Introduction to AI, Search, and Problem Spaces

What is Artificial Intelligence?

Book's definition: AI is the study of how to make computers do things that people (generally) do better. AI is a relatively new field, that is still under change. Although it can be traced back to pre-1800’s (for an interesting read, look up "The Turk"), the field was not more formally defined until the late 40’s and into the 60’s and 70’s. AI has many intersections with other disciplines, and many approaches to the AI problem.

We will draw from many different areas that contribute to AI.

Define three task domains: Expert tasks (you might hire a professional consultant to do), formal tasks (logic, constraints), mundane tasks (common things you do every day).

Mundane:
- Vision, Speech
- Natural Language Processing, Generation, Understanding
- Reasoning
- Motion

Formal:
- Board Game-Playing, chess, checkers, billabong
- Logic
- Calculus
- Algebra
- Verification, Theorem Proving
People learn the mundane tasks first. The formal and expert tasks are the most difficult to learn. It made sense to focus early AI work on these task areas, in particular, playing chess, performing medical diagnosis, etc. However, it turns out that these expert tasks actually require much less knowledge than do the mundane skills. Consequently, AI is doing very well in the formal and expert tasks; however it is doing very poorly in the mundane tasks.

Example of a mundane task: You are hungry. You have the goal of not being hungry. What do you do to get food? To solve this problem, you have to know what constitutes edible food. You have to know where the food is located. If you do not know where the food is located, you have to find some way to find where it is located, such as looking in phone book or asking someone. You need to navigate to the food. Perhaps the food is in a restaurant. You need to know how to pay for the food, what a restaurant is, what money is, ways to communicate your goals to others, etc… the knowledge necessary to perform this simple task is enormous.

Mundane tasks and the area of broad knowledge understanding are sometimes referred to as “Commonsense Reasoning” and has been termed “AI-Complete” by some researchers.

Generality/Performance curve observed in current AI systems:

![Generality/Performance curve](image_url)
**Major branches of AI**

1) **Weak AI.** The study and design of machines that perform intelligent tasks. Not concerned with how tasks are performed, mostly concerned with performance and efficiency, such as solutions that are reasonable for NP-Complete problems. E.g., to make a flying machine, use logic and physics, don’t mimic a bird.

2) **Strong AI.** The study and design of machines that simulate the human mind to perform intelligent tasks. Borrow many ideas from psychology, neuroscience. Goal is to perform tasks the way a human might do them – which makes sense, since we do have models of human thought and problem solving. Includes psychological ideas in STM, LTM, forgetting, language, genetics, etc. Assumes that the physical symbol hypothesis holds.

3) **Evolutionary AI.** The study and design of machines that simulate simple creatures, and attempt to evolve and have higher level emergent behavior. For example, ants, bees, etc.

The so-called “Agent” approach to artificial intelligence spans all of these branches, and typically involves specific applications and activities. Agent-based activity has focused on the issues of:

1) **Autonomy.** Agents should be independent and communicate with others as necessary.
2) **Situated.** Agents should be sensitive to their own surroundings and context.
3) **Interactional.** Often an interface with not only humans, but also with other agents.
4) **Structured.** Agents cooperate in a structured society.
5) **Emergent.** Collection of agents more powerful than an individual agent.

**Philosophical Foundations**

Underlying assumption of Strong AI is the *physical symbol hypothesis*, defined by Newell and Simon in 1976.

Physical symbol hypothesis states: The thinking mind consists of the manipulation of symbols. That is, a physical symbol system has the necessary and sufficient means for general intelligent action.

If this hypothesis is true, then it means that a computer (which merely manipulates symbols) can perform generally intelligent actions. This claim has been rebuked by many researchers citing arguments of consciousness, self-awareness, or quantum theory. David Chalmers has proposed some interesting thought experiments if brain cells were replaced by transistors, and consciousness is graphed vs. transistors.

Turing Test: Proposed by Alan Turing in 1950 as a way to define intelligence. His test is that if the computer should be interrogated by a human through a modem or remote link, and passes the test if the interrogator cannot tell if there is a human or computer at the other end. No computer today can pass the test in a general domain, although computers have used “tricks” to pass in limited domains (e.g. Eliza, Julia). But is something intelligent if it is perceived to be intelligent?

Many complaints about the Turing Test; note that humans often mistake humans on the other end as computers! A famous argument is Searle’s Chinese Room. Consider a room, closed off from the world except for an envelope drop. Inside the room is a human with a rule book written in English and stacks of paper for writing. The rule book tells the human how to transcribe from Chinese to English.
Naturally, the set of rules is terribly complex, but one can imagine it possible. Now, if someone drops a letter written in Chinese through the slot, the human can follow the rules in the book (perhaps writing intermediate steps) and produce some English output. Question: Does the human understand Chinese? Searle says no, he is just following rules; consequently computers will never “understand” a language like Chinese the same was as humans. (Searle does claim consciousness is an emergent process of neural activity).

