Artificial Intelligence @ KIIT

CS 3011

Author: Ajay Anand

Department: School of Computer Engineering

Institute: Kalinga Institute of Industrial Technology (DU), Bhubaneswar

Date: 2023/08/28 04:38:31

Version: 2307a

Contents

1 Notices **1**

1.1 Top Announcement . . . 1
1.2 Course Information . . . 1
1.3 Aug 24: Quiz and class notes inspection . . . 4
1.4 Aug 25: Assignment 1 . . . 4
1.5 Aug 28: Activities updated . . . 9

2 Introduction **10**

2.1 What is artificial intelligence? . . . 10
2.2 A Brief History of AI . . . 12
2.3 Current State . . . 13
2.4 Applications . . . 13

3 Intelligent Agents **15**

3.1 Agent . . . 15
3.2 Table-driven agent . . . 16
3.3 Simple reflex agent . . . 17
3.4 Structure of agent . . . 18
3.5 Model-based reflex agent . . . 18
3.6 Goal-based agent . . . 19
3.7 Utility-based agent . . . 20
3.8 Rationality and agent learning automatically . . . 21
3.9 Environment . . . 22

4 Searching **25**

4.1 Search problem . . . 26
4.2 General search algorithm . . . 27
4.3 Uninformed search strategies . . . 29
4.4 Informed search strategies . . . 34
4.5 Discovering good heuristic functions . . . 38
4.6 Variations of basic search . . . 40
4.7 Comparison . . . 41

5 Local Search Algorithms and Optimization Problems **43**

5.1 Hill-climbing search . . . 44
5.2 Simulated annealing . . . 47

5.3 Local beam search 48

Section 1 Notices

1.1 Top Announcement

Refer to (click here) section 1.3 for Quiz 1 announcement.

Refer to (click here) section 1.4 for Assignment 1 announcement.

1.2 Course Information

1.2.1 Links

Note *This file may be old. Refer to the latest version of this file from the course folder.*

Course folder

https://drive.google.com/drive/folders/1ZC32z3UAWf-BX-h25Ca6koIMXrxk86L1?usp=sharing

Whatsapp group

https://chat.whatsapp.com/KatvgJ8BLEM88hbAt4izgL

1.2.2 Time table

Batch: CS-10

Mon	*11 a.m.*	*A-LH-005*
Wed	*4 p.m.*	*A-LH-005*
Thu	*12 noon*	*A-LH-005*

1.2.3 Lesson plan

Module	Topic	Hours	Lectures
1. Introduction	1. What is AI? 2. The foundations of AI 3. The history of AI 4. The State of the Art	3	1-3

2. Intelligent Agents	1. Agents and Environments 2. The good behavior: The concept of rationality 3. The nature of Environments 4. The Structure of Agents 5. The Learning Agent	6	4-9
3. Solving Problems by Searching	1. Problem Solving Agents 2. Example Problems 3. Searching for Solutions 4. Uninformed Search Strategies 5. Avoiding repeated states 6. Informed Search Strategies 7. Heuristic functions	12	10-21
4. Beyond Classical Search	1. Local search algorithms 2. Optimization problems 3. Hill Climbing 4. Simulated Annealing 5. Local Beam Search 6. Genetic Algorithm	6	22-27
5. Constraint Satisfaction Problems	1. Definitions 2. Australian color mapping 3. Job shop scheduling 4. Sudoku game 5. Cryptarithmetic 6. Types of variables 7. Types of constraints 8. Types of consistencies 9. Constraint propagation 10. Backtracking search for CSPs 11. Local search for CSPs	4	28-31

6. Adversarial Search	1. Games 2. Optimal decision in games 3. Minimax values 4. Minimax algorithm 5. Alpha-Beta pruning	5	32-36
7. Logical Agents	1. Knowledge-based agents 2. Wumpus World 3. Entailment 4. Inference 5. Sound and complete inference algorithms 6. Propositional logic 7. Various inference procedures such as model checking and theorem proving 8. Forward and backward chaining	4	37-40
8. First-Order Logic and its Inference	1. Syntax and semantics of First-Order Logic 2. Propositional vs First-Order Inference	3	41-43
9. Planning	1. Definition of classical planning 2. Planning Domain Definition Language 3. Application of PDDL in some example problems 4. Algorithms for classical planning	2	44-45

1.2.4 Resources

Text Book

Artificial Intelligence: A Modern Approach, Stuart Russel, Peter Norvig, Pearson Education

Reference Books

1. Artificial Intelligence, Rich, Knight and Nair, Tata McGraw Hill.
2. Principles of Artificial Intelligence, Nils J. Nilsson, Elsevier, 1980.

1.2.5 Activities

Before Mid-Semester Exam

1. *Quiz [5 marks]*
2. *Assignment [5 marks]*
3. *Notes inspection [2.5 marks]*

After Mid-Semester Exam

1. *Project-in-pairs or Quiz [10 marks]*
2. *Assignment [5 marks]*
3. *Notes inspection [2.5 marks]*
4. *Defaulter activity (to make up for single missed activity)*

1.3 Aug 24: Quiz and class notes inspection

1. An online MCQ quiz will be conducted on 31-Aug-2023.
2. Your class notes will be inspected on the same day.

1.4 Aug 25: Assignment 1

1. Submit on or before 5-Sep-2023 in the classroom or at Room 13, Campus 15-A (library building) between 2 p.m. to 4:30 p.m. on working days.
2. Write on standard A4 size paper.
3. Answer any 5 out of 7 questions assigned to you as per the table below.

Roll #	Question #							Roll #	Question #						
2105001	3	4	16	18	23	25	27	2105031	1	2	5	8	19	23	28
2105043	8	9	13	15	17	20	21	2105065	4	6	7	10	12	18	24
2105073	1	5	11	16	18	22	25	2105120	1	3	7	10	12	15	21
2105243	2	5	10	11	14	15	23	2105297	1	4	7	14	16	21	22
2105420	3	5	8	12	16	17	19	2105512	7	8	13	15	17	21	28
2105589	4	14	15	19	21	24	26	2105605	6	9	11	13	15	16	22
2105707	2	3	4	6	9	19	20	2105776	3	6	11	18	19	20	21
2105787	4	8	9	16	17	18	23	2105790	9	20	21	22	26	27	28
2105812	1	4	5	6	9	24	25	2105866	3	4	9	21	24	25	26
2105876	5	6	12	15	20	23	26	2105898	5	6	10	13	14	20	21
2105911	3	9	10	12	18	26	27	2105982	5	8	16	17	20	22	28
2105983	7	8	11	19	20	26	27	21051008	4	7	11	14	15	22	24
21051023	5	13	15	16	20	21	22	21051057	2	8	9	11	12	13	26
21051090	1	3	6	10	19	20	28	21051109	2	16	19	20	21	25	26

21051150	3	5	8	10	18	21	25	21051179	1	2	4	13	17	18	25
21051257	5	7	12	18	19	20	22	21051329	3	10	13	18	19	20	23
21051374	2	3	11	13	15	26	27	21051381	2	5	8	16	22	25	27
21051484	2	4	8	12	24	25	28	21051498	2	7	9	12	21	25	26
21051530	14	16	17	18	19	23	27	21051531	3	6	15	21	22	24	28
21051611	8	11	14	18	19	20	21	21051618	1	5	10	13	17	23	24
21051691	6	8	9	11	22	23	26	21052007	3	5	11	12	20	22	23
21052022	10	16	19	20	21	23	24	21052035	4	9	14	18	20	21	26
21052082	2	6	9	10	13	17	21	21052091	5	7	8	9	13	14	19
21052165	4	7	10	12	18	19	21	21052182	3	7	8	12	13	16	18
21052206	2	3	4	6	18	24	25	21052308	2	7	10	20	21	26	28
21052401	3	8	11	13	14	23	28	21052402	2	7	11	12	19	20	24
21052481	4	6	10	14	17	21	28	21052519	6	13	14	20	21	22	26
21052549	4	5	11	13	14	22	23	21052559	8	11	12	15	20	24	26
21052566	2	10	12	15	17	23	28	21052619	3	5	6	10	15	16	27
21052655	1	5	9	17	18	22	23	21052694	2	4	8	16	17	19	20
21052701	3	4	5	9	12	15	19	21052706	2	13	17	21	23	24	28
21052837	4	6	12	14	15	17	20	21052843	3	6	10	11	13	15	18
21052898	1	4	12	19	22	24	25	21052940	1	2	5	6	7	8	15
21052949	3	4	5	14	17	20	26	21052960	9	10	15	18	20	22	23
21053233	10	11	12	19	20	22	25	21053234	1	3	6	11	17	18	25
21053258	4	5	12	15	16	24	26	21053264	7	9	13	15	20	23	27
21053286	6	10	15	18	20	21	26	21053304	2	4	7	9	12	18	27
21053357	6	9	12	14	17	23	26	21053457	3	5	7	10	11	18	25
21053470	1	3	7	10	21	25	26	22057029	5	7	10	15	18	25	26

Questions

1. Describe the criteria used to measure the performance of search algorithms. Explain with suitable sketch of the state-space landscape the issues that an agent may face with a simple hill-climbing search.
2. Explain the various operators in genetic algorithm with an example.
3. Differentiate between informed and uninformed search strategies and provide two examples of each.
4. Explain A* search with an example. Show that A* search is optimal.
5. How are agent and its environment related to each other? Describe the architecture of simple reflex agent with a diagram.
6. Describe the architecture of goal-based reflex agent with a diagram. When is it recommended and when is it not recommend to use goal-based agent?

