Unit-01: Introduction and Basic Search Strategies

Artificial Intelligence

Dr. Mohammad Fayazur Rahaman
Associate Professor, mfrahaman_ece@mgit.ac.in

Dept. of Electronics and Communications Engineering,
Mahatma Gandhi Institute of Technology, Gandipet, Hyderabad-75

Mar-Jun 2022

Unit-01: Introduction and Basic Search Strategies [1, 3]

1. Introduction

1.1 AI problems [2]

1.2 Agents and Environments [1]

1.3 Structure of Agents [1]

- Agent Programs
- Simple Reflex Agents
- Model-based Reflex Agents
- Goal-based Agents
- Utility-based Agents
- Learning Agents
- How agent programs work

2. Problem Solving Agents [1]

2.1 Search problems and solutions

3. Basic Search Strategies

3.1 Problem Spaces [2]

3.2 Uninformed Search [1]

- I. Breadth-First Search (BFS)
- II. Depth-First Search (DFS)
- III. Depth-First with Iterative Deepening Search (IDS)

3.3 Heuristic Search [1]

- II. Generic Best-First (GBF)
- III. A* Search
- I. Hill Climbing (HC)

3.4 Constraint Satisfaction [1]

- I. Backtracking
- II. Local Search

Introduction (Definition)

i. **Artificial intelligence (AI)** is the study of how to make computers do things which, at the moment people do better.
   - → **Intelligence** is the most important differentiating factor of human beings.

ii. The field of AI, attempts not just to understand but also to build intelligent entities.

iii. AI encompasses a huge variety of sub fields ranging from the **general** (learning and perception) to the **specific**, such as playing chess, proving mathematical theorems, driving a car and diagnosing diseases.

AI definition in four categories

I. **Acting humanly:** The Turing test approach suggests that a computer passes the Intelligent test if a human interrogator, after using some written questions, cannot tell whether the written response has come from a person or from a computer. To acheive this, the computer need to possess the following capabilities:

   i. Natural language processing
   ii. Knowledge representation
   iii. Automated reasoning
   iv. Machine learning

   To pass the total Turing test, the computer will need

   i. Computer vision
   ii. Robotics

II. **Thinking humanly:** There are three ways to determine how a human mind thinks

   i. Through introspection
   ii. Through psychological experiments
   iii. Observing the brain in action

III. **Thinking rationally:** The 'laws of thought' govern the operation of the mind, their study initiated the field called logic.

   i. "Socrates is a man; all men are mortal; therefore, Socrates his mortal".

IV. **Acting rationally:** A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainity, the best expected outcome.

Where are we ?

AI problems [2]

i. Some of the tasks that are the targets of work in AI are
   - I. **Mundane tasks**
     - a. Perception
       - Vision, Speech
     - b. Natural language
       - Understanding, Generation, Translation
     - c. Commonsense reasoning
     - d. Robot control
   - II. **Formal tasks**
     - a. Games
       - Chess, Backgammon, Checkers - Go
     - b. Mathematics
       - Geometry, Logic, Integral calculus, proving properties of programs
   - III. **Expert tasks**
     - a. Engineering
       - Design, Fault finding, Manufacturing planning
     - b. Scientific analysis
     - c. Medical diagnosis
     - d. Financial analysis

ii. As AI research advanced, some progress was made on the tasks like perception (vision, speech), natural language understanding and problem solving in specialised domains such as medical diagnosis
iii. **Perceptual** tasks are difficult because they involve analog signals, and the signals are typically very noisy and usually a large number of things must be perceived at once
iv. In order to **understand sentences** about the topic it is necessary to know not only about the language itself but also a good deal about the topic so that **un-stated assumptions** can be recognised
v. **Specialised tasks** such as engineering design, medical diagnosis need carefully acquired **expertise**
vi. Although expert skills require knowledge that many of us do not have, they often require much **less knowledge** than do the more mundane skills and that knowledge is usually **easier to represent** and deal with inside programmes
vii. As a result, AI programs called **expert systems** are now flourishing primarily in the domains that require specialized expertise without the assistance of **commonsense knowledge**

Where are we ?

Agents and Environments [1]

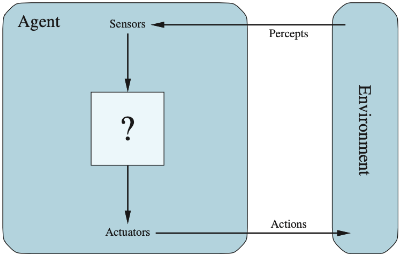


Figure 2.1 Agents interact with environments through sensors and actuators.

I. **Agents:** An agent is anything that can be viewed as **perceiving its environment** through **sensors** and **acting upon** that environment through **actuators**
   i. A **robotic agent** might have cameras and in for a range finders for **sensors** and various motors for **actuators**
   ii. A **software agent** receives keystrokes, file contents, network packets as sensory inputs and acts on the environment by displaying on the screen writing files

II. **Percept:** The term percept refers to the agents perceptual inputs at any given instance
   i. **Percept Sequence:** An agent's percept sequence is the complete history of everything the agent has ever perceived
   ii. In general and agent's **choice of action** at any given instance can depend on the entire percept sequence it has observed to date, but not on anything it hasn't perceived

III. **Agent function:** Mathematically we say that an agent's behaviour is described by the **agent function** that maps any given **percept sequence** to **an action**
   i. **Agent program:** Internally the **agent function** for an artificial agent will be **implemented** by an **agent program**

Example: Vacuum-cleaner world

i. In this example, **its world** has just two locations Squares A and B

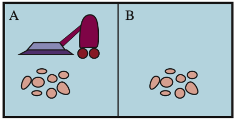


Figure 2.2 A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always succeed, and so on.

→ The **vacuum agent perceives which square** it is in and whether **there is dirt** in the square

→ The agent can choose to **move left**, **move right**, **suck up** the dirt, or **do nothing**

→ One very simple **agent function** is the following: If the current square **is dirty**, **then suck**; **otherwise move to** the other Square

ii. A partial tabulation of this **agent function** is shown in the figure below and an **agent program** implements it

Percept sequence	Action
[A, *Clean*]	*Right*
[A, *Dirty*]	*Suck*
[B, *Clean*]	*Left*
[B, *Dirty*]	*Suck*
[A, *Clean*], [A, *Clean*]	*Right*
[A, *Clean*], [A, *Dirty*]	*Suck*
⋮	⋮
[A, *Clean*], [A, *Clean*], [A, *Clean*]	*Right*
[A, *Clean*], [A, *Clean*], [A, *Dirty*]	*Suck*
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2. The agent cleans the current square if it is dirty, otherwise it moves to the other square. Note that the table is of unbounded size unless there is a restriction on the length of possible percept sequences.