Other objections to the Turing test point out that it is biased purely toward symbolic problem-solving skills. Perceptual skill or manual dexterity are left out. Similarly, the test is biased towards humans – it may be possible to have intelligence that is entirely different from human intelligence. After all, why should a computer be as slow as humans to add numbers? Perhaps one of the largest objections is that of the situational intelligence required. To really pass the Turing Test, some have argued that a machine must be raised and brought up in the same culture and society of humans. How else would a machine know that it is not appropriate to call a “throne” a “chair”? (One answer is to painstakingly enter information like this by hand).

In 1990 Hugh Loebner agreed with The Cambridge Center for Behavioral Studies to underwrite a contest designed to implement the Turing Test. Dr. Loebner pledged a Grand Prize of $100,000 and a Gold Medal for the first computer whose responses were indistinguishable from a human's. Each year an annual prize of $2000 and a bronze medal is awarded to the most human computer. The winner of the annual contest is the best entry relative to other entries that year, irrespective of how good it is in an absolute sense. Here is a short snippet of interaction from Alice, the winning program from 2001:

JUDGE02[02:36:40] may I ask you some questions?
PROGRAM[02:36:40] Sure ask me anything.
JUDGE02[02:36:52] who composed Beethoven's piano concerto?
PROGRAM[02:36:52] Charlie Parker. :-)
JUDGE02[02:37:08] excellent. can you name another composer of music for me?
PROGRAM[02:37:08] Thanks for the compliment.
PROGRAM[02:37:08] Linus Torvalds. :-)
JUDGE02[02:37:21] have you heard of Caspar Hauser?
PROGRAM[02:37:21] No, tell me about him, her or it.
JUDGE02[02:37:40] later maybe. When did the 1820 settlers arrive?
PROGRAM[02:37:40] I understand.
PROGRAM[02:37:40] That is all water under the bridge.
JUDGE02[02:37:50] I agree. Are you a computer?
PROGRAM[02:37:50] You and I are on the same wavelength, Judge02.

As you can see, sometimes the program gives good answers, other times it only picks up on keywords. More information on the Loebner contest is available at http://www.loebner.net/Prizef/loebner-prize.html
AI Applications

Although AI has sometimes been loudly criticized by industry, the media, and academia, there have been many success stories. The criticism has come mainly as a result of hype. For many years, AI was hailed as solving problems such as natural language processing and commonsense reasoning, and it turned out that these problems were more difficult than expected. Here are just a few applications of artificial intelligence.

1. Game-playing. IBM’s deep-blue has beaten Kasparov, and we have a world-champion caliber Backgammon program. The success here is due to heuristic search and the brute-force power of computers. AI path-finding algorithms and strategy have also been applied to many commercial games, such as Quake or Command & Conquer.

2. Automated reasoning and theorem-proving. Newell and Simon are pioneers in this area, when they created the Logic Theorist program in 1963. Logic Theorist proved logical assertions and this helped define propositional calculus and eventually programming languages like Prolog. Formal mathematical logic has been important in fields like chip verification and mission-critical applications such as space missions.

3. Expert Systems. An expert system is a computer program with deep knowledge in a specific niche area that provides assistance to a user. Famous examples include DENDREAL, an expert system that inferred the structure of organic molecules from their spectrographic information, and MYCIN, an expert system that diagnosed blood diseases with better accuracy than human experts. More common examples of expert systems include programs like “Turbo Tax” or Microsoft’s help system. Typically, a human has to program the expert knowledge into these systems, and they operate only within one domain with little or no learning.

4. Machine Learning. Systems that can automatically classify data and learn from new examples has become more popular, especially as the Internet has grown and spawned applications that require personalized agents to learn a user’s interests. Some examples include cars capable of driving themselves, face and speech recognition, and Internet portals with pre-classified hierarchies.

5. Natural Language Understanding, Semantic Modeling. This area has been successful in limited domains. Most attention has shifted to a shallow understanding of natural language, i.e. witness the various search-engine technologies on the WWW, some that understand rudimentary questions.

6. Modeling Human Performance. As described earlier, machine intelligence need not pattern itself after human intelligence. Indeed, many AI programs are engineered to solve useful problems without regard for their similarities to human mental architecture. These systems give us another benchmark to understand and model human performance. Many cognitive scientists use computer techniques to construct their psychological models.

7. Planning and Robotics. Planning research began as an effort to design robots that could perform their task. For example, the Sojourner robot on Mars was able to perform some of its own navigation tasks since the time delay to earth makes real-time control impossible. Planning is the task of putting together some sequence of atomic actions to achieve a goal. This area of work extends beyond robots today; for example, consider a web “bot” that puts together a complete travel or vacation package for a customer. It must find reasonable connections and activities in each stop.

8. Languages and Environments. LISP and PROLOG were designed to help support AI, along with constructs such as object-oriented design and knowledge bases. Some of these ideas are now common in mainstream programming languages.
9. Alternative Representations, e.g. Neural Networks and Genetic Algorithms. These are bottom-up approaches to intelligence, based on modeling individual neurons in a brain or the evolutionary process.

