Unit I

Introduction to AI and Searching

Introduction to AI – AI Applications – Problem solving agents – Search algorithms – Uninformed search strategies – Heuristic Search Strategies: A* algorithm – Game Playing: Alpha Beta Pruning – Constraint Satisfaction Problem

1. Introduction to AI

* Concept of AI

- How to think?
- How to make system think?

→ Steps involved in process of thinking

- perceive
- understand
- predict
- manipulate

* Definitions of AI

- "Branch of computer Sience that is concerned with the automation of intelligent behaviour" (Luger - 1993)
- "The study of how to make computers work (or) do things at which, at the moment people are better" (Rich and Knight - 1991)
- "The art of creating machines that perform functions that require intelligence, when performed by people" (Kurzweil - 1990)
- "The study of the computations that make it possible to perceive, reason and act". (Winston - 1992)

* Approaches followed in AI

① Acting Humanly

- Computers need to possess the following capabilities:

(i) Natural language processing (NLP): To enable it to communicate successfully in English

(ii) Knowledge representation – To store what it knows, what it hears

(iii) Automated reasoning – To make use of stored information to answer questions being asked and to draw conclusions

(iv) Machine Learning – To adapt to new circumstances and to detect and make new predictions by finding patterns

(v) Computer vision – To perceive objects

(vi) Robotics :- To manipulate objects

② Thinking Humanly

- Human minds can be explored through
  - Introspection (catch your own thoughts)
  - psychological experiments

- Cognitive Science : Brings together computer models from AI and experimental techniques from psychology that try to construct precise and testable theories of the workings of human mind.

③ Thinking Rationally – the "laws of thought approach"

- Thinking rationally means thinking rightly
- Eg: If something is true it must be true
- Laws of thought approach – If someone thinking rightly always in a given circumstance with a given amount of information.
- Eg: "Socrates is a man, all men are mortal, therefore Socrates is mortal
- Approach needs 100% knowledge
- More computation required

④ Acting Rationally: Rational Agent Approach

- Agent: Anything that perceives its environment through sensor and acts upon that environment through effectors.
- Human Agent
  - ↳ Sensors – Sense organs (Eyes, ears, nose, tongue, skin)
  - ↳ Effectors – Hands, legs, mouth
- Robotic Agent
  - ↳ Sensors – Cameras, infrared range finders
  - ↳ Effectors – Motors, Actuators
- S/w Agent
  - ↳ Sensors – Keystroke, file content, received N/w packages
  - ↳ Effectors – Display on the screen, file, sent network (N/w) packages

2. AI Applications

- The state of the art means the various applications of AI in different fields

① Autonomous planning and scheduling

- NASA developed Remote Agent program
- First on-board autonomous planning program
- Controls the scheduling of operations for spacecraft
- Such remote agents can do:
  - ↳ Detecting of problems
  - ↳ Diagnosing of problems
  - ↳ Recovering from problems

} as they occured

② Game playing

- Deep Blue – A computer chess program
- Developed by IBM
- Defeated World Chess champion Garory Kasparov in exhibition match in 1997
- Such games can be developed using AI

③ Autonomous Control

- ALVINN – Computer Vision system
- Trained to steer car to keep it following a lane
- Made to travel 2850 miles
- 98% of the time control was with the system
- 2% of the time human took over

④ Diagnosis

- Machine can explain the diagnosis
- Machine points out the major factors influencing its decision
- Explain interaction of several of the symptoms

⑤ Logistic Planning

- DART
- Automated Logistics planning and scheduling the transportation
- US forces deployed this dynamic analysis and replanning tool during the Persian Gulf Crisis in 1991

⑥ Robotics

- To do complex and critical tasks
- Eg: Surgeons can use robot assistants in microsurgery which can generate 3D vision of patients internal anatomy

⑦ Language understanding and problem Solving

- PROVERB - A computer program which is an expert in solving crossword puzzles
- It uses
  - ↳ Constraints or possible word fillers
  - ↳ Large database of past puzzles
  - ↳ Variety of information sources including dictionaries and online databases.

BENEFITS OF ARTIFICIAL INTELLIGENCE

① Reduction in Human Error

- Decrease errors
- Increase accuracy
- Increase precision

② Zero Risks

- Accurate work with greater responsibility
- Not wear out easily

③ 24 x 7 Availability

- AI can work endlessly without breaks
- Think faster
- Multiple tasks at a time
- Accurate results.

④ Digital Assistance

- User-requested content in websites

⑤ New Inventions

- AI is the driving force behind numerous innovations that will aid humans in resolving the majority of challenging issues.

⑥ Unbiased Decisions

AI is,
- Devoid of emotions
- Highly practical
- Rational

⑦ Perform Repititive Jobs

- Automate menial chores
- Eliminate "boring" tasks for people

⑧ Daily Applications

- Google Maps
- Alexa
- Siri
- Cortona on Windows
- OK Google

(9) AI in Risky Situations

- Natural (or) Manmade calamity

(10) Medical Applications

- Diagnosis
- Treatment
- Drug discovery
- clinical trials
- Analyze patient data
- Identify potential health risks
- Develop personalized treatment plans

RISKS (OR) DISADVANTAGES OF AI

(1) High cost

- Requires plenty of time
- Requires plenty of resources
- Can cost a huge deal of money
- Latest H/w and S/w

(2) No Creativity

- Cannot think outside the box
- AI
  - ↳ Learn over time
  - ↳ Pre-fed data
  - ↳ Past experiences

(3) Unemployment

- ↳ Displace occupations
- ↳ Increase unemployment
- ↳ Chatbots and robots replacing humans

② No Creativity

- Cannot think outside the box
- AI
  ↳ Learn over time
  ↳ Pre-fed data
  ↳ Past experiences

③ Unemployment

↳ Displace occupations
↳ Increase unemployment
↳ Chatbots and robots replacing humans

④ Make Humans Lazy

↳ Automate tedious and repetitive tasks

⑤ No Ethics

↳ Ethics and morality can be difficult to incorporate into an AI.

⑥ Emotionless

↳ Neither computer nor other machines have feelings

⑦ No improvement

↳ Machines can only complete tasks they have been developed or programmed for.
↳ Unable to make anything conventional

Nature of Environment

- **Task environments**, are essentially the "problems" to which rational agents are the "solutions.

a) Specifying the task environment (PEAS Elements)

- In designing an agent, the first step must always be to specify the task environment as fully as possible.
- Problem: An automated taxi driver.

- We should point out, before the reader becomes alarmed, that a fully automated taxi is currently somewhat beyond the capabilities of existing technology.
- The full driving task is extremely open-ended.
- There is no limit to the novel combinations of circumstances that can arise—another reason we chose it as a focus for discussion.
- Figure 2.4 summarizes the PEAS description for the taxi's task environment.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

- First, what is the ***performance measure*** to which we would like our automated driver to aspire? Desirable qualities include
  - getting to the correct destination;
  - minimizing fuel consumption and wear and tear;
  - minimizing the trip time or cost;
  - minimizing violations of traffic laws and disturbances to other drivers;
  - maximizing safety and passenger comfort;
  - maximizing profits.
  - Obviously, some of these goals conflict, so tradeoffs will be required.
- What is the driving ***environment*** that the taxi will face?
  - Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways.
  - The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles, and potholes.
  - The taxi must also interact with potential and actual passengers.
  - There are also some optional choices.
  - The taxi might need to operate in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not.
  - It could always be driving on the right, or we might want it to be flexible enough to drive on the left when in Britain or Japan.
  - Obviously, the more restricted the environment, the easier the design problem
- The ***actuators*** for an automated taxi include those available to a human driver:
  - control over the engine through the accelerator and control over steering and braking.
  - In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise.
- The basic ***sensors*** for the taxi will include one or more controllable video cameras so that it can see the road; it might augment these with infrared or sonar sensors to detect distances to other cars and obstacles.
  - To avoid speeding tickets, the taxi should have a speedometer, and to control the vehicle properly, especially on curves, it should have an accelerometer.
  - To determine the mechanical state of the vehicle, it will need the usual array of engine, fuel, and electrical system sensors.

- Like many human drivers, it might want a global positioning system (GPS) so that it doesn't get lost.
- Finally, it will need a keyboard or microphone for the passenger to request a destination.

- Some software agents (or software robots or ***softbots***) exist in rich, unlimited domains.
- Imagine a softbot Web site operator designed to scan Internet news sources and show the interesting items to its users, while selling advertising space to generate revenue.
- To do well, that operator will need some natural language processing abilities, it will need to learn what each user and advertiser is interested in, and it will need to change its plans dynamically—for example, when the connection for one news source goes down or when a new one comes online.
- The Internet is an environment whose complexity rivals that of the physical world and whose inhabitants include many artificial and human agents.

b) Properties of task environments

- ***Fully observable vs. partially observable***:
  - If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
  - A task environment is effectively fully observable if the sensors detect all aspects that are relevant to the choice of action; relevance, in turn, depends on the performance measure.
  - ***Fully observable*** environments are convenient because the agent need not maintain any internal state to keep track of the world.
  - An environment might be ***partially observable*** because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data—for example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, and an automated taxi cannot see what other drivers are thinking.
  - If the agent has no sensors at all then the environment is ***unobservable***.
- ***Single agent vs. multiagent***:
  - The distinction between single-agent and multiagent environments may seem simple enough.
  - For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment.
  - There are, however, some subtle issues.
  - First, we have described how an entity may be viewed as an agent, but we have not explained which entities must be viewed as agents.
  - Does an agent A (the taxi driver for example) have to treat an object B (another vehicle) as an agent, or can it be treated merely as an object behaving according to the laws of physics, analogous to waves at the beach or leaves blowing in the wind?
  - The key distinction is whether B's behavior is best described as maximizing a performance measure whose value depends on agent A's behavior.
  - For example, in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure.
  - Thus, chess is a ***competitive*** multiagent environment. In the taxi-driving environment, on the other hand, avoiding collisions maximizes the performance measure of all agents, so it is a partially ***cooperative*** multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space.
  - The agent-design problems in multiagent environments are often quite different from those in single-agent environments; for example, communication often emerges as a rational behavior in multiagent environments; in some competitive environments, ***randomized*** behavior is rational because it avoids the pitfalls of predictability.

- ***Deterministic vs. stochastic.***
  - If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.
  - In principle, an agent need not worry about uncertainty in a fully observable, ***deterministic*** environment. (In our definition, we ignore uncertainty that arises purely from the actions of other agents in a multiagent environment; thus, a game can be deterministic even though each agent may be unable to predict the actions of the others.)
  - If the environment is partially observable, however, then it could appear to be stochastic.
  - Most real situations are so complex that it is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as ***stochastic***.
  - Taxi driving is clearly stochastic in this sense, because one can never predict the behavior of traffic exactly; moreover, one's tires blow out and one's engine seizes up without warning.
  - The vacuum world as we described it is deterministic, but variations can include stochastic elements such as randomly appearing dirt and an unreliable suction mechanism.
  - We say an environment is ***uncertain*** if it is not fully observable or not deterministic.
  - The word "stochastic" generally implies that uncertainty about outcomes is quantified in terms of probabilities; a ***nondeterministic*** environment is one in which actions are characterized by their possible outcomes, but no probabilities are attached to them. Nondeterministic environment descriptions are usually associated with performance measures that require the agent to succeed for all possible outcomes of its actions.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

- ***Episodic vs. sequential***:
  - In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action.
  - Crucially, the next episode does not depend on the actions taken in previous episodes.
  - Many classification tasks are episodic.

- For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective.
- In sequential environments, on the other hand, the current decision could affect all future decisions.
- Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences.
- Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

- ***Static vs. dynamic***:
  - If the environment can change while an agent is deliberating, then we say the environment is ***dynamic*** for that agent; otherwise, it is static.
  - ***Static*** environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time.
  - Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing.
  - If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is ***semidynamic***.
  - Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next.
  - Chess, when played with a clock, is semidynamic.
  - Crossword puzzles are static.
- ***Discrete vs. continuous***:
  - The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.
  - For example, the chess environment has a finite number of distinct states (excluding the clock).
  - Chess also has a discrete set of percepts and actions.
  - Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
  - Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

- ***Known vs. unknown***:
  - In a known environment, the outcomes (or outcome probabilities if the environment is stochastic) for all actions are given.

- Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.
- Note that the distinction between known and unknown environments is not the same as the one between fully and partially observable environments.
- It is quite possible for a known environment to be partially observable—for example, in solitaire card games, I know the rules but am still unable to see the cards that have not yet been turned over.
- Conversely, an unknown environment can be fully observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them.