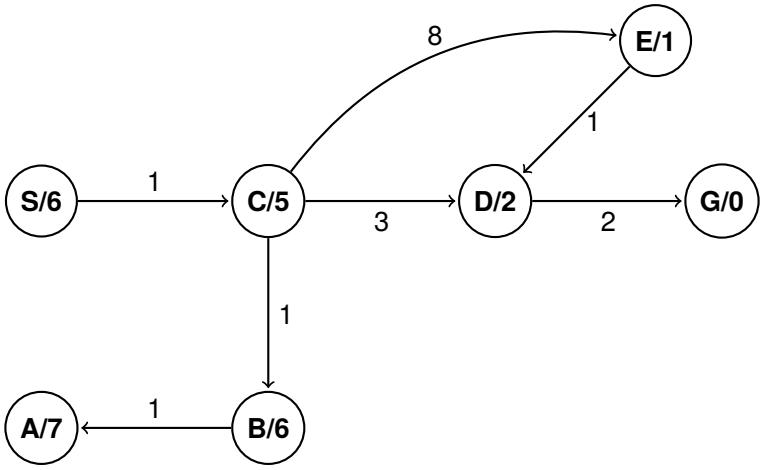


Figure 1.1: Graph 1

7. What are the common four measures to measure the performance of any search strategy? Give in a tabular manner, the comparison of performance measures of following four search strategies:
   i) Breadth first search
   ii) Depth first search
   iii) Depth limited search
   iv) Iterative deepening search
8. In figure 1.1 find the path from S to G using
   - depth-first search
   - breadh-first search

   Show the contents of the priority queue after each step in both the algorithms. Ignore the edge costs and heurist values. Break ties by visiting lexographically earlier nodes first.
9. In figure 1.1 find the path from S to G using uniform-cost search. Show the contens of the frontier (priority queue) and visited nodes (dictionary) after each step. Ignore the heurist values given at the nodes.
10. In figure 1.1 find the path from S to G using greedy best-first search. Show the contens of the frontier (priority queue) and visited nodes (dictionary) after each step.
11. In figure 1.1 find the path from S to G using $A^\star$ search. Show the contens of the frontier (priority queue) and visited nodes (dictionary) after each step.
12. Justify the statement: ”Consistency implies admissibility”. State the condition under which A* algorithm is optimal. What are the advantages of having a consistent heuristic?
13. Describe 4 types of AI: Describe 4 types of AI: Reactive machines, Limited memory, Theory of mind, Self-awareness.
14. What is the role of turing test in AI? Illustrate difference between thinking rationally and thinking humanly with an example.
15. Draw the schematic diagram of learning agent? Explain its various components through

a suitable example.

16. Give a PAGE (Percepts, Actions, Goals, Environment) description for an automated crossword player. What are the possible problems that can be encountered while developing it?
17. Solve the following problem and specify

    a) Solution Space

    b) Algorithm

    c) Optimal Path Length

    You are given two jugs, a 4-gallon and 3-gallon. Neither has any measuring markers on it. How can you exactly get 2 gallons of water into the 4-gallons jug.
18. Briefly explain why a table-driven agent would fail. What are four basic kinds of agent programs/structures that embody the principle underlying almost all intelligent systems?
19. What is Task environment? How is it specified? What is PEAS specification? Write the PEAS specifications for the task environments of the following agents in a tabular manner:

    i) Medical diagnosis system

    ii) Part-picking robot system

    iii) Automated taxi driving agent

    iv) Vacuum world
20. What is the branching factor in 24-puzzle (sliding tile) problem averaged over all locations? What are the main advantages and disadvantage of depth-limited search over breadth-first?
21. Consider the vacuum cleaner agent moving between two rooms. In each time step the agent can perform only one of the following tasks:

    i) change its location and check the status of the room,

    ii) stay in the same location and check the status of the room,

    iii) suck the dirt and check the status of the room.

    The reward for each action is -1 unit and the reward of checking the status of the room and finding it clean is 2 units.

    In every 6 time steps one of the two rooms randomly becomes dirty.

    Compute the total reward of the agent after 600 time steps.
22. Write the algorithm for best-first search. Suggest the modifications required to make this algorithm behave like following by providing suitable variations of the cost and priority functions.

    a) Uniform-cost search

    b) Breadth-first search

    c) Depth-first search
23. Why is it important to choose an appropriate heuristic function? Why uninformed search

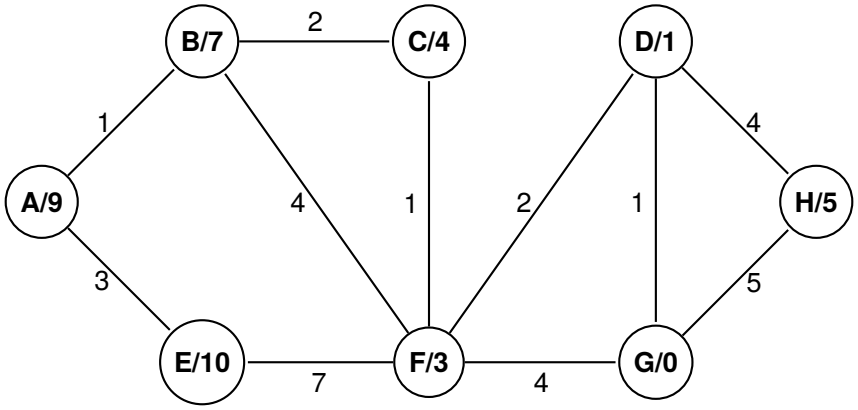


Figure 1.2: Graph 2

techniques in some condition are better than informed search techniques?

24. In GA, for Eight-Queen problem, suppose following four real numbered chromosomes are selected as parents in the population pool:

    1) 32748552

    2) 24752411

    3) 32752124

    4) 24415411

    If the objective function (i.e. fitness function) is defined as "Number of non-attacking pairs of queens", then evaluate the fitness value for each of these parents.

    If the crossover is done at crossover point 3 for the first pair of parents (i.e. parents 1 and 2) and if the crossover is done at crossover point 5 for the second pair of parents (i.e. parents 3 and 4) then enlist the four offspring chromosomes (i.e. children) those get generated due to these crossover operations. Also evaluate the fitness value for each of these children.

25. In figure 1.2 find the path from A to G using
    - depth-first search
    - breadh-first search

    Show the contents of the priority queue after each step in both the algorithms. Ignore the edge costs and heurist values. Break ties by visiting lexographically earlier nodes first.

26. In figure 1.2 find the path from A to G using uniform-cost search. Show the contens of the frontier (priority queue) and visited nodes (dictionary) after each step. Ignore the heurist values given at the nodes.

27. In figure 1.2 find the path from A to G using greedy best-first search. Show the contens of the frontier (priority queue) and visited nodes (dictionary) after each step.

28. In figure 1.2 find the path from A to G using $A^\star$ search. Show the contens of the frontier (priority queue) and visited nodes (dictionary) after each step.

1.5 Aug 28: Activities updated

Project has been added as an alternative to quiz 2 and a special activity has been added for the defaulters.

Section 2 Introduction

Prelude

- ❏ *Definition*
- ❏ *Turing Test*
- ❏ *History*
- ❏ *Applications*

2.1 What is artificial intelligence?

Artificial Intelligence

AI is the science of making machines that

1. *Think like people*
2. *Think rationally*
3. *Act like people*
4. *Act rationally*

♣

2.1.1 Think like people

To say that a program thinks like a human, we must know how humans think. We can learn about human thought in three ways:

- introspection-trying to catch our own thoughts as they go by;
- psychological experiments-observing a person in action;
- brain imaging-observing the brain in action.

Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input-output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind.

2.1.2 Think rationally

Laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic. Logic as conventionally understood requires knowledge of the world that is certain, a condition that, in reality, is seldom achieved. The theory of probability fills this gap, allowing rigorous reasoning with uncertain information. Being rational means maximizing your expected utility.

2.1.3 Act like people

The Turing test, proposed by Alan Turing (1950), was designed as a thought experiment that would sidestep the philosophical vagueness of the question "Can a machine think?" A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.

Turing Test

"Can machines think?" , "Can machines behave intelligently?"

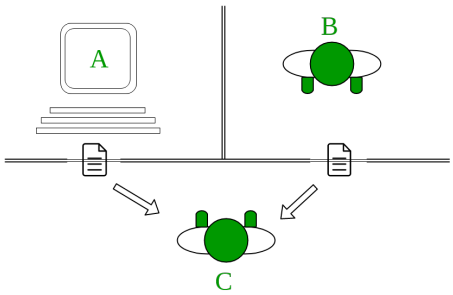


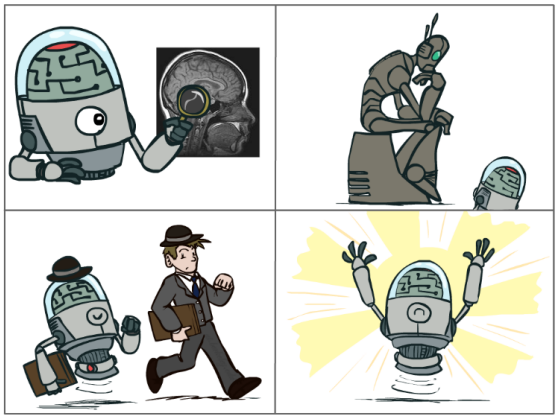
Figure 2.1: Turing test

Six Capabilities of a Computer to pass Total Turing Test

1. Natural Language Processing (to communicate successfully in a human language)
2. Knowledge Representation (to store what it knows or hears)
3. Automated Reasoning (to answer questions and to draw new conclusions)
4. Machine Learning (to adapt to new circumstances and to detect and extrapolate patterns)
5. Computer Vision (and speech recognition to perceive the world)
6. Robotics (to manipulate objects and move about)

2.1.4 Act rationally

An agent is just something that acts (agent comes from the Latin agere, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. All the skills needed for the Turing test also allow an agent to act rationally.



2.2 A Brief History of AI

- Initial Stage

  In 1943 McCulloch and Pitts proposed boolean circuit model of brain. During same time Turing's computing machine was developed.

- Initial Excitement

  In 1950s AI programs were developed for playing checkers and simple games. Theories on logical reasoning were enhanced. In 1956 we had the first conference related to AI knows as Dartmouth meeting. In 1965 Robinson proposed a complete algorithm for logical reasoning.

- Knowledge-based Intelligence

  In 1970s knowledge based systems emerged that formed the foundation for expert systems adopted for industrial applications in 1980s. But the expert system was found extremely limited which led to "AI winter" in 1990s.

- Mathematics and Statistics

  In the mid-1980s at least four different groups reinvented the back-propagation learning algorithm first developed in the early 1960s. This led to the return of neural networks. Late 1990s saw the extensive use of probability and uncertainty which helped to better understanding of the nature of problems. Several machine learning models were invented during this period: Hidden Markov models, Bayesian networks, Support vector machines etc. This led to the emergence of agents that could learn. Finally a spring after the winter.

- Big data

  From about 2001 onward, remarkable advances in computing power and the creation of the World Wide Web have facilitated the creation of very large data sets - a phenomenon sometimes known as big data. This has led to the development of learning algorithms specially designed to take advantage of very large data sets.

- Deep learning

  The term deep learning refers to machine learning using multiple layers of simple, ad-

justable computing elements. It was not until 2011, however, that deep learning methods really took off. This occurred first in speech recognition and then in visual object recognition.

In the 2012 ImageNet competition, which required classifying images into one of a thousand categories (armadillo, bookshelf, corkscrew, etc.), a deep learning system created in Geoffrey Hinton's group at the University of Toronto (Krizhevsky et al., 2013) demonstrated a dramatic improvement over previous systems, which were based largely on handcrafted features. Deep learning relies heavily on powerful hardware. Whereas a standard computer CPU can do 10^9 or 10^{10} operations per second. a deep learning algorithm running on specialized hardware (e.g., GPU, TPU, or FPGA) might consume between 10^{14} and 10^{17} operations per second, mostly in the form of highly parallelized matrix and vector operations.