Good Behavior: The Concept of Rationality [1]

I. **Rational agent**: A rational agent is one that does the right thing
   - → As a general rule it is better to design **performance measures** according to what one actually **wants in the environment** rather than according to how one thinks the agent should behave
   - → **Definition:** For each possible percept sequence a **rational agent** should **select an action** that is expected to **maximise** its performance measures; given **the evidence** provided by the percept sequence and what ever **built in knowledge** the agent has

II. **Omniscience:** An **Omniscient agent** knows the **actual outcome** of its actions and can act accordingly but omniscience is **impossible in reality**

III. **Information Gathering:** **Doing actions** in order to **modify future percepts** is sometimes called **information gathering** and is an important part of rationality

IV. **Learning:** A **rational agent** not only requires to **gather information** but also to **learn** as much as possible from **what he perceives**
   - → The agents **initial configuration** could reflect some **prior knowledge** of the environment but as the agent **gains experience** this may be **modified and augmented**

V. **Autonomy:** If an agent relies on the **prior knowledge** of its designer rather than on it **own percepts**, then we say the agent **lacks autonomy**
   - → A **rational agent** should be **autonomous**; it **should learn** what it can, to **compensate** for **partial or incorrect** prior knowledge
   - → After **sufficient experience** the behaviour of a **rational agent** can become **effectively independent** of its prior knowledge

The Nature of Environments [1]

i. **Task environment** : The task environments are essentially **the problems** to which rational agents are **the solutions**

ii. **PEAS**: In designing an agent the first steps must always be to specify the task environment as fully as possible, i.e., its **Performance**, **Environment**, **Actuators** and **Sensors** description

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.

Where are we ?

Structure of Agents [1]

I. The job of AI is to design an **agent program** that implements an **agent function**, that is the mapping from **Percepts to actions**

II. We assume this **program** will run on some sort of computing device with physical **sensers and actuators** that is called as the **architecture**

$$\text{agent} = \text{architecture} + \text{program}$$

III. The **architecture** might be an ordinary PC or be a robotic car with several on board computers, cameras and other sensers. In general, the architecture

   i. Makes the Perspects from the sensors available to the program
   ii. Runs the program, and
   iii. Feeds the program's action choises to the actuators

IV. The **agent program** takes the **current percept** as input from the sensors and **return action** to the actuators

→ Note that the **agent program** takes the current percept only as the input, and the **agent function** takes the **entire percept history**

Agent Programs

i. The figure below shows a trivial agent program that keeps track of the **percept sequence** and then uses it to index into a **table of actions**

ii. However, the **table driven** approach to agent construction is **doomed to failure**
   - → Let $\mathcal{P}$ be the set of possible percepts, and
   - → let T be the lifetime of the agent, then
   - → the lookup table with contain $\sum_{t=1}^{T} |\mathcal{P}|^t$ entries
   - → Example: The lookup table for Chess - would have at least 10^{150} entries

iii. The **key challenge** for AI is to find out how to write **small programs** that produce **rational behaviour** from a small program rather than from a table

iv. **Four basic** kind of agent programs that embody the principles underlying almost all **intelligent systems** are
   - I. **Simple reflex agents**
   - II. **Model-based reflex agents**
   - III. **Goal-based agents and**
   - IV. **Utility-based agents**

```
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.

I. Simple Reflex Agents

i. A **simple reflex agent** selects action on the basis of **current percept** ignoring the rest of the **percept history**
   - → In the case of VACUUM-AGENT, this cuts down the number of possible actions from 4^T to just 4

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Figure 2.8 The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3.

ii. In this agent, some **processing is** done on the percept input to **establish the condition** (dirty or clean), this triggers some **established connection** in the agent program to the action. Such a connection is called as **condition-action rule**

iii. The `INTERPRET-INPUT` function generates a description of the current state from the percept

iv. The `RULE-MATCH` function returns the first rule in the set of rules that matches the given state

v. The simple reflex agents often get into **infinite loops**

→ **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`

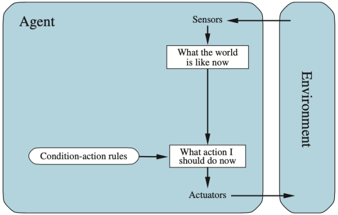


Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

```
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.

II. Model-based Reflex Agents

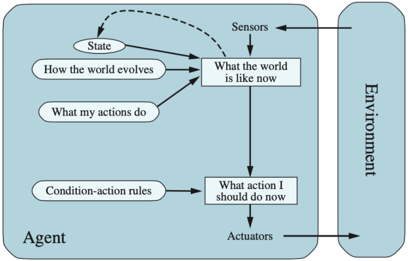


Figure 2.11 A model-based reflex agent.

i. The agent maintain some sort of **internal state** that depends on the **percept history** and thereby reflects at least some of the un-observed aspects of the current state

ii. The knowledge about how the world works is called a **model** of the world

→ First, we need some information about how the **world** evolves **independently** of the agent

→ Second, we need some information about how the **agents on actions** affect the world

iii. The function `UPDATE-STATE` is responsible for creating the new internal state description

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

  state ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_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.

III. Goal-based Agents

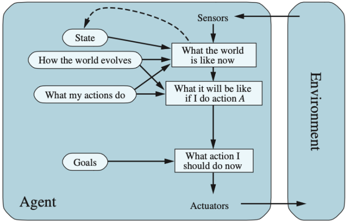


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

i. In a goal based agent, in addition to the current state description, the agents needs some sort of **goal information** that describes situations that are **desirable**. For example, driver-less taxi being at the passengers destination.

ii. Sometimes the goal based action selection will be **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 agents goals

iii. Note that **decision making** in this agent involves consideration of **the future** - both
   - → "`what will happen if I do such and such`" and
   - → "`will that make me happy`"

iv. Note that, the goal-based agent's **behaviour** can **easily be changed** to go to a different destination by **simply specifying** the new destination as the goal
   - → Whereas in case of **reflex agent**, rules will work only for a single destination they must **all be replaced** to go somewhere new

IV. Utility-based Agents

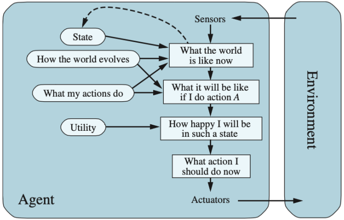


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

i. Goals alone or **not enough** to generate **high-quality behaviour** in most environments
   → For example **many action sequences** will get the taxi to its destination, but some are **quicker**, **safer**, **more reliable**, or **cheaper** than others

ii. An **utility function** is used to capture the **performance measures** of an Agent

iii. In two kinds of cases, **goals are in adequate** but a **utility-based** agent can still make **rational decisions**
   → First, when there are **conflicting goals** like **speed and safety**, the **utility function** specifies the appropriate **trade-off**
   → Second, when there are **several goals** that the agent can aim for, utility provides a way in which the **likelihood of success** can wieghted against the **importance of the goals**