Structure of Agent

- The job of AI is to design an ***agent program*** that implements the agent function — the mapping from percepts to actions.
- We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the architecture:

agent = architecture + program

a) Agent Program

- The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.
- For example, Figure 2.7 shows a rather trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do.
- The table—an example of which is given for the vacuum world in Figure 2.3—represents explicitly the agent function that the agent program embodies.
- To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.
- The TABLE – DRIVEN – AGENT implements the desired agent function.

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

b) Simple reflex agents

- The simplest kind of agent is the simple reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history.
- For example, the vacuum agent whose agent function is tabulated in Figure 2.3 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt.
- An agent program for this agent is shown in Figure 2.8.
- Notice that the vacuum agent program is very small indeed compared to the corresponding table.
- The most obvious reduction comes from ignoring the percept history, which cuts down the number of possibilities from 4T to just 4.
- A further, small reduction comes from the fact that when the current square is dirty, the action does not depend on the location.

- Simple reflex behaviors occur even in more complex environments.
- Imagine yourself as the driver of the automated taxi.
- If the car in front brakes and its brake lights come on, then you should notice this and initiate braking.
- In other words, some processing is done on thevisual input to establish the condition we call "The car in front is braking."
- Then, this triggers some established connection in the agent program to the action "initiate braking."
- We call such a connection a ***condition–action rule***, written as

if car-in-front-is-braking then initiate-braking.

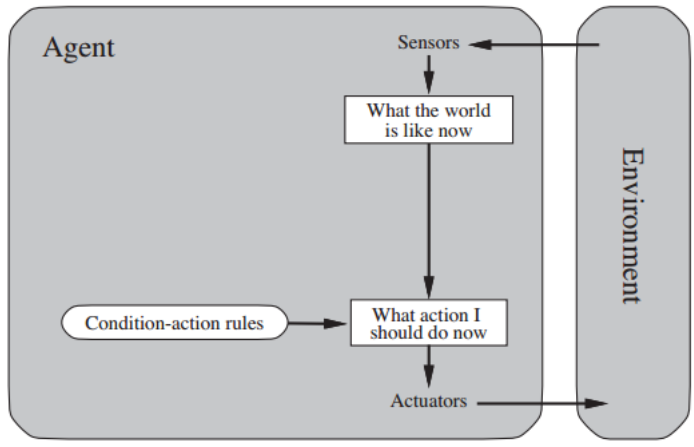


Figure 2.9 Schematic diagram of a simple reflex agent.

- Humans also have many such connections, some of which are learned responses (as for driving) and some of which are innate reflexes (such as blinking when something approaches the eye).
- The program in Figure 2.8 is specific to one particular vacuum environment.
- A more general and flexible approach is first to build a general-purpose interpreter for condition–action rules and then to create rule sets for specific task environments.
- Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.
- We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.
- The agent program, which is also very simple, is shown in Figure 2.10.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

- The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULE-MATCH function returns the first rule in the set of rules that matches the given state description.
- Note that the description in terms of "rules" and "matching" is purely conceptual; actual implementations can be as simple as a collection of logic gates implementing a Boolean circuit.
- Simple reflex agents have the admirable property of being simple, but they turn out to be of limited intelligence.

- The agent in Figure 2.10 will work only if the correct decision can be made on the basis of only the current percept—that is, only if the environment is fully observable.
- Even a little bit of unobservability can cause serious trouble.
- For example, the braking rule given earlier assumes that the condition car-in-front-is-braking can be determined from the current percept—a single frame of video.
- This works if the car in front has a centrally mounted brake light.
- Unfortunately, older models have different configurations of taillights brake lights, and turn-signal lights, and it is not always possible to tell from a single image whether the car is braking.
- A simple reflex agent driving behind such a car would either brake continuously and unnecessarily, or, worse, never brake at all.
- We can see a similar problem arising in the vacuum world. Suppose that a simple reflex vacuum agent is deprived of its location sensor and has only a dirt sensor.
- Such an agent has just two possible percepts: [Dirty] and [Clean]. It can Suck in response to [Dirty]; what should it do in response to [Clean]? Moving Left fails (forever) if it happens to start in square A, and moving Right fails (forever) if it happens to start in square B. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.
- Escape from infinite loops is possible if the agent can **randomize** its actions.
- For example, if the vacuum agent perceives [Clean], it might flip a coin to choose between Left and Right.
- It is easy to show that the agent will reach the other square in an average of two steps.
- Then, if that square is dirty, the agent will clean it and the task will be complete.
- Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.
- In single-agent environments, randomization is usually not rational.
- It is a useful trick that helps a simple reflex agent in some situations, but in most cases we can do much better with more sophisticated deterministic agents.

c) ***Model – based reflex agents***

- The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now.
- That is, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.
- For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously.
- For other driving tasks such as changing lanes, the agent needs to keep track of where the other cars are if it can't see them all at once.
- And for any driving to be possible at all, the agent needs to keep track of where its keys are.
- Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program.
- First, we need some information about how the world evolves independently of the agent—for example, that an overtaking car generally will be closer behind than it was a moment ago.
- Second, we need some information about how the agent's own actions affect the world—for example, that when the agent turns the steering wheel clockwise, the car turns to the right, or that after driving for five minutes northbound on the freeway, one is usually about five miles north of where one was five minutes ago.
- This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a model of the world.

- An agent that uses such a model is called a ***model-based agent***.

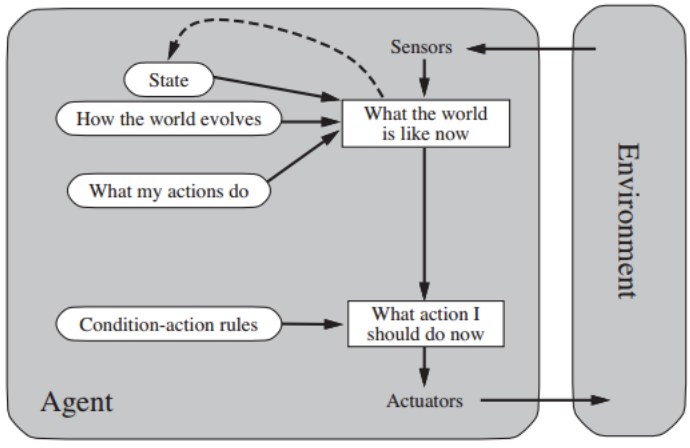


Figure 2.11 A model-based reflex agent.

- Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works.
- The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design. Detailed examples of models and updating algorithms appear in Chapters 4, 12, 11, 15, 17, and 25.
- Regardless of the kind of representation used, it is seldom possible for the agent to determine the current state of a partially observable environment exactly.
- Instead, the box labeled "what the world is like now" (Figure 2.11) represents the agent's "best guess" (or sometimes best guesses).
- The agent program is shown in Figure 2.12.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

- For example, an automated taxi may not be able to see around the large truck that has stopped in front of it and can only guess about what may be causing the hold-up.
- Thus, uncertainty about the current state may be unavoidable, but the agent still has to make a decision.
- A perhaps less obvious point about the internal "state" maintained by a model-based agent is that it does not have to describe "what the world is like now" in a literal sense.
- For example, the taxi may be driving back home, and it may have a rule telling it to fill up with gas on the way home unless it has at least half a tank.
- Although "driving back home" may seem to an aspect of the world state, the fact of the taxi's destination is actually an aspect of the agent's internal state.
- If you find this puzzling, consider that the taxi could be in exactly the same place at the same time, but intending to reach a different destination.

d) Goal – based Agent

- Knowing something about the current state of the environment is not always enough to decide what to do.
- For example, at a road junction, the taxi can turn left, turn right, or go straight on.
- The correct decision depends on where the taxi is trying to get to.
- In other words, as well as a current state description, the agent needs some sort of ***goal*** information that describes situations that are desirable—for example, being at the passenger's destination.
- The agent program can combine this with the model (the same information as was used in the model based reflex agent) to choose actions that achieve the goal.
- Figure 2.13 shows the goal-based agent's structure.
- Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action.
- Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal.
- Search and planning are the subfields of AI devoted to finding action sequences that achieve the agent's goals.
- Notice that decision making of this kind is fundamentally different from the condition–action rules described earlier, in that it involves consideration of the future—both "What will happen if I do such-and-such?" and "Will that make me happy?"
- In the reflex agent designs, this information is not explicitly represented, because the built-in rules map directly from percepts to actions.
- The reflex agent brakes when it sees brake lights. A goal-based agent, in principle, could reason that if the car in front has its brake lights on, it will slow down.
- Given the way the world usually evolves, the only action that will achieve the goal of not hitting other cars is to brake.
- Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.
- If it starts to rain, the agent can update its knowledge of how effectively its brakes will operate; this will automatically cause all of the relevant behaviors to be altered to suit the new conditions.
- For the reflex agent, on the other hand, we would have to rewrite many condition–action rules.
- The goal-based agent's behavior can easily be changed to go to a different destination, simply by specifying that destination as the goal.
- The reflex agent's rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

e) Utility – based agents

- Goals alone are not enough to generate high-quality behavior in most environments.
- For example, many action sequences will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others.
- Goals just provide a crude binary distinction between "happy" and "unhappy" states.
- A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term ***utility*** instead.
- We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination.
- An ***agent's utility function*** is essentially an internalization of the performance measure.

- If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure.
- Let us emphasize again that this is not the only way to be rational—we have already seen a rational agent program for the vacuum world (Figure 2.8) that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning.
- Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions.
- First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff.
- Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.
- Partial observability and stochasticity are ubiquitous in the real world, and so, therefore, is decision making under uncertainty.
- Technically speaking, a rational utility-based agent chooses the action that maximizes the ***expected utility*** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome.
- Any rational agent must behave as if it possesses a utility function whose expected value it tries to maximize.
- An agent that possesses an explicit utility function can make rational decisions with a general-purpose algorithm that does not depend on the specific utility function being maximized.
- In this way, the "global" definition of rationality—designating as rational those agent functions that have the highest performance—is turned into a "local" constraint on rational-agent designs that can be expressed in a simple program.

f) Learning Agents

- A learning agent can be divided into four conceptual components, as shown in Figure 2.15.

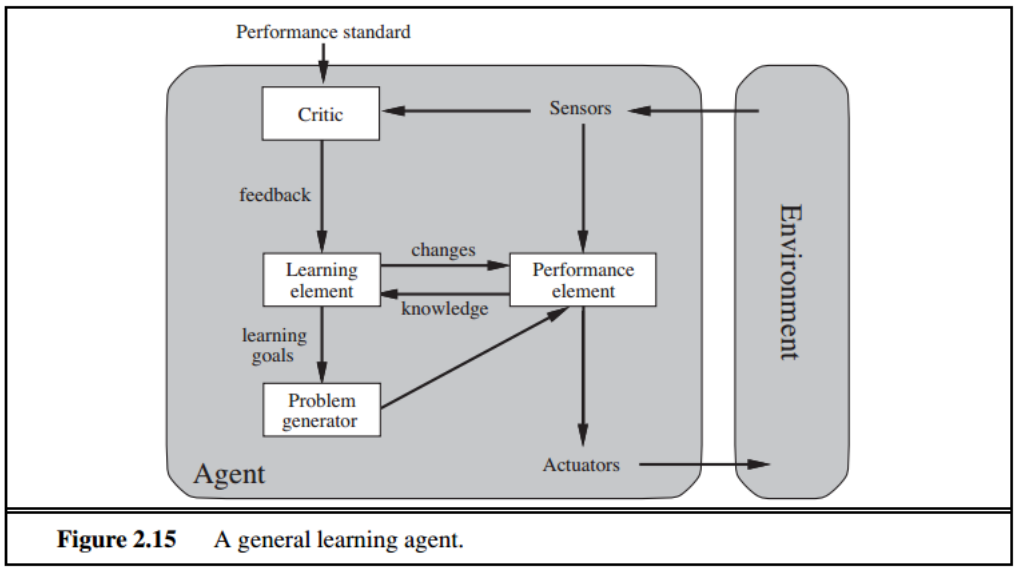


Figure 2.15 A general learning agent.