2.3 Current State

- Drive safely along a curving mountain road?
- Drive safely in Bangalore traffic?
- Discover and prove new mathematical theorems?
- Talk successfully with another person for an hour?
- Perform a surgical operation?
- Fold laundry?
- Translate spoken Chinese into spoken English in real time?
- Create a brand new exciting and funny story?

2.4 Applications

1. Vision

   Training time for image recognition dropped by factor of 100. Error rates for object detection (as achieved in LSVRC, the Large-Scale Visual Recognition Challenge) improved from 28% in 2010 to 2% in 2017, exceeding human performance.
2. Language

   Accuracy on question answering, as measured by F1 score on the Stanford Question Answering Dataset (SQ UAD), increased from 60 to 95 from 2015 to 2019.
3. Robotic vehicles

   In 2005, 132-mile DARPA Grand Challenge was won successfully. In 2018, Waymo test vehicles passed the landmark of 10 million miles driven on public roads, the human driver stepping in to take over control only once every 6,000 miles. Soon after, the company began offering a commercial robotic taxi service. Multiple companies are working on advancing drone delivery and flying taxi.

4. Legged locomotion

   BigDog and Spot can play soccer and are no longer the slow, stiff-legged, side-to-side gait of movie robots, but something closely resembling an animal.

5. Autonomous planning and scheduling

   EUROPA planning toolkit was used for daily operations of NASA's Mars rovers.

   Every day, ride hailing companies such as Uber and mapping services such as Google Maps provide driving directions for hundreds of millions of users with traffic updates.

6. Machine translation

   Most translation is understandable if not that accurate. For closely related languages, like French and English, translation is at par with human proficiency.

7. Speech recognition

   Skype provides real-time speech-to-speech translation in ten languages.

   Alexa, Siri, and Google assistants

   Google Duplex service

8. Recommendations

   Companies like Amazon, Facebook, Netflix, Spotify and YouTube recommend what you might like based on your past experiences and those of others like you.

9. Game playing

   In 1997 Deep Blue defeated Garry Kasparov.

   In 2018, ALPHA ZERO used no input from humans (except for the rules of the game), and was able to learn through self-play alone to defeat all opponents, human and machine, at Go, chess, and shogi.

10. Medicine

    AI algorithms now equal or exceed expert doctors at diagnosing many conditions, particularly when the diagnosis is based on images. Examples include Alzheimer's disease (Ding et al., 2018), metastatic cancer (Liu et al., 2017; Esteva et al., 2017), ophthalmic disease (Gulshan et al., 2016), and skin diseases (Liu et al., 2019c). A systematic review and meta-analysis (Liu et al., 2019a) found that the performance of AI programs, on average, was equivalent to health care professionals.

11. Climate science

    A team of scientists won the 2018 Gordon Bell Prize for a deep learning model that discovers detailed information about extreme weather events that were previously buried in climate data. They used a supercomputer with specialized GPU hardware to exceed 10^{18} operations per second.

Section 3 Intelligent Agents

Prelude

- *AI Agent*
- *Agent types*
- *Learning agent and Rationality*
- *Environment*
- *PEAS*

3.1 Agent

Agent

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- The environment could be everything-the entire universe!
- Sensors are like our sense organs or camera fitted on a robot.
- Actuators are like our hands and feet or wheels fitted on a robot.
- Perceive means data stream from sensors, generally called percept sequence.
- The action is decided through agent function.

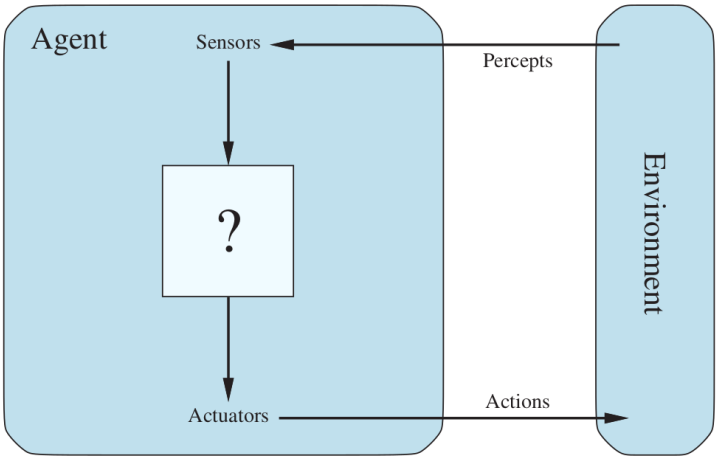


Figure 3.1: AI agent

A vacuum-cleaner world is an environment 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.

3.2 Table-driven agent

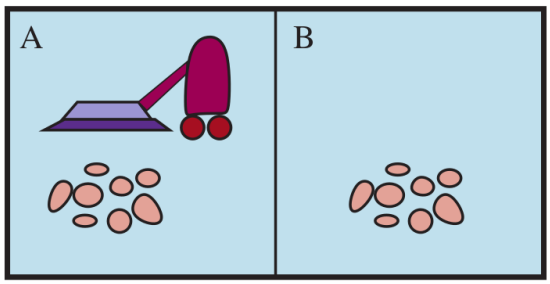


Figure 3.2: Simple reflex agent

The simplest form of agent function is a table listing appropriate action for different percept sequences.

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 3.3: Function table

Such a table can be extremely long. The implementation of agent function is called agent program.

```
tableDrivenAgent(A, p)    // A is agent, p is a percept
  A.q.push(p)             // q is percept sequence
  a = A.t.lookup(A.q)     // t is agent function table
  return a                // a is the action decided by the agent
```

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.

3.3 Simple reflex agent

Vacuum world

Simple reflex 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 was tabulated earlier, is also a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. Here is an example of a rule table.

Rules

if state is	then action is
(A, Dirty) or (B, Dirty)	suck
A, Clean	right
B, Clean	left

Such 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 relevant percept sequences from 4^n 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. Here's an agent program that uses a rule table.

```
simpleReflexAgent(A, p)      // A is agent, p is percept
  s = A.interpret(p)         // s is agent's state derived from p
  r = A.t.select(s)          // t is table containing rule for each state
  return r.a                 // r is selected rule and a is action dictated by r
```

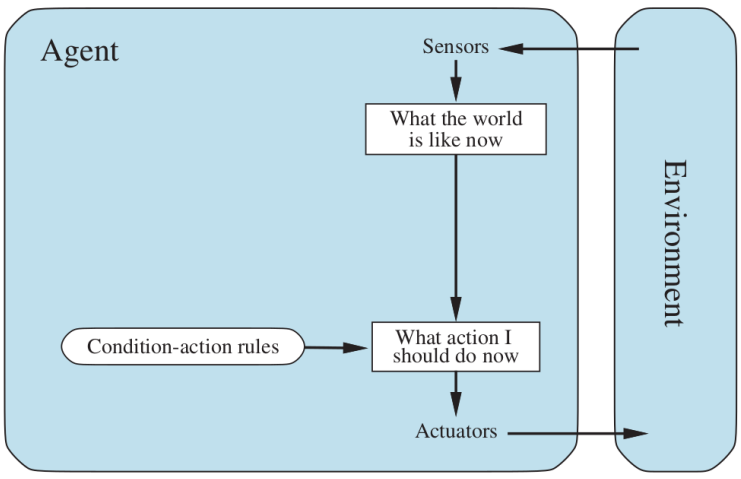


Figure 3.4: Simple reflex agent

3.4 Structure of agent

An agent consists of architecture (hardware) and agent program (software). *agent = architecture + program* The sensor inputs and actions that can be used in agent function are determined by the architecture of the agent. Here are some different types of most common agent programs:

- Simple reflex agents *(already discussed above)*
- Model-based reflex agents
- Goal-based agents
- Utility-based agents

3.5 Model-based reflex agent

Model-based agent

Instead of storing entire history or instead of storing no history at all, an agent can maintain some sort of internal state that depends on the percept history and thus captures some of the unobserved aspects of the current situation.

For example, to apply a brake at red signal, a self-driving agent needs to know only of the signal, but to change lane, it needs to know and remember all other surrounding vehicles. The state can have simple or complex representation like,

- Atomic representation: state is indivisible. For example current speed.
- Factored representation: state may contain multiple variables. For example speed, direction, distance covered, GPS coordinates all may be stored in the state.
- Structured representation: in real life scenarios, a state should store details of the agent and its environment explicitly, like information about other vehicles, objects and humans in the surroundings. For this structured representation is used where the attributes are no longer fixed but are flexible just like in relational databases.

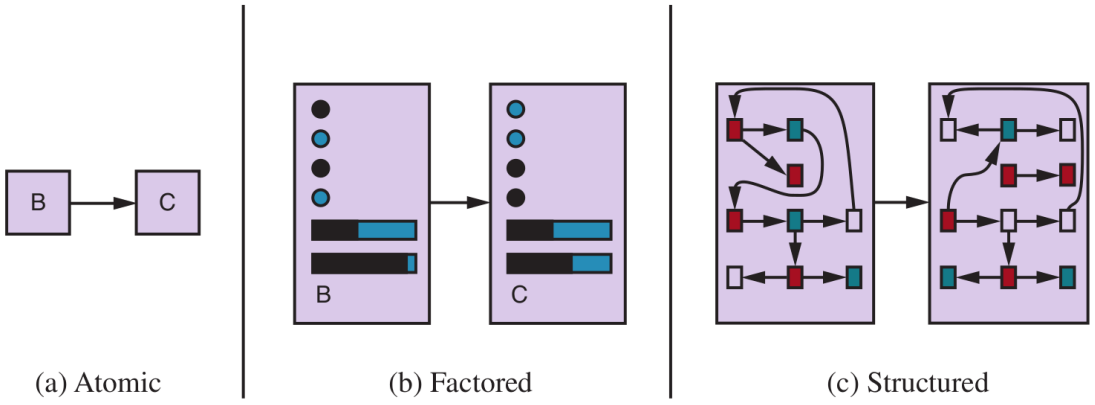


Figure 3.5: State representations

```
modelBasedReflexAgent(A, p) //A is agent, p is percept
  A.s = A.updateState(A.s, A.a, p, A.tm, A.sm)  //s,a are previous state and action
                                                //tm is transition model, sm is sensor model
  r = A.t.select(A.s)                           //t is table containing rule for each state
  return r.a                                    //r is selected rule and a is action dictated by r
```

Agent's state should capture two types of information.

- The process of natural evolution of environment without agent's involvement and the effect of agent's actions on the environment need to be encoded. Such encoding is called transition model of the world. For example, steering right will change the direction from North to East.
- A part of agent's state need to be inferred from what percepts tell about the agent and its environment. This is encoded through sensor model. For example if image of the road is noisy, it may be due to rain or dust storm.