Learning Agents

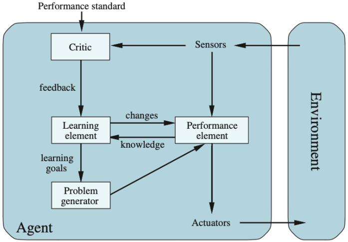


Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

i. One of the **advantages** of **Learning** is that it allows the agent to operate in initially **unknown environment** and to become **more competent** than its own initial knowledge

ii. The learning agent can be divided into **four conceptual components**

   I. **Learning Element:** This is responsible for **making improvements**

   II. **Performance Element:** This is responsible for selecting **external actions**, i.e., it takes in percepts and decides on actions

   III. **Critic:** The critic tells the **learning element** how well the **agent is doing** with respect to fixed performance standards

   IV. **Problem Generator:** This is responsible for **suggesting actions** that will lead to **new and informative** experiences

   → With **exploration**, it might discover **much better** actions for the long run

How the components of agent programs work

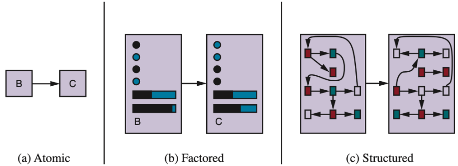


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

i. Some of the basic ways that **components** can **represent the environment** that the agent inhabits are

   I. **Atomic:** In this each **state** of the world is **indivisible**, i.e., it has no **internal structure**
      → The algorithms underlying **search** and **gameplay**, **hidden Markov models** and **Markov decision processes** all work with atomic presentations

   II. **Featured:** This representation splits up each **state** into a fixed **set of variables** or **attributes**, each of which can have a **value**
      → Many areas of AI or based on this representation including **constrain satisfaction algorithms**, **proportional logic**, **planning**, **Bayesian networks** and the **machine learning** algorithms

   III. **Structured:** Note that the **world has things** in it that are **related to each** other, and are **not just variables** with values
      → Example: It is unlikely to have a variable `TruckBackingBlockedByLooseCow` with value `true` or `false`
      → Structure representations underline **relational databases**, **first-order logic**, **first-order probability models** and **knowledge-based learning**

ii. The **atomic**, **factored**, and **structured representations** lie on the increasing **expressiveness** axis
   → Give more **expressive representation** can capture **larger knowledge** and **more concisely**
   → On the other hand **reasoning and learning** becomes more **complex** as the expressive power of the representation **increases**

Summary

I. An ideal **intelligent agent** takes the best **possible action** in a situation
II. An **agent** is something that perceives and acts in an environment
III. The **agent function** for an agent specifies the action taken by the agent in response to any percept sequence
IV. The **performance measure** evaluates the behavior of the agent in an environment. A **rational agent** acts so as to maximise the expected value of the performance measure

Where are we ?

Problem Solving Agents [1]

I. In **solving problems by searching** we see how an agent can **look ahead** to find a **sequence of actions** that will eventually **achieve its goal**

II. When the correct action to take is not immediately obvious, an agent may need to plan ahead; to consider a sequence of actions that form a path to a goal state

III. Such an agent is called a **problem-solving agent**, and the **computational process** it undertakes is called **search**

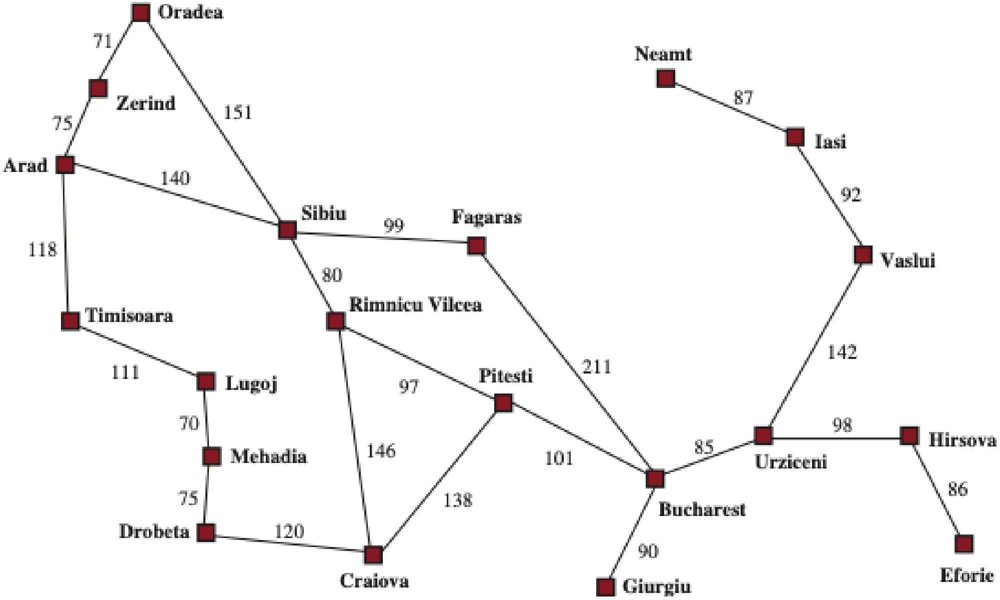


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

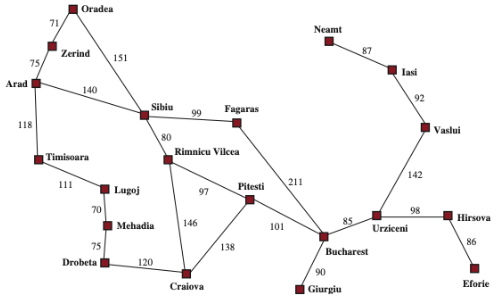


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

i. Let us assume our agent is in `Arad` and has to travel to `Bucharest`

ii. Further, assuming that the agent has access to information about the world such as the map in the figure, the agent can follow a four-phase problem-solving process, involving

   I. **Goal Formulation:** Goals **organise behaviour** by **limiting the objectives** and hence the actions to be considered

   II. **Problem Formulation:** The agent devices a description of the **states** and **actions** necessary to reach the goal

   → One good model is to consider the **actions** of travelling from one city to an adjacent city, and

   → therefore the change in the **state** due to the action is the current city

   III. **Search:** Before taking any action in the real world, the agent **simulates sequences of actions** in its model, **searching** until it **finds** a sequence of action that reaches the goal

   → Such a sequence is called a **solution**

   IV. **Execution:** The agent can now **execute the actions** in the solution one at a time

iii. In a **fully observable**, **deterministic**, **known environment**, the solution to any problem is a **fixed sequence of actions**

iv. In a **partially observable** or **nondeterministic environments** (road-closed sign) a solution would be a **branching strategy** that **recommends** different **future actions** depending on what percept arrive

