4	x_5	
	G	O	O	G	L	E
+	D	O	T	C	O	M
	W	W	W	D	O	T

Figure 45 Variables for the Google cryptarithmetic problem.

- **Variables**: G, O, L, E, D, T, C, M, W, D, x_1, x_2, x_3, x_4, x_5

[64] http://googleblog.blogspot.com/2004/09/pencils-down-people.html

[65] Annoyed I gave the answer away? Here are some more for you to try:
SEND + MORE = MONEY (from "Mathematical Puzzles for Beginners and Enthusiasts" by Geoffrey Mott-Smith)
EAT + THAT = APPLE (from "Take a Look at a Good Book" by Steven Kahan)
TAKE + A + CAKE = KATE
BE × BE = MOB (from "Madachy's Mathematical Recreations" by Joseph S. Madachy)
NO + GUN + NO = HUNT (from "Entertaining Mathematical Teasers and How to Solve Them" by J. A. H. Hunter)

- **Domain**: 0,1 for the x_n variables, and 0 ... 9 for the others
- **Constraints**: (E + M) mod 10 = T ("mod" means remainder after integer division)
  (E + M) div 10 = x_5 ("div" means integer division)
  (x_5 + L + O) mod 10 = O
  (x_5 + L + O) div 10 = x_4
  (x_4 + G + C) mod 10 = D
  (x_4 + G + C) div 10 = x_3
  (x_3 + O + T) mod 10 = W
  (x_3 + O + T) div 10 = x_2
  (x_2 + O + O) mod 10 = W
  (x_2 + O + O) div 10 = x_1
  (x_1 + G + D) = W (Notice no mod or div needed)
  G, D, and W may not be 0 (No leading zeros. This could be put in the domain)
  None of G, O, L, E, D, T, C, M, W, or D may have the same values

Let's take another problem: the **N-Queens Problem** described in Section 7. The figure at right shows one of multiple solutions to the 5-Queens problem. We can formulate this as a constraint satisfaction problem by realizing that each column must hold exactly one queen. We can thus treat the columns as variables, and the row-location of the queens on those columns as values.

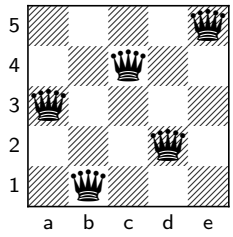


Figure 46 A solution to the 5-Queens problem, showing variables (a, b, c, d, e) and values (1, 2, 3, 4, 5).

- **Variables**: a, b, c, d, e
- **Domain**: 0, 1, 2, 3, 4, 5
- **Constraints**: No two queens may threaten one another

Obviously this single constraint is a doozy: you'll have to compute a lot about the board to determine it. We could divide it into dozens of smaller constraints (such as "if a = 5, then b may not = 4"). But you get the idea.

Here's another example. At right is the *Petersen Graph*, a classic 10-node graph structure. The objective is to color each node such that no two nodes of the same color are connected via an edge. You have three colors to work with. The figure shows one possible solution. We can formulate this **graph coloring** problem as:

- **Variables**: A, B, C, D, E, F, G, H, I, J
- **Domain**: red, green, blue
- **Constraints**: A and B may not share the same value
  B and C may not share the same value
  C and D may not share the same value
  D and E may not share the same value
  E and A may not share the same value
  G and I may not share the same value
  H and J may not share the same value
  I and F may not share the same value

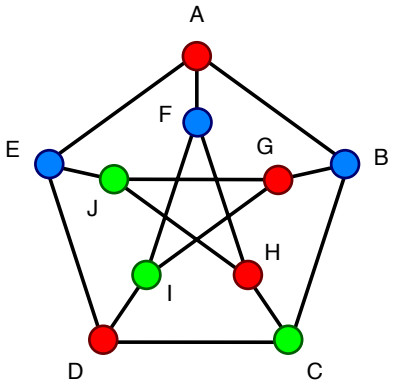


Figure 47 A 3-colored Petersen Graph. Courtesy of Wikipedia.

