

R.M.K **GROUP OF** **ENGINEERING** **INSTITUTIONS**



R.M.K
GROUP OF
INSTITUTIONS

R.M.K GROUP OF INSTITUTIONS



R.M.K
GROUP OF
INSTITUTIONS



Please read this disclaimer before

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.

21AI401 ARTIFICIAL INTELLIGENCE

Department: C.S.E

Batch/Year:2021-2025/III

Created by: Dr . M. Arun Manicka Raja

Ms.V. Kalaipriya

Date: 15.08.2023

1.TABLE OF CONTENTS

S.NO.	CONTENTS	SLIDE NO.
1	CONTENTS	5
2	COURSE OBJECTIVES	7
3	PRE REQUISITES (COURSE NAMES WITH CODE)	8
4	SYLLABUS (WITH SUBJECT CODE, NAME, LTPC DETAILS)	9
5	COURSE OUTCOMES	10
6	CO- PO/PSO MAPPING	11
7	LECTURE PLAN –UNIT 2	13
8	ACTIVITY BASED LEARNING –UNIT 2	15
9	LECTURE NOTES – UNIT 2	16
10	ASSIGNMENT 2- UNIT 2	64
11	PART A Q & A (WITH K LEVEL AND CO) UNIT 2	65
12	PART B Q s (WITH K LEVEL AND CO) UNIT 2	72
13	SUPPORTIVE ONLINE CERTIFICATION COURSES UNIT 2	73
14	REAL TIME APPLICATIONS IN DAY TO DAY LIFE AND TO INDUSTRY UNIT 2	74

S.NO.	CONTENTS	SLIDE NO.
15	ASSESSMENT SCHEDULE	76
16	PRESCRIBED TEXT BOOKS & REFERENCE BOOKS	77
17	MINI PROJECT SUGGESTIONS	78



