2

The Golden Age

ALTHOUGH TURING'S ARTICLE "Computing Machinery and Intelligence," which introduced the Turing test, made what we now recognize as the first substantial scientific contribution to the discipline of AI, it was a rather isolated contribution, because AI as a discipline simply did not exist at the time. It did not have a name, there was no community of researchers working on it, and the only contributions at the time were speculative conceptual ones, such as the Turing test—there were no AI systems. Just a decade later, by the end of the 1950s, all that had changed: a new discipline had been established, with a distinctive name, and researchers were able to proudly show off the first tentative systems demonstrating rudimentary components of intelligent behavior.

The next two decades were the first boom in AI. There was a flush of optimism, growth, and apparent progress, leading to the era called the **golden age of AI,** from about 1956 to 1974. There had been no disappointments yet; everything seemed possible. The AI systems built in this period are legends in the AI canon. Systems with quirky, geeky names like SHRDLU, STRIPS, and SHAKEY—short names, all in uppercase, supposedly because those were the constraints of computer file names at the time (the tradition of naming AI systems in this way continues to the present day, although it has long since ceased to be necessary). The computers used to build these systems were, by modern standards, unimaginably limited, painfully slow, and tremendously hard to use. The tools we take for granted when developing software today did not exist then and indeed could not have run on the computers of the time. Much of the "hacker" culture of computer programming seems to have emerged at the time. AI researchers worked at night because then they could get access to the computers that were used for more important work during normal office hours; and they had to invent all sorts of ingenious programming tricks to get their complicated programs to run at all—many of these tricks subsequently became standard techniques, with their origins in the AI labs of the 1960s and 1970s now only dimly remembered, if at all.[1]

But by the mid-1970s, progress on AI stalled, having failed to progress far beyond the earliest simple experiments. The young discipline came close to being snuffed out by research funders and a scientific community that came to believe AI, which had promised so much, was actually going nowhere.

In this chapter, we'll look at these first two decades of AI. We'll look at some of the key systems built during this period and discuss one of the most important techniques developed in AI at the time—a technique called a *search,* which to the present day remains a central component of many AI systems. We'll also hear how an abstract mathematical theory called *computational complexity,* which was developed in the late 1960s and early 1970s, began to explain why so many problems in AI were fundamentally hard. Computational complexity cast a long shadow over AI.

We'll begin with the traditional starting point of the golden age: the summer of 1956, when the field was given its name by a young American academic by the name of John McCarthy.

THE FIRST SUMMER OF AI

McCarthy belonged to that generation who seem to have created the modern technological United States. With almost casual brilliance, throughout the 1950s and 1960s, he invented a range of concepts in computing that are now taken so much for granted that it is hard to imagine that they actually had to be invented. One of his most famous developments was a programming language called LISP, which for decades was the programming language of choice for AI researchers. At the best of times, computer programs are hard to read, but even by the frankly arcane standards of my profession, LISP is regarded as bizarre, because in LISP (all (programs (look (like this))))—generations of programmers learned to joke that LISP stood for Lots of Irrelevant Silly Parentheses.[2]

McCarthy invented LISP in the mid-1950s, but astonishingly, nearly seventy years later, it is still regularly taught and used across the world. (I use it every day.) Think about that for a moment: when McCarthy invented LISP, Dwight D. Eisenhower was president of the United States. There were no more than a handful of computers in the whole world. And the programming language McCarthy invented then is still routinely used today.

In 1955, McCarthy submitted a proposal to the Rockefeller Institute in the hope of obtaining funds to organize a summer school at Dartmouth College. If you are not an academic, the idea of "summer schools" for adults may sound a little strange, but they are a well-established and fondly regarded academic tradition even today. The idea is to bring together a group of researchers with common interests from across the world and give them the opportunity to meet and work together for an extended period. They are held in summer because, of course, teaching has finished for the year, and academics have a big chunk of time without lecturing commitments. Naturally, the goal is to organize the summer school in an attractive location, and a lively program of social events is essential.

When McCarthy wrote his funding proposal for the Rockefeller Institute in 1955, he had to give a name to the event, and he chose *artificial intelligence.* In what would become something of a weary tradition for AI, McCarthy had unrealistically high expectations: "We think that a significant advance can be made … if a carefully selected group of scientists work on it together for a summer."[3]

By the end of the summer school, the delegates had made no real progress, but McCarthy's chosen name had stuck, and thus was a new academic discipline formed.

Unfortunately, many have subsequently had occasion to regret McCarthy's choice of name for the field he founded. For one thing, *artificial* can be read as *fake* or *ersatz*—and who wants *fake intelligence*? Moreover, the word *intelligence* suggests that *intellect* is key. In fact, many of the tasks that the AI community has worked so hard on since 1956 don't seem to require *intelligence* when people do them. On the contrary, as we saw in the previous chapter, many of the most important and difficult problems that AI has struggled with over the past sixty years don't seem to be *intellectual* at all—a fact that has repeatedly been the cause of consternation and confusion for those new to the field.