10. AI and Philosophy. We have briefly touched on some of the philosophical issues, but there are many more. What happens if we do have intelligent computers? Should they have the same rights as people? What are the ethical issues? What is knowledge? Can knowledge be represented? The questions go on…

**Sampling of AI Techniques and Problems**: Tic-Tac-Toe, Question Answering

Tic-Tac-Toe: examine solution complexity, generalization, knowledge, extensibility

Program 1: Represent board in an array.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Each element holds “ “, “X”, or “O”

Possible states: (possible values)\(\text{elements}^3 = 3^9 = 19,683\). (less if account for symmetry, invalid states)

This is different than the number of games, or game paths, which loosely is 9! (less accounting for symmetry, some games complete earlier than others).

Algorithm for tic-tac-toe player:

- Create a table of size 19,683 containing all the states and the optimal move to make.
- Compute current state of board (O(1) operation)
- Lookup state in table (O(1) operation)
- Make move (O(1) operation)

Analysis:

Very fast, Could play optimal game, Someone has to sit down and enter all the moves - this is the knowledge problem. Would not work for larger boards - this is the scale/generalization problem. Consider chess - there are about 5\(^{44}\) states and 10\(^{120}\) game paths! Someone could never enter and analyze all these moves.

Program 2: Can keep board as before.

Algorithm: Compute in advance the moves to make.

- If computer moves first, go in center.
- If human moves first to upper-left corner, then move in center.
- If human moves first to center, move to upper left corner
- If human moves first to middle edge, move to center.
Analysis

Still runs in O(1) time, but not quite as fast as previous program since we have to do make several checks. May play optimally, but dependent on programmer’s tic-tac-toe knowledge. May not work as well for larger boards, does not scale well.

Modified Program 2: Keep algorithm the same, but change board slightly to make magic square:

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

Checking for a win is simplified for the computer - may have slight speedup. The numbers form a magic square, so if someone has won then the row, column, or diagonal should sum to 15.

Analysis

Board representation may have a profound impact on algorithm efficiency, so choose the representation for your board carefully. Note the difference between human win-checking and computer-win checking, humans have a much harder time using the magic square method.

Program 3:

Board representation the same.

Algorithm

To decide next move, look ahead at resulting board positions if certain moves are made.

Assign a rating for board states, and make those moves that lead to good states.

Seeing good positions:

1) See if your move is a win, if yes, give high rating

2) See what moves opponent can make. See which is worst by recursively calling the same function, and assume opponent will make that move. Can repeat process by examining what further moves we can make.

3) Choose move that results in the highest rating later on

Analysis

This technique is called minimax search. It is very general, scales up to larger games. Can use rules to limit the number of moves examined, as humans may do. May not too much for small problems.
Story: Goal is to have a computer be able to answer questions regarding a story.

“Lee was hungry. He went to McDonalds and ordered a Big Mac. It was bloody. In disgust, Lee left.”

Q1: Who was hungry?
Q2: What did Lee find that he disliked?
Q3: Did Lee buy anything?

Program 1
Data structures: Table matching template of questions to answers.
Algorithm:
1) Compare each element of the table to the question, and use all those that match to generate a set of text patterns.
2) Expand text patterns with verbs, adjectives, substitute in proper nouns, etc.
3) Reply with set of answers.
Example: “X saw Y” then if “Who saw Y?” reply with X.
Analysis
Q1 can match the template “Who was Y” and match that with the string “X was Y” to return X=Lee and print out “Lee was hungry” as the answer.
Q2 would not be answered unless the database knows about “disgust” leading to “dislike”. The database would also have to resolve the pronoun reference “it” to be the big mac. This question would probably not be answered.
Q3 is not specifically mentioned in the story and could not be answered. Consider the difference if Lee went to the Seven Glaciers Restaurant in Girdwood instead of McDonalds. Did he pay for the food then?

Program 2
Data Structures:
EnglishKnow - Knowledge of English language, semantics, syntactic constructs
StructuredText - Knowledge representation to capture input story. For example, the big mac could be represented as:

```
Food
Type = Meat
Animal = Cow
Name = Big Mac
Purchaser = Lee
```
This is a slot-filler structure we will discuss more later. These entities are connected to other entities that help give it meaning and an enriched representation.

Algorithm
- Convert story to StructuredText form, using EnglishKnow knowledge.
- Convert question to StructuredText form, using EnglishKnow knowledge.
- Match forms, use match to output answer to question.

Analysis
- Much more effective than previous approach since this forces us to use one explicit representation for knowledge and questions that might be posed in different ways; scale is still an issue, since all of the knowledge needs to be input somehow. A few efforts underway to do this on a large scale, incorporated World Knowledge in addition to just English Knowledge.