J and G may not share the same value
F and H may not share the same value
A and F may not share the same value
B and G may not share the same value
C and H may not share the same value
D and I may not share the same value
E and J may not share the same value

Now consider **Tetromino Tiling**. A tetromino is a shape made out of four adjacent squares, as shown at right. You might recognize these shapes from Tetris.[66]

The objective of tetromino tiling is to fill a specific region with a provided set of tetrominos, which may be rotated or flipped as necessary. We can formulate this problem as a constraint satisfaction problem as well:

- **Variables**: Each tetromino in the set
- **Domain**: Possible rotations (4), flips (2), and $\langle x, y \rangle$ positions (40 in the 5×8 problem) for a tetromino, for a total of 320 values.
- **Constraints**: No two tetrominos may overlap
  No tetromino may lie, in part or in whole, outside the problem region (in this case, the 5×8 rectangle)

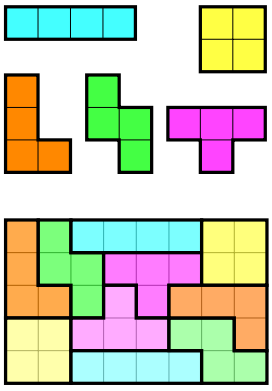


Figure 48 The five tetrominos, and one solution to the 5×8 tetromino tiling problem (using two copies of each tetromino). Courtesy of Wikipedia.

We presume that the tetromino variables are hard-set to specific tetromino types: else this can be defined in the variable domains or as constraints.

Finally, the classic puzzle craze **Sudoku** is also a straightforward constraint satisfaction problem. In short:

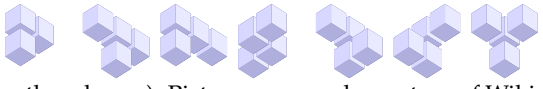
- **Variables**: Each open square location
- **Domain**: 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Constraints**: A value may not appear twice in any column
  A value may not appear twice in any row
  A value may not appear twice in any outlined 9×9 box

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 49 A Sudoku puzzle and its solution. Courtesy of Wikipedia.

Logistics Constraint satisfaction is particularly common in planning and scheduling problems: everything from determining optimal factory floor plans to scheduling astronomer usage of the Hubble Space Telescope.[67] How might you cast a problem like these as a constraint satisfaction problem? Consider the problem of devising a student schedule for all four years as a computer science major. To keep things simple, let's assume that every course is 3 credits. You might construct the problem like this:

[66] A classic 3-D version of this problem is the **Soma Cube**, where you are given the following **polycubes** consisting of 3 or 4 cubes glued together:  The objective is to put them together to form a 3×3 cube (or certain other shapes). Pictures, as usual, courtesy of Wikipedia. I'll leave it as an exercise to work out the formulation of the Soma Cube as a constraint satisfaction problem.

[67] Seriously. I knew someone at NASA in charge of this.

- **Variables**: 60 courses
- **Domain**: All valid combinations of course numbers, semesters, years, and sections (for example, one value would be CS480/Fall/2011/001), or *nil*
- **Constraints**: The courses must satisfy all general education requirements
  The courses must satisfy all major requirements
  No two courses may overlap in time
  No more than six non-*nil* courses may be taken in a semester
  No fewer than three non-*nil* courses may be taken in a semester
  Courses may only be taken after their prerequisites have been taken
  At least 40 courses must be non-*nil*

I included *nil* to make it easier to take more than 40 credits of courses. Notice that several of these constraints are of a different kind than we've seen so far. In previous examples, the constraints were always $A = B$ or *"foo is not permitted"*. Those kinds of constraints can be checked immediately in the middle of building an incomplete solution. But "The courses must satisfy all general education requirements" can only be checked when the solution is complete (or near enough that it's easy to tell that it's impossible to satisfy the constraint with the remaining unassigned variables). There are several other constraints like this (which ones?).