But *artificial intelligence* was the name that McCarthy chose and the name that persists to this day. From McCarthy's summer school, there is an unbroken thread of research by way of the participants in the summer school and their academic descendants, right down to the present day. AI in its recognizably modern form began that summer, and the beginnings of AI seemed to be very promising indeed.

The period following the Dartmouth summer school was one of excitement and growth. And for a while at least, it seemed like there was rapid progress. Four delegates of the summer school went on to dominate AI in the decades that followed. McCarthy himself founded the AI lab at Stanford University in the heart of what is now Silicon Valley; Marvin Minsky founded the AI lab at MIT in Cambridge, Massachusetts; and Allen Newell and his Ph.D. supervisor Herb Simon went to Carnegie Mellon University (CMU). These four individuals, and the AI systems that they and their students built, are totems for AI researchers of my generation.

But there was a good deal of naivety in the golden age, with researchers making reckless and grandiose predictions about the likely speed of progress in the field, which have haunted AI ever since. By the mid-1970s, the good times were over, and a vicious backlash began—an AI boom-and-bust cycle destined to be repeated over the coming decades.

DIVIDE AND CONQUER

As we've seen, general AI is a large and very nebulous target—it is hard to approach directly. Instead, the strategy adopted during the golden age was that of *divide and conquer*. Thus, instead of starting out trying to build a complete general intelligent system, the approach adopted was to identify the various different individual *capabilities* that seemed to be required for general-purpose AI and to build systems that could demonstrate these capabilities. The implicit assumption was that if we could succeed in building systems that demonstrate each of these individual capabilities, then, later on, assembling them into a whole would be straightforward. This fundamental assumption—that the way to progress toward general AI was to focus on the component capabilities of intelligent behavior—became embedded as the standard methodology for AI research. There was a rush to build machines that could demonstrate these component capabilities.

So what were the main capabilities that researchers focused on? The first, and as it turned out one of the most stubbornly difficult, is one that we take for granted: **perception.** A machine that is going to act intelligently in a particular environment needs to be able to get information about it. We perceive our world through various mechanisms, including the five senses: sight, hearing, touch, smell, and taste. So one strand of research involved building **sensors** that provide analogues of these. Robots today use a wide range of artificial sensors to give them information about their environment—radars, infrared range finders, ultrasonic range finders, laser radar, and so on. But *building* these sensors—which in itself is not trivial—is only part of the problem. However good the optics are on a digital camera and no matter how many megapixels there are on the camera's image sensor, ultimately all that camera does is break down the image it is seeing into a grid and then assign numbers to each cell in the grid, indicating color and brightness. So a robot equipped with the very best digital camera will, in the end, only receive a long list of numbers. The second challenge of perception is therefore to interpret those raw numbers: to understand what it is seeing. And this challenge turned out to be far harder than the problem of actually building the sensors.

Another key capability for general intelligent systems seems to be the ability to learn from experience, and this led to a strand of AI research called **machine learning.** Like the name *artificial intelligence* itself, *machine learning* is perhaps an unfortunate choice of terminology. It sounds like a machine somehow bootstrapping its own intelligence: starting from nothing and progressively getting smarter and smarter. In fact, machine learning is not like human learning: it is about learning from and making predictions about data. For example, one big success in machine learning over the past decade is in programs that can recognize faces in pictures. The way this is usually done involves providing a program with examples of the things that you are trying to learn. Thus, a program to recognize faces would be trained by giving it many pictures labeled with the names of the people that appear in the pictures. The goal is that, when subsequently presented with solely an image, the program would be able to correctly give the name of the person pictured.

Problem solving and **planning** are two related capabilities that also seem to be associated with intelligent behavior. They both require being able to achieve a goal using a given repertoire of actions; the challenge is to find the right sequence of actions. Playing a board game such as chess or Go would be an example: the goal is to win the game; the actions are the possible moves; the challenge is to figure out which moves to make. As we will see, one of the most fundamental challenges in problem solving and planning is that while they appear easy to do *in principle,* by considering all the possible alternatives, this approach doesn't work in practice, because there are far too many alternatives for it to be feasible.

Reasoning is perhaps the most exalted of all the capabilities associated with intelligence: it involves deriving new knowledge from existing facts in a robust way. To pick a famous example, if you know that "all men are mortal" and you know that "Michael is a man," then it is reasonable to conclude that "Michael is mortal." A truly intelligent system would be able to derive this conclusion and then use the newly derived knowledge to make further conclusions. For example, knowing that "Michael is mortal" should then allow you to conclude that "at some point in the future, Michael will die" and that "after Michael is dead, he will stay dead forever," and so on. Automated reasoning is all about giving computers this kind of capability: giving them the capability to reason *logically.* In chapter 3, we will see that for a long time, it was believed that this kind of reasoning should be the main goal for AI, and although this is not a mainstream view anymore, automated reasoning remains an important thread of the field.

Finally, **natural language understanding** involves computers interpreting human languages like English and French. At present, when a programmer gives a computer a recipe to follow (an algorithm or program), she must do so using an *artificial* language: a language specially constructed accordingly to precisely defined unambiguous rules. These languages are *much* simpler than natural languages like English. For a long time, the main approach to understanding natural languages involved trying to come up with precise rules that define these languages, in the same way that we have precise rules defining computer languages. But this turned out to be impossible. Natural languages are too flexible, vague, and fluid to be rigorously defined in this way, and the way in which language is used in everyday situations resists attempts to give it a precise definition.