What we need to do in AI

- Search - find solution or knowledge from a large space
- Knowledge - must have enough knowledge to solve problems effectively and be useful in more than trivial problems
- Abstraction - solutions must be general enough to be useful and scale

Problem and Search Spaces

As described earlier, chess has approximately $10^{120}$ game paths. These positions comprise the problem search space. Typically, AI problems will have a very large space, too large to search or enumerate exhaustively.

Consider the state space for the Cannibals and Missionaries problem. You have 3 cannibals and 3 missionaries, all who have agreed they want to get to the other side of the river. You don’t want to ever have more cannibals than missionaries on one side alone or the cannibals may eat the missionaries. The boat available only holds two people.

State: number of cannibals and missionaries on each side of the river, location of boat

Production rules, or Move-Generator:

For side the boat is on:
- if c>=2 send 2 cannibals
- if c>=1 send 1 cannibals
- if c>=1 and m>=1 send 1 cannibal and 1 missionary
- if m>=1 send 1 missionary
- if m>=2 send 2 missionaries
Searching for a state: apply all applicable rules to the current state to generate a new state, and repeat.

Note the space grows exponentially, difficult and almost impossible to enumerate for large spaces, at least to find a goal state (in this case, the goal state is when all 6 people are on the other side of the river).

One solution:

Initial State: 3m3cb 0m0c
Send 2c: 3m1c 0m2cb
Return 1c 3m2cb 0m1c
Send 2c 3m0c 0m3cb
Return 1c 3m1cb 0m2c
Send 2m 1m1c 2m2cb
Return 1m1c 2m2cb 1m1c
Send 2m 0m2c 3m1cb
Return 1c 0m3cb 3m0c
Send 2c 0m1c 3m2cb
Return 1c 0m2cb 3m1c
Send 2c 0m0c 3m3cb

Note that this problem is somewhat difficult for people to solve because it involved “backwards” moves where we take away people from the other side of the river. People tend to think in terms of heuristics, which are essentially rules of thumb for progress, and these types of move violate the heuristic of “the more people on the other side of the river, the better”. We will also have the problem of returning back to states we’ve already been in, which can raise the potential for endless loops in searching for a solution.
Searching Problem Space

The problem space is a theoretical construct; the entire space “exists”. However, only a portion of this space need (or can) be represented in the computer. The rest will need to be generated. Question: best way to search/generate the nodes in the problem space?

Breadth First Search (BFS)

function BFS(state)

    node-list:=Apply-Production-Rules(state)  ; return all possible “moves”

    loop
        if node-list==empty then return false
        node:=Remove-Front(node-list)
        if Goal-Test(node) then return true
        node-list:=Enqueue-At-End(Apply-Production-Rules(node),node-list)

By expanding new states at the end of the node-list, BFS systematically explores all children of a given state first. (Show order of node generation on Missionary/Cannibal problem). This is different from “normal” BFS that is taught in algorithms classes because here we are not attempting to explore all states, but we are attempting to find a particular state. Consequently, not all states will be explicitly enumerated.

Advantages:

    Will never get trapped exploring a blind alley.

    Guaranteed to find solution, if it exists. Good if solution is only a few steps away.

Disadvantages:

    Time and memory may be poor.

    Call b the branching factor - the number of paths possible to take at a node. (show binary tree, branching factor=2). At depth=0, nodes=2^0=1. At depth=1, nodes=2^1=2. At depth=2, nodes=2^2=4… at depth=10 nodes=2^10=1024. In general, nodes = b^d. This is fine for b=2, but consider a game like GO where b=50. Then, if we want to look ahead to a depth of 6, we need to generate 50^6=15600000000 states. If each state required 100 bytes, this would occupy 1.56 terabytes (1560 gigabytes). Too large to do and even store in memory, let alone most hard drives!

    BFS requires O(b^d) space to store the results. Exponential time to generate! Not feasible for large problems, these could take years to compute.
**Depth-First Search (DFS)**

```plaintext
function DFS(state)
    node-list:=Apply-Production-Rules(state)
    loop
        if node-list==empty then return false
        node:=Remove-Front(node-list)
        if Goal-Test(node) then return true
        node-list:=Enqueue-At-Front(Apply-Production-Rules(node),node-list)
    end loop
end DFS
```

DFS always expands one of the nodes at the deepest level of the tree, and only when a dead end is hit, does search go back to expand nodes at shallower levels. (Show example on Missionary/Cannibals problem).

**Advantages:**
- Requires space only in storing the path traversed. For maximum depth \( m \), requires \( b^m \) nodes to be expanded, as opposed to \( b^d \) for BFS.
- Might get lucky and find solution right away.

**Disadvantages:**
- Still has \( O(b^m) \) time complexity.
- May get stuck following an unfruitful path, or never find a solution if stuck in loop.

**Iterative Deepening Search**

Later we will combine both DFS and BFS together to get something called A*. For now we will discuss DFID, Depth-First Iterative Deepening.