- The most important distinction is between the learning element, which is responsible for making improvements, and the ***performance element***, which is responsible for selecting external actions.
- The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

- The learning element uses feedback from the ***critic*** on how the agent is doing and determines how the performance element should be modified to do better in the future.
- The design of the learning element depends very much on the design of the performance element.
- When trying to design an agent that learns a certain capability, the first question is not "How am I going to get it to learn this?" but "What kind of performance element will my agent need to do this once it has learned how?" Given an agent design, learning mechanisms can be constructed to improve every part of the agent.
- The critic tells the learning element how well the agent is doing with respect to a fixed performance standard.
- The critic is necessary because the percepts themselves provide no indication of the agent's success.
- For example, a chess program could receive a percept indicating that it has checkmated its opponent, but it needs a performance standard to know that this is a good thing; the percept itself does not say so.
- It is important that the performance standard be fixed.
- Conceptually, one should think of it as being outside the agent altogether because the agent must not modify it to fit its own behavior.
- The last component of the learning agent is the **problem generator**.
- It is responsible for suggesting actions that will lead to new and informative experiences.
- The point is that if the performance element had its way, it would keep doing the actions that are best, given what it knows.
- But if the agent is willing to explore a little and do some perhaps suboptimal actions in the short run, it might discover much better actions for the long run.
- The problem generator's job is to suggest these exploratory actions.
- This is what scientists do when they carry out experiments.
- Galileo did not think that dropping rocks from the top of a tower in Pisa was valuable in itself.
- He was not trying to break the rocks or to modify the brains of unfortunate passers-by.
- His aim was to modify his own brain by identifying a better theory of the motion of objects.
- To make the overall design more concrete, let us return to the automated taxi example.
- The performance element consists of whatever collection of knowledge and procedures the taxi has for selecting its driving actions.
- The taxi goes out on the road and drives, using this performance element.
- The critic observes the world and passes information along to the learning element.
- For example, after the taxi makes a quick left turn across three lanes of traffic, the critic observes the shocking language used by other drivers.
- From this experience, the learning element is able to formulate a rule saying this was a bad action, and the performance element is modified by installation of the new rule.
- The problem generator might identify certain areas of behavior in need of improvement and suggest experiments, such as trying out the brakes on different road surfaces under different conditions.
- The learning element can make changes to any of the "knowledge" components shown in the agent diagrams (Figures 2.9, 2.11, 2.13, and 2.14).
- The simplest cases involve learning directly from the percept sequence.
- Observation of pairs of successive states of the environment can allow the agent to learn "How the world evolves," and observation of the results of its actions can allow the agent to learn "What my actions do."
- For example, if the taxi exerts a certain braking pressure when driving on a wet road, then it will soon find out how much deceleration is actually achieved.
- Clearly, these two learning tasks are more difficult if the environment is only partially observable.

- The forms of learning in the preceding paragraph do not need to access the external performance standard—in a sense, the standard is the universal one of making predictions that agree with experiment.
- The situation is slightly more complex for a utility-based agent that wishes to learn utility information.
- For example, suppose the taxi-driving agent receives no tips from passengers who have been thoroughly shaken up during the trip.
- The external performance standard must inform the agent that the loss of tips is a negative contribution to its overall performance; then the agent might be able to learn that violent maneuvers do not contribute to its own utility.
- In a sense, the performance standard distinguishes part of the incoming percept as a reward (or penalty) that provides direct feedback on the quality of the agent's behavior.

Hard-wired performance standards such as pain and hunger in animals can be understood in this way.

---

3. Problem Solving Agents

- Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.
- The agent's performance measure contains many factors:
  - it wants to improve its suntan,
  - improve its Ro-manian, take in the sights, enjoy the nightlife (such as it is), avoid hangovers, and so on.
  - The decision problem is a complex one involving many tradeoffs and careful reading of guide books. Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day.
  - In that case, it makes sense for the agent to adopt the goal of getting to Bucharest.
  - Courses of action that don't reach Bucharest on time can be rejected without further consideration and the agent's decision problem is greatly simplified.
- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- ***Goal formulation***, based on the current situation and the agent's performance measure, is the first step in problem solving.
- We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied.
- The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.
- Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider.
- If it were to consider actions at the level of "move the left foot forward an inch" or "turn the steering wheel one degree left," the agent would probably never find its way out of the parking lot, let alone to Bucharest, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution.
- ***Problem formulation*** is the process of deciding what actions PR and states to consider, given a goal. We discuss this process in more detail later.
- For now, let us assume that the agent will consider actions at the level of driving from one major town to another.
- Each state therefore corresponds to being in a particular town.
- Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.
- In other words, the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action.

- But suppose the agent has a map of Romania.
- The point of a map is to provide the agent with information about the states it might get itself into and the actions it can take.
- We assume that the environment is ***observable***, so the agent always knows the current state.
- For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers.
- We also assume the environment is ***discrete***, so at any given state there are only finitely many actions to choose from. This is true for navigating in Romania because each city is connected to a small number of other cities.
- We will assume the environment is ***known***, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.)
- Finally, we assume that the environment is ***deterministic***, so each action has exactly one outcome. Under ideal conditions, this is true for the agent in Romania—it means that if it chooses to drive from Arad to Sibiu, it does end up in Sibiu.
- Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey.
- In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.
- The process of looking for a sequence of actions that reaches the goal is called ***search***.
- A search algorithm takes a problem as input and returns a ***solution*** in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the ***execution*** phase.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

- Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1.
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it.
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence.
- Once the solution has been executed, the agent will formulate a new goal.
- While the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance what they will be.
- An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going OPEN-LOOP on.

- Control theorists call this an open-loop system, because ignoring the percepts breaks the loop between agent and environment.

Well – defined problems and solutions

- A ***problem*** can be defined formally by five components:
  - The ***initial state*** that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad)
  - A description of the possible ***actions*** available to the agent. Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is ***applicable*** in s. For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.
  - A description of what each action does; the formal name for this is the ***transition model***, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term ***successor*** to refer to any state reachable from a given state by a single action.2 For example, we have

RESULT(In(Arad),Go(Zerind)) = In(Zerind)

Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or ***graph*** in which the nodes are states and the links between nodes are actions. (The map of Romania shown in Figure 3.2 can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction.) A ***path*** in the state space is a sequence of states connected by a sequence of actions.

  - The ***goal test***, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set {In(Bucharest)}. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.
  - A ***path cost*** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. We assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The ***step cost*** of taking action a in state s to reach state s is denoted by c(s, a, s). The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are nonnegative.

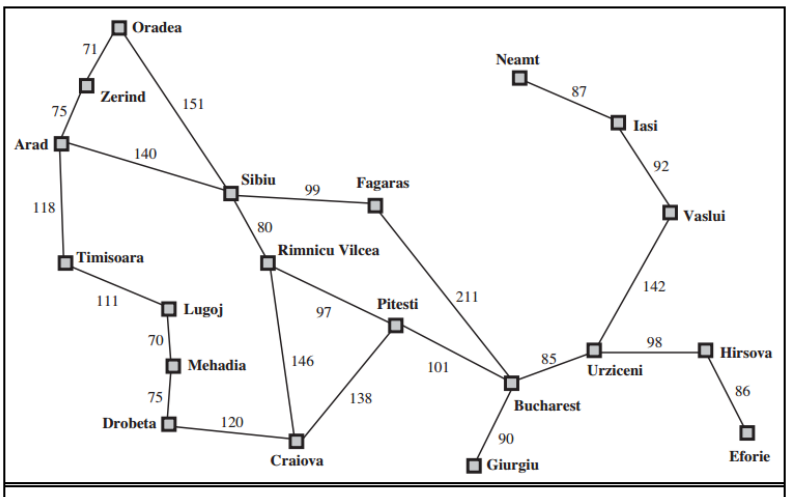


Figure 3.2 A simplified road map of part of Romania.

- In(Arad), to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, and so on.
- All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called ***abstraction***.

Formulating Problems

Toy Problem:

A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.

Real – World Problem:

A real-world problem is one whose solutions people care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

Toy Problems:

a) ***Vacuum World***

This can be formulated as a problem as follows:

- ***States*:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2n$ states.
- ***Initial state:*** Any state can be designated as the initial state.
- ***Actions***: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
- ***Transition model***: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure 3.3.
- ***Goal test***: This checks whether all the squares are clean.
- ***Path cost***: Each step costs 1, so the path cost is the number of steps in the path.

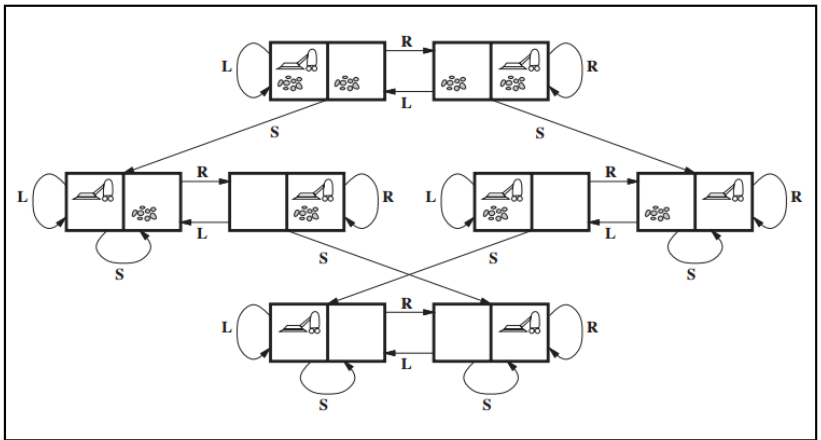


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

b) *8-puzzle*

The 8-puzzle, an instance of which is shown in Figure 3.4, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- ***States***: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- ***Initial state***: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- ***Actions***: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
- ***Transition model***: Given a state and action, this returns the resulting state; for example if we apply Left to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- ***Goal test***: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- ***Path cost***: Each step costs 1, so the path cost is the number of steps in the path.
- ***Abstraction***: The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding. We have abstracted away actions such as shaking the board when pieces get stuck and ruled out extracting the pieces with a knife and putting them back again. We are left with a description of the rules of the puzzle, avoiding all the details of physical manipulations.
- The 8-puzzle belongs to the family of ***sliding-block puzzles***, which are often used as test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has 9!/2 = 181, 440 reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5 × 5 board) has around 1025 states, and random instances take several hours to solve optimally.

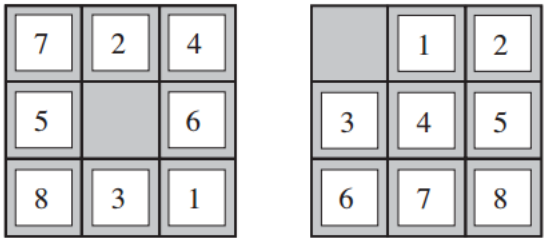


Figure 3.4 A typical instance of the 8-puzzle.

c) *8 – Queens Problem*

- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.)
- Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.
- Although efficient special-purpose algorithms exist for this problem and for the whole n-queens family, it remains a useful test problem for search algorithms.
- There are two main kinds of formulation.
- An ***incremental formulation*** involves operators that augment the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.

- A ***complete-state formulation*** starts with all 8 queens on the board and moves them around.
- In either case, the path cost is of no interest because only the final state counts.
- The first incremental formulation one might try is the following:
- ***States***: Any arrangement of 0 to 8 queens on the board is a state.
- ***Initial state***: No queens on the board.
- ***Actions***: Add a queen to any empty square.
- ***Transition model***: Returns the board with a queen added to the specified square.
- ***Goal test***: 8 queens are on the board, none attacked.
- In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ possible sequences to investigate.
- A better formulation would prohibit placing a queen in any square that is already attacked:
- ***States***: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- ***Actions***: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- This formulation reduces the 8-queens state space from 1.8×1014 to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the reduction is from roughly 10400 states to about 1052 states — a big improvement, but not enough to make the problem tractable.

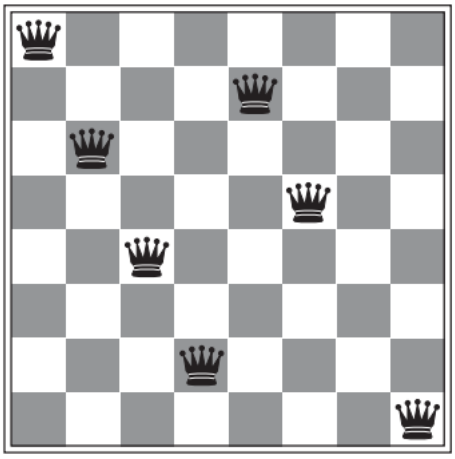


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

d) How infinite spaces can arise?

- Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.
- For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

The problem definition is very simple:

- ***States***: Positive numbers.
- ***Initial state***: 4.
- **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
- ***Transition model***: As given by the mathematical definitions of the operations.
- ***Goal test***: State is the desired positive integer.