SHRDLU AND THE BLOCKS WORLD

The **SHRDLU** system was one of the most acclaimed achievements of the golden age (the odd name derives from the order in which letters were arranged on printing machines of the time—computer programmers enjoy obscure jokes). Developed by Stanford University Ph.D. student Terry Winograd in 1971,[4] SHRDLU aimed to demonstrate two key capabilities for AI: problem solving and natural language understanding.

The problem-solving component of SHRDLU was based around what became one of the most famous experimental scenarios in AI: the **Blocks World.** The Blocks World was a simulated environment containing a number of colored objects (blocks, boxes, and pyramids). The rationale for using a simulated environment, rather than trying to build a real robot, was simply to reduce the complexity of the problem to something manageable. Problem solving in the SHRDLU Blocks World involves arranging objects according to instructions from a user, using a simulated robot arm to manipulate them. Figure 2 illustrates: we see the initial configuration of the Blocks World and the goal configuration. The challenge is to figure out *how* to transform the initial configuration to the goal configuration.

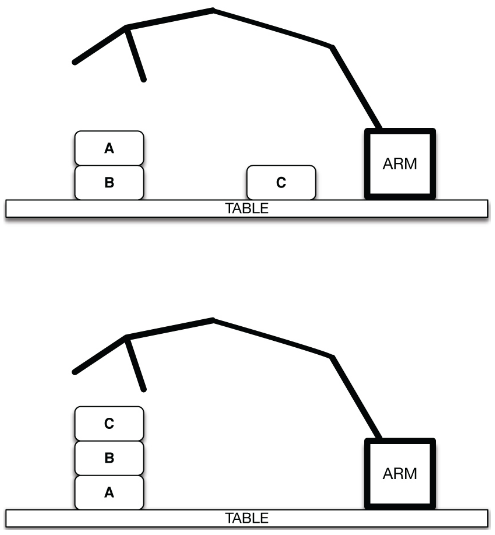

Figure 2: The Blocks World. Above is the initial configuration—below, the goal configuration. How do you get from one to the other?

To achieve this transformation, you are only allowed to use a small repertoire of actions:

- ***Pick up object* x *from the table.*** Here, the robot arm picks up object *x* (which could be a block or pyramid) from the table. The robot arm will only be able to do this successfully if both object *x* is currently on the table and the robot arm is currently empty.
- ***Place object* x *on the table.*** The robot arm will only be able to do this if it is currently carrying object *x*.
- ***Pick up object* x *from object* y.** For the robot arm to be able to do this, the robot arm must be empty, object *x* must be currently on top of object *y,* and object *x* must currently have nothing on top of it.
- ***Place object* x *on top of object* y.** For the robot arm to be able to do this, it has to actually be carrying object *x*, and there must be nothing currently on top of object *y.*

Everything that happens in the Blocks World reduces to these actions, and only these actions. Using these actions, a plan to achieve the transformation illustrated in figure 2 might begin as follows:

- **Pick up object *A* from object *B*.**
- **Place object *A* on the table.**
- **Pick up object *B* from the table.**

The Blocks World is probably the most-studied scenario in the whole of AI, because tasks in the Blocks World—picking up packages and moving them around—sound like the kind of tasks we might envisage for robots in the real world. But the Blocks World as it appears in SHRDLU (and much subsequent research) has severe limitations as a scenario for developing practically useful AI techniques.

First, it is assumed that the Blocks World is *closed.* This means that the only thing that causes change in the Blocks World is SHRDLU. This is rather like saying that you live alone. If you live alone, you can safely assume that when you wake up, your keys will be where you left them the night before; if you don't live alone, there is always the possibility that someone else has moved them overnight. So when SHRDLU places block *x* on top of block *y,* it can safely assume that block *x* remains on top of block *y* unless it has moved the block itself. The real world is not like that: any AI system that relies on the assumption that it is the only actor in its world is likely to get things wrong a lot of the time.

Second, and perhaps more important, the Blocks World is *simulated.* SHRDLU is not *actually* picking up objects and moving them around with a robot arm—it is just pretending to. It does this by maintaining a model of the world, and it models the effects of its actions on this world. It never actually looks at the world to build its model, or to check that the model it has is correct. This is a huge simplifying assumption, and researchers later argued that it therefore ignored most of the genuinely hard problems that a robot would face in the real world.

To better understand this point, consider the action "Pick up object *x* from object *y.*" From the point of view of SHRDLU, this is a single action: the robot is assumed to carry out the *whole* of the action in a single indivisible step, and nowhere does it need to worry about what this action *actually involves.* Thus, all the program "controlling" the arm has to do is to find the right order of such actions to carry out the required task. It doesn't need to worry about the messy business of actually carrying them out. But think about a robot in the physical world trying to undertake this action, perhaps in a warehouse. Here, the robot would have to be able to identify the two objects in question, successfully execute a complex locomotion problem to get the arm to the right place, and grasp the relevant object. Even the final part of this "atomic" action, grasping the relevant object is, in reality, far from trivial—getting robots to manipulate even simple objects in the real world is phenomenally difficult, and remains a significant challenge to the present day. The fact that *we* find such tasks trivial leads us to believe, wrongly, that it must also be trivial to get machines to do them.