Tries to combine DFS and BFS by trying all possible depths, starting with a very short depth. Algorithm is to simply perform a DFS to depth 0, then to depth 1, then to depth 2, etc. This is similar to BFS, but uses the DFS algorithm instead. (Show example on Missionary/Cannibal problem).

**Advantages:**
- Optimal and complete; will find a solution if one exists.
- Same memory requirements as DFS.

**Disadvantages:**
- Some states will be expanded multiple times, especially initial moves and moves up in the search tree. BUT the computation in an exponential search tree is dominated by the number of leaves; i.e. \( O(b^{d+1}) \) dominates \( O(b^d) \). So this does not add significantly to runtime.
Ex: Tree with $b=10$, $d=5$. Number nodes = $1 + 10 + 100 + 1,000 + 10,000 + 100,000$

Using DFID:
- 1st pass we examine 1 node
- 2nd pass we examine $1 + 10 = 11$ nodes
- 3rd pass we examine $1 + 10 + 100 = 111$ nodes
- 4th pass we examine $1 + 10 + 100 + 1000 = 1111$ nodes
- 5th pass we examine $1 + 10 + 100 + 1000 + 10000 = 11,111$ nodes
- 6th pass we examine $1+10 +100+ 1000+10000+100000 = 111,111$ nodes
- Total nodes = 123,456. All previous passes contribute little, dominated by the leaves on the last level of computation.

In general, DFID is the preferred search method when there is a large search space and the depth of the solution is not known.

**Summary of BFS, DFS, DFID**

- $b$=branch factor, $d$=depth of solution, $m$=maximum depth of search tree

<table>
<thead>
<tr>
<th>Criteria</th>
<th>BFS</th>
<th>DFS</th>
<th>DFID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$bm$</td>
<td>$bd$</td>
</tr>
<tr>
<td>Optimal? (Find best solution)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete? (Guaranteed to find solution if exists)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Duplicate States**

So far we’ve been ignoring the possibility of making a cycle and returning to repeated states. In general, we don’t want to return to a duplicate state. Typical solution is to either ignore the problem or use a hash table to store visited states, and then check for them before visiting. $O(1)$ time, although $O(s)$ storage space required (using a good hash function).

**Constraints**

Constraints are one way to limit the search space. In the missionaries/cannibals problem, we could instead view the problem as having the constraint that we can’t make a move that results in more cannibals than missionaries, rather than create the death state. As a result, there are fewer moves that can be generated. Constraints are inherent in many search problems (e.g., cryptoarithmetic).
Example: CryptoArithmetic

\[
\begin{array}{ccccc}
S & E & N & D \\
+ & M & O & R & E \\
\hline
= & M & O & N & E & Y
\end{array}
\]

Here, no two letters have the same value. The sums of the digits must be as shown in the problem.

We could use normal DFS, BFS, or DFID. In this case, we’d assign a value to each letter until we find a solution that works. But the constraints of arithmetic help us solve the problem faster.

Steps:
1. Propagate constraints using the rules of the problem domain. In this case, the rules of arithmetic.
2. Guess value for some variable
3. Repeat process by propagating constraints based on the guess
4. If we reach a dead end, back up and take a different guess

How the constraints help:

Let’s rewrite the problem with carry’s:

\[
\begin{array}{cccc}
C_3 & C_2 & C_1 \\
S & E & N & D \\
+ & M & O & R & E \\
\hline
= & M & O & N & E & Y
\end{array}
\]

We know that M=1 because two single digits + a carry can’t be more than 19

If M=1, then S=8 or 9, has to be big enough to generate the carry

If M=1 and S=8 or 9, then the sum of M+S+C3 will be either 10 or 11. This means that O=0 or 1. But since no two letters can be the same, and M is already 1, then O=0.

If O=0, then then N=E+C2. This means that N=E or N=E+1. But since N can’t be E due to the constraint that no two letters be the same, then N=E+1.

If N=E+1, then C2=1

If C2=1, then N+R+C1\geq 10

Without doing any search we’ve already identified several variables!

When we can’t find any more constraints, we just need to guess a value for a variable.

Guess that E=2.

Then N=3

R=8 or 9
Guess that C1=1

Etc… until we find some set of variables that satisfy the equation. At this point we’re doing a normal DFS search, propagating constraints each time, to narrow what moves we can make next. It’s left as an exercise to the reader to find the rest of the numbers that satisfy the equation.

**Back to Searching:**

Covered so far: BFS, DFS, DFID

*Bi-Directional Search:*

Not used too often; idea is to search simultaneously both forward from the initial state, and backward from the goal, and stop when the two searches meet in the middle.

Ex: With Missionaries and Cannibals, search backwards from everyone on the right side of the river, along with searching everyone from the left.

If solution found at depth d, the solution will be found in \( O(2b^{d/2}) = O(b^{d/2}) \) steps since the algorithm will only have to search halfway. Can make a big difference! If b=10 and d=6, BFS generates 1,111,111 nodes, and bidirectional search only 2,222 nodes.