Together, the transition model and sensor model allow an agent to keep track of the state of the world to the greatest extent allowed by its sensor system. An agent that uses such models is called a model-based agent.

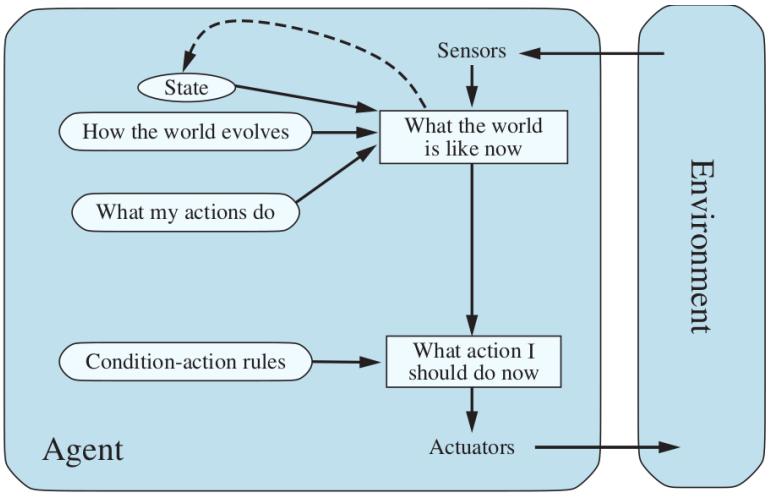


Figure 3.6: Model-based agent

3.6 Goal-based agent

Knowing something about the current state of the environment is not always enough to decide what to do. For example, a self-driving agent may turn or keep going straight, may speed up or slow down or maintain a constant speed.

Goal-based agent

In many situations the goal is known but the actual sequence of actions to be performed to reach the goal is flexible. This information is then combined with the basic reflex

based model. Such agent is called goal-based agent and it makes selection of actions dynamically based on well defined goal.

The problem reduces to finding sequence of actions that achieve the agent's goals by considering future results of available actions. No rule table associating percepts with actions is available. Due to extra task of searching a good sequence of actions, goal-based agent appears less efficient, but it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. For example, a self driving agent's destination can be changed without worrying about how exactly it will reach there.

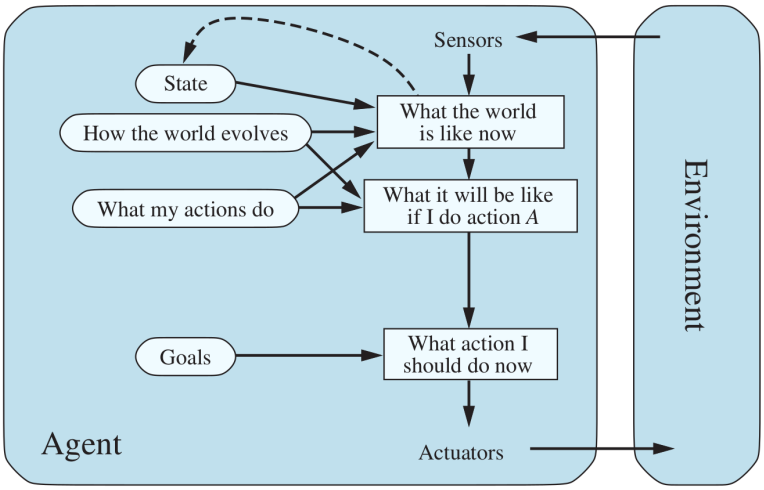


Figure 3.7: Goal-based agent

3.7 Utility-based agent

Many action sequences will get a self-driving agent to its destination (thereby achieving the goal), but some are quicker, safer, more reliable, or cheaper than others. For a high quality behavior, that not just focuses on the goal but also on the quality of actions, a more general performance measure should be provided that allow a comparison of different world states according to exactly how beneficial it is to the agent. Technically it is called state of happiness of the agent or the agent's utility function, which is used as performance measure along with the goal. So the problem reduces to searching a good sequence of actions that maximizes the overall utility function.

Utility-based agent

The agent's happiness in terms of a utility function is maximized. Utility-based agent works very efficiently in an unknown and uncertain environment, making the agent more rational due to its probabilistic reasoning.

There are utility-based agents that are model-free, where agent can learn how to act without learning exactly about the environment, that will be considered later.

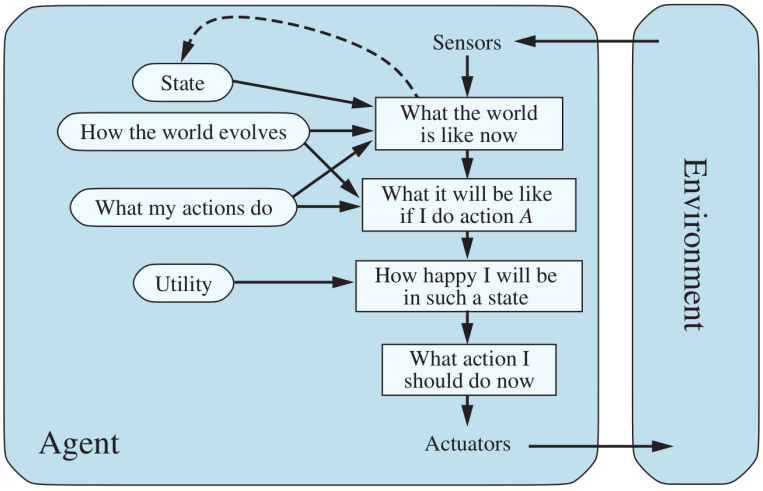


Figure 3.8: Utility-based agent

3.8 Rationality and agent learning automatically

It is easy to automate the learning of rule table, transition model and sensor model, goals and utility functions, if the agent's knowledge system is split from the decision making system. Such a decoupled agent is called learning agent. Learning agent has two decoupled parts,

- **learning element**, which is responsible for improvement in knowledge components,
- **performance element**, which is responsible for selecting actions.

Apart from decoupling, two new components can be added to assist in automatic learning.

- **critic**, compares percepts with a fixed performance standard and generates a feedback based on its idea of good or bad percepts,
- **problem generator**, suggests new scenarios that lead to new informative experiences.

Learning agent

Learning agent decouples the knolwedge component from the action component such that with the help of performance standards the knolwedge component may be updated. ♣

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 use to do this once it has learned how?" Given a design for the performance element, learning mechanisms can be constructed to improve every part of the agent.

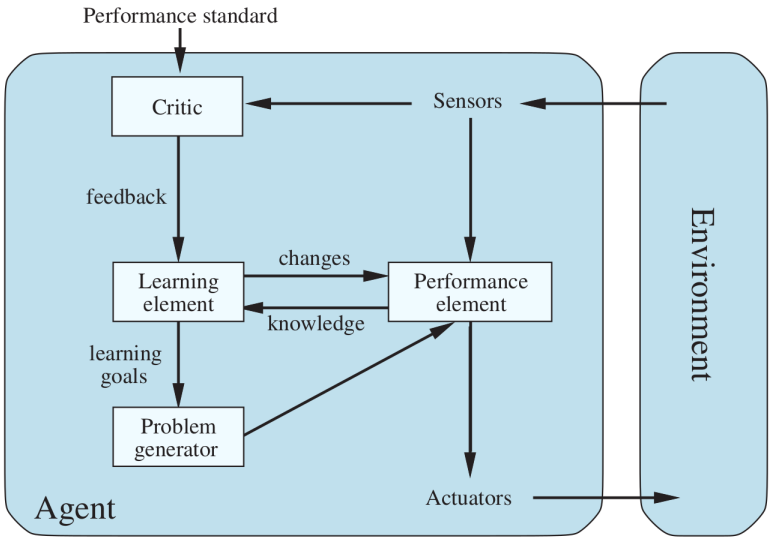


Figure 3.9: Learning agent

The learning takes place when the sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a performance measure that evaluates any given sequence of environment states. An agent that maximizes its performance measure by selecting appropriate actions against each possible percept based on the percept sequence and overall built-in knowledge is called a rational agent. Rationality depends on four things,

- prior knowledge of the environment
- current percept sequence
- list of available actions
- performance measure

3.9 Environment

Task environment

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

We can specify a task environment by a set of descriptios of its Performance, Environment, Actuators and Sensors also called PEAS description. Task environments come in a variety of flavors. The nature of the task environment directly affects the appropriate design for the agent program.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors

Environments can be categorized based on several dimensions. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation. Here's a list of dimensions,

Property	**Values**	**Description**
Observable	fully	Sensors capture all aspects of environment relevant to the agent
	partially	Several aspects of the environment are relevant but can not be known
Population	single-agent	Agent is working alone
	multiagent	There are other agents with whom agent can compete or cooperate
Model	deterministic	Current state and action of agent are enough to determine the next state
	stochastic	Current state and action of agent do not guarantee what state will be the next
Events	episodic	Events and actions in one time frame do not affect the other
	sequential	Events and actions in current time frame can affect all future decisions
Changeability	static	Agent does not need to look at the world while making decision
	dynamic	Even while the agent is thinking, the environment can change
Smoothness	discrete	States are discrete and finite
	continuous	States can have any of the infinite continuous values
Knowledge	known	Agent has complete information about the environment
	unknown	Some information about the environment is not known and has to be learned

The table below lists some examples of task environments along with their classification.

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
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

Section 4 Searching

Prelude

- ❏ *Problem solving agents*
- ❏ *Searching for solutions*
- ❏ *Uninformed search*
- ❏ *Informed search*
- ❏ *Variations of basic search*
- ❏ *Comparison of search algorithms*

Problem solving agent

It has the ability to plan ahead, mainly in the situations where correct action to take is not immediately obvious.

The process of planning is called **search**. In special case, when state is not atomic (like factored or structured), it is called **planning agent**. Problem is solved using four steps.