Search problems and solutions

i. A **Search problem** can be defined formally as follows

   I. **State space:** A set of possible states that an environment can be in

   II. **Initial state:** The state that the agent starts in. Ex: `Arad`

   III. **Goal States:** A **set** of one or more goal states.

   → Is specified by `IS-GOAL` method for a problem

   IV. **Actions available:** The **actions** available to the agent

   → Given a state s, `ACTIONS(s)` returns a finite set of actions that can be executed in s

   → For example:

   $$\texttt{ACTIONS(Arad)} = \{\texttt{ToSibiu, ToTimisoara, ToZerind}\}$$

   V. **Transition model**, describes what each action does

   → `RESULT(s,a)` returns the state that results from doing action **a** in state **s**.

   $$\texttt{RESULT(Arad, ToZerind)} = \{\texttt{Zerind}\}$$

   VI. **Action cost function,**, denoted by $\texttt{ACTION-COST(s,a,s}'\texttt{)}$, gives the numeric cost of applying action **a** in state **s** to reach state $\texttt{s}'$

   → For route-finding agents, the cost of an action might be the length in miles, or the time it takes to complete the action

ii. A sequence of actions form a **path**, and a **solution** is a path from the initial state to your goal state

iii. An **optimal solution** has the lowest path cost among all solutions

iv. The **state space** can be represented as a **graph** in which the vertices are **states** and the directed edges between them are **actions**

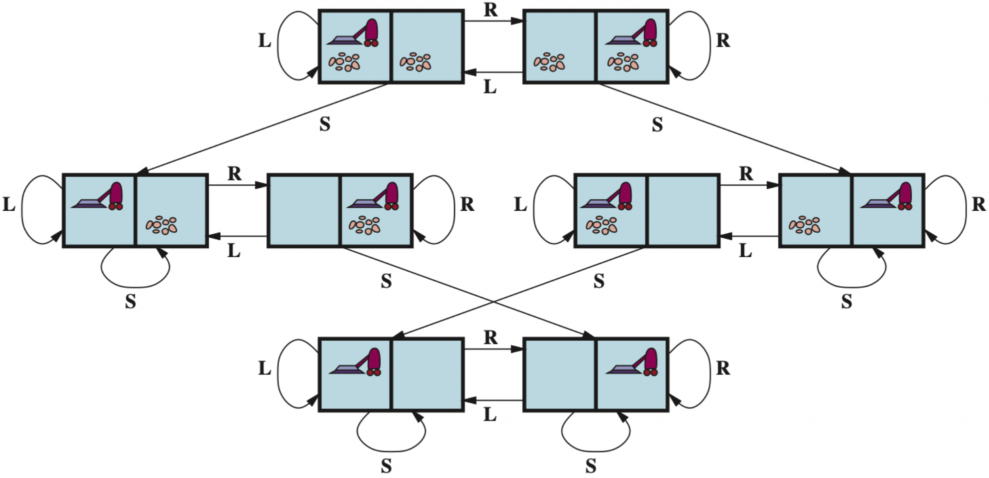


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

The **vacuum world** can be formulated as a grid world problem as follows:

i. **States:** A **state** of the world says which **objects** are in which cells
   - → In this example, the objects are **the agent** and **any dirt**
   - → In the simple **two-cell** version, the agent can be in either of the two cells, and each cell can either contain dirt or not. As a result there are $2 \times 2 \times 2 = 8$ states
   - → A n cell environment has $n \times 2^n$ states

ii. **Initial State:** Any state can be designated as the initial state

iii. **Actions:** Three actions - `Suck`, `move Left`, and `move Right`

iv. **Transition Model:** Suck **removes** any dirt from the agent's cell

v. **Goal States:** The states in which every cell is clean

vi. **Action cost:** Each action costs 1

Where are we ?

To build a system to solve a particular problem, the following four steps need to be done

I. **Define the problem** precisely.
   - → Must include specifications of what the **initial situation** will be and
   - → what **final situations** constitute acceptable solutions to the problem

II. **Analyze** the problem
   - → A few very **important features** can have an immense impact for solving the problem

III. Isolate and **represent** the task **knowledge** that is necessary to solve the problem

IV. Choose the best **problem-solving** technique and apply it

Problem Spaces [2]

To build a program that could "Play Chess", we would first have to specify

i. the **starting position** of the chess board,
ii. the **rules** that define the **legal moves**, and
iii. the board positions that represent **a win** for one side or the other

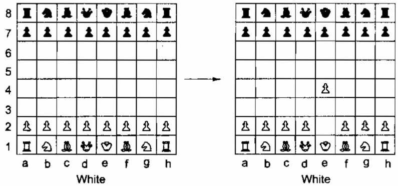


Fig. 2.1 *One Legal Chess Move*

State space is defined as a representation where each **state** corresponds to a legal position of the board.

→ We can then play chess (or solve any problem) by starting at an **initial state**,
→ using a **set of rules** to move form one state to another, and
→ attempt to end up in one of a set of **final states**

White pawn at
Square(file e, rank 2)
AND
Square(file e, rank 3)
is empty
AND
Square(file e, rank 4)
is empty

→ move pawn from
Square(file e, rank 2)
to Square(file e, rank 4)

Fig. 2.2 *Another Way to Describe Chess Moves*

The **state space representation** forms the basis of most of the **AI problems**

i. It allows for a **formal definition of a problem** as the need to convert some **given** situation into some **desired** situation using a set of permissible operations
ii. It permits us to define the **process** of solving a particular problem as a combination of known **techniques** and **search**
   I. Each **technique** is represented as a **rule** defining a single step in the space
   II. **Search** is a very important process in the solution for which **no more** direct techniques are available

Summarising, in order to provide a formal description of a problem we must do the following

I. Define a **state space** that contains all the possible configuration of the relevant objects
   - → Note that, it is possible to define the space **without explicitly** enumerating all of the states it contains

II. Specify **one or more states** that describe possible situations from which the problem-solving process: may start: called the **initial states**

III. Specify one or more states that would be **acceptable** as solutions to the problem: called the **goal states**

IV. Specify a **set of rules** that describe the **actions available**. Need to consider
   - → What unstated **assumptions** are present
   - → How **general** should the rules to be
   - → How much of **work required** to solve the problem should be **pre computed** and represented in the **rules**?

Where are we ?