Problem: One search needs to retain in memory all nodes examined so it is known if they meet. Requires \( O(b^{d/2}) \) memory to store. Also, you may not know the goal state or there may be multiple goal states (e.g. chess).

**Generate and Test:**

Algorithm:
1) Generate possible solution. This might be a entire path from the start state to the end state (as in Cannibals/Missionaries) or it might be a single state (trying to find the right values to optimize some function, say, the cryptarithmetic problem \((SEND+MORE=MONEY)\)).
2) Test to see if solution works.
3) If found quit, otherwise return to step 1.

Can be done systematically as in DFS. May also be performed randomly to perform a more random search ; this has been called the British Museum Algorithm, named after visitors that randomly wander through the museum.

The algorithm is typically implemented through DFS with backtracking.
Means-End Analysis

Only discuss briefly here, see book for details (it’s discussed under Newell and Simon’s General Problem Solver). Inspired from plan of attack humans sometimes take to solve problems.

General plan of attack: Look at operators available. If one of them gets you closer to the goal state then now, take it. Repeat the process, results in a divide & conquer style search. If I want to get to somewhere else:

Start------------| Drive |-------------------------------Goal
               Drive precond   Drive postcond

Now, try to get from the start state to the Drive-Precond state, and from the Drive-Result/PostCond state to the goal state via recursive calls.

Best-First Search Methods

So called Best-First Search methods should really be named “Possibly Best-First Search” since if we actually made the best move possible at every move, we wouldn’t be searching at all. We’d just find a direct path to the goal.

Most searches are based on heuristics:

Heuristic \( h(n) \) = estimated cost of the cheapest path from the state at node \( n \) to a goal state
i.e. a guess as to the “goodness” of a particular state. A heuristic only applies to a particular state.

History: Heuristic comes from the Greek word “heuriskein” that means “to find”. Archimedes is supposed to have said “Heureka”, for ‘I have found it!’ not “Eureka”.

Sample heuristics:
Consider navigation through Anchorage in a car. You want to get to UAA from the convention center. There are buildings in the way, and also one way streets, so the problem is not trivial. A heuristic for how close you are to the goal might just be the air distance from your current location to the goal.
A second heuristic might be the Manhattan Distance, which is the sum of the horizontal and vertical distances to the goal.

Manhattan distance has been extensively studied with solving 8-puzzles.

Initial state:

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>
Goal state:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Heuristic function \( h = \text{sum of Manhattan distance for each tile} \)

**Hill Climbing**

If you view the heuristic function applied to child states, then you can “climb the hill” that results:

1) Evaluate initial state. If goal, then quit.
2) Loop until solution is found or no new operators left to apply to current state:
   3) Apply new operator to current state
   4) Evaluate new state. If goal, then quit. If not a goal but better than current state, then make it the current state. If worse than current state, continue looping.

Hill Climbing is often referred to as a “greedy” strategy since it follows whatever path looks best at the time. It also relies on an accurate heuristic to determine which operator to select.

The simple algorithm is most commonly used as **Steepest Ascent Hill Climbing**. That is, apply all operators to the current state, and then pick the best resulting state as the new state.

Example (show hill climbing for different goal=M and goal=K):

Problems:

Local maximum : stuck at small hill
Plateau: All neighboring states look the same (hard to find goal)

Ridge: Moves don’t allow moving to higher region (1-way street)

Solutions: Backtrack to old solutions, then pick new direction
Make a random jump
Genetic Algorithms
Apply other rules before making tests

*Simulated Annealing*

Method intended to reduce risk of local maxima and ridges. Based upon annealing of materials, such as steel. In annealing, a material is heated to a high temperature and then cooled according to an annealing scheduling. Quick/slow annealing results in material that may be brittle, smooth, etc. Typically the material is cooled to produce a minimal energy state - i.e. energy-min is the goal.

Allows for transition from low energy to high energy by:

\[ p = e^{-\Delta E/kT} \]

i.e. the probability of an uphill move decreases as the temperature decreases. This is analogous to initially allowing uphill moves, but as we zero in on the goal state, allow fewer and fewer uphill moves until a minimum is attained.
Algorithm:

1) If initial state is goal, quit
2) Set Best-So-Far to current state
3) Initialize T to annealing schedule (rate at which to cool)
4) Loop until solution is found or no new operators to apply
5) Apply new operator to current state
6) Compute change in energy; h(current) - h(new state)
7) If new state is goal, quit
8) If h(new state) > h (best-so-far) set best-so-far to new state and make it the current state
9) If not better than current state, make it the current state with probability \( p = e^{-\Delta E/kT} \); this probability decreases as T is lowered. Right units must be found, typically k is a unit conversion factor.

Things to fiddle with to improve performance based on problem: Annealing schedule, mapping values, backtracking, etc.

Used to be a hot algorithm in AI, would help solve many search problems, but is now not used very widely.