So the Blocks World is appealing because it is intuitive and easy to explain, and it seems to resemble a problem that we could imagine robots working on in the real world. But digging deeper, it turns out that it grossly simplifies the problems that would be faced by a real robot.

One of the most lauded features of SHRDLU was that the user gave instructions to the system and received feedback from it in what seemed to be ordinary English. Here is a snippet of dialogue with SHRDLU (SHRDLU's responses are in italics):

PICK UP A BIG RED BLOCK.

OK. (*Does so.*)

GRASP THE PYRAMID.

I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.

FIND A BLOCK WHICH IS TALLER THAN THE ONE YOU ARE HOLDING AND PUT IT INTO THE BOX.

BY "IT," I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM HOLDING.

OK. (*Does so.*)

WHAT DOES THE BOX CONTAIN?

THE BLUE PYRAMID AND THE BLUE BLOCK.

WHAT IS THE PYRAMID SUPPORTED BY?

THE BOX.

HOW MANY BLOCKS ARE NOT IN THE BOX?

FOUR OF THEM.

IS AT LEAST ONE OF THEM NARROWER THAN THE ONE WHICH I TOLD YOU TO PICK UP?

YES, THE RED CUBE.

This dialogue is far beyond ELIZA,[5] and it was this aspect of the system—the way that a user interacted with it, via what seemed at first like ordinary English—which led to SHRDLU receiving a rapturous reception in the early 1970s. However, it later became clear that SHRDLU was able to generate rich dialogues only because those dialogues related to a very constrained scenario—the Blocks World. The dialogues were not quite "canned" in the way that dialogues in ELIZA were canned, but they were, nevertheless, very constrained. There was hope, when the system first appeared, that the techniques it embodied might provide a route to much more general natural-language understanding systems, but this hope was not realized.

It is easy for us, fifty years later, to identify the limitations of SHRDLU. But it was hugely influential, and remains one of the landmark AI systems.

SHAKEY THE ROBOT

Robots have always been closely associated with AI—particularly in the media. The "machine man" in Fritz Lang's classic 1927 film *Metropolis* set the template for the endless portrayals of robotic AI that followed: a robot with two arms, two legs, a head … and a murderous temperament. Even today, it seems, every article about AI in the popular press is illustrated with a picture of a robot that could be a direct descendant of the *Metropolis* "machine man." It is, I think, no surprise that robots in general, and humanoid robots in particular, should become the public signifier for AI. After all, the idea of robots that inhabit our world and work among us is probably the most obvious manifestation of the AI dream—and most of us would be delighted to have robot butlers to do our every bidding.

It may therefore come as a surprise to learn that during the golden age, robots were only a comparatively small part of the AI story. I'm afraid the reasons for this are rather mundane: building robots is expensive, time-consuming, and, frankly, *difficult.* It would have been impossible for a Ph.D. student working alone in the 1960s or 1970s to build a research-level AI robot. They would have needed an entire research lab, dedicated engineers, and workshop facilities—and in any case, computers powerful enough to drive AI programs were not mobile enough to install on autonomous robots. It was easier and much cheaper for researchers to build programs like SHRDLU, which *pretended* to be working in the real world, than it was to build robots that *actually* operated in the real world, with all its complexity and messiness.

But although the roster of AI robots in the early period of AI is rather sparse, there was one glorious experiment with AI robots during this time: the **SHAKEY** project, which was carried out at the Stanford Research Institute (SRI) between 1966 and 1972.

SHAKEY was the first serious attempt to build a mobile robot that could be given tasks in the real world and that could figure out on its own how to accomplish them. To do this, SHAKEY had to be able to perceive its environment and understand where it was and what was around it; receive tasks from users; figure out for itself appropriate plans to achieve those tasks; and then actually carry out these plans, all the while making sure that everything was progressing as intended. The tasks in question involved moving objects such as boxes around an office environment. This may sound like SHRDLU, but unlike SHRDLU, SHAKEY was a real robot, manipulating real objects. This was a far greater challenge.

To succeed, SHAKEY had to integrate a daunting array of AI capabilities. First, there was a basic engineering challenge: the developers needed to build the robot itself. It had to be small enough and agile enough to operate in an office, and it needed sensors that were powerful and accurate enough that the robot could understand what was around it. For this, SHAKEY was equipped with a television camera and laser range finders for determining distance to objects; to detect obstacles, it had bump detectors, called *cats' whiskers.* Next, SHAKEY had to be able to navigate around its environment. Then, SHAKEY needed to be able to plan how to carry out the tasks it was given.[6] And finally, all these capabilities had to be made to work together in harmony. As any AI researcher will tell you, getting any one of these components to work is a challenge in itself; getting them to work as an ensemble is an order of magnitude harder.