8.1 Backtracking

The goal of a constraint satisfaction search algorithm is to produce a solution which is both **complete** and **correct**. By *correct* we mean that, for every variable assigned a value, those values collectively do not violate any constraints (that it, it is **consistent** with them). By *complete* we mean that every variable has been assigned a variable. Recall that there are two general ways of searching for a solution: by hunting through the space of *incorrect but complete* states until we find one which is both complete and correct; or by hunting through the space of *incomplete but correct* states until we find one which is both complete and correct. In many cases it's more efficient to do the latter in constraint satisfaction search. So that's what we'll do here.

The general procedure will be to do the following:

1. Select a variable
2. For each possible value you could set that variable to, set the value and recurse to see if this lead to a solution
3. If it does, return it

This procedure is known as **backtracking**: since if trying one value setting doesn't lead to a solution, we backtrack and try the next, and so on. It's a variation on depth-first search.

The first interesting thing is that **correctness doesn't matter what variable you select first**. If there exists a solution, you'll get to it by first working on variable X, then later Y, just as likely as you will if you start on variable Y, then later X. However you can make the process *more efficient* by carefully selecting which variables to go after first, and also by carefully selecting which values to try first. This is where the **heuristic** nature of the search comes into play. We'll get to that in a bit.

The second interesting thing is that if you take this route (incomplete but correct states), your state space has no cycles: since you're always adding another assigned variable, at some point you'll run out of unassigned variables. Thus the depth of your search tree is bounded to the number of variables. So we don't need a "maxdepth" parameter.

Algorithm 22 *Backtracking Search*

```
1: S ← current state                          ▷ Some variables may have been assigned to values, others not
2: select(...) ← returns an unassigned variable to work on
3: domain(...) ← returns all possible values which may be assigned to a given variable
4: infer(...) ← reduces the values of a given variable to the ones still consistent in a given state
5: expand(...) ← produces a single new state from an old state and the assignment of a variable
6: consistent(...) ← consistency predicate function
7: complete(...) ← completeness predicate function

8: if complete(S) then                                                          ▷ Found a solution!
9:     return S
10: else
11:     var ← select(S)                                                         ▷ pick a variable to work on
12:     V ← domain(var)
13:     V ← infer(V, var, S)          ▷ V may be empty if there are no more consistent values for var
14:     for each v ∈ V do
15:         C ← expand(S, "var = v")                                            ▷ Copy S and set var = v in the copy
16:         if consistent(C) then
17:             result ← Backtracking Search with C, select, domain, infer, expand, correct, and complete
18:             if result ≠ FAILURE then
19:                 return result
20:     return FAILURE                                                          ▷ Couldn't find an answer
```

This algorithm requires you to provide several functions. Let's go through these functions one by one.

- **select(...)** Given a state S, this function selects an unassigned variable to start working on. Technically it doesn't matter what variable you return, but the variable you select has a huge impact on how long it takes to find a solution. We'll get to that in a bit.
- **domain(...)** Given a variable *var*, this function returns all the possible values you could set *var* to (the *domain* of *var*).
- **infer(...)** Given a variable *var*, its domain V, and a state S this function returns only those values in V which wouldn't make S inconsistent. You don't **have** to do this because you'll check later on with the consistent(...) function.
- **expand(...)** Given a state S and an assignment "$var = v$", creates a new state C which is a copy of S but where that assignment is now set.
- **consistent(...)** Returns whether or not all variables in S are consistent given the constraints. You don't have to do this if you're weeding out all possible inconsistent variable settings in the infer(...) function.

- **complete(...)** Returns whether or not S is complete, that is, all variables have been assigned.

Figure 50 shows the search tree produced by this algorithm on the 5-Queens problem, where at each level select(S) is defined as just picking the next column, left to right.