*Best-First Search*

There is an actual algorithm named “Best-First Search” which is essentially a general search using a heuristic function (hopefully this specific algorithm is not to be confused with the general class of best-first search algorithms). Unlike DFS or BFS, Best-First Search remembers all previous moves and takes the next move that looks best, according to the heuristic function.
Best-First-Search(start)
  open ← start
  closed ← empty
  while open <> empty do
    x ← remove leftmost state from open
    if Goal(x) return x
    succ ← Production-Rules(x)
    for each child c of succ do
      if c is not on open or closed
        assign c a heuristic value
        add c to open
      if c is on open
        if c was reached with a shorter cost
          assign shorter path to c
      if c is on closed
        if c was reached with a shorter cost
          assign shorter path to c
          remove from closed
          add c to open
    Add x to closed
  Reorder states on open by heuristic (leftmost node=best)
  Return failure

heuristic: higher=better

Show example on graph below. Cost of move=1
For now we’ll skip the case where we might have reached a node from a previous path. The cost in this case for each move is 1.

Start at A:
  Open = [ (A, 2) ]
  Generator successors for A: [ (B, 4) (C, 1) (D, 3) ]
  Open = [ (B,4) (D,3) (C,1) ]
  Closed = [ (A, 2) ]
  Generate successors for B: [ (E, 1) (F, 2) (G,2) ]
Open = [ (D, 3) (F, 2) (G, 2) (C, 1) (E, 1) ]
Closed = [ (B, 4) (A, 2) ]
Generate successors for D: [ (I, 4) ]

Open = [ (I, 4) (F, 2) (G, 2) (C, 1) (E, 1) ]
Closed = [ (B, 4) (D, 3) (A, 2) ]
Generate successors for I: [ (J, 6) (K, Goal) ]

Open = [ (K, Goal) (J, 6) (F, 2) (G, 2) (C, 1) (E, 1) ]
Closed = [ (B, 4) (D, 3) (I, 4) (A, 2) ]
K=Goal, quit

We didn’t come across the case where a succ node was in open or closed. We’ll look at this case next in A* search.

**A* Search: Minimize Total Cost**

h(n) attempts to address a way to minimize the cost to the goal state. Let’s define:

g(n) as the cost of the path so far.

A natural method to approach search is to minimize the total cost: f(n)=g(n)+h(n).
Consequently, f(n) should give us the estimated cost of the cheapest solution throughout the problem space. Note that if h(n)=0 then we essentially have BFS. Also note that here we are MINIMIZING the heuristic, not maximizing it. That is, a small heuristic value is better than a large one.

Algorithm:

Run Best-First Search with a heuristic of $f(n) = g(n) + h(n)$.
Small values are better than larger ones.

Example: Run on the same graph as before, but with heuristic values flipped so that smaller is better. The cost g(n) of making a move is 1.
Open = [ (A, 8) ]
Generator successors for A: [(B, 3) (C, 6) (D, 4)]

Open = [ (B,3) (D,4) (C,6) ]
Generate successors for B: [ (E, 7) (F, 6) (G,6) ]

Open = [ (D, 4) (C, 6) (F, 6) (G, 6) (E, 7) ]
Closed = [ (B, 3) (A, 8) ]
Generate successors for D: [ (I, 6) ]

Open = [ (I,6) (C, 6) (F, 6) (G, 6) (E, 7)]
Closed = [ (B, 3) (D, 4) (A,8)]
Generate successors for C (tie, we could have done I,C,F, or G): [(H,7)]

Open = [ (I,6)(F, 6) (G, 6) (E, 7) (H,7)]
Closed = [ (B, 4) (D, 3) (C,6) (A,8)]
Generator successors for I (tie): [ (J, 4) (K, 3) ]

Open = [ (K,3) (J,4) (F, 6) (G, 6) (E, 7) (H,7)]
Closed = [ (B, 4) (D, 3) (C,6) (I,6) (A,8) ]
K=Goal, Quit

More complicated example: Finding shortest path to goal. Use Manhattan distance as the heuristic, where the Manhattan distance is just the number of edges to the goal.
Start in lower left corner.

Open = [ (0,3 f=0+6) ]
Generate succ for (0,3): [ (0,2 f=2+5), (1,3 f=1+5) ]

Open = [(0,2 f=2+5), (1,3 f=1+5) ]
Closed = [ (0,3 g=0) ]
Generate succ for (1,3) : [ (1,2 f=6+4), (2,3 f=2+4), (0,3 f=2+6) ]
Since (0,3) is on closed but g=3 while old g=0, we don’t update (and we never will for the start node!)

Open = [ (2,3 f=6) (0,2 f=7) (1,2 f=10) ]
Closed = [ (1,3 g=1) (0,3 g=0) ]
Generate succ for (2,3) : [ (1,3 f=3+5) (2,2 f=6+3) (3,3 f=8+3) ]
Since (1,3) is on closed but g=3 while old g=1, we don’t update

Open = [ (0,2 f=7) (2,2 f=9) (1,2 f=10) (3,3 f=11) ]
Closed = [ (1,3 g=1) (2,3 g=2) (0,3 g=0)]
Generate succ for (0,2) : [ (0,1 f=6+4) (1,2 f=4+4) ]
(1,2) is on Open. We found a shorter path, so update it.