1. Goal formation: Deciding goal like to reach Bucharest from Arad.

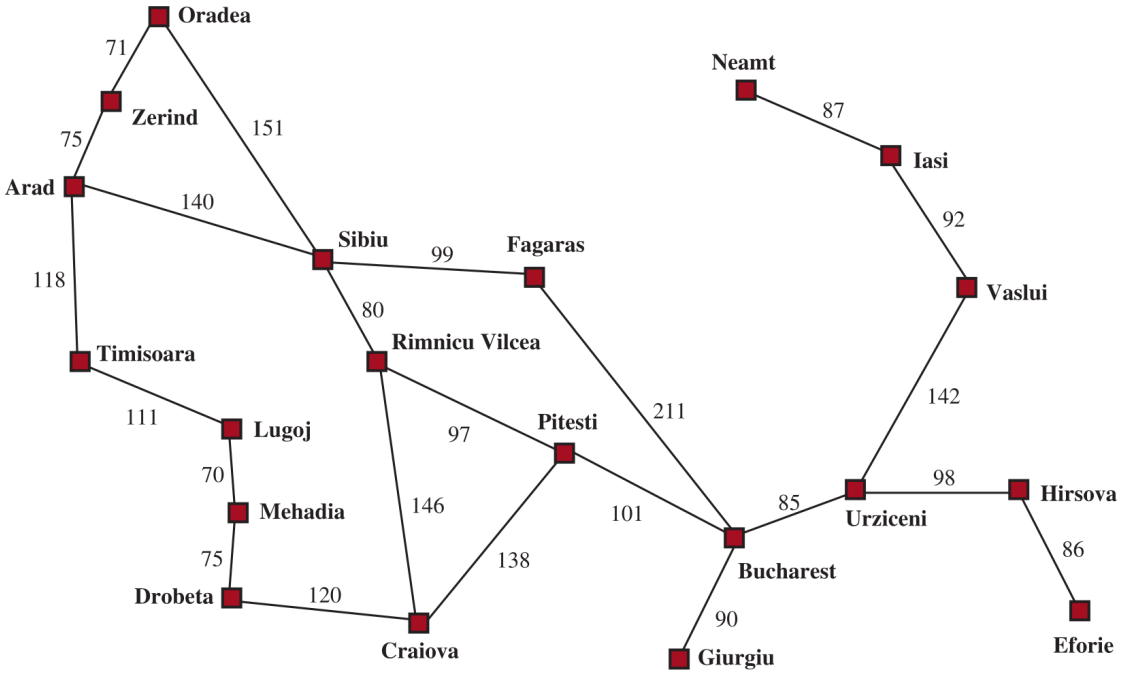


Figure 4.1: State-space for routing problem

2. Problem formulation: Describing states and actions, like traveling from one city to the adjacent city.
3. Search: Search for a sequence of actions that lead to goal state. Such a sequence is called solution, like traveling from Arad to Bucharest via Sibiu and Fagaras.
4. Execution: Execute the solution.

4.1 Search problem

A search problem consists of

1. **State space**: which is a set of possible states. For a traveling route problem each state includes a location (e.g., an airport) and the current time.
2. **Initial state**: like starting from Arad at a specific time.
3. **Set of goal states**: like a destination city, say Bucharest, and time frame to reach the city.
4. **Actions available** from each state: like flights from Zerind are {toArad, toOradea}. Therefore the action from current city is to take any flight from the current location at certain time.

   States and actions can be represented by graph where states are vertices and actions form the edges as in the map of Romania above.
5. **Transition model**: describes results of each action. The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time. For example, $(Arad|8\ a.m, toZerind) \rightarrow Zerind|8:45\ a.m$.
6. **Action cost function**: the cost of taking an action, like distance between Arad and Sibiu. It also involves a combination of monetary cost, waiting time and flight time. A solution with minimum overall cost is called optimum solution.

4.1.1 Example of vacuum world

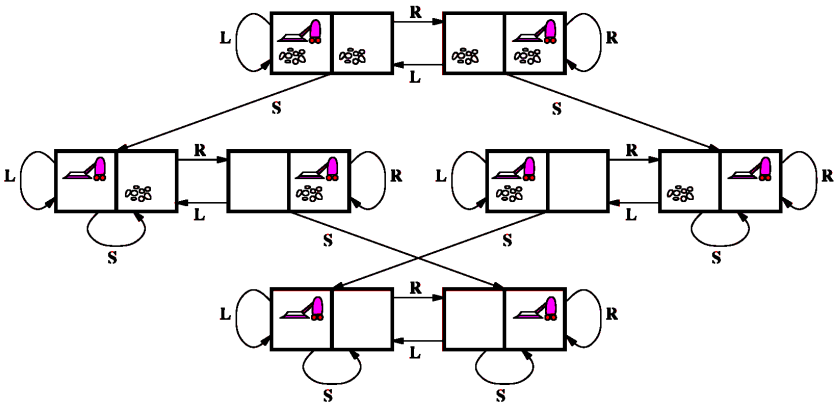


Figure 4.2: State-space of vacuum world

State space: location, dirt A, dirt B

A state of the world says which objects are in which cells. For the vacuum world, 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, so there are $2 \cdot 2 \cdot 2 = 8$ states.

Initial state: Any state can be designated as the initial state.

Actions: left, right, suck

In the two-cell world we defined three actions: Suck, move Left, and move Right. In a two-dimensional multi-cell world we need more movement actions. We could add Upward and Downward, giving us four absolute movement actions, or we could switch to egocentric actions, defined relative to the viewpoint of the agent—for example, Forward, Backward, TurnRight, and TurnLeft.

Transition model: As show in graph above.

Suck removes any dirt from the agent's cell; Forward moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. Backward moves the agent in the opposite direction, while TurnRight and TurnLeft change the direction it is facing by 90 degrees.

Goal state: State in which every cell is clean.

Action cost: Each action costs 1 unit.

4.1.2 Example of pacman

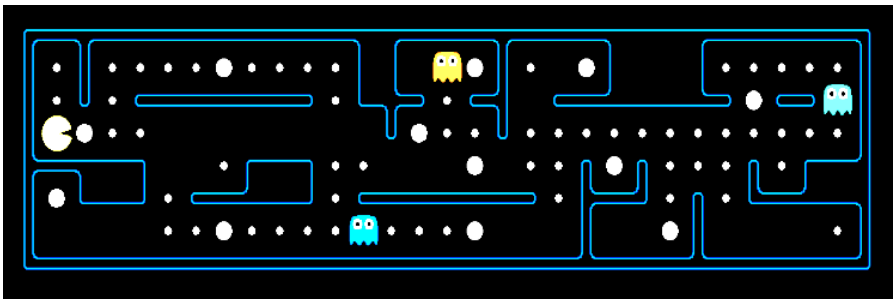


Figure 4.3: Pacman game

The states consist of possible positions and list of dots. Initial state can be any scenario in Pacman game. Goal state is reached when list of dots is empty. Available actions are N, E, S, W. Transition model updates location and may be reduces a dot when eaten. Action cost may be a score that increases when dots are eaten but diminishes gradually with passage of time.

4.2 General search algorithm

Search algorithm

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

♣

When we begin searching a search tree is formed automatically from the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal

state.

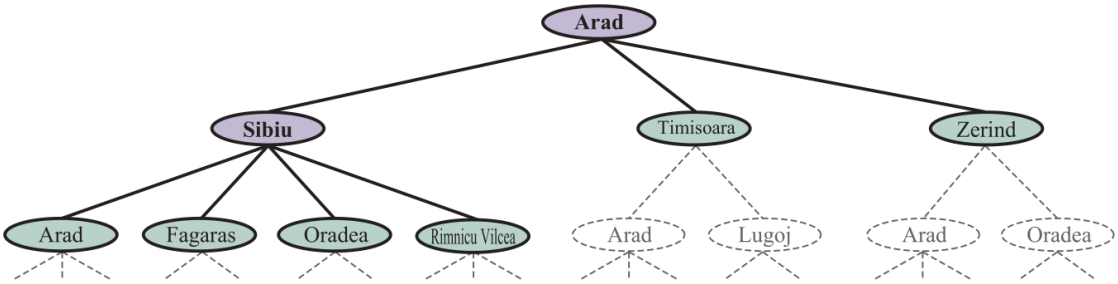


Figure 4.4: Search tree

Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem. The set of states that have been reached (discovered) but not yet expanded (processed) is called frontier.

Best-first search

Best-first search

It uses a cost function and extracts the node with minimum cost from the frontier.

function BEST-FIRST-SEARCH(*problem*, *f*) **returns** a solution node or *failure*
  node ← NODE(STATE=*problem*.INITIAL)
  frontier ← a priority queue ordered by *f*, with *node* as an element
  reached ← a lookup table, with one entry with key *problem*.INITIAL and value *node*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if *s* is not in *reached* **or** *child*.PATH-COST < *reached*[*s*].PATH-COST **then**
        reached[*s*] ← *child*
        add *child* to *frontier*
  return *failure*

function EXPAND(*problem*, *node*) **yields** nodes
  s ← *node*.STATE
  for each *action* **in** *problem*.ACTIONS(*s*) **do**
    s′ ← *problem*.RESULT(*s*, *action*)
    cost ← *node*.PATH-COST + *problem*.ACTION-COST(*s*, *action*, *s*′)
    yield NODE(STATE=*s*′, PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

Figure 4.5: Sample algorithm from textbook.

In each iteration we choose a node *n* on the frontier with minimum cost *d*[*n*], return it if its state is a goal state, and otherwise follow each possible action to generate successor nodes (child nodes). 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. The algorithm returns either an indication of failure, or a node that represents a path to a goal.

```
bestFirstSearch(p, f) //p is problem and f is priority function
  D[p. init ] = {∅ , 0} // D is dictionary and init is initial state
  Q = {#0, p. init , f(p. init )} // Q is priority queue
  while Q≠ ∅
    n = Q.pop()
    if p.isGoal(n) then return success
    for c ∈ p.children(n)
      cost = D[n].cost + p.actionCost(n, c)
      if c ∉ D or cost < D[c].cost
        D[c] = {n, cost}
        Q.push({#nextId, c, f(c)})
  return ∅
```

4.3 Uninformed search strategies

If agent has no idea about the distance to the goal, then the search is uninformed.

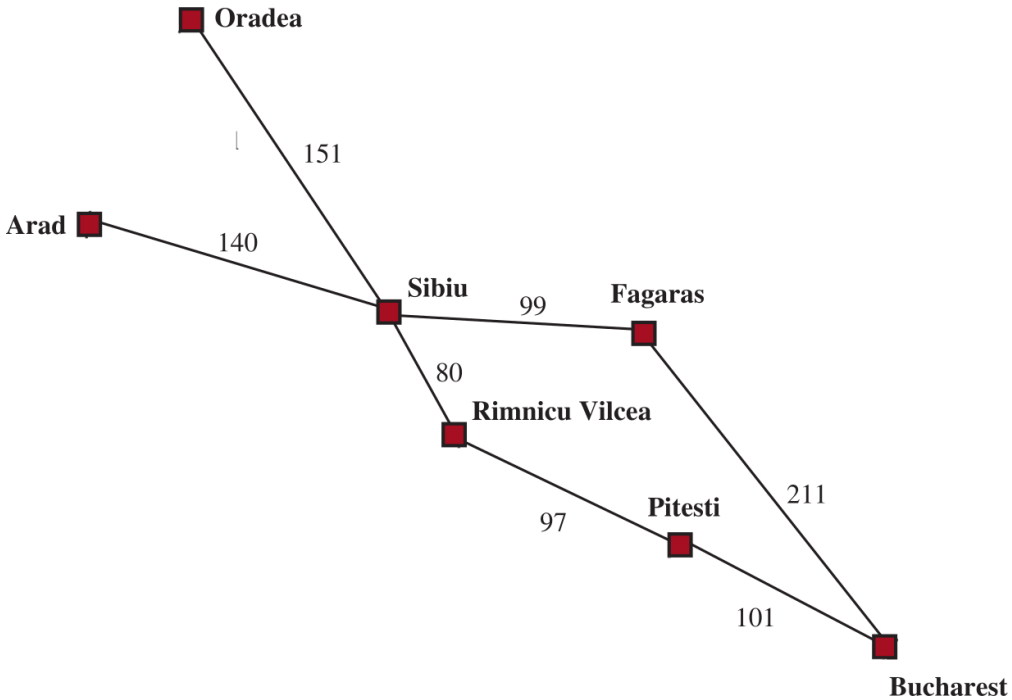


Figure 4.6: Minimap

4.3.1 Breadth-first search

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, then their successors, and so on. Breadth-first search is best-first search where the evaluation function f(n) is the depth of the node.