Where Do You Check for Inconsistency? Notice that there are **three places** where you could check for consistency: the infer(...) function (Line 13) or the consistent(...) function (Line 16) (ignore Line 8 for now). Between the two of them, you must weed out all inconsistent variable settings. You could do this all in infer(...) beforehand, or do it all in consistent(...), or share the burden between the two.

Sometimes it's cheap to do some checking beforehand to weed out the stupid assignments, as a preprocessing step before doing further recursion. For example, you might have some constraints which only involve a single variable (like "G may not be set to 0" or "Tetromino 4 may not lie outside the problem region". These **unary constraints** are often efficiently checked beforehand in the infer(...) step. This is known as checking for **1-consistency** (sometimes called **node consistency**).

It's *possibly* worthwhile to also check for **binary constraints** at this step: constraints involving two variables, such as "J and G may not share the same value" or "No two courses may overlap in time". This is known as **2-consistency** (sometimes called **arc consistency**).

Some larger constraints, involving more variables, (so-called **k-consistency** checking) may be more expensive to check at the infer(...) step and it might make more sense to only whittle down variables with these constraints as you need to, during the consistent(...) step. These are constraints such as " $(x_5 + \mathrm{L} + \mathrm{O}) \text{ div } 10 = x_4$" or "no more than six non-*nil* courses may be taken in a semester". This last constraint example is often called a **resource constraint**: you have constraints on a resource (here, a semester) that may be shared among variables (courses).

Finally, so-called **global constraints**, which involve *all* the variables, are usually worthwhile doing at the infer(...) step because they can be done quickly and you'll need to do them regardless. These include constraints like "None of G, O, L, E, D, T, C, M, W, or D may have the same values". Some other global constraints can only be checked at the very end. An example of these is "The courses must satisfy all general education requirements." You can check for these in either the infer(...) or consistent(...) functions. If you use infer(...), check when S has all but one variable left to assign (the one you're assigning now). If you use consistent(...), check when C has had all its variables assigned.

8.2 Heuristic Backtracking

Besides the consistency checking options discussed above, there are two major places where you can add heuristic knowledge about your problem to help the search go faster than brute-force depth-first search:

- Choose the right variable to work on next
- Choose the right order in which you'll try assigning values to that variable

Like consistency checking, these heuristic decisions are often domain-specific. But here are some good heuristics:

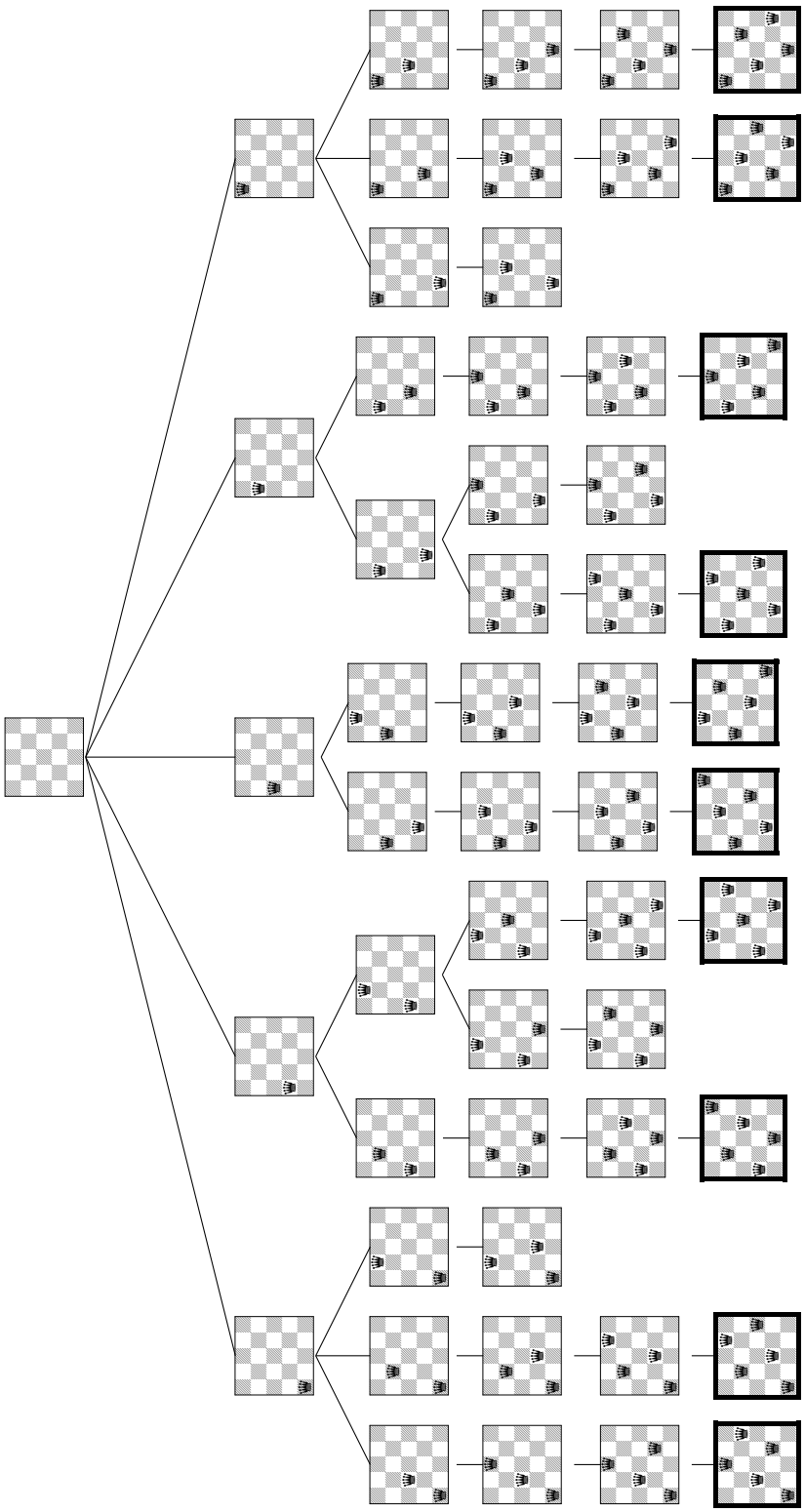


Figure 50 Full constraint satisfaction search tree for the 5-queens problem, where the select(S) is defined as selecting columns left to right. Solutions shown in bold.

- **Select the variable which has the fewest consistent values left as options.** The idea here is that if there's a variable with very few options left, you want to try it first because if it fails, you'll probably find very soon. That way you can avoid deep searches which all end with this variable failing. As an extreme example: if the variable has *no options left*, then you've already failed! So you'd want to check that first always. This is often called the **Minimum Remaining Values** or **MRV** heuristic. It might be useful here to have handled a lot of variable reduction in the infer(...) function.
- **Select the variable which is involved in as many constraints as possible.** The idea here again is to pick a variable likely to cause early failure, because assigning it to a value will deeply constrain so many other variables (wiping out as many remaining options as possible). This isn't as powerful as MRV, but it's useful to use as a tie-breaker for MRV. It's known as the **Degree** heuristic. Again, it might be useful here to have handled a lot of variable reduction in the infer(...) function.
- **Select the value which reduces the options for the remaining variables as little as possible.** Strangely, this is the opposite of the fail-as-soon-as-possible approach for setting variables. The idea here is that we want to increase our chances of finding an answer, and picking a value which reduces options for other variables will reduce that chance. This heuristic, called the **Least Constraining Value** or **LCV** heuristic, might essentially require you to call the infer(...) function internally as part of its computation. (You might want to store away those results so you don't have to recompute them!)

Why are the first and last heuristics so different? Here's something to ruminate on. In constraint satisfaction search, you are not backtracking on variable selection (it doesn't matter what order you pick variables, as you'll still find the solution if one exists). You are only backtracking on the *value selection* for a variable. If you select a particular variable and it doesn't work out, you're done: no solution exists. But if you try a *value* and it doesn't work out, you have to try the next value and see if that one works out, and so on.