But impressive as it was, SHAKEY also highlighted the limitations of the AI technology of the time. To make SHAKEY work, its designers had to greatly simplify the challenges faced by the robot. For example, SHAKEY's ability to interpret data from its TV camera was very limited—amounting to not much more than detecting obstacles. And even to enable this, the environment had to be specially painted and carefully lit. Because the TV camera required so much power, it was only switched on when it was needed, and it took about ten seconds after the power was turned on before it produced a usable image. And the developers constantly struggled with the limitations of computers at that time: it took up to fifteen minutes for SHAKEY to figure out how to carry out a task, during which time SHAKEY would sit, immobile and inert, utterly isolated from its environment. Since computers with sufficient power to run SHAKEY's software were too large and heavy for SHAKEY to carry around, SHAKEY used a radio link to a computer that actually ran its software.[7] Overall, SHAKEY could not possibly have been put to use on any substantial problem.

SHAKEY was arguably the first real autonomous mobile robot, and like SHRDLU, it deserves a place of honor in the AI history books for this reason. But SHAKEY's limitations demonstrated just how far AI actually was from the dream of practical autonomous robots and just how daunting these challenges really were.

PROBLEM SOLVING AND SEARCH

The ability to solve complex problems is surely one of the key capabilities that distinguishes humans from other animals. And of course, problem solving seems like the kind of thing that requires intelligence—if we can build programs that can solve problems that people find hard, then surely this would be a key step on the road to AI. Problem solving was therefore studied intensely during the golden age, and a standard exercise for AI researchers of the time was to get a computer to solve a problem of the kind that you might come across on the puzzle page of a newspaper. Here is a classic example of such a puzzle, called the Towers of Hanoi:

In a remote monastery, there are three columns and sixty-four golden rings. The rings are of different sizes and rest over the columns. Initially, all the rings rested on the farthest left column, and since then, the monks have been moving the rings, one by one, between columns. Their goal is to move all the rings to the farthest right column. In doing so, the monks must obey two rules:

1. They may only move one ring at a time between columns.

2. At no time may any ring rest upon a smaller ring.

So to solve the problem, you have to find the right sequence of moves, which will get all the rings from the farthest left column to the farthest right one, without breaking the rule that a larger ring can never be above a smaller ring.

In some versions of the story, the world is supposed to end when the monks complete their task—although, as we will see later, even if this story were true, we would not need to lose sleep over it, because life on planet Earth will surely be extinct long before the monks would be able to move all sixty-four rings. For this reason, the puzzle is usually deployed with far fewer rings. Figure 3 illustrates the puzzle, showing the initial configuration (all rings are on the farthest left column), then the goal state (with all rings on the farthest right column), and finally an illegal configuration, for the three-ring version of the puzzle.

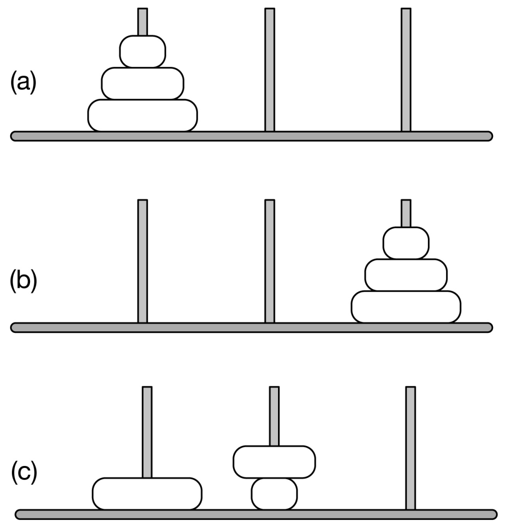

Figure 3: The Towers of Hanoi, a classic puzzle of the type studied in the golden age of AI. Panel (a) shows the initial state of the puzzle, and panel (b) shows the goal state. Panel (c) shows a disallowed configuration of the puzzle (it isn't allowed because a larger ring is on top of a smaller ring).

So how do we go about solving problems like the Towers of Hanoi? The answer is to use a technique called **search.** I should clarify that when we use the term *search* in AI, we don't mean searching the web: search in AI is a problem-solving technique, which involves systematically considering all possible courses of action. Any program that plays a game like chess will be based on search, as will the satellite navigation system in your car. Search arises time and time again: it is one of the cornerstone techniques in AI.

All problems like the Towers of Hanoi have the same basic structure. As in the Blocks World, we want to find a sequence of actions that will take us from some **initial state** of the problem to some designated **goal state.** The term **state** is used in AI to refer to a particular configuration of a problem at some moment in time.

We can use search to solve problems like the Towers of Hanoi through the following procedure:

- **First, starting from the initial state, we consider the effects of every available action on that initial state. The effect of performing an action is to transform the problem into a new state.**
- **If one of the actions has generated the goal state, then we have succeeded: the solution to the puzzle is the sequence of actions that got us from the initial state to the goal state.**
- **Otherwise, we repeat this process for every state we just generated, considering the effect of each action on those states, and so on.**

Applying this recipe for search generates a **search tree:**

- **Initially, we can only move the smallest ring, and our only choices are to move it to the middle or farthest right column. So we have two possible actions available for the first move and two possible successor states.**
- **If we chose to move the small ring to the center column (shown via the left arrow emerging from the initial state), then we subsequently have three possible actions available to us: we can move the smallest ring to the farthest left column or the farthest right column, or we can move the middle-sized ring to the farthest right column (we can't move it to the center column, because then it would be on top of the smallest ring, which violates the rules of the puzzle).**
- **… and so on.**

Because we are systematically generating the search tree, level by level, we are considering all possibilities—and so if there is a solution, we will be guaranteed to eventually find it using this process.