Search Algorithms (Terminologies)

i. A **search algorithm** takes a **search problem** as input and returns **solution**, or an indication of **failure**

ii. An algorithm **superimposes** a **search tree** over the **state-space** graph forming various **paths** from the initial state to reach the goal state
   - → Each **node** in the search tree corresponds to a **state** in the state space and the **edges** correspond to the **actions**
   - → The **root node** corresponds to the **initial state** of the problem

iii. The **state space** describes the **set of states** in the world, and the **actions** that allow transitions from one state to another
   - → Whereas the **search tree** describes **paths** between the states reaching towards the **goal**

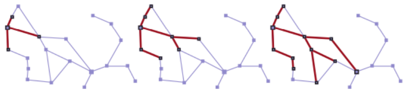


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

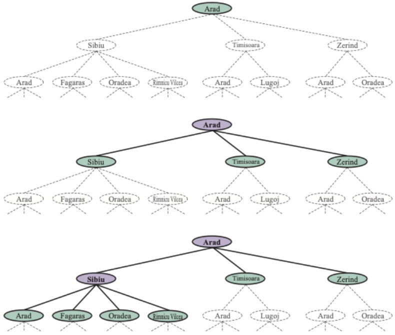


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

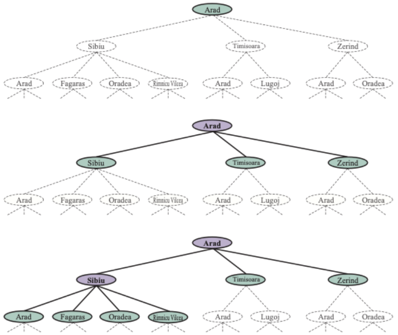


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

i. **Expand:** We can expand a node by considering the **available actions** from that state.
   - → Each action from a node leads to (**generates**) a new (**child** or successor) node
   - → Each child node has a **parent node**

ii. **Search:** Note, we must choose which of the child nodes to **expand next**. This is the **essence of search**

iii. **Frontier:** The set of the unexpanded child nodes of the search tree is the called the **frontier** of the tree at that instance
   - → The frontier separates two regions; and **interior region** where every state has been **expanded** and an **exterior region** that has not yet been **reached**

iv. **Reached:** We say that any state that has had a node generated for it has been **reached**

I. **Best-First Search:** This is a very **general approach** to decide which node from the frontier to **expand next**
   - → On each iteration we choose a node n on the frontier with minimum **evaluation function** $f(n)$ value, and return if its state is a **goal state**, and otherwise **expand** to generate **child notes**
   - → Each child node is added to the frontier if it has not been **reached before**, or is **re-added** if it is now being reached with a path that has a **lower path cost** than any previous path

II. **Node data structures:** A node in the tree is represented by a **data structure** with four components
   - → `node.STATE`, `node.PARENT`, `node.ACTION`, `node.PATH-COST`

III. **Frontier data structure:** A **queue** data structure is used to store the **frontier** nodes. Three kinds of queues are used in search algorithms
   - i. **Priority Queue**: This pops the first node with the **minimum cost** according to the evaluation function $f(n)$. This is used in the **best-first** search
   - ii. **FIFO queue**: This first pops the node that was added to the **queue first**
   - iii. **LIFO queue**: This first pops the **most recently** added node, (this is also known as **Stack**). This is used in **depth-first** search

IV. **Redundant paths:** When a state is repeated in the search tree due to a **cycle** (loopy path), we call it a **redundant path**
   - → We call a search algorithm a **graph-search** if it checks for redundant paths and a **tree-like** search if it does not check
   - → The `BEST-FIRST-SEARCH` algorithm is a **graph search** algorithm

V. **Measuring problem-solving performance:** The **criteria** used to choose among the **various search algorithms**, are
   - i. **Completeness:** Is the algorithm **guaranteed** to find a solution when there is one, and to correctly **report failure** when there is not ?
   - ii. **Cost optimality:** Does it find a solution with the **lowest path cost** of all solutions ?
   - iii. **Time complexity:** **How long** does it take to find a solution ?
   - iv. **Space complexity:** How much **memory** is needed to perform the search ?

Uninformed Search [1]

I. An **uninformed search** algorithm is given **no clue** about how **close** a state is to the goal
   → An agent with **no knowledge** of **Romanian geography** has no clue whether going to Zerind or Sibia is a better first step

II. In contrast, an **informed agent** who knows the location of each city knows that Sibiu is much closer to Bucharest (i.e., goal) and thus **more likely** to be on the **shortest path**

I. Breadth-First Search (BFS)

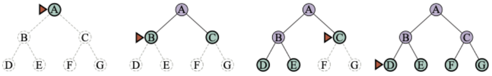


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

i. When all actions have the same cost, an appropriate strategy is `BREADTH-FIRST-SEARCH`, in which the root node is expanded first, then all the successors of the root node are expanded next and so on
   - → This is a **systematic search** that is there for **complete** even on infinite state space

ii. We could implement `BREADTH-FIRST-SEARCH` as a call to `BEST-FIRST-SEARCH` where the evaluation function $f(n)$ is the depth of the node

iii. The `BREADTH-FIRST-SEARCH` always finds a solution with a **minimum number** of actions, because when it is generating nodes at d, it has already generated all the nodes at depth $d-1$, so if one of them were a solution, it would have been found
   - → i.e., It is **cost-optimal** for problems where all actions have the same cost

iv. If we consider a uniform tree where every state has b successors, i.e.,
   - → the root generates b nodes, each of which generates b more nodes, and
   - → a total of b^2 at the second level.
   - → Suppose that the solution is at depth d, then the total number of nodes generated is

$$1 + b + b^2 + \cdots + b^d = \mathcal{O}(b^d)$$

v. In general, exponential-complexity search problems cannot be solved by uninformed search for $d > 10$

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

II. Depth-First Search (DFS)

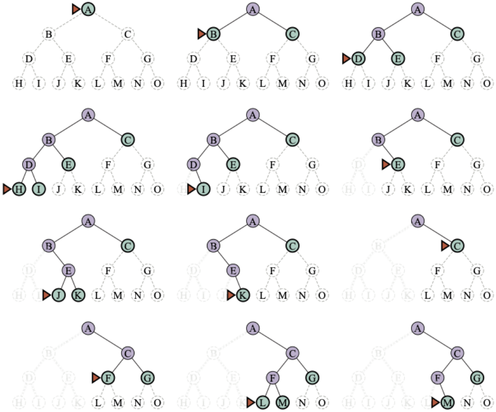


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

i. `DEPTH-FIRST-SEARCH` (DFS) always expands the **deepest node** in the frontier first

- → It is usually implemented as a tree-like search that **does not** keep a table of reached states
- → Search proceeds immediately to the **deepest level** of the search tree, where the nodes have no successors
- → The search then **"backs up"** to the **next deepest** node that still has unexpanded successors

ii. DFS is not **cost-optimal**, it returns the **first solution** it finds, even if it is not the cheapest

iii. Note that DFS,

- → For **finite state spaces** that are trees it is **efficient** and **complete**
- → For **acyclic state spaces** it may end up expanding the same state **many times** via different parts but will **systematically** explore the entire space
- → In **cyclic state space** it can get stuck in an infinite loop
- → In an **infinite state space** DFS is **not systematic**, it can **get stuck** going down an infinite path, thus DFS is **incomplete**

iv. However, for problems where a tree-like search is feasible, the DFS has **much smaller** needs for memory
   - $\rightarrow$ We **don't keep** a reached table at all, and the **frontier is very small**

v. For a finite tree-shaped state space, the DFS takes **time proportional** to the number of states, and has **memory complexity** of only $\mathcal{O}(b\,m)$,
   - $\rightarrow$ where b is the **branching factor** and m is the **maximum depth** of the tree

vi. Some problems that would require **exabytes** of memory with BFS can be handled with **only kilobytes** using DFS