Where to Go From Here? Constraint Satisfaction is a very old, and very well studied problem with an enormous number of algorithms and approaches. Just within the search framework described earlier, there are techniques such as **backjumping** and **forward checking** which trade off preprocessing steps for recursive steps. And decisions regarding variable and value ordering, and when you check for consistency, are domain-specific and require a lot of thought to keep from searching too much to find a solution. There are a lot of algorithm variants here.

8.3 Greedy Search

An alternative is to instead search through the space of **incorrect but complete** solutions until you find one which is both complete and correct. Recall in Figure 36 (page 82) that, for the **N-Queens problem** this might consist of plopping down N random queens, one for each column and then searching though the space of states where you move a queen (within her column) to transition from one state to another. Ordinarily this is a bad way of doing constraint satisfaction: but surprisingly you can often do a great job on some constraint satisfaction problems with a greedy version. Instead of recursively searching the space in a depth-first fashion, simply move forward through the space by changing one queen at a time and never go back. Keep moving queens until you find an answer.

Surprisingly this often works really well. But to do it right you have to *move a queen to the right place* each time. This is again an opportunity for a heuristic. The heuristic will be:

- **The Minimum Conflicts Heuristic** Change the variable's value to the one which violates the fewest constraints. In the N-Queens problem, this means: move the queen to the spot in her column where she attacks the fewest other queens.

The algorithm is simple: we repeatedly pick a random queen which is violating a constraint. We then move the queen to the spot where she violates the fewest constraints. And we repeat until no queen is violating any constraints. And that's it!

Algorithm 23 *Min-Conflicts Greedy Search*

```
1: S ← initial state                                   ▷ With random assignments to all variables
2: randomVariable(...) ← returns a random variable violating a constraint in S
3: bestValue(...) ← returns the value for a given variable which violates the fewest constraints in S
4: expand(...) ← produces a single new state from an old state and the (re-)assignment of a variable
5: consistent(...) ← consistency predicate function

6: loop
7:     if consistent(S) then
8:         return S                                    ▷ Found a solution!
9:     else
10:        var ← randomVariable(S)
11:        value ← bestValue(var, S)
12:        S ← expand(S, "var = v")
```

Note that the algorithm, unlike Backtracking, won't terminate if it can't find a solution: it'll just keep on going. You might modify it to run for a limited (if long) period of time.

8.4 Constrained Optimization

So far the constraints we've seen have been either met or unmet: either the states are correct (consistent) or incorrect. Note however that in Figure 50 there were no less than ten solutions to the 5-Queens problem.[68] What if, for example, we **preferred** a solution where a Queen sat directly in the center of the board (that is, at C3)?

Now we've introduced a **soft constraint**: a *quality assessment*[69] we apply to a state rather than simply declaring it right or wrong. We cannot accept wrong solutions: but among the right solutions we'd prefer the ones with the highest quality. We've seen quality assessments before: in metaheuristics. Most metaheuristics are pure optimization (soft constraints), and the constraint search algorithms described so far are pure hard constraints.

There exist quite a number of algorithms which handle both hard and soft constraints in certain situations, such as the **Simplex Algorithm** or the algorithm known as **Branch and Bound**. In the more general context, we need an algorithm which allows for an arbitrary quality function in the context of hard constraints. One easy way to do this is to use an evolutionary algorithm. Here's a

[68] This isn't always the case: often there are very few solutions, or only one (or maybe zero).

[69] The constraints we've seen up to now are usually called **hard constraints**.

simplistic approach. Let's imagine that the soft constraint was a value s from 0 to n, with 0 being most preferred. And let's say that the number of hard constraints violated is h. We could construct a fitness function as $s + h \times n$. Lower fitnesses are preferred. The idea would be that a solution which violated fewer hard constraints would be preferred, but ties would be broken by the solution which had the lowest soft constraint result.