So how many moves would you need to make to solve the Towers of Hanoi problem *optimally* (i.e., with the smallest possible number of moves from the initial state)? The answer, for three rings, turns out to be seven: you will never be able to find a solution that solves the puzzle in fewer than seven moves. There are solutions that solve it in more than seven moves (in fact there are *infinitely many* such solutions) but they are not optimal, because you could have done it in fewer. Now, because the search process we have described is exhaustive—in the sense that it will systematically unwind the search tree, considering all possibilities at every stage—it is guaranteed not just to find a solution but to find the *shortest* solution.

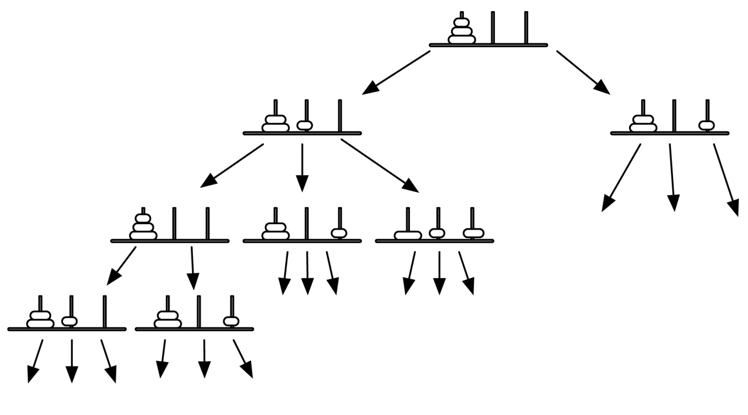

Figure 4: A small fragment of the search tree for the Towers of Hanoi puzzle.

So this process of naive exhaustive search is guaranteed to find a solution to the problem if one exists, and it is also guaranteed to find the shortest solution. Moreover, as computational recipes go, exhaustive search is pretty simple—it is easy to write a program to implement this recipe.

But even a cursory study of the search tree in figure 4 shows that, in the simple form we just described it, naive exhaustive search is actually a pretty stupid process. For example, if you study the leftmost branch of the search tree, you will see that, after just two moves, we have returned to the initial state of the problem, which seems like a pointless waste of effort. If *you* were solving the Towers of Hanoi, you might make this mistake once or twice while looking for a solution, but you would quickly learn to spot and avoid such wasted effort. But a computer carrying out the recipe for naive exhaustive search does not: the recipe involves systematically generating all the alternatives, even if those alternatives are a waste of time, in the sense that they are revisiting earlier failed configurations.

But apart from being hopelessly inefficient, exhaustive search has a much more fundamental problem. If you do a little experimentation, you will see that in almost all possible configurations of the Towers of Hanoi puzzle, you will have three possible moves available to you. We say that the **branching factor** of the puzzle is three. Different search problems will have different branching factors. In the board game Go, for example, the branching factor is 250, meaning that, on average, each player will have about 250 moves available in any given state of the game. So let's look at how the *size* of a search tree grows with respect to the branching factor of the tree—how many states there will be in any given level of the tree. So consider the game of Go:[8]

- **The first level of the tree will contain 250 states, because there are 250 moves available from the initial state of the game.**
- **The second level of the search tree will have 250 × 250 = 62,500 states, because we have to consider each possible move for each of the 250 states in the first level of the tree.**
- **The third level of the search tree will contain 250 × 62,500 = 15.6 million states.**
- **The fourth level of the search tree will contain 250 × 15.6 million = 3.9 billion states.**

So at the time of writing, a typical desktop computer would not have enough memory to be able to store the search tree for the game of Go for more than about four moves—and a typical game of Go lasts for about 200 moves. The number of states in the Go search tree for 200 moves is a number that is so large that it defies comprehension. It is hundreds of orders of magnitude larger than the number of atoms in our universe. No improvement in conventional computer technology is ever going to be able to cope with search trees this large.

Search trees grow fast. Ludicrously, unimaginably fast. This problem is called **combinatorial explosion,** and it is the single most important practical problem in AI, because search is such a ubiquitous requirement in AI problems.[9] If you could find a foolproof recipe for solving search problems quickly—to get the same result as exhaustive search, without all that effort—then you would make yourself very famous, and many problems that are currently very difficult for AI would suddenly become easy. But you won't, I'm afraid. We can't get around combinatorial explosion: we need to work with it. Attention therefore shifted to making search more efficient. To do this, there are several obvious lines of attack.