Smart structure	**Priority queue**
Arad () 0	#0 Arad 0
Processing #0 Arad 0	
Arad () 0	
Sibiu (Arad) 1	#1 Sibiu 1
Processing #1 Sibiu 1	
Arad () 0	
Sibiu (Arad) 1	
Fagaras (Sibiu) 2	#2 Fagaras 2
Oradea (Sibiu) 2	#3 Oradea 2
Rimnicu-Vilcea (Sibiu) 2	#4 Rimnicu-Vilcea 2
Processing #2 Fagaras 2	
Arad () 0	
Sibiu (Arad) 1	
Fagaras (Sibiu) 2	
Oradea (Sibiu) 2	#3 Oradea 2
Rimnicu-Vilcea (Sibiu) 2	#4 Rimnicu-Vilcea 2
Bucharest (Fagaras) 3	#5 Bucharest 3
Processing #3 Oradea 2	
Arad () 0	
Sibiu (Arad) 1	
Fagaras (Sibiu) 2	
Oradea (Sibiu) 2	
Rimnicu-Vilcea (Sibiu) 2	#4 Rimnicu-Vilcea 2
Bucharest (Fagaras) 3	#5 Bucharest 3
Processing #4 Rimnicu-Vilcea 2	
Arad () 0	
Sibiu (Arad) 1	
Fagaras (Sibiu) 2	
Oradea (Sibiu) 2	
Rimnicu-Vilcea (Sibiu) 2	
Bucharest (Fagaras) 3	#5 Bucharest 3
Pitesti (Rimnicu-Vilcea) 3	#6 Pitesti 3
Processing #5 Bucharest 3	
Goal reached in 6 iterations: Arad Sibiu Fagaras Bucharest	

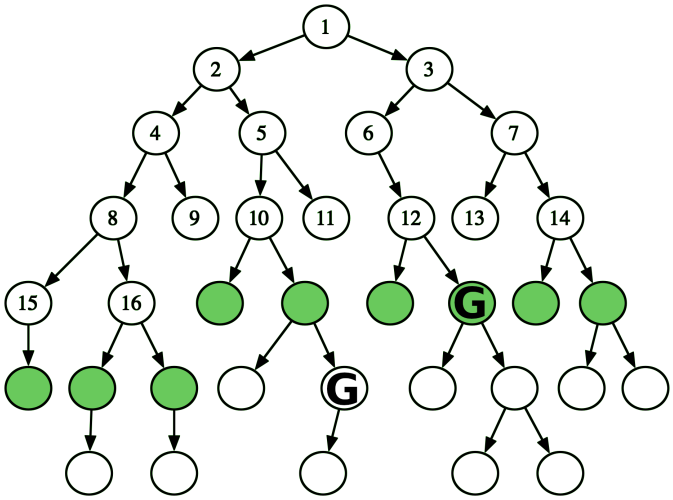


Figure 4.7: BFS

Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth 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.

4.3.2 Depth-first search

Depth-first search always expands the deepest node in the frontier first. It could be implemented as a call to best-first search where the evaluation function f is the negative of the depth. 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.

Smart structure	**Priority queue**
Arad () 0	#0 Arad 0
Processing #0 Arad 0	
Arad () 0	
Sibiu (Arad) -1	#1 Sibiu -1
Processing #1 Sibiu -1	
Arad () 0	
Sibiu (Arad) -1	
Fagaras (Sibiu) -2	#2 Fagaras -2
Oradea (Sibiu) -2	#3 Oradea -2
Rimnicu-Vilcea (Sibiu) -2	#4 Rimnicu-Vilcea -2
Processing #4 Rimnicu-Vilcea -2	
Arad () 0	
Sibiu (Arad) -1	
Fagaras (Sibiu) -2	

Oradea (Sibiu) -2	#2 Fagaras -2
Rimnicu-Vilcea (Sibiu) -2	#3 Oradea -2
Pitesti (Rimnicu-Vilcea) -3	#5 Pitesti -3
Processing #5 Pitesti -3	
Arad () 0	
Sibiu (Arad) -1	
Fagaras (Sibiu) -2	
Oradea (Sibiu) -2	
Rimnicu-Vilcea (Sibiu) -2	#2 Fagaras -2
Pitesti (Rimnicu-Vilcea) -3	#3 Oradea -2
Bucharest (Pitesti) -4	#6 Bucharest -4
Processing #6 Bucharest -4	
Goal reached in 5 iterations: Arad Sibiu Rimnicu-Vilcea Pitesti Bucharest	

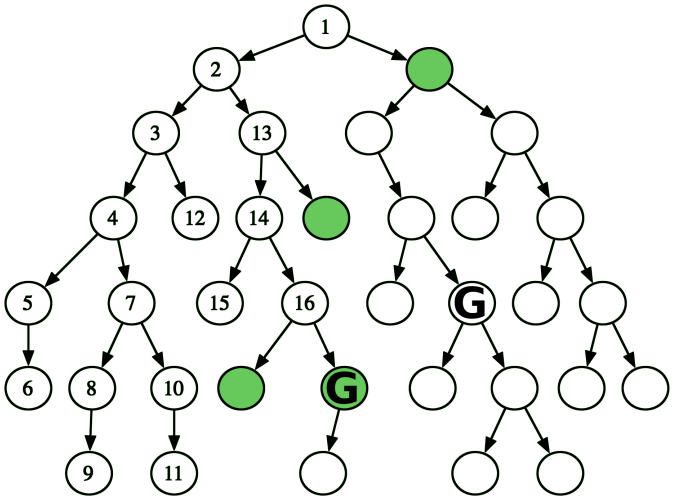


Figure 4.8: DFS

Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.

4.3.3 Uniform-cost search

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's algorithm or uniform-cost search. Uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to best-first search with path cost as the evaluation function.

Smart structure	**Priority queue**
Arad () 0	#0 Arad 0

Processing #0 Arad 0	
Arad () 0	
Sibiu (Arad) 140	#1 Sibiu 140
Processing #1 Sibiu 140	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	#2 Fagaras 239
Oradea (Sibiu) 291	#3 Oradea 291
Rimnicu-Vilcea (Sibiu) 220	#4 Rimnicu-Vilcea 220
Processing #4 Rimnicu-Vilcea 220	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	#2 Fagaras 239
Rimnicu-Vilcea (Sibiu) 220	#3 Oradea 291
Pitesti (Rimnicu-Vilcea) 317	#5 Pitesti 317
Processing #2 Fagaras 239	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	
Rimnicu-Vilcea (Sibiu) 220	#3 Oradea 291
Pitesti (Rimnicu-Vilcea) 317	#5 Pitesti 317
Bucharest (Fagaras) 450	#6 Bucharest 450
Processing #3 Oradea 291	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	
Rimnicu-Vilcea (Sibiu) 220	
Pitesti (Rimnicu-Vilcea) 317	#5 Pitesti 317
Bucharest (Fagaras) 450	#6 Bucharest 450
Processing #5 Pitesti 317	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	

Rimnicu-Vilcea (Sibiu) 220	
Pitesti (Rimnicu-Vilcea) 317	#6 Bucharest 450
Bucharest (Pitesti) 418	#7 Bucharest 418

Processing #7 Bucharest 418

Goal reached in 7 iterations: Arad Sibiu Rimnicu-Vilcea Pitesti Bucharest

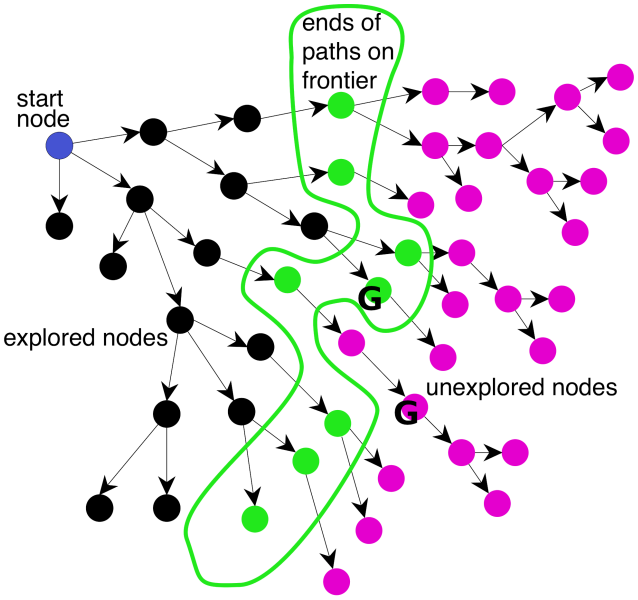


Figure 4.9: UCS

Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path.

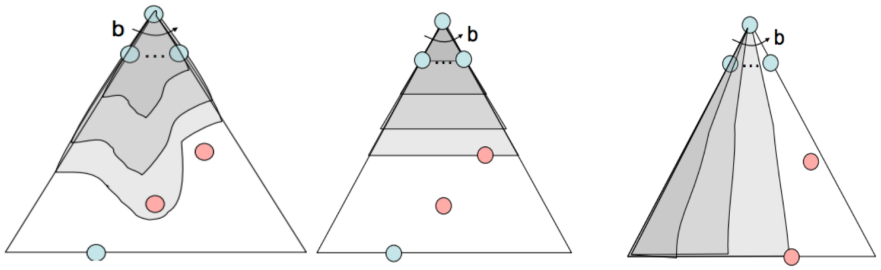


Figure 4.10: Identify the search algorithms

4.4 Informed search strategies

If it is possible for agent to estimate the distance to the goal, the agent performs an informed search. 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. Such a hint is called heuristic function, denoted $h(n)$.

Heuristic

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

In the map below the heuristic which is straight-line distance of a city to Bucharest is given next to the city.