To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target—for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5—so the state space for this problem is infinite. Such state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

Real-World Problems

a) Route Finding Problem

- The route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example.
- Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.
- Consider the airline travel problems that must be solved by a travel-planning Web site:
- ***States***: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- ***Initial state***: This is specified by the user's query.
- ***Actions***: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- ***Transition model***: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- ***Goal test***: Are we at the final destination specified by the user?
- ***Path cost***: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

b) Touring Problems

- Touring problems are closely related to route-finding problems, but with an important difference.
- Consider, for example, the problem "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities.
- The state space, however, is quite different.
- Each state must include not just the current location but also the set of cities the agent has visited.
- So, the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui), Visited({Bucharest, Urziceni, Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

c) Traveling Salesman Problem (TSP)

- The traveling salesperson problem (TSP) is a touring problem in which each city
- must be visited exactly once. The aim is to find the shortest tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.
- In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

d) VLSI Layout

- A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- The layout problem comes after the logical design phase and is usually split into two parts: cell layout and channel routing.

- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function.
- Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells.
- The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.
- Channel routing finds a specific route for each wire through the gaps between the cells.
- These search problems are extremely complex, but worth solving.

e) Robot Navigation

- Robot navigation is a generalization of the route-finding problem described earlier.
- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional.
- When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional.
- Advanced techniques are required just to make the search space finite.
- In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls

f) Automatic Assembly Sequencing

- Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972).
- Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible.
- In assembly problems, the aim is to find an order in which to assemble the parts of some object.
- If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.
- Thus, the generation of legal actions is the expensive part of assembly sequencing.
- Any practical algorithm must avoid exploring all but a tiny fraction of the state space.

g) Protein Design

- Another important assembly problem is protein design, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

---

4. Search Algorithms

- Uninformed Search Strategies
- Heuristic Search Strategies

---

5. Uninformed Search Strategies

The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded.

The following are a few examples of uninformed search strategies:

- Breadth First Search
- Uniform – Cost Search
- Depth – First Search
- Depth – Limited Search

Breadth First Search

- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breadth-first search is an instance of the general graph-search algorithm (Figure 3.7) in which the shallowest unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier.
- Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity.
- Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found.
- Thus, breadth-first search always has the shallowest path to every node on the frontier.
- Pseudocode is given in Figure 3.11.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

- Figure 3.12 shows the progress of the search on a simple binary tree.

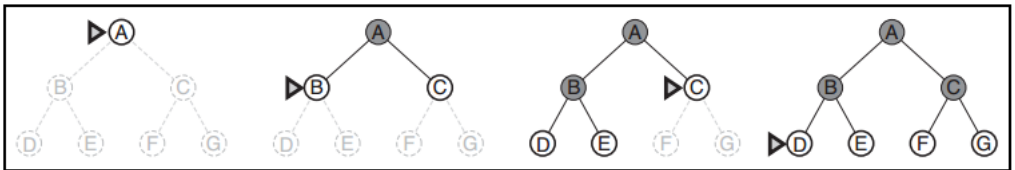


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

How does breadth-first search rate according to the four criteria from the previous section?

- We can easily see that it is complete—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test.
- Now, the shallowest goal node is not necessarily the optimal one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node.
- The most common such scenario is that all actions have the same cost.
- So far, the news about breadth-first search has been good.

- The news about time and space is not so good.
- Imagine searching a uniform tree where every state has b successors.
- The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.
- Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d) .$$

- (If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be O(bd+1).)
- As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity.
- For breadth-first graph search in particular, every node generated remains in memory.
- There will be O(bd−1) nodes in the explored set and O(bd) nodes in the frontier, so the space complexity is O(bd), i.e., it is dominated by the size of the frontier.
- Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time.
- An exponential complexity bound such as O(bd) is scary. Figure 3.13 shows why. It lists, for various values of the solution depth d, the time and memory required for a breadth-first search with branching factor b = 10. The table assumes that 1 million nodes can be generated per second and that a node requires 1000 bytes of storage.
- Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

- Two lessons can be learned from Figure 3.13.
- First, the memory requirements are a bigger problem for breadth-first search than is the execution time. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Fortunately, other strategies require less memory.

The second lesson is that time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it. In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Dijkstra's algorithm or uniform-cost search

- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost. uniform-cost search does not care about the number of steps a path has, but only about their total cost.

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, uniform-cost search UNIFORM-COST SEARCH expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g. The algorithm is shown in Figure 3.14.
- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is selected for expansion (as in the generic graph-search algorithm shown in Figure 3.7) rather than when it is first generated.
- The reason is that the first goal node that is generated may be on a suboptimal path.
- The second difference is that a test is added in case a better path is found to a node currently on the frontier.
- Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+ 97+ 101 = 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.
- It is easy to see that uniform-cost search is optimal in general. First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found. (Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n, by the graph separation property of Figure 3.9; by definition, n would have lower g-cost than n and would have been selected first.) Then, because step costs are nonnegative, paths never get shorter as nodes are added.
- These two facts together imply that uniform-cost search expands nodes in order of their optimal path cost.
- Hence, the first goal node selected for expansion must be the optimal solution.
- Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of No Op actions.6 Completeness is guaranteed provided the cost of every step exceeds some small positive constant .
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d. Instead, let C∗ be the cost of the optimal solution,7 and assume that every action costs at least .
- Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than bd. This is because uniform-cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, is just bd+1. When all step $b^{1+\lfloor C^*/\epsilon \rfloor}$ costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

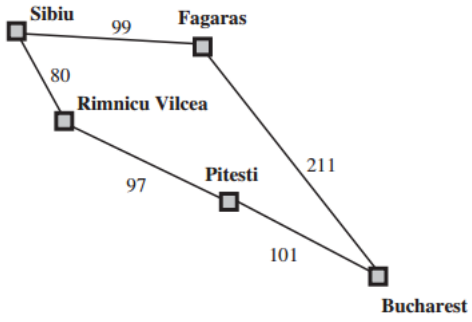


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Depth First Search

- Depth-first-search always expands the deepest node in the current fringe of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.
- Depth-first-search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored(Refer Figure 2.12).
- For a state space with a branching factor b and maximum depth m, depth-first-search requires storage of only bm + 1 nodes.
- Using the same assumptions as Figure 2.11,and assuming that nodes at the same depth as the goal node have no successors, we find the depth-first-search would require 118 kilobytes instead of 10 petabytes,a factor of 10 billion times less space.

- The progress of the search is illustrated in figure 3.16

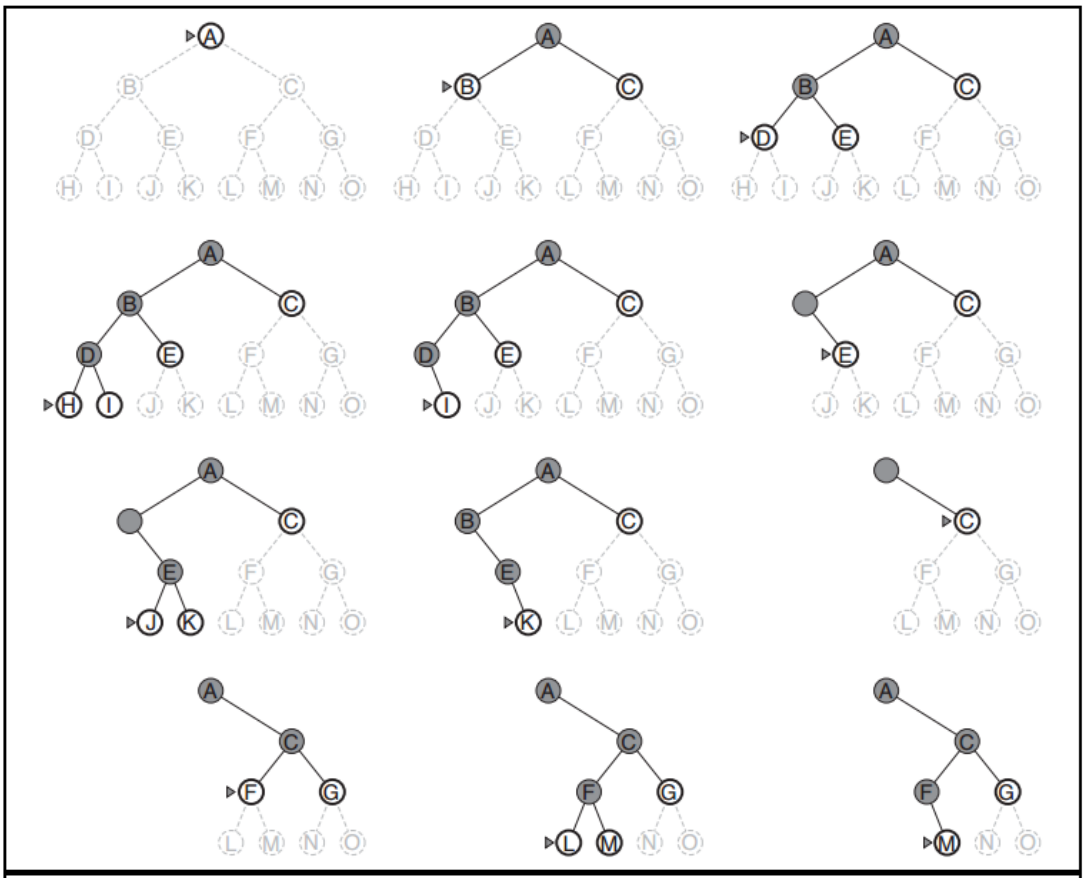


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Drawback of Depth-first-search

- The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree.
- For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

- A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

Depth Limited Search

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l. Nodes at depth l are treated as if they have no successors. This approach is called depth-limited-search.

- The depth limit solves the infinite path problem.
- Depth limited search will be nonoptimal if we choose l > d.
- Its time complexity is O(bl) and its space complexity is O(bl).
- Depth-first-search can be viewed as a special case of depth-limited search with l = oo
- Sometimes, depth limits can be based on knowledge of the problem.
- For example, on the map of Romania there are 20 cities.
- Therefore, we know that if there is a solution., it must be of length 19 at the longest, So l = 10 is a possible choice.

- However, it can be shown that any city can be reached from any other city in at most 9 steps.
- This number known as the diameter of the state space, gives us a better depth limit.
- Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm.
- The pseudocode for recursive depth-limited-search is shown in Figure 3.17**.**
- It can be noted that the above algorithm can terminate with two kinds of failure : the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.
- Depth-limited search = depth-first search with depth limit l

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

6. Heuristic Search Strategies: A* algorithm

- Informed search algorithms can do quite well given some guidance on where to look for solutions
- Strategies that know whether one non – goal state is "more promising" than another are called ***informed search or heuristic search strategies***
- ***Informed search strategy uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.***
- ***Best-first search*** is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, f(n).
- The ***evaluation function*** is construed as a cost estimate, so the node with the lowest evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search (Figure 3.14), except for the use of f instead of g to order the priority queue.
- The choice of f determines the search strategy.
- Most best-first algorithms include as a component of f a ***heuristic function, denoted h(n)***:
  h(n) = estimated cost of the cheapest path from the state at node n to a goal state.
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. W
- For now, we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n)=0.

Greedy Best First search

- ***Greedy best-first search*** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n).
- Let us see how this works for route-finding problems in Romania; we use the straight-line distance heuristic, which we will call hSLD .
- If the goal is Bucharest, we need to know the ***straight-line distances*** to Bucharest, which are shown in Figure 3.22.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

- For example, hSLD (In(Arad)) = 366. Notice that the values of hSLD cannot be computed from the problem description itself.
- Moreover, it takes a certain amount of experience to know that hSLD is correlated with actual road distances and is, therefore, a useful heuristic.
- Figure 3.23 shows the progress of a greedy best-first search using hSLD to find a path from Arad to Bucharest.