III. Depth-First with Iterative Deepening Search (IDS)

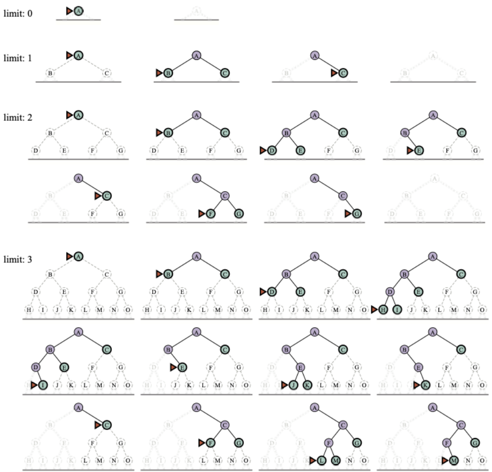


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

i. To keep DFS going down an **infinite path**, we can use **depth-limited search**, which is a version of DFS with a **depth limit** l, where all nodes at depth l are treated as if they had **no successors**
   - → The **time complexity** is $\mathcal{O}(b^l)$ and the **space complexity** is $\mathcal{O}(b\,l)$
   - → Unfortunately, if we make a poor choice for l the algorithm will **fail** to reach the solution, making it **incomplete** again

ii. **Iterative Deepening Search** (IDS) solves the problem of picking a good value for l by trying **all values**: first 0, then 1, then 2, and so on - until either a **solution is found**, or the depth-limited search **returns the failure**

iii. **IDS combines** many of the benefits of DFS and BFS
   - → Like **DFS**, its **memory requirements** are modest: $\mathcal{O}(bd)$ where there is a solution, or $\mathcal{O}(bm)$ on finite state spaces with no solution
   - → Like **BFS**, IDS is **optimal** for problems where all actions have the same cost, and is complete on finite acyclic state spaces or on any finite state space

iv. The **time complexity** is $\mathcal{O}(b^d)$ where there is a **solution**, $\mathcal{O}(b^m)$ where there is **none**

v. In an IDS, the nodes on the bottom level (depth d) are **generated once**, those on the **next-to-bottom** level are generated **twice**, and so on
   - $\rightarrow$ where children of the root are generated d times.
   - $\rightarrow$ The total number of **nodes generated**ß in the worst case is

$$N(IDS) = (d)b^1 + (d-1)b^2 + (d-2)b^3 + \cdots + b^d$$

vi. For example, if $b = 10$ and $d = 5$, the numbers are

$$N(IDS) = 50 + 400 + 3000 + 20,000 + 100,000 = 123,450$$
$$N(BFS) = 10 + 100 + 1000 + 10,000 + 100,000 = 111,110$$

vii. In general, **IDS** is the preferred **uniformed search** method when the search **state space is larger** than can fit in memory and the **depth** of the solution is **not known**

Summary

I. Search algorithms are judged on the basis of **completeness**, **cost of optimality**, **time complexity**, and **space complexity**
II. **Uninformed search** methods have access only to the problem definition. Algorithms build a **search tree** in an attempt to find a solution
III. Algorithms differ based on which node they expand first
   - **Breadth-first search** expands the **shallowest** nodes first, it is **complete**, **optimal for unit** action costs, but has **exponential** space complexity
   - **Depth-first search** expands the deepest unexpanded nodes first. It is neither complete nor optimal but has **linear space complexity**
   - **Iterative deepening search** calls depth first search with **increasing depth limits** until a goal is found. It is **complete** when full cycle checking is done, **optimal** for unit action course, has time complexity **comparable** to breath-first search, and has **linear** space complexity

Where are we ?

Heuristic Search [1, 2]

i. These search strategies use **domain-specific** hints about the location of goals and can find solutions more **efficiently** than an **uninformed** strategy

   $\rightarrow$ The **hints** come in the form of a **heuristic function**, denoted $h(n)$

   $h(n) =$ estimated cost of the **cheapest path** from the state at node n to a goal state

ii. For example, in **route-finding** problems, we can estimate the distance from the current state to a goal by computing the **straight-line** distance on the map between the two points

II. Generic/Greedy Best-First (GBF)

i. **GBF** search is a form of **best-first** search that **expands** first the node with **the lowest** $h(n)$ value, i.e., the node that appears to be **closest** to the goal

   → So the **evaluation funciton** $f(n) = h(n)$

ii. If the goal is Bucharest, the **straight-line** distances to Bucharest can be used as the **heuristic**. i.e., $h_{SLD}(Arad) = 366$, etc.

   → Note that h_{SLD} is **correlated** with actual distances and is therefore, a **useful** heuristic

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

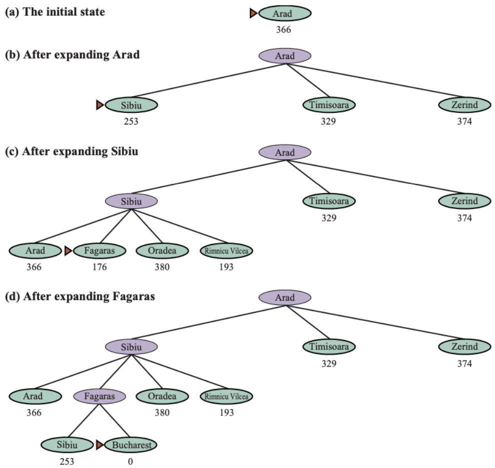


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

iii. Figure 3.17 shows the progress of a **Greedy Best-first** search using h_{SLD} to find a path from Arad to Bucharest
  - → The **first node** to be expanded from Arad will be Sibiu because the **heuristic says** it is closer to Bucharest than the other cities

iv. Note that in this case, GBF finds a solution **without ever** expanding a node that is **not on** the solution path
  - → The solution it found **does not** have **optimal cost**
  - → The path via Sibiu and Fagaras to Bucharest is 32 miles **longer** than the path through Rimnicu and Pitesti

v. On each **iteration** GBF tries to **get as close** to a goal as it can. This is why the algorithm is called "**Greedy**"
  - → But **greediness** can lead to **worse results** than being careful

III. A* Search

i. A* search is an **informed** search algorithm that uses the **evaluation function**

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

where

$g(n) =$ the **path cost** from the **initial state to node** n

$h(n) =$ the **estimated cost** of the **shortest path** from n to a goal state

so, we have

$f(n) =$ **estimated cost** of the **best path** that continues from n to a goal

ii. Note that Bucharest first appears on the frontier at step (e), but it is **not selected** for expansion (and thus not detected as a solution) because at $f = 450$ it is **not the lowest-cost node** on the frontier

→ **Lowest-cost** would be Pitesti, at $f = 417$.