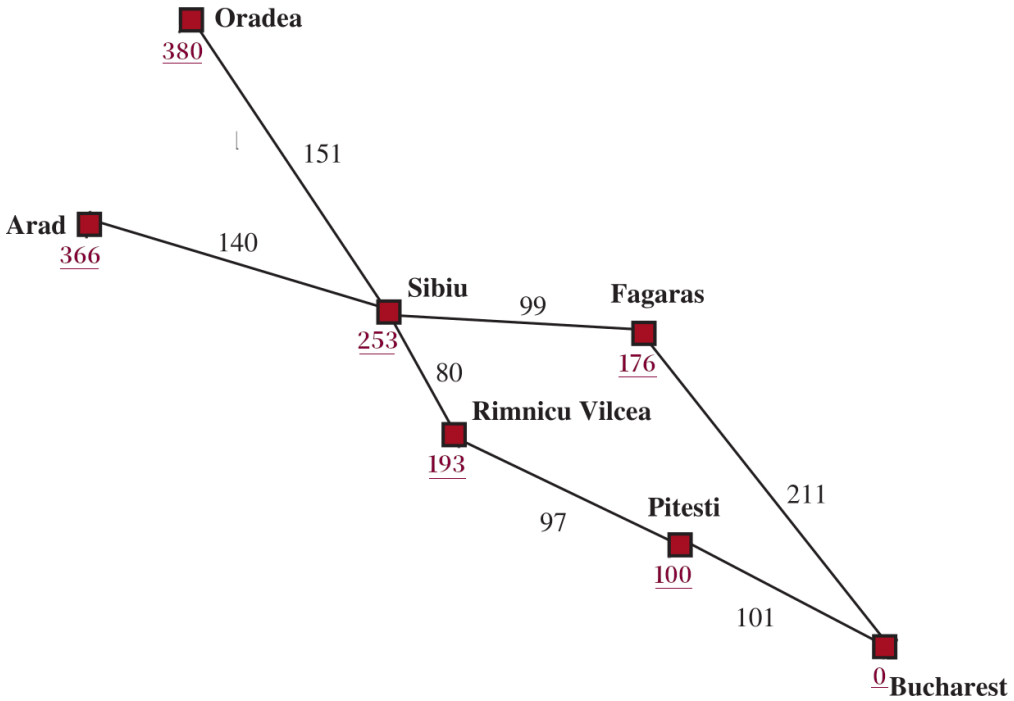


Figure 4.11: Informed minimap

4.4.1 A* search

The most common informed search algorithm is *A** search.

A* search

It is a best-first search algorithm that uses the priority function,

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

where g(*n*) *is the cost from start state to current state and h*(*n*) *is the heuristic.*

Smart structure	Priority queue
Arad () 0	#0 Arad 366
Processing #0 Arad 366 Arad () 0 Sibiu (Arad) 140	#1 Sibiu 393
Processing #1 Sibiu 393 Arad () 0 Sibiu (Arad) 140 Fagaras (Sibiu) 239	#2 Fagaras 415

Oradea (Sibiu) 291	#3 Oradea 671
Rimnicu-Vilcea (Sibiu) 220	#4 Rimnicu-Vilcea 413
Processing #4 Rimnicu-Vilcea 413	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	#2 Fagaras 415
Rimnicu-Vilcea (Sibiu) 220	#3 Oradea 671
Pitesti (Rimnicu-Vilcea) 317	#5 Pitesti 417
Processing #2 Fagaras 415	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	
Rimnicu-Vilcea (Sibiu) 220	#3 Oradea 671
Pitesti (Rimnicu-Vilcea) 317	#5 Pitesti 417
Bucharest (Fagaras) 450	#6 Bucharest 450
Processing #5 Pitesti 417	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	
Rimnicu-Vilcea (Sibiu) 220	#3 Oradea 671
Pitesti (Rimnicu-Vilcea) 317	#6 Bucharest 450
Bucharest (Pitesti) 418	#7 Bucharest 418
Processing #7 Bucharest 418	
Goal reached in 6 iterations: Arad Sibiu Rimnicu-Vilcea Pitesti Bucharest	

Properties of a heuristic function

First of all a heuristic should be inexpensive . The heuristic should be much easier to compute than solving for the exact remaining distance.

Whether A^* is cost-optimal depends on certain properties of the heuristic. A key property is admissibility: an admissible heuristic is one that never overestimates the cost to reach a goal. (An admissible heuristic is therefore optimistic.)

function

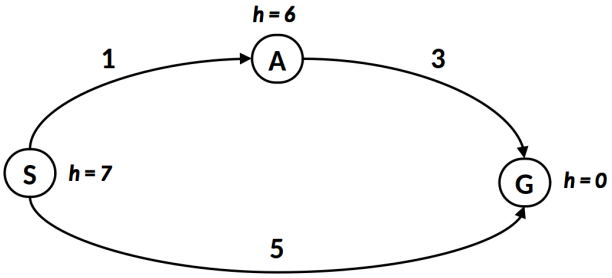


Figure 4.12: Admissible?

A heuristic may even have the stronger property of consistency, which means that it satisfies the triangle inequality.

$$h(a) \leq cost(a, b) + h(b)$$

Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, $A^\star$ is cost-optimal. With an inconsistent heuristic, we may end up with multiple paths reaching the same state, and if each new path has a lower path cost than the previous one, then we will end up with multiple nodes for that state in the frontier, costing us both time and space.

Search contours

A useful way to visualize a search is to draw contours in the state space, just like the contours in a topographic map. With uniform-cost search, we also have contours, but of g-cost, not $g + h$. The contours with uniform-cost search will be ”circular” around the start state, spreading out equally in all directions with no preference towards the goal. With $A^\star$ search using a good heuristic, the $g + h$ bands will stretch toward a goal state and become more narrowly focused around an optimal path.

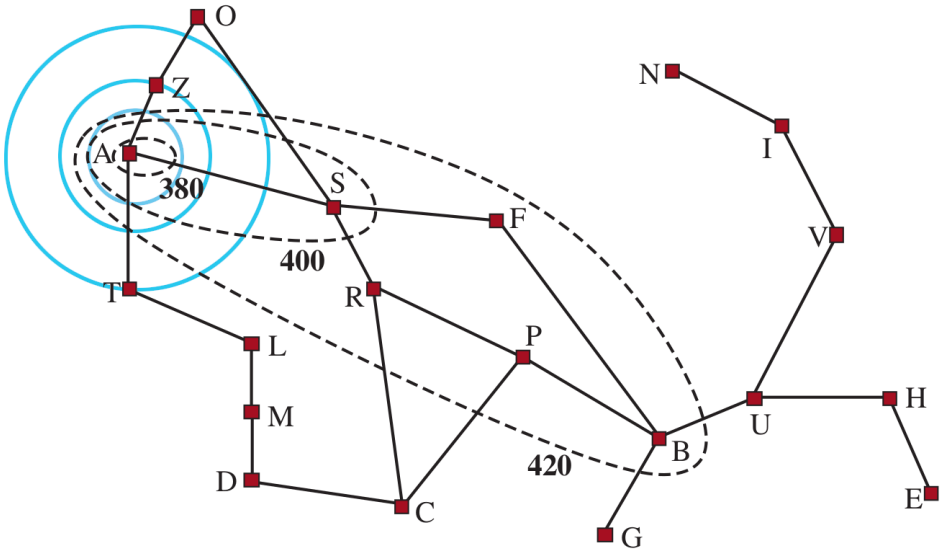


Figure 4.13: Search contours

4.4.2 Greedy search

Greedy best-first search is a form of best-first search that expands first the node with the lowest h(n) value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the priority function $f(n) = h(n)$.

Smart structure	**Priority queue**
Arad () 0	#0 Arad 366
Processing #0 Arad 366	
Arad () 0	
Sibiu (Arad) 140	#1 Sibiu 253
Processing #1 Sibiu 253	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	#2 Fagaras 176
Oradea (Sibiu) 291	#3 Oradea 380
Rimnicu-Vilcea (Sibiu) 220	#4 Rimnicu-Vilcea 193
Processing #2 Fagaras 176	
Arad () 0	
Sibiu (Arad) 140	
Fagaras (Sibiu) 239	
Oradea (Sibiu) 291	#3 Oradea 380
Rimnicu-Vilcea (Sibiu) 220	#4 Rimnicu-Vilcea 193
Bucharest (Fagaras) 450	#5 Bucharest 0
Processing #5 Bucharest 0	
Goal reached in 4 iterations: Arad Sibiu Fagaras Bucharest	

The algorithm is called greedy because on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful. With a good heuristic function, the complexity of such functions can be reduces substantially.

4.5 Discovering good heuristic functions

4.5.1 from relaxed problem

A problem with fewer restrictions on the actions is called a relaxed problem. The state-space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph. Because the relaxed problem adds edges to the state-space graph, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have better solutions if the

added edges provide shortcuts. Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

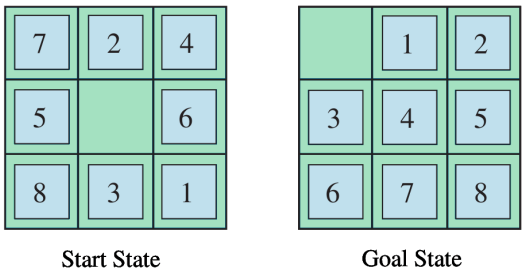


Figure 4.14: Eight puzzle that requires 26 moves to solve

For 8-puzzle, following two heuristics are easy to compute.

1. h_1, number of misplaced tiles (not including blank), which is 8 for the puzzle above.
2. h_2, sum of Manhattan distances of tiles from their goal positions, which is 18 for the puzzle above.

In what way do h_1 and h_2 correspond to relaxed problems?

4.5.2 from subproblems

Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem. The idea behind pattern databases is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank.

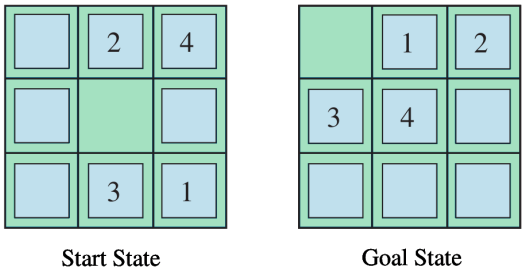


Figure 4.15: Solving a pattern

4.5.3 from landmarks

We could generate a perfect heuristic by precomputing and storing the cost of the optimal path between every vertex to a set of few specially selected vertices called landmarks. Although the precomputation can be time-consuming, it need only be done once. Then an efficient and admissible heuristic will be the landmark who has maximum absolute difference between its optimal distance from current vertex and its optimal diance from the goal.

$$h(n) = \max_{L \in landmarks} |C^{\star}(n, L) - C^{\star}(goal, L)| \tag{4.1}$$

4.6 Variations of basic search

The main issue with basic search techniques is use of memory. Memory is split between the frontier and the reached states. In our implementation of best-first search, a state that is on the frontier is stored in two places: as a node in the frontier (priority queue) and as an entry in the table of reached states (dictionary). Here are some implementation tricks that save space, and then some entirely new algorithms that take better advantage of the available space.