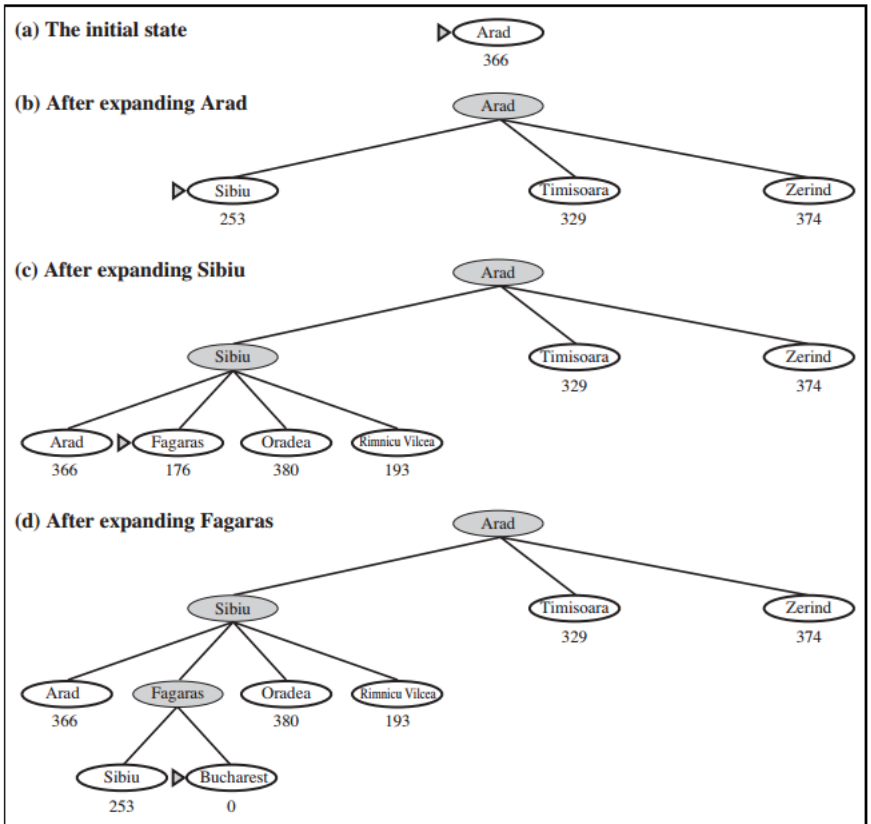


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara.
- The next node to be expanded will be Fagaras because it is closest.

- Fagaras in turn generates Bucharest, which is the goal.
- For this particular problem, greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.
- It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.
- This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can.
- Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search. Consider the problem of getting from Iasi to Fagaras.
- The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end.
- The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras.
- The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop.
- The worst-case time and space complexity for the tree version is O(bm), where m is the maximum depth of the search space.
- With a good heuristic function, however, the complexity can be reduced substantially.
- The amount of the reduction depends on the particular problem and on the quality of the heuristic.

---

7. A* Search

- The most widely known form of best-first search is called ***A* search*** (pronounced "A-star search").
- It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

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

- Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \text{estimated cost of the cheapest solution through } n .$$

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n).
- It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A* search is both complete and optimal.
- The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses g + h instead of g.

Conditions for optimality: Admissibility and consistency

- The ***first condition*** we require for optimality is that h(n) be an ***admissible heuristic***.
- An ***admissible heuristic*** is one that never overestimates the cost to reach the goal. Because g(n) is the actual cost to reach n along the current path, and f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.
- An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest.
- Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

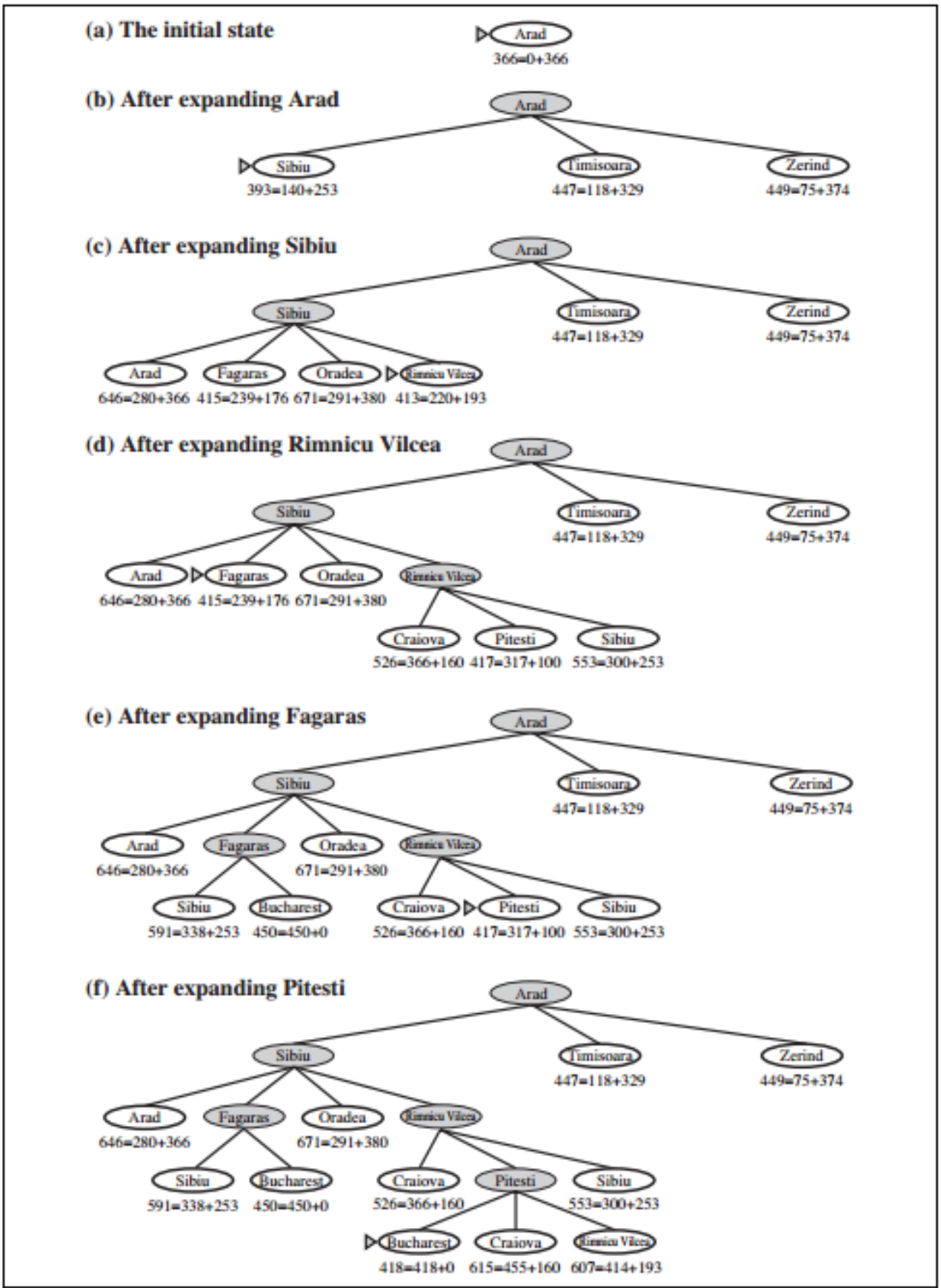


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

- In Figure 3.24, we show the progress of an A* tree search for Bucharest.
- The values of g are computed from the step costs in Figure 3.2, and the values of hSLD are given in Figure 3.22.

- Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417).
- Another way to say this is that there might be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.
- A ***second, slightly stronger condition*** called ***consistency*** (or sometimes ***monotonicity***) is required only for applications of A* to graph search.
- A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n:

$$h(n) \leq c(n, a, n) + h(n) .$$

- This is a form of the general ***triangle inequality***, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides.
- Here, the triangle is formed by n, n',and the goal G_n closest to n.
- For an admissible heuristic, the inequality makes perfect sense: if there were a route from n to Gn via n' that was cheaper than h(n), that would violate the property that h(n) is a lower bound on the cost to reach G_n.
- It is fairly easy to show that every consistent heuristic is also admissible.
- Consistency is therefore a stricter requirement than admissibility, but one has to work quite hard to concoct heuristics that are admissible but not consistent.
- Consider, for example, hSLD .
- We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance and that the straight-line distance between n and n is no greater than c(n, a, n').
- Hence, hSLD is a consistent heuristic.

Optimality of A*

- A* has the following properties: the tree-search version of A* is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent
- We show the second of these two claims since it is more useful.
- The argument essentially mirrors the argument for the optimality of uniform-cost search, with g replaced by f—just as in the A* algorithm itself.
- The first step is to establish the following: if h(n) is consistent, then the values of f(n) along any path are nondecreasing.
- The proof follows directly from the definition of consistency.
- Suppose n is a successor of n; then g(n) = g(n) + c(n, a, n') for some action a, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

- The next step is to prove that whenever A* selects a node n for expansion, the optimal path to that node has been found. Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n, by the graph separation property of Figure 3.9; because f is nondecreasing along any path, n would have lower f-cost than n and would have been selected first.
- From the two preceding observations, it follows that the sequence of nodes expanded by A∗ using GRAPH-SEARCH is in nondecreasing order of f(n).
- Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes(which have h = 0) and all later goal nodes will be at least as expensive.
- The fact that f-costs are nondecreasing along any path also means that we can draw ***contours*** in the state space, just like the contours in a topographic map.
- Figure 3.25 shows an example.

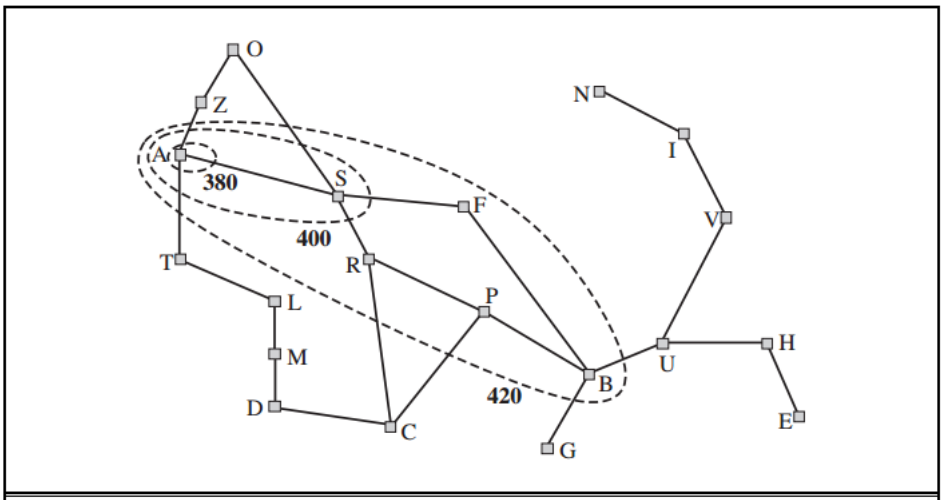


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

- Inside the contour labeled 400, all nodes have f(n) less than or equal to 400, and so on.
- Then, because A* expands the frontier node of lowest f-cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost
- With uniform-cost search (A∗ search using h(n)=0), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path.
- If C∗ is the cost of the optimal solution path, then we can say the following:
  - A* expands all nodes with f(n) < C*
  - A* might then expand some of the nodes right on the "goal contour" (where f(n) = C*) before selecting a goal node
- Completeness requires that there be only finitely many nodes with cost less than or equal to C*, a condition that is true if all step costs exceed some finite and if b is finite.
- Notice that A∗ expands no nodes with f(n) > C*—for example, Timisoara is not expanded in Figure 3.24 even though it is a child of the root.
- We say that the subtree below Timisoara is ***pruned***; because hSLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality.
- The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.
- One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root and use the same heuristic information—A* is ***optimally efficient*** for any given consistent heuristic.
- That is, no other optimal algorithm is guaranteed to expand fewer nodes than A∗ (except possibly through tie-breaking among nodes with f(n) = C*). This is because any algorithm that does not expand all nodes with f(n) < C* runs the risk of missing the optimal solution
- That A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying.
- Unfortunately, it does not mean that A* is the answer to all our searching needs.
- The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution.
- The details of the analysis are beyond the scope of this book, but the basic results are as follows.
- For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the the ***absolute error*** or the ***relative error*** of the heuristic.

- The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal
- The relative error is defined as $\equiv (h^* - h)/h^*$.
- The complexity results depend very strongly on the assumptions made about the state space.
- The simplest model studied is a state space that has a single goal and is essentially a tree with reversible actions. (The 8-puzzle satisfies the first and third of these assumptions.)
- In this case, the time complexity of A* is exponential in the maximum absolute error, that is,$O(b^{\epsilon d})$.
- For constant step costs, we can write this as O(bd), where d is the solution depth.
- For almost all heuristics in practical use, the absolute error is at least proportional to the path cost h^*, so is constant or growing and the time complexity is exponential in d.
- We can also see the effect of a more accurate heuristic: $O(b^{\epsilon d}) = O((b^{\epsilon})^{d})$, so the effective branching factor (defined more formally in the next section) is b^{ϵ}.
- When the state space has many goal states—particularly near-optimal goal states—the search process can be led astray from the optimal path and there is an extra cost proportional to the number of goals whose cost is within a factor of the optimal cost.
- Finally, in the general case of a graph, the situation is even worse.
- There can be exponentially many states with $f(n) < C^*$ even if the absolute error is bounded by a constant.
- For example, consider a version of the vacuum world where the agent can clean up any square for unit cost without even having to visit it: in that case, squares can be cleaned in any order.
- With N initially dirty squares, there are 2N states where some subset has been cleaned and all of them are on an optimal solution path—and hence satisfy $f(n) < C^*$—even if the heuristic has an error of 1.
- The complexity of A* often makes it impractical to insist on finding an optimal solution.
- One can use variants of A* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible.
- In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search.
- Computation time is not, however, A*'s main drawback.
- Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A* usually runs out of space long before it runs out of time.
- For this reason, A* is not practical for many large-scale problems.