One possibility is to *focus* the search in some way—to identify promising directions in the search tree and explore those, rather than trying to explore them all. A simple way of doing this is as follows: instead of developing the tree level by level, we instead build the tree *along just one branch.* This approach is called **depth-first search.** With depth-first search, we carry on expanding a branch until we get a solution or become convinced that we won't get a solution; if we ever get stuck (for example by re-creating a configuration that we already saw, as in the farthest left branch of figure 4), then we stop expanding that branch and go back up the tree to start working on the next branch.

The main advantage that depth-first search has is that we don't have to store the whole of the search tree: we only need to store the branch we are working on. But it has a big disadvantage: if we pick the wrong branch to explore, then we may go on expanding our search along this branch without ever finding a solution. So if we want to use depth-first search, we really want to know which branch to investigate. And this is what **heuristic search** tries to help us with.

The idea of heuristic search is to use rules of thumb called **heuristics,** which indicate where to focus our search. We can't typically find heuristics that are *guaranteed* to focus search in the best possible direction, but we can often find heuristics that work well in cases of interest.

Heuristic search is such a natural idea that it has been reinvented many times over the years, and so it seems pointless to debate who invented it. But we can identify with reasonable confidence the first substantial application of heuristic search in an AI program: a program to play the game of checkers, written by an IBM employee called Arthur Samuel in the mid-1950s. Perhaps the key component of Samuel's program was a way of evaluating any given board position, to estimate how "good" the position was for a particular player: intuitively, a "good" board position for a particular player is one that would probably lead to a win for that player, whereas a "bad" board position would probably lead to that player losing. To do this, Samuel used a number of *features* of the board position in order to compute its value. For example, one feature might be the number of pieces you have on the board: the more you have, the better you are doing. These different features were then aggregated by the program to give a single overall measure of the quality of a board position. Then a typical heuristic would involve picking a move that leads to the best board position, according to this heuristic.

Samuel's program played a credible game of checkers. This was an impressive achievement in many respects but is all the more so when one considers the primitive state of practical computers of the time—the IBM 701 computer that Samuel used could only handle programs that were the equivalent length of just a few pages of text. A modern desktop computer will have millions of times more memory than this and will be millions of times faster. Given these restrictions, it is impressive that it could play the game at all, let alone play it competently.

AI CRASHES INTO THE COMPLEXITY BARRIER

We previously saw that computers were for all intents and purposes invented by Alan Turing in order to solve one of the great mathematical problems of his time. It is one of the great ironies of scientific history that Turing invented computers in order to show that there are things that computers are fundamentally incapable of doing—that some problems are inherently undecidable.

In the decades following Turing's results, exploring the limits of what computers can and cannot do became a small industry in university mathematics departments across the world. The focus in this work was on separating out those problems that are inherently undecidable (cannot be solved by computer) from those that are decidable (can be solved by computer). One intriguing discovery from this time was that there are *hierarchies* of undecidable problems: it is possible to have a problem that is not just undecidable but *highly* undecidable. (This kind of thing is catnip for mathematicians of a certain type.)

But by the 1960s, it was becoming clear that whether a problem is or is not decidable was far from being the end of the story. The fact that a problem was decidable in the sense of Turing's work did not mean that it was *practically* solvable at all: some problems that were solvable according to Turing's theory seemed to resist all practical attempts to attack them—they required impossible amounts of memory or were too impossibly slow to ever be practicable. And it was becoming uncomfortably obvious that many AI problems fell into this awkward class.

Here's an example of a problem that is easily seen to be solvable according to Turing's theory—the problem is rather famous, as computational problems go, and you may have heard of it. It is called the *traveling salesman problem:*[10]

A salesman must visit a group of cities in a road network, returning finally to his starting point. Not all the cities are connected by road, but for those that are, the salesman knows the shortest route. The salesman's car can drive for a certain, known number of miles on a single tank of gas. Is there a route for the salesman, which will visit all the cities and return to the starting destination, which can be done on a single tank of gas?

We can solve this problem, using naive search, just by listing all the possible tours of the cities and checking whether each tour is possible on a single tank of gas. However, as you might now guess, the number of possible tours increases very rapidly as the number of cities increases: for 10 cities, you would have to consider up to 3.6 million possible tours; for 11 cities, you would have to consider up to 40 million.

This is another example of combinatorial explosion. We were introduced to combinatorial explosion when looking at search trees, where each successive layer in the search tree multiplies the size of the tree.

Combinatorial explosion occurs in situations where you must make a series of successive choices: each successive choice *multiplies* the total number of possible outcomes.

So our naive approach, of exhaustively searching through all the candidate tours, is not going to be feasible. It works *in principle* (given enough time, we will get the right answer), but it doesn't work *in practice* (because the amount of time required to consider all the alternatives quickly becomes impossibly large).

But as we noted earlier, naive exhaustive search is a very crude technique. We could use heuristics, but they aren't guaranteed to work. Is there a *smarter* way that is *guaranteed* to find an appropriate tour, which doesn't involve exhaustively checking all the alternatives? *No!* You might find a technique that improves things marginally, but ultimately, *you won't be able to get around that combinatorial explosion.* Any recipe you find that is guaranteed to solve this problem is not going to be feasible for most cases of interest.