While the number of nodes in the priority queue of breadth-first search increases exponentially as b^d where b is the branching factor and d is the depth, the number of nodes in the queue for depth-first search is only bd.

4.6.1 Depth-limited search

To keep depth-first search from wandering down an infinite path, we can use depth-limited search, a version of depth-first search in which we supply a depth limit, m, and treat all nodes at depth m as if they had no successors. Memory is saved further by not using any dictionary. The time complexity is $O(b^m)$ and the space complexity is $O(bm)$. Unfortunately, if we make a poor choice for m the algorithm will fail to reach the solution, making it incomplete.

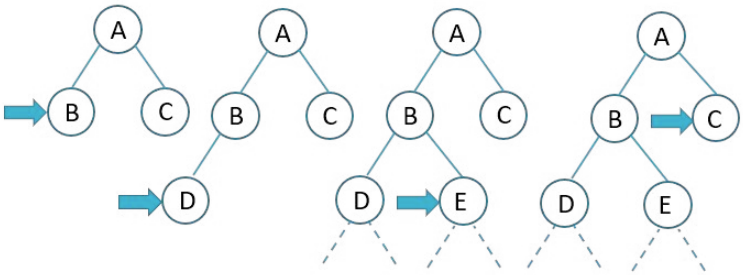


Figure 4.16: Depth-limited search

```
depthLimitedSearch(p, m)
  Q = {#0, p. init }   // Q is LIFO queue
  result = ∅
  while Q≠ ∅
    n = Q.pop()
    if p.isGoal(n) then return success
    if p.depth(n)> m then result = cutoff
    else if not p. cyclic (n)
      for c ∈ p.children(n)
        Q.push({#nextId, c})
```

```
  return result
```

4.6.2 Iterative deepening search

Iterative deepening search solves the problem of picking a good value for m 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 value rather than the cutoff value. Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: $O(bd)$ when there is a solution at depth d. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces.

```
iterativeDeepeningSearch(p)
  for depth = 0 to ∞
    result = depthLimitedSearch(p, depth)
    if result ≠ cutoff then return result
```

4.6.3 Bi-directional search

An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d. We need to keep track of two priority queues and two dictionaries. We have a solution when the two frontiers collide, that is, a state is successor state in both forward and backward direction.

4.7 Comparison

For comparison between various search algorithms we will consider the following criteria based on depth and branching factor.

1. Completeness: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
2. Cost optimality: Does it find a solution with the lowest path cost of all solutions?
3. Time complexity: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
4. Space complexity: How much memory is needed to perform the search?

In the table below, d is depth of shallowest solution; m is maximum depth of search tree or maximum depth allowed; $C^\star$ is optimum cost and ϵ is minimum value of improvement per iteraton.

Search Algorithm	Complete and optimal	Time complexity	Space complexity
Breadth-first search	Yes	$O(b^d)$	
Bi-directional search		$O(b^{d/2})$	
Uniform-cost search		$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	
Iterative deepening search		$O(b^d)$	$O(bd)$
Depth-first search	No	$O(b^m)$	$O(bm)$
Depth-limited search			

Section 5 Local Search Algorithms and Optimization Problems

Prelude

- *Hill-climbing search*

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.)

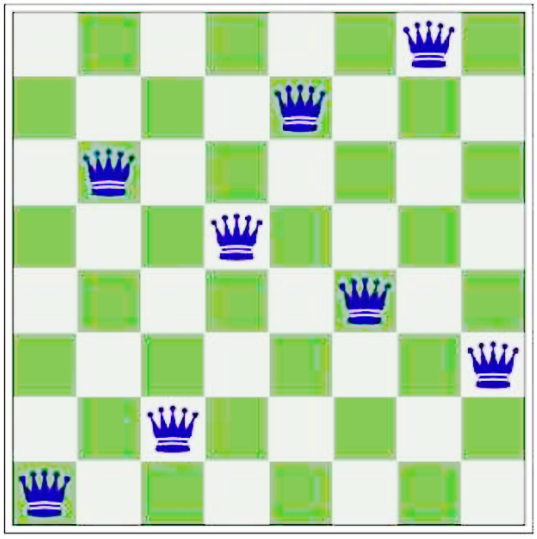


Figure 5.1: 8-queens, almost

This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. Let us define a heuristic function that counts number of attacking pairs.

$h(n)$ = number of attacking pairs in n

Figure 5.2: One-move heuristic table for a state with h=17

How many best options are availabe in the image? If we want to convert this to a maximization problem, what will be the function to be maximixed? What will be the maximum value of that function?

$$f(n) = 28 - h(n)$$

or simply,

$$f(n) = -h(n)$$

Such a function that needs to be maximized is called objective function.

< *Demo* >

5.1 Hill-climbing search

Hill-climbing search

It is a search algorithm that keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the steepest ascent. It terminates when it reaches a peak.

```
hillClimbing (p)          // function  maximization problem
  current = p. initial
  while true
    neighbor = max(current.successors)
    if  neighbor.value ≤ current.value
      return current
    current = neighbor
```

Let us say we start from this configuration and by moving each queen one-by-one we solve the 8 queen problem. Does the solution path from current state to the solution matter? (The solution path mattered in routing problem from Arad to Bucharest.)

Optimization problem

Such problems were the objective is to find the best state that has the maximum value of the objective function are called optimization problems. Here only discovery of the best state matters; the path used to reach from the starting state to best state does not matter.

♣

Algorithms that operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached are called local search algorithms.

5.1.1 Obstacles

Unfortunately, hill climbing can get stuck for any of the following reasons:

1. **Local maxima**: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
2. **Ridges**: A ridge is shown in Figure 5.3. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

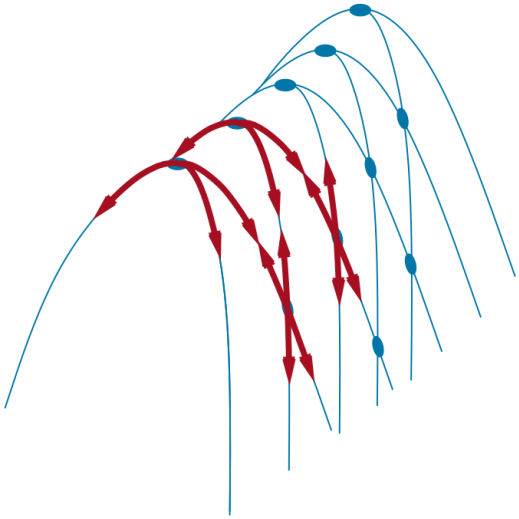


Figure 5.3: Ridge

This is an illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in

low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

3. **Plateaus**: A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search can get lost wandering on the plateau.

In one dimension, such spread of heuristic function like on the chess board can be visualized as state-space landscape, which is "elevation" plot of objective function to be maximized.

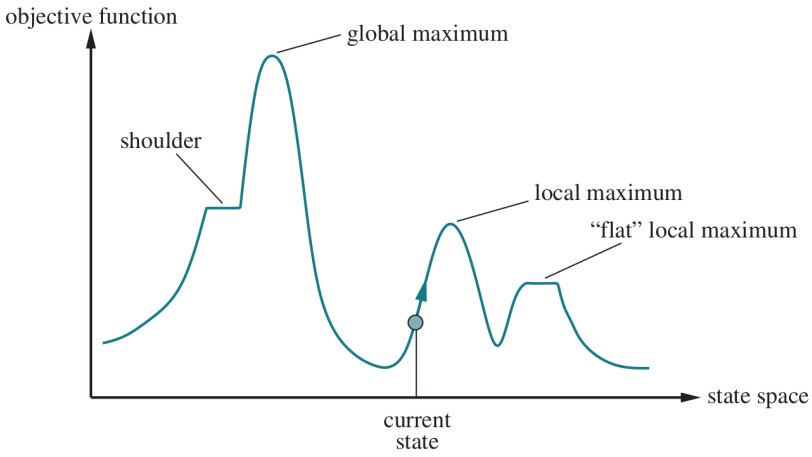


Figure 5.4: State-space landscape

5.1.2 Variations

Many variants of hill climbing have been invented.

1. Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
2. First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.
3. Another variant is random-restart hill climbing, which adopts the adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is complete with probability 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$. For 8-queens instances with no sideways moves allowed, p $\approx$ 0.14, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1 - p)/p$ times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, 1/0.94 $\approx$ 1.06 iterations are

needed on average and $(1 \times 21) + (0.06/0.94) \times 64 \approx 25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in few seconds.

5.1.3 Requirements

The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaus, random-restart hill climbing will find a good solution very quickly. On the other hand, many real problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle. NP-hard problems typically have an exponential number of local maxima to get stuck on. Despite this, a reason- ably good local maximum can often be found after a small number of restarts.

5.2 Simulated annealing

- A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum.
- In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum, but will be extremely inefficient.
- Therefore, it seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing

A powerful enhancement for hill climbing search in which there is a small probability that random moves will be allowed in directions different from the preferred up-hill direction. Initially large moves are allowed but a "cooling" schedule continuously lowers the probability of random directions during the search and thus the solution tends to settle on the best local maximum.

♣

Concept of temperature

The temperature is a parameter in simulated annealing that affects two aspects of the algorithm:

1. The distance of a trial point from the current point.
2. The probability of accepting a trial point with higher objective function value.

Temperature decreases gradually as the algorithm proceeds. The slower the rate of temperature decrease, the better the chances are of finding an optimal solution, but the longer the run time.

The overall structure of the simulated-annealing algorithm is similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.

```
simulatedAnnealing(p, sch)        // sch is cooling schedule
  current = p. initial
  for t = 1 to ∞
    T = sch[t]
    if T == 0
      return current
    neighbor = randomChoice(current.successors)
    ΔE = neighbor.value − current.value
    if ΔE > 0
      current = neighbor
    else if random() < e^(ΔE/T)
      current = neighbor
```

5.3 Local beam search

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

	Local beam search	**Random-restart search**
1.	Runs with k states in parallel.	Runs with k states sequentially.
2.	Useful information is passed among the parallel threads.	Each process runs independently of others.
3.	The algorithm abandons unfruitful searches and moves its resources to where the most progress is being made.	The idea is that some process may find a suitable goal.
4.	Diversity is eventually lost.	Diversity is preserved.

Local bean search may suffer from lack of diversity. A variant called stochastic beam

search, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the top k successors, stochastic beam search chooses successors with probability proportional to the successor's value, thus increasing diversity.