→ I.e., 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

iii. At step (f), a different path to Bucharest is now the **lowest-cost** node, at $f = 418$, so it is selected and detected as the **optimal solution**

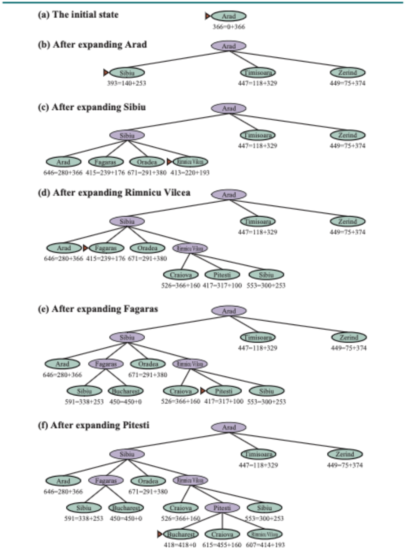


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

iii. Ath search is **complete**, and whether Ath is **cost-optimal** depends on certain **properties** of the heuristic

$\rightarrow$ A key property is **admissibility**; an admissible heuristic is one that **never overestimates** the cost to reach a goal

iv. A slightly stronger property is called **consistency**.

$\rightarrow$ A heuristic $h(n)$ is **consistent** if, for every node n and every successor n' of n generated by an action a, we have

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

$\rightarrow$ This is a form of the **triangle inequality**

v. Every **consistent heuristic is admissible** (but not vice versa), so with a **consistent heuristic**, A* is **cost-optimal**

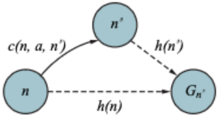


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

I. Hill Climbing (HC)

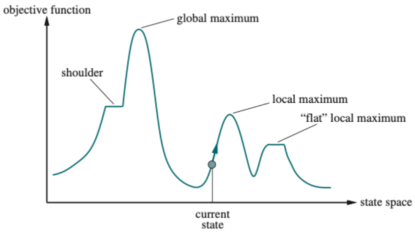


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

i. **Hill Climbing** algorithm is a type of **Local search** which operates by searching from a start state to neighbouring states
   - → They use very **little memory** and they can often find **reasonable solution** in large or **infinite state spaces**

ii. Consider the **states** of a problem laid out in a **state space** as shown in figure
   - → Each **point (state)** in the state space has an **elevation** defined by the value of the **objective function**
   - → The **aim** of the algorithm is to find the **highest peak** also called the **Global Maximum** and we call the process **hill climbing**

iii. The HC keeps track of **one current state** and on each iteration moves to the **neighbouring state** with **highest value**
   - → It heads in the direction that provides the **steepest ascent**
   - → It **terminates** when it reaches a **peak** where **no neighbour** has a higher value
   - → The HC **does not** look ahead **beyond the immediate** neighbours of the current state

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

v. HC is sometimes called **greedy Local search** because it grabs a good neighbour state **without thinking** ahead about where to go next

vi. HC can make **rapid progress** toward a solution because it is usually quite easy to **improve a bad state**. However, HC can **get stuck** for any of the following reasons

   I. **Local Maxima:** A **local maxima** is **a peak** that is higher than each of its **neighbouring states** but **lower than the global maximum**

   II. **Ridges:** Ridges result in a **sequence of local maxima** that is very difficult for greedy algorithm is to navigate

   III. **Plateaus:** This is a **flat area** of the state space landscape. It can be a flat local maxima from which **no uphill** exit exists. A HC search can get lost wandering on the plateau

vii. Many variants of HC have been invented,

   I. **Stochastic hill climbing**

   II. **First-choice hill climbing**

   III. **Random-restart hill climbing**

Summary

I. **Informed search** methods have access to a **heuristic function** $h(n)$ that estimate the cost of a solution from n.
II. **Generic Best-first** search expands nodes with **minimal** $h(n)$, it is **not optimal** but is often efficient
III. **A* search** expands nodes with minimal $f(n) = g(n) + h(n)$. A* is **complete** and **optimal**, provided that $h(n)$ is **admissible**
IV. **Hill climbing** keeps only a small number of states in memory
V. The **performance** of heuristic search algorithms depend on the **quality** of the heuristic function

Where are we ?

Constraint Satisfaction [1]

I. In this section we use **factored representation** for each **state**: a set of **variables**, each of which has a **value**

II. A problem is **solved** when each **variable** has a value that **satisfies** all the **constraints** on the variable

$\rightarrow$ A problem described this way is called a **constraint satisfaction problem**, or **CSP**

III. The main idea of the **CSP search** algorithms 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

i. A CSP consist of **three components**, $\mathcal{X}, \mathcal{D}$, and $\mathcal{C}$:

$\mathcal{X}$ is a set of **variables**, $\{X_1, \cdots, X_n\}$

$\mathcal{D}$ is a set of **domains**, $\{D_1, \cdots, D_n\}$

$\mathcal{C}$ is a set of **constaints** that specify allowable combinations of values.

ii. A domain D_i, consists of a set of allowable values, $\{v_1, \cdots, v_k\}$
  - → For example, a Boolean variable would have the domain { `true`, `false` }
  - → Different variables can have different domains of different sizes

iii. Each constraint C_j consists of a pair $\langle$`scope`, `rel`$\rangle$, 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
  - → For Example, if X_1 and X_2 both have the domain $\{1, 2, 3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as

$$\langle\, (X_1, X_2),\ \{(3,1), (3,2), (2,1)\}\, \rangle$$

or as

$$\langle\, (X_1, X_2),\ X_1 > X_2\, \rangle$$

iv. CSPs deal with assignments of values to variables $\{X_i = v_i, X_j = v_j, \cdots\}$
  - → **Consistent assignment:** An assignment that does **not violate** any constraints is called a **consistent** or legal assignment
  - → **Complete Assignment:** A complete assignment is one in which **every variable** is assigned a value, and
    - ◦ a **solution** to a CSP is a **consistent**, **complete assignment**
  - → **Partial Assignment:** A partial assignment is one that leaves some variables **unassigned**, and
    - ◦ a **partial solution** is a partial assignment that is consistent