Open = [ (1,2 f=8) (2,2 f=9) (0,1 f=10) (3,3 f=11) ]
Closed = [ (1,3 g=1) (2,3 g=2) (0,2 g=2) ]
Generate succ for (1,2) : [ (0,2 f=6+5), (1,1 f=6+3) (2,2 f=7+3) ]
Ignore (0,2) since g > old path
Ignore (2,2) since g > old path

Open = [ (2,2 f=9) (1,1 f=9) (0,1 f=10) (3,3 f=11) ]
Closed = [ (1,2 g=4), (1,3 g=1) (2,3 g=2) (0,2 g=2) ]
Generate succ for (2,2) : (2, 1 f=7+2) (1,2 f=9+4) (2,3 f=10+4) ]
Ignore (1,2) since g > old path
Ignore (2,3) since g > old path

Open = [ (2, 1 f=9) (1,1 f=9) (0,1 f=10) (3,3 f=11) ]
Closed = [(1,2 g=4), (1,3 g=1) (2,3 g=2) (0,2 g=2) (2,2 g=6) ]
Generate succ for (2,1) : …. Etc. continue until reach the goal

Combines properties of breadth first search, but directs only along “good” paths.

About the heuristic, h:

If h is a perfect estimation of the actual distance to the goal, A* will converge immediately to the goal along the best path without doing any search.

What about the properties of finding the optimal solutions and ensuring that we will find a solution if one exists?

If h is possibly an overestimate of the distance to the goal, we could possibly be fooled into taking the wrong path and finding a non-optimal goal state. Consequently, we need the property that h always return an underestimate of the actual distance to the goal to have the guarantee that we find the optimal solution. This is called an admissible heuristic:

An admissible heuristic never overestimates the cost to reach the goal.

Note that the trivial admissible heuristic, h(n)=0, resorts to BFS which is optimal and complete.

If h(n) is admissible, then A* will find the optimal solution, and is complete on locally finite graphs (graphs with finite branching factor). The completeness follows from the monotonicity of h(n); i.e. h(a) <= h(b) + cost(a,b) if a is a successor of b. The h(n)-cost will generally decrease as search along the path progresses (and f(n) increase). Almost all admissible heuristics result in monotone increasing f(n) functions. If not, it’s possible to force an admissible heuristic to give monotone results.

To clarify on admissible heuristics and monotonicity:
Consider the following tree:

```
Above is a tree with a nonadmissible heuristic. Might find non-optimal answer.
A* search will go from A to B to E to F.
```
This is an admissible heuristic. Search does go the wrong path, to B first, and then it may go to E, but it will proceed to C and then D before going to F.

The property of *monotonicity* says that f will be *nondecreasing* as we move to child states if the heuristic is admissible. This is true in the second example; notice how f=1, 2, 3 or f=2,2 as we move along down the tree.

Computation Time: Depends on h(n), could result in same computation time as BFS (O(b^d)). The main problem is space: if the heuristic is not very good we could also generate exponential space to remember what nodes are on the OPEN list.

**IDA* - Iterative Deepening A**

Will only mention here; just as we did DFID, we can also perform Iterative Deepening on A* to remove the memory constraints. The idea is to perform a DFS using the f-cost as the limit rather than the depth-cost. Once again we will revisit some nodes, but the complexity remains the same, and we only use O(bd) storage for a solution at depth d to store each node along the solution path.

**AND-OR Graphs: Problem Reduction**

Useful for reducing problems to smaller subproblems.

Example: Impress a date. The numbers by the links are the arc costs. There are no child costs in this first example (i.e. cost to actually do the task in the box).
Cost of nodes:

- **AND**: Sum of child costs + arc costs
- **OR**: Minimum of children + arc costs

Can’t use A* to search due to the AND links. Can also be tricky with interacting subgoals, e.g. if borrowing clothes had a relation to goto store.

Algorithm:

1. Initialize graph to the starting node.
2. Loop until solved or cost is above futility threshold
   3. Compute f of all children of the current node
   4. Propagate their values back to the current node
   5. Repeat at the node with the best f value (may be an old node)

Example:

Initially: Only one node with heuristic guess. Cost of all arcs is 1 and the cost of executing the tasks is shown as h.

\[
\text{Goal: Impress Date}\hspace{1cm} h=5
\]

Expand children:
Choose Comb Hair, since cheapest. Expand this one:

Comb hair is now more expensive than getting money, so expand getting money:

Update Get Money to 6, since in the OR we pick the lowest value, that of stealing money. The cheapest path is now back to getting hair, so we would continue by expanding Grow Hair and Buy Comb.