There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

---

8. Game Playing (Adversarial Game and Search)

- The competitive environments, in which the agents' goals are in conflict, giving rise to adversarial search problems—often known as games.

Game theory

- Mathematical game theory, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.
- In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, ***zero-sum games*** of ***perfect information*** (such as chess).
- In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses.
- It is this opposition between the agents' utility functions that makes the situation adversarial.

- Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed.
- For AI researchers, the abstract nature of games makes them an appealing subject for study.
- The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules. Physical games, such as croquet and ice hockey, have much more complicated descriptions, a much larger range of possible actions, and rather imprecise rules defining the legality of actions.
- With the exception of robot soccer, these physical games have not attracted much interest in the AI community.
- Games, unlike most of the toy problems are interesting because they are too hard to solve.
- For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35100 or 10154 nodes (although the search graph has "only" about 1040 distinct nodes).
- Games, like the real world, therefore require the ability to make some decision even when calculating the optimal decision is infeasible.
- Games also penalize inefficiency severely.
- Whereas an implementation of A* search that is half as efficient will simply take twice as long to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal.
- Game-playing research has therefore spawned a number of interesting ideas on how to make the best possible use of time
- We begin with a definition of the optimal move and an algorithm for finding it.
- We then look at techniques for choosing a good move when time is limited.
- ***Pruning*** allows us to ignore portions of the search tree that make no difference to the final choice, and heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search.
- Finally, we look at how state-of-the-art game-playing programs fare against human opposition and at directions for future developments.
- We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious.
- MAX moves first, and then they take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.
- A game can be formally defined as a kind of search problem with the following elements:
- S0: The initial state, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The transition model, which defines the result of a move.
- TERMINAL-TEST(s): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- UTILITY(s, p): A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2 . Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192. A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either 0+1, 1+0 or 1/2 + 1/2. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of 1/2 .

- The initial state, ACTIONS function, and RESULT function define the game tree for the game—a tree where the nodes are game states and the edges are moves.
- Figure 5.1 shows part of the game tree for tic-tac-toe (noughts and crosses).

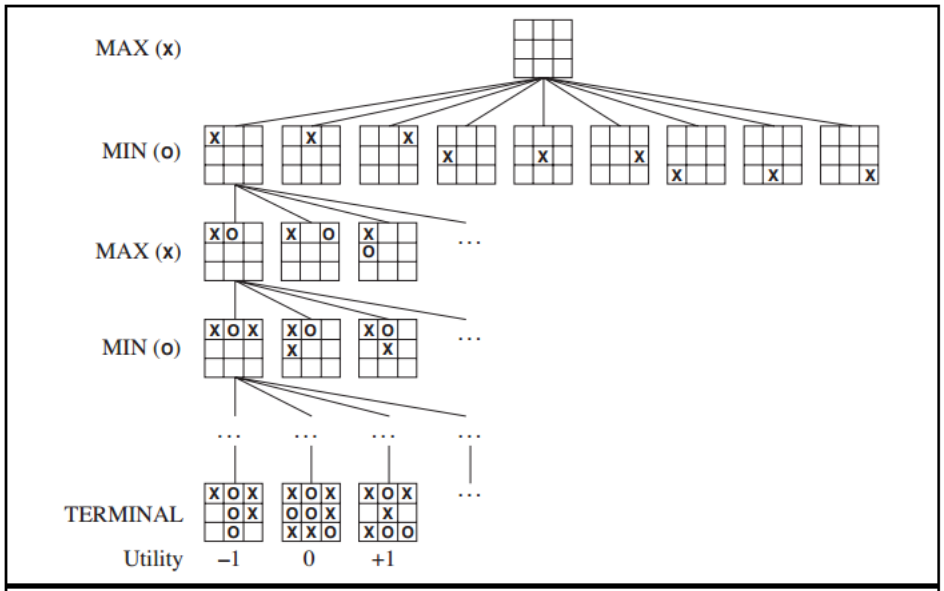


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

- From the initial state, MAX has nine possible moves.
- Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).
- For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes. But for chess there are over 1040 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. But regardless of the size SEARCH TREE of the game tree, it is MAX's job to search for a good move.
- We use the term search tree for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

Optimal decisions in game

- In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win.
- In adversarial search, MIN has something to say about it. MAX therefore must find a contingent ***strategy***, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.
- This is exactly analogous to the AND–OR search algorithm (Figure 4.11) with MAX playing the role of OR and MIN equivalent to AND.
- Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.
- We begin by showing how to find this optimal strategy.
- Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will switch to the trivial game in Figure 5.2.

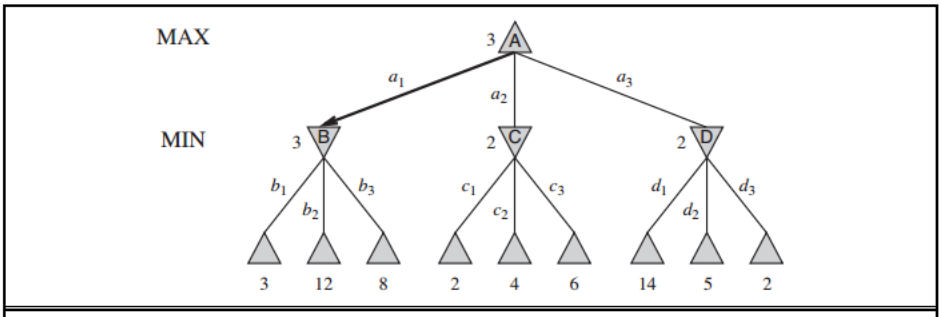


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

- The possible moves for MAX at the root node are labeled a1, a2, and a3.
- The possible replies to a1 for MIN are b1, b2, b3, and so on.
- This particular game ends after one move each by MAX and MIN. (In game parlance, we say that this tree is one move deep, consisting of two half-moves, each of which is called a ***ply***.) The utilities of the terminal states in this game range from 2 to 14.
- Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as MINIMAX(n).
- The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game.
- Obviously, the minimax value of a terminal state is just its utility.
- Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.
- So we have the following:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

- Let us apply these definitions to the game tree in Figure 5.2.
- The terminal nodes on the bottom level get their utility values from the game's UTILITY function.
- The first MIN node, labelled B, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2.
- The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3.
- We can also identify the minimax decision at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.
- This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the worst-case outcome for MAX. What if MIN does not play optimally?
- Then it is easy to show that MAX will do even better.
- Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

Min Max Search algorithm

- The minimax algorithm (Figure 5.3) computes the minimax decision from the current state.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.
- For example, in Figure 5.2, the algorithm first recurses down to the three bottom left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively.
- Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B.
- A similar process gives the backed-up values of 2 for C and 2 for D.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.
- The minimax algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$.
- The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time.

For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

---

9. Alpha Beta Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half.
- The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.
- That is, we can borrow the idea of pruning to eliminate large parts of the tree from consideration.
- The particular technique we examine is called alpha–beta pruning.
- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
- Consider again the two-ply game tree from Figure 5.2.
- Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process.

- The steps are explained in Figure 5.5.
- The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.
- Another way to look at this is as a simplification of the formula for MINIMAX.
- Let the two unevaluated successors of node C in Figure 5.5 have values x and y.
- Then the value of the root node is given by

$$\begin{aligned}\text{MINIMAX}(root) &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2)) \\ &= \max(3,\min(2,x,y),2) \\ &= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \le 2 \\ &= 3.\end{aligned}$$

- the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.
- Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

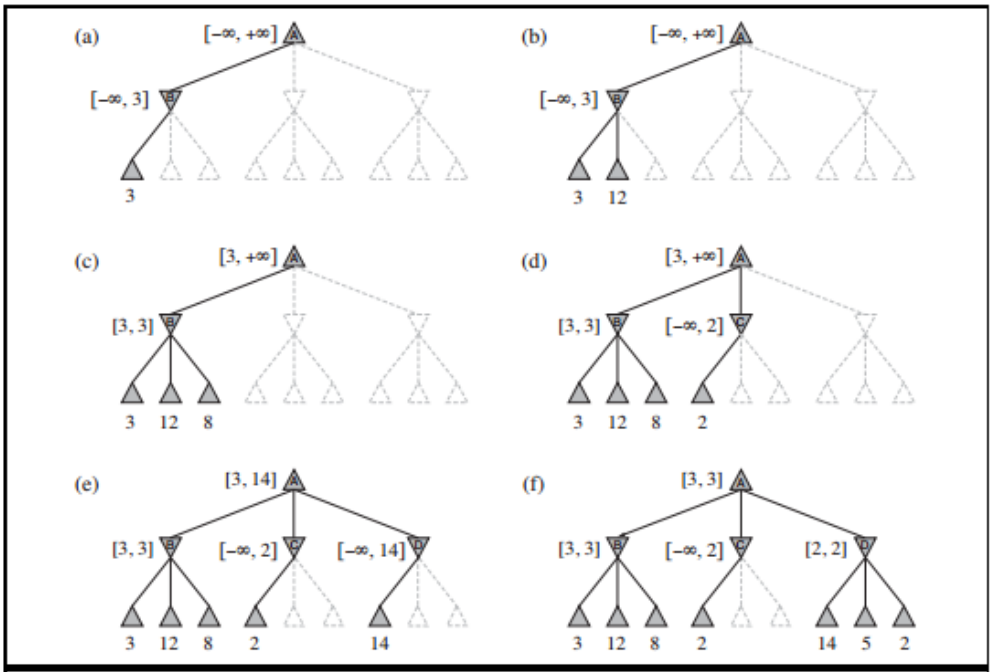


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

- The general principle is this: consider a node n somewhere in the tree (see Figure 5.6), such that Player has a choice of moving to that node.
- If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play.

- So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.
- Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree.
- Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:
  α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.
- Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.
- The complete algorithm is given in Figure 5.7.
- We encourage you to trace its behavior when applied to the tree in Figure 5.5.

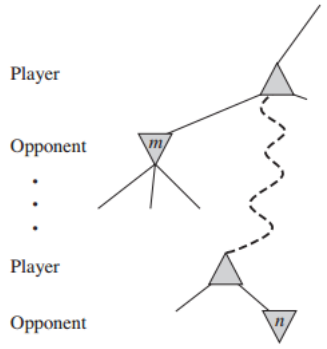


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

Move Ordering

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined.
- For example, in Figure 5.5(e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first.
- If the third successor of D had been generated first, we would have been able to prune the othertwo.
- This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.
- If this can be done,2 then it turns out that alpha–beta needs to examine only O(bm/2) nodes to pick the best move, instead of O(bm) for minimax.
- This means that the effective branching factor becomes √b instead of b—for chess, about 6 instead of 35.
- Put another way, alpha–beta can solve a tree roughly twice as deep as minimax in the same amount of time.
- If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly O(b3m/4) for moderate b.
- For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case O(bm/2) result.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

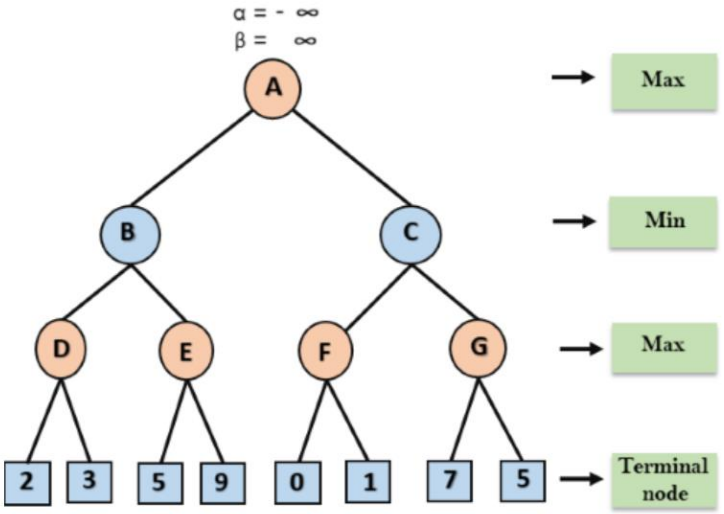
Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

- Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit.
- The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move.
- One way to gain information from the current move is with iterative deepening search.
- First, search 1 ply deep and record the best path of moves.
- Then search 1 ply deeper, but use the recorded path to inform move ordering.
- Iterative deepening on an exponential game tree adds only a constant fraction to the total search time, which can be more than made up from better move ordering.
- The best moves are often called killer moves and to try them first is called the killer move heuristic.
- Repeated states in the search tree can cause an exponential increase in search cost.
- In many games, repeated states occur frequently because of transpositions—different permutations of the move sequence that end up in the same position.
- For example, if White has one move, a1, that can be answered by Black with b1 and an unrelated move a2 on the other side of the board that can be answered by b2, then the sequences [a1, b1, a2, b2] and [a2, b2, a1, b1] both end up in the same position.
- It is worthwhile to store the evaluation of the resulting position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences.
- The hash table of previously seen positions is traditionally called a transposition table; it is essentially identical to the explored list in GRAPH-SEARCH
- Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million nodes per second, at some point it is not practical to keep all of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard

Working of Alpha-Beta Pruning:

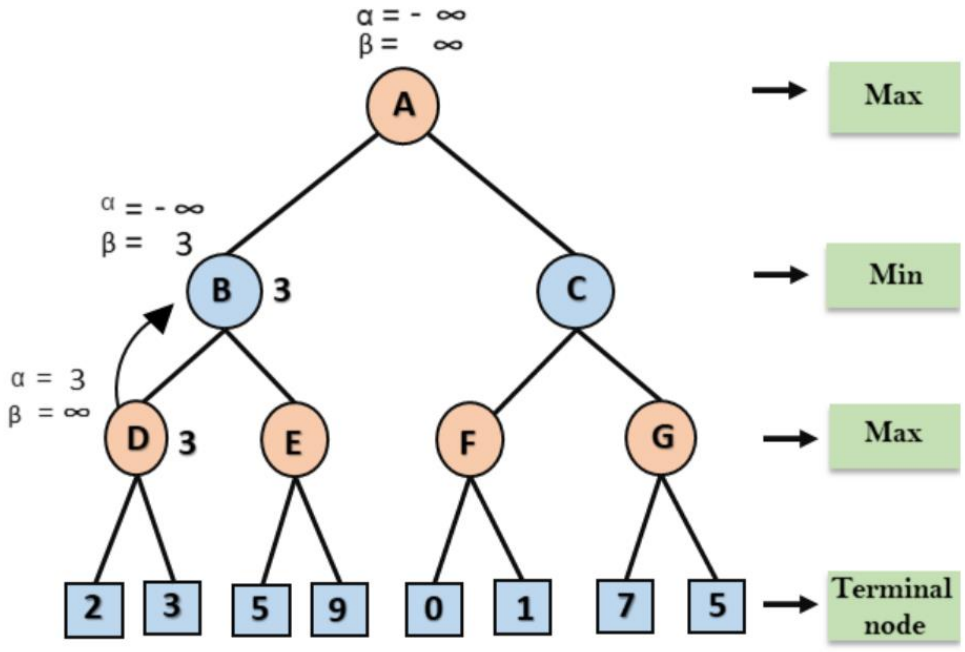
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



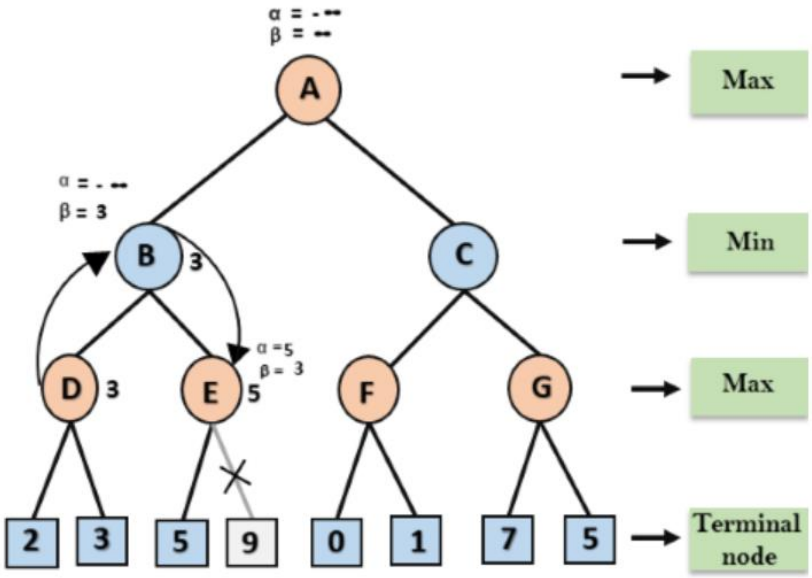
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

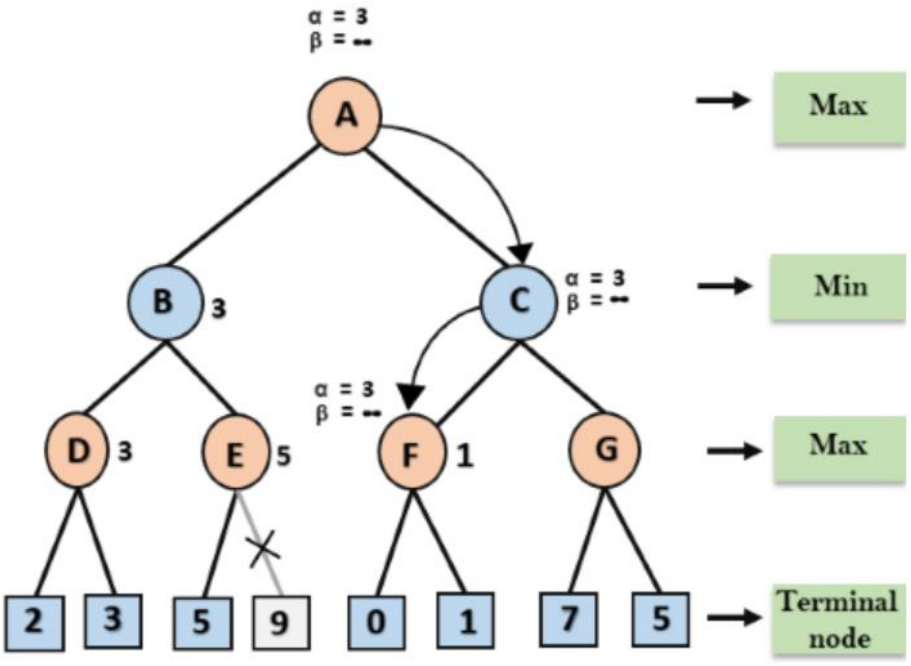
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



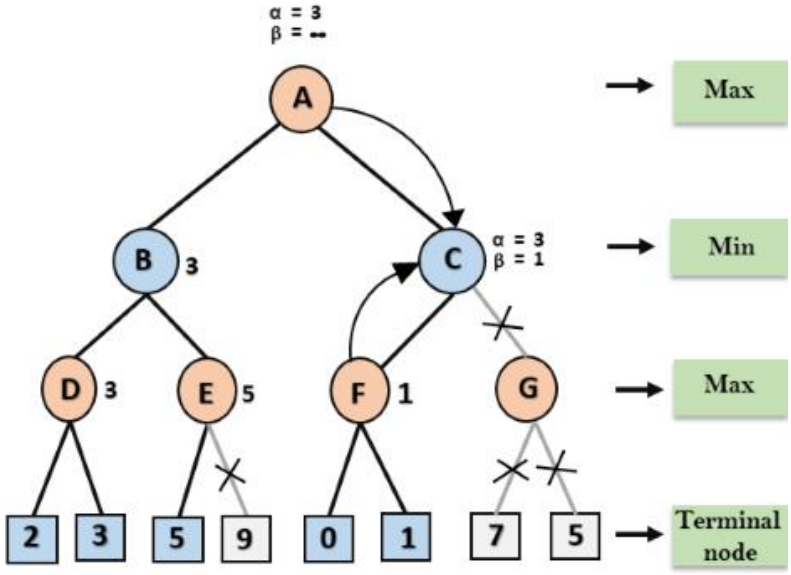
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

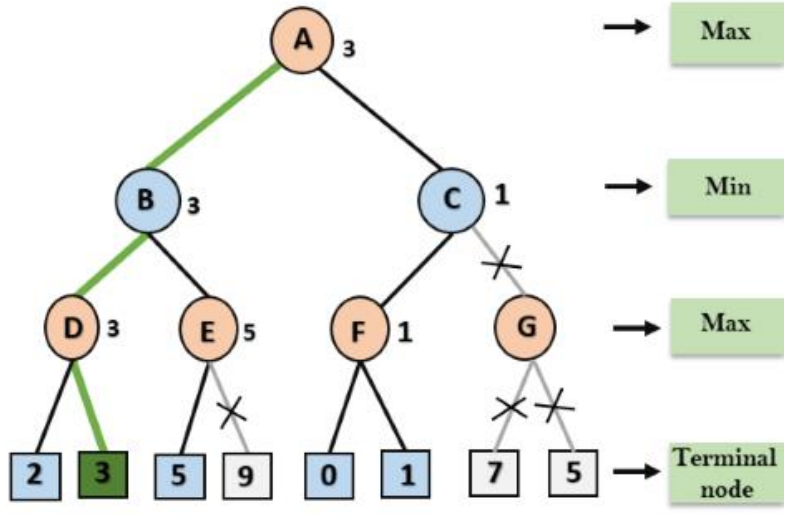
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



---

10. Constraint Satisfaction Problem

- Problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- Each state is atomic, or indivisible—a black box with no internal structure
- Use Factored representation for each state: a set of variables, each of which has a value.
- A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a constraint satisfaction problem, or CSP.

- CSP search algorithms take advantage of the structure of states and use general-purpose rather than problem-specific heuristics to enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Defining Constraint Satisfaction Problems

- A constraint satisfaction problem consists of three components, X, D, and C:
  X is a set of variables, {X1,...,Xn}.
  D is a set of domains, {D1,...,Dn}, one for each variable.
  C is a set of constraints that specify allowable combinations of values.
- Each domain Di consists of a set of allowable values, {v1,...,vk} for variable Xi.
- Each constraint Ci consists of a pair <scope, rel>, where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.
- A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.
- For example, if X1 and X2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as <(X1, X2), [(A, B),(B,A)]> or as <(X1, X2), X1≠ X2>.
- To solve a CSP, we need to define a state space and the notion of a solution.
- Each state in a CSP is defined by an assignment of values to some or all of the variables, {Xi = vi CONSISTENT , Xj = vj ,...}.
- An assignment that does not violate any constraints is called a consistent or legal assignment.
- A complete assignment is one in which every variable is assigned, and a solution to a CSP is a consistent, complete assignment.
- A partial assignment is one that assigns values to only some of the variables.

Examples

a. Map Coloring

- Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)).
- We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.
- To formulate this as a CSP, we define the variables to be the regions
  X = {WA, NT, Q, NSW ,V, SA, T} .
- The domain of each variable is the set Di = {red, green, blue}.
- The constraints require neighboring regions to have distinct colors.
- Since there are nine places where regions border, there are nine constraints:
  C = {SA≠WA, SA≠NT, SA≠Q, SA≠NSW , SA≠V,WA≠NT, NT≠Q, Q≠NSW , NSW≠V } .
- Here we are using abbreviations; SA≠WA is a shortcut for <(SA,WA), SA≠ WA>, where SA≠WA can be fully enumerated in turn as
  {(red, green),(red, blue),(green, red),(green, blue),(blue, red),(blue, green)} .
- There are many possible solutions to this problem, such as
  {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red }
- It can be helpful to visualize a CSP as a constraint graph, as shown in Figure 6.1(b).
- The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.
- Why formulate a problem as a CSP? One reason is that the CSPs yield a natural representation for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique.