Example problem: Map coloring

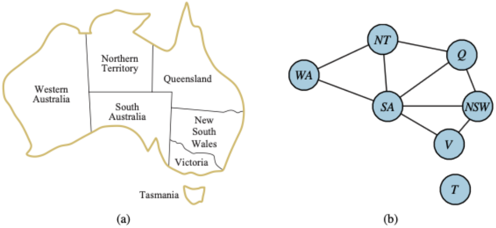


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

i. Consider we are given the task of **colouring** each region of Australia either `red`, `green`, or `blue` in such a way that **no two neighbouring** regions have the same colour

ii. To **formulate** this as a CSP, we define the **variables** to be the **regions**

$$\mathcal{X} = \{\texttt{WA},\ \texttt{NT},\ \texttt{Q},\ \texttt{NSW},\ \texttt{V},\ \texttt{SA},\ \texttt{T}\}$$

iii. The **domain** of every variable is the set

$$\mathcal{D}_i = \{\ \texttt{red},\ \texttt{green},\ \texttt{blue}\}$$

iv. The **constraints** require neighbouring regions to have distinct colours

$$\mathcal{C} = \{\ \texttt{SA}{\neq}\texttt{WA},\ \texttt{SA}{\neq}\texttt{NT},\ \texttt{SA}{\neq}\texttt{Q},\ \texttt{SA}{\neq}\texttt{V},\ \texttt{WA}{\neq}\texttt{NT},\ \texttt{NT}{\neq}\texttt{Q},\ \texttt{Q}{\neq}\texttt{NSW},\ \texttt{NSW}{\neq}\texttt{V}\ \}$$

v. There are many possible **solutions** to this problem, such as

$$\{\ \texttt{WA}{=}\texttt{red},\ \texttt{NT}{=}\texttt{green},\ \texttt{Q}{=}\texttt{red},\ \texttt{NSW}{=}\texttt{green},\ \texttt{V}{=}\texttt{red},\ \texttt{SA}{=}\texttt{blue},\ \texttt{T}{=}\texttt{red}\ \}$$

vi. A CSP can be **visualised** as a **constraint graph**, where the **nodes** of the graph correspond to **variables** of the problem and **an edge** connect any two variables that participate in a **constraint**

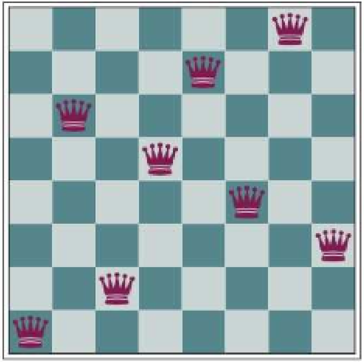
vii. Note, once we have chosen {`SA=blue`}, we can conclude that **none** of the five neighbouring variables can take one the value `blue`

viii. A search procedure that **doesn't use** constraints would have to consider $3^5 = 243$ assignments, however **with constraints**, we have only $2^5 = 32$ assignments to consider

ix. With CSPs, once we find out that a **partial assignment** **violates a constraint**, we can immediately **discard** further refinement of the partial assignment

→ As a result, many problems that are **intractable** for **atomic state space** search can be **solved quickly** when formulated as a CSP

Example problem: n-Queen Problem



(a)

Figure 4.3 (a) The 8-queens problem: place 8

i. The **8-queens** problem: place 8 queens on a chess board so that no queen **attacks** another. (A queen attacks any piece in the same row, column, or diagonal.)

ii. The positions in the figures are **almost a solution**, except for the two queens in the fourth and seventh columns that attack each other along the diagonal.

Constraint Propagation in CSPs

i. An **atomic state space search** algorithm makes progress by **expanding a node** to visit the successors

ii. Where as, a **CSP algorithm** generates successesors by doing a **specific type of inference** called **constraint propagation**

→ The constraints **reduce the number of legal values** for a variable, which in turn can reduce the legal values for **another variable** and so on

→ The **idea** is that this will **leave fewer choices** to consider when we make the next choice of variable assignment

I. Backtracking

i. Sometimes we can finish the **constraint propagation** process and still have variables with multiple possible values
   - → In that case we have to **search** for a solution

ii. In this section we cover **backtracking search** algorithms that work on **partial assignments**

iii. In figure 5.5, the **backtracking search** procedure for CSP repeatedly chooses **unassigned variable**, and then **tries all values** in the domain of that variable in turn, trying to extend each one into a solution via a recursive call
   - → If the call succeeds, the **solution** is returned, and
   - → if it fails, the assignment is **restored** to the previous state, and we **try the next value**.
   - → If no value works then we return **failure**

iv. Part of the search tree is shown in Fig 5.6, where we have assigned variables in the order `WA, NT, Q,` etc.,

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

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

Figure 5.5 A simple backtracking algorithm for constraint satisfaction problems.

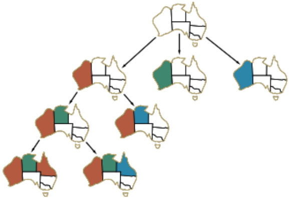


Figure 5.6 Part of the search tree for the map-coloring problem in Figure 5.1.

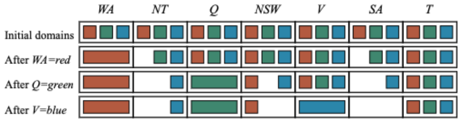


Figure 5.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After $Q = green$ is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After $V = blue$ is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

v. **Forward Checking:** One of the simplest form of **inference** is called **forward checking**

- → Whenever a **variable X is assigned**, the forward checking process establishes **consistency** for it
- → For each **unassigned variable** Y that is connected to X by a **constraint**, delete from Y's domain in any value that is **inconsistent** with the value chosen for X

II. Local Search

i. **Local search** algorithms turn out to be **very effective** in solving many CSPs
   - → They use a **complete state formulation** where each state assigns a value to every variable, and the **search** changes the value of **one variable at a time**

ii. For example, we use the 8-Queens problem as in the figure
   - → A random **complete assignment** to the 8 variables will typically **violate** several constraints
   - → We then randomly choose a **conflicted variable**, which turns out to be Q_8
   - → We would like to change the **value** to something that brings us **closer to be a solution**, i.e., select the value that results in the **minimum number** of **conflicts** with other variables - the **min-conflicts** heuristic
     - ○ We pick $Q_8 = 3$, as that only violates one constraint
   - → On the **next iteration**, we select Q_6 as the variable to change, and note that moving the Queen to $Q_6 = 8$ results in no conflict
   - → At this point there are no more conflicted variables, so we have a **solution**

iii. Amazingly, on the n-Queens problem, the run-time of min-conflicts is roughly `independent of problem size`
   - → It solves even the **million-queens** problem in an average of **50 steps**
   - → Roughly speaking, n-queens is **easy** for **local search** because solutions are **densely distributed** throughout the state space

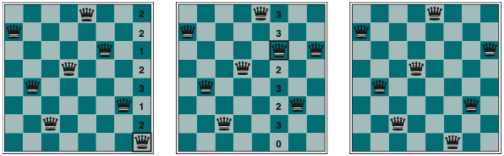


Figure 5.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or *failure*
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

```
current ← an initial complete assignment for csp
for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
    set var = value in current
return failure
```

Figure 5.9 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Text Books

[1] R. S and N. P, *Artificial Intelligence: A Modern Approach*.
Third Edition, Prentice-Hall, 2009.

[2] S. B. N. Elaine Rich, Kevin Knight, *Artificial Intelligence*.
Third Edition, The McGraw Hill Publications, 2009.

[3] G. F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*.
Sixth Edition, Pearson Education, 2009.

Thank you