The reason for this is that our problem is an example of what is called an **NP-complete** problem. I'm afraid the acronym is not helpful for the uninitiated: *NP* stands for *non-deterministic polynomial time,* and the technical meaning is rather complex. Fortunately, the intuition behind NP-complete problems is simple.

An NP-complete problem is a combinatorial problem, like the traveling salesman problem, in which it is *hard to find solutions* (because there are too many of them to exhaustively consider each one, as we discussed above) but where it is *easy to verify whether you have found a solution* (in the traveling salesman problem, we can check a possible solution by simply verifying that the tour is possible on a single tank of gas).

To date, nobody has found an efficient recipe for any NP-complete problem. And the question of whether NP-complete problems can be efficiently solved is, in fact, one of the most important open problems in science today. It is called the **P-versus-NP problem,**[11] and if you can settle it one way or the other to the satisfaction of the scientific community, you stand to receive a prize of $1 million from the Clay Mathematics Institute, who in the year 2000 identified it as one of the "millennium problems" in mathematics. Smart money says that NP-complete problems *cannot* be solved efficiently; but smart money also says that we are a long way from knowing for certain whether this really is the case.

If you discover that a problem you are working on is NP-complete, then this tells you that techniques to solve it based on simple brute force are just not going to work: your problem is, in a precise mathematical sense, *hard.*

The basic structure of NP-complete problems was unraveled in the early 1970s, and researchers in AI started to look at the problems that they had been working on using the new theory. The results were shocking. Everywhere they looked—in problem solving, game playing, planning, learning, reasoning—it seemed that key problems were NP-complete (or worse). The phenomenon was so ubiquitous that it became a joke—the term *AI-complete* came to mean "a problem as hard as AI itself"—if you could solve one AI-complete problem, so the joke went, you would solve them all.

Before the theory of NP-completeness and its consequences were understood, there was always a hope that some sudden breakthrough would render these problems easy—**tractable,** to use the technical term. And technically, such a hope still remains, since we don't yet know for certain that NP-complete problems cannot be solved efficiently. But by the late 1970s, the specter of NP-completeness and combinatorial explosion began to loom large over the AI landscape. The field had hit a barrier, in the form of computational complexity, and progress ground to a halt. The techniques developed in the golden age seemed unable to scale up beyond toy scenarios like the Blocks World, and the optimistic predictions of rapid progress made throughout the 1950s and 1960s started to haunt the early pioneers.

AI AS ALCHEMY

By the early 1970s, the wider scientific community was beginning to be more and more frustrated by the very visible lack of progress made on core AI problems and by the continuing extravagant claims of some researchers. By the mid-1970s, the criticisms reached a fever pitch.

Of all the critics (and there were many), the most outspoken was the American philosopher Hubert Dreyfus. He was commissioned by the RAND Corporation in the mid-1960s to write a report on the state of progress in AI. The title he chose for his report, *Alchemy and AI,* made very clear his disdain for the field and those who worked in it. Publicly equating the work of a serious scientist to alchemy is extraordinarily insulting. Rereading *Alchemy and AI* now, the contempt is palpable and still rather shocking more than half a century later.

While we might frown upon the way in which Dreyfus articulated his critique of AI, it is hard to avoid the conclusion that he had a few valid points—particularly about the inflated claims and grand predictions of AI pioneers. In hindsight, it is painfully obvious that such claims and predictions were hopelessly unrealistic. I think at least some of this exuberance should be forgiven. In particular, there was no reason to suppose that the problems being tackled were *by their very nature* hard for computers to solve. Back then, there had always been the possibility that some breakthrough would render the difficult problems easy. But when understanding of NP-completeness began to spread, the community began to understand what it really meant for a computational problem to be *hard.*

THE END OF THE GOLDEN AGE

In 1972, a key funding body for scientific research in the UK asked the eminent mathematician Sir James Lighthill to evaluate the state of and prospects for AI. The request supposedly came after reports of academic infighting among members of the AI community at the University of Edinburgh, then and now one of the world's leading centers for AI research. Lighthill was at the time Lucasian Professor of Mathematics at the University of Cambridge, the most prestigious academic position that can be held by a UK mathematician (his successor in the post was Stephen Hawking). He had a wealth of experience in practical applied mathematics. However, reading the ***Lighthill Report*** today, he seems to have been left utterly bemused by his exposure to the AI culture of the time. His report was fiercely dismissive of mainstream AI—he specifically identified combinatorial explosion as a key problem that the AI community had failed to tackle. His report immediately led to severe funding cuts to AI research across the UK.

In the United States, the main funders of AI research historically had been the military funding agency DARPA and its predecessor, ARPA, but by the early 1970s, they too were becoming frustrated with the failure of AI to deliver on its many promises. Funding cuts to AI research in the United States followed.

The decade from the early 1970s to the early 1980s later became known as the **AI winter,** although it should perhaps better be known as the *first* AI winter, because there were more to come. The stereotype was established, of AI researchers making hopelessly overoptimistic and unwarranted predictions, then failing to deliver. Within the scientific community, AI began to acquire a reputation somewhat akin to homeopathic medicine. As a serious academic discipline, it appeared to be in terminal decline.