- In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space.
- For example, once we have chosen {SA = blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue.
- Without taking advantage of constraint propagation, a search procedure would have to consider 35 = 243 assignments for the five neighboring variables; with constraint propagation we never have to consider blue as a value, so we have only 25 = 32 assignments to look at, a reduction of 87%.
- In regular state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment.
- Furthermore, we can see why the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

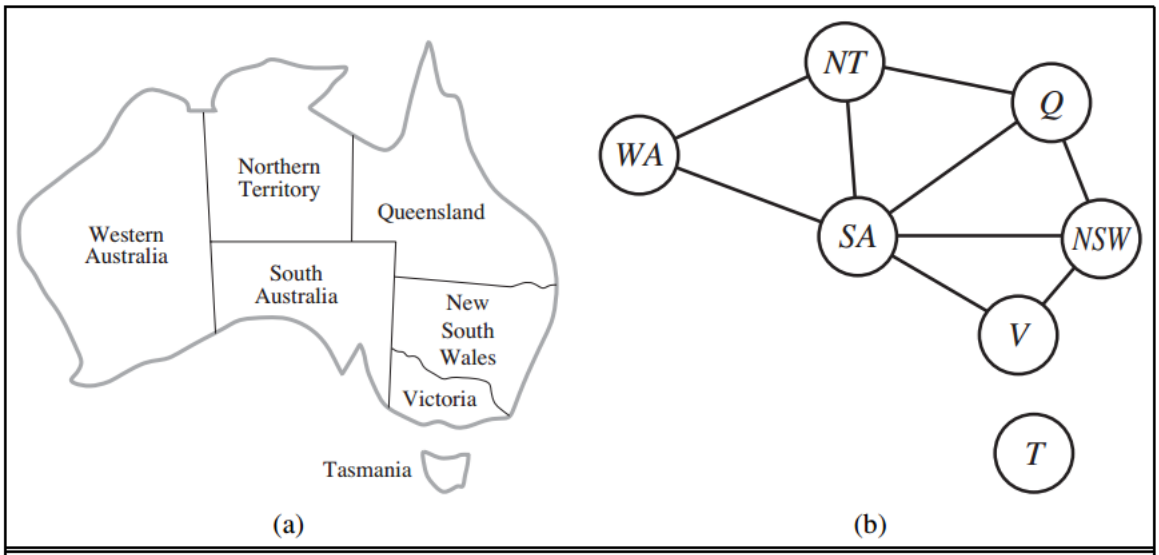


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

b. Job Scheduling

- Factories have the problem of scheduling a day's worth of jobs, subject to various constraints.
- Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.
- We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly.
- We can represent the tasks with 15 variables:
- $X = \{Axle_F , Axle_B, Wheel_{RF} , Wheel_{LF} , Wheel_{RB}, Wheel_{LB}, Nuts_{RF} , Nuts_{LF} , Nuts_{RB}, Nuts_{LB}, Cap_{RF} , Cap_{LF} , Cap_{RB}, Cap_{LB}, Inspect\}$.
- The value of each variable is the time that the task starts.
- We represent precedence constraints between individual tasks.

- Whenever a task T1 must occur before task T2, and task T1 takes duration d1 to complete, we add an arithmetic constraint of the form
  $T1 + d1 \leq T2$
- In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write
  $Axle_F + 10 \leq Wheel_{RF}$; $Axle_F + 10 \leq Wheel_{LF}$;
  $Axle_B + 10 \leq Wheel_{RB}$; $Axle_B + 10 \leq Wheel_{LB}$.
- For each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):
  $Wheel_{RF} + 1 \leq Nuts_{RF}$; $Nuts_{RF} + 2 \leq Cap_{RF}$;
  $Wheel_{LF} + 1 \leq Nuts_{LF}$; $Nuts_{LF} + 2 \leq Cap_{LF}$;
  $Wheel_{RB} + 1 \leq Nuts_{RB}$; $Nuts_{RB} + 2 \leq Cap_{RB}$;
  $Wheel_{LB} + 1 \leq Nuts_{LB}$; $Nuts_{LB} + 2 \leq Cap_{LB}$.
- Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place.
- We need a disjunctive constraint to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:
  $(Axle_F + 10 \leq AxleB)$ or $(Axle_B + 10 \leq Axle_F)$
- This looks like a more complicated constraint, combining arithmetic and logic.
- But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_F$ can take on.
- We also need to assert that the inspection comes last and takes 3 minutes.
- For every variable except Inspect we add a constraint of the form $X + dX \leq Inspect$.
- Finally, suppose there is a requirement to get the whole assembly done in 30 minutes.
- We can achieve that by limiting the domain of all variables:
  $Di = \{1, 2, 3,..., 27\}$.
- This problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables.
- In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used.

Backtracking Search for CSP

- We could apply a standard depth-limited search.
- A state would be a partial assignment, and an action would be adding var = value to the assignment.
- But for a CSP with n variables of domain size d, we quickly notice something terrible: the branching factor at the top level is nd because any of d values can be assigned to any of n variables.
- At the next level, the branching factor is $(n - 1)d$, and so on for n levels.
- We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!
- Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: commutativity.
- A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order.
- Therefore, we need only consider a single variable at each node in the search tree.
- For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between SA = red, SA = green, and SA = blue, but we would never choose between SA = red and WA = blue.
- With this restriction, the number of leaves is d^n, as we would hope.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

- The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- The algorithm is shown in Figure 6.5.
- It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution.
- If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.
- Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT, Q, ... Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.
- BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones
- We improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem.
- It turns out that we can solve CSPs efficiently without such domain-specific knowledge.
- Instead, we can add some sophistication to the unspecified functions in Figure 6.5, using them to address the following questions:
    - Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and inwhat order should its values be tried (ORDER-DOMAIN-VALUES)?
    - What inferences should be performed at each step in the search (INFERENCE)?
    - When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

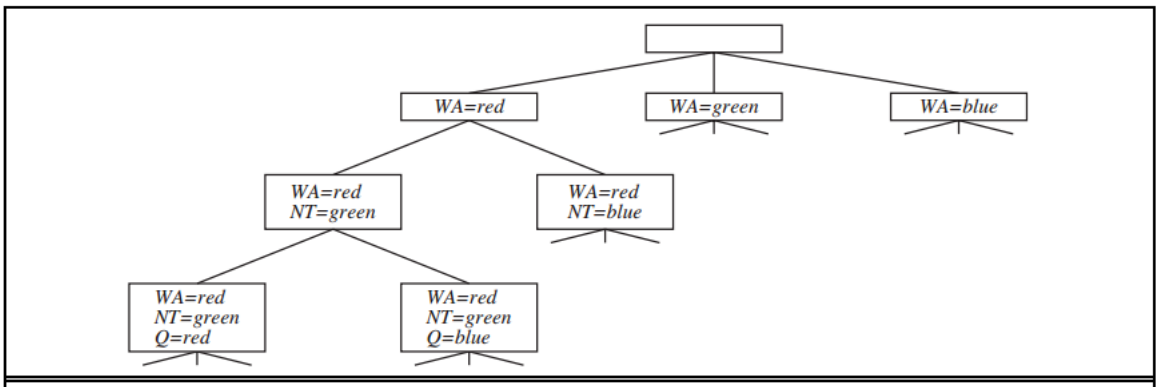


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

Variable and Value Ordering

- The backtracking algorithm contains the line
  var ← SELECT-UNASSIGNED-VARIABLE(csp) .
- The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, {X1, X2,...}.
- This static variable ordering seldom results in the most effi-cient search. For example, after the assignments for WA = red and NT = green in Figure 6.6, there is only one possible value for SA, so it makes sense to assign SA = blue next rather than assigning Q.
- In fact, after SA is assigned, the choices for Q, NSW , and V are all forced.
- This intuitive idea—choosing the variable with the fewest "legal" values—is called the minimum-remaining-values (MRV) heuristic.
- It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.
- The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem
- The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors.
- In this case, the degree heuristic comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.
- In Figure 6.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has degree 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose any consistent color at each choice point and still arrive at a solution with no backtracking.
- The minimum-remaining-values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.
- Once a variable has been selected, the algorithm must decide on the order in which to
- examine its values. For this, the least-constraining-value heuristic can be effective in some cases.
- It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.
- For example, suppose that in Figure 6.1 we have generated the partial assignment with WA = red and NT = green and that our next choice is for Q.
- Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA.
- The least-constraining-value heuristic therefore prefers red to blue.
- In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

- Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway.
- The same holds if there are no solutions to the problem.
- Why should variable selection be fail-first, but value selection be fail-last? It turns out that, for a wide variety of problems, a variable ordering that chooses a variable with the minimum number of remaining values helps minimize the number of nodes in the search tree by pruning larger parts of the tree earlier.
- For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first.
- If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

Interleaving search and inference

- One of the simplest forms of inference is called forward checking. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent with the value chosen for X.
- Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.
- Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking.
- There are two important points to notice about this example.
- First, notice that after WA = red and Q = green are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. A second point to notice is that after V = blue, the domain of SA is empty. Hence, forward checking has detected that the partial assignment
- {WA = red, Q = green, V = blue} is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.
- For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 6.7 after assigning {WA = red}.
- Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those variables next, and then all the other variables will fall into place.
- That's exactly what happens with MRV: NT and SA have two values, so one of them is chosen first, then the other, then Q, NSW , and V in order.
- Finally T still has three values, and any one of them works.
- We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.
- Although forward checking detects many inconsistencies, it does not detect all of them.
- The problem is that it makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent. For example, consider the third row of Figure 6.7.
- It shows that when WA is red and Q is green, both NT and SA are forced to be blue.
- Forward checking does not look far enough ahead to notice that this is an inconsistency:
- NT and SA are adjacent and so cannot have the same value.
- The algorithm called MAC (for Maintaining Arc Consistency (MAC)) detects this inconsistency. After a variable Xi is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (Xj , Xi) for all Xj that are unassigned variables that are neighbors of Xi. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately.
- MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

Intelligent backtracking: Looking backward

- The BACKTRACKING-SEARCH algorithm in Figure 6.5 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it.
- This is called chronological backtracking because the most recent decision point is revisited. In this subsection, we consider better possibilities.
- Consider what happens when we apply simple backtracking in Figure 6.1 with a fixed variable ordering Q, NSW , V , T, SA, WA, NT. Suppose we have generated the partial assignment {Q = red, NSW = green, V = blue, T = red}. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly resolve the problem with South Australia.
- A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of SA impossible.
- To do this, we will keep track of a set of assignments that are in conflict with some value for SA.
- The set (in this case {Q = red, NSW = green, V = blue, }), is called the conflict set for SA.
- The backjumping method backtracks to the most recent assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V .
- This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign.
- If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.
- The sharp-eyed reader will have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment X = x deletes a value from Y 's domain, it should add X = x to Y 's conflict set.
- If the last value is deleted from Y 's domain, then the assignments in the conflict set of Y are added to the conflict set of X.
- Then, when we get to Y , we know immediately where to backtrack if needed.
- The eagle-eyed reader will have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that every branch pruned by backjumping is also pruned by forward checking.
- Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC
- Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure.
- Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment {WA = red, NSW = red} (which, from our earlier discussion, is inconsistent).
- Suppose we try T = red next and then assign NT, Q, V , SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT.
- Now, the question is, where to backtrack? Backjumping cannot work, because NT does have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail.
- We know, however, that the four variables NT, Q, V , and SA, taken together, failed because of a set of preceding variables, which must be those variables that directly conflict with the four.
- This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, together with any subsequent variables, to have no consistent solution.
- In this case, the set is WA and NSW , so the algorithm should backtrack to NSW and skip over Tasmania.

- A backjumping algorithm that uses conflict sets defined in this way is called conflict-directed backjumping.
- We must now explain how these new conflict sets are computed.
- The method is in fact quite simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set.
- In our example, SA fails, and its conflict set is (say) {WA, NT, Q}.
- We backjump to Q, and Q absorbs the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is {NT, NSW }; the new conflict set is {WA, NT, NSW }.
- That is, there is no solution from Q onward, given the preceding assignment to {WA, NT, NSW }.
- Therefore, we backtrack to NT, the most recent of these. NT absorbs {WA, NT, NSW }−{NT} into its own direct conflict set {WA}, giving {WA, NSW } (as stated in the previous paragraph).
- Now the algorithm backjumps to NSW , as we would hope.
- To summarize: let Xj be the current variable, and let conf (Xj) be its conflict set.
- If every possible value for Xj fails, backjump to the most recent variable Xi in conf (Xj), and set conf (Xi) ← conf (Xi) ∪ conf (Xj) − {Xi} .
- When we reach a contradiction, backjumping can tell us how far to back up, so we don't waste time changing variables that won't fix the problem.
- But we would also like to avoid running into the same problem again.
- When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem.
- Constraint learning is the idea of finding a minimum set of variables from the conflict set that causes the problem.
- This NO-GOOD set of variables, along with their corresponding values, is called a no-good.
- We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.
- For example, consider the state {WA = red, NT = green, Q = blue} in the bottom row of Figure 6.6. Forward checking can tell us this state is a no-good because there is no valid assignment to SA.
- In this particular case, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again. But suppose that the search tree in Figure 6.6 were actually part of a larger search tree that started by first assigning values for V and T. Then it would be worthwhile to record {WA = red, NT = green, Q = blue} as a no-good because we are going to run into the same problem again for each possible set of assignments to V and T.
- No-goods can be effectively used by forward checking or by backjumping.
- Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.