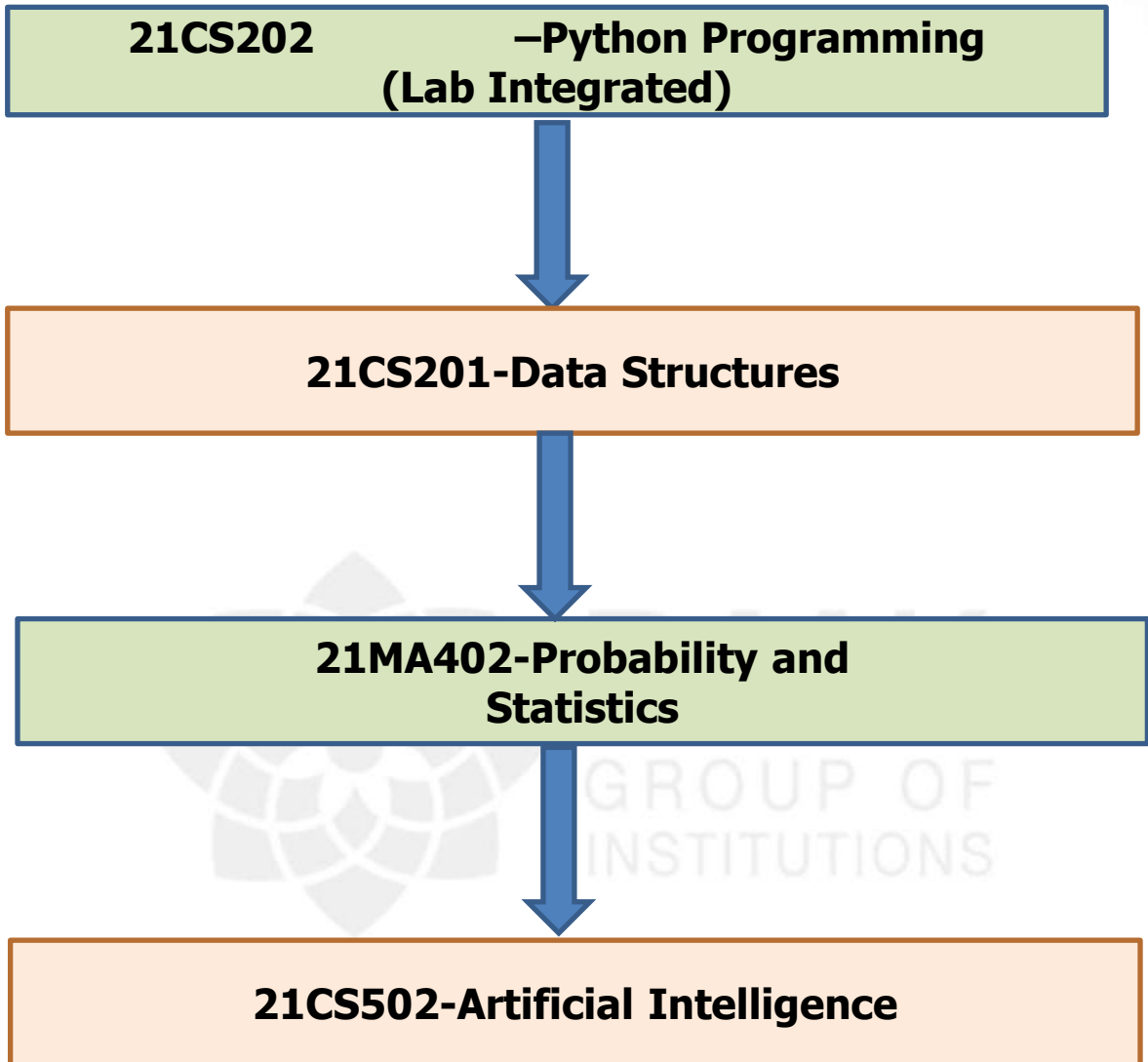
2. COURSE OBJECTIVES

- ❖ To Explain the Foundations of AI and various intelligent agents
- ❖ To discuss problem solving search strategies and game playing
- ❖ To describe logical agents and first-order logic
- ❖ To illustrate problem-solving strategies with knowledge representation mechanism for hard problems
- ❖ To Explain the basics of learning and expert systems



3. PRE REQUISITES

PRE-REQUISITE CHART



4.SYLLABUS

21AI401-ARTIFICIAL INTELLIGENCE

L T P C

3 0 0 3

Unit-I ARTIFICIAL INTELLIGENCE AND INTELLIGENT AGENTS 9

Introduction to AI–Foundations of Artificial Intelligence–Intelligent Agents–Agents and Environment–Concept of rationality – Nature of environments – Structure of agents – Problem Solving Agents–Example Problems – Search Algorithms – Uninformed Search Strategies

Unit II : PROBLEM SOLVING 9

Heuristics Search Strategies – Heuristic Functions - Game Play ing – Mini Max Algorithm- Optimal Decisions in Games – Alpha - Beta Search – Monte Carlo Search for Games - Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search for CSP- Local Search for CSP-Structure of CSP

Unit III : LOGICAL AGENTS 9

Knowledge Based Agents-Logic-Propositional logic- Propositional theorem proving- Propositional model Checking- Agents based on propositional Logic-First Order Logic – Propositional Vs First Order Inference – Unification and First Order Inference – Forward Chaining- Backward Chaining – Resolution

Unit IV : KNOWLEDGE REPRESENTATION AND PLANNING 9

Ontological Engineering -Categories and Objects – Events - Mental Events and Mental Objects - Reasoning Systems for Categories - Reasoning with Default Information-Classical planning-Algorithms for Classical Planning- Heuristics for planning-Hierarchical planning - Non-Deterministic domains- Time, Schedule and resources-Analysis

Unit V : LEARNING AND EXPERT SYSTEMS 9

Forms of Learning AI applications – Developing Machine Learning Systems-Statistical Learning- Deep Learning: Simple Feed Forward network-Neural Networks-Reinforcement Learning : Learning from rewards-Passive and active Reinforcement learning . Expert Systems : Functions-Main structure – if-then rules for representing knowledge- developing the shel-Dealingg with uncertainty



5.COURSE OUTCOME

Course Code	Course Outcome Statement	Cognitive / Affective Level of the Course Outcome	Course Outcome
Course Outcome Statements in Cognitive Domain			
21AI401	Explain the foundations of AI and various Intelligent Agents	Apply K3	CO1
21AI401	Apply Search Strategies in Problem Solving and Game Playing	Apply K3	CO2
21AI401	Explain logical agents and First-order logic	Apply K3	CO3
21AI401	Apply problem-solving strategies with knowledge representation mechanism for solving hard problems	Apply K4	CO4
21AI401	Describe the basics of learning and expert systems.	Apply K4	CO5

6.CO-PO/PSO MAPPING

Course Outcomes (Cos)		Programme Outcomes (POs), Programme Specific Outcomes (PSOs)														
		PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
21AI401.1	K 2	3	3	1	-	-	-	-	-	-	-	-	-	-	-	-
21AI401.2	K 3	3	2	1	-	-	-	-	-	-	-	-	-	-	-	-
21AI401.3	K 3	3	2	1	-	-	-	-	-	-	-	-	-	-	-	-
21AI401.4	K 3	3	3	2	-	-	-	-	-	-	-	-	-	-	-	-
21AI401.5	K 2	3	2	2	-	-	-	-	-	-	-	-	-	-	-	-

UNIT II

PROBLEM SOLVING












R.M.K.
GROUP OF
INSTITUTIONS

LECTURE PLAN – UNIT II

UNIT II PROBLEM SOLVING METHODS							
Sl. No	TOPIC	NO OF PERIODS	PROPOSED LECTURE	ACTUAL LECTURE	PERTAINING CO(s)	TAXONOMY LEVEL	MODE OF DELIVERY
			PERIOD	PERIOD			
1	Heuristic search strategies	1	18.08.2023		CO2	K3	MD1, MD5
2	Heuristic Functions	1	19.08.2023		CO2	K3	MD1, MD5
3	Game Playing –Mini-Max Algorithm	1	22.08.2023		CO2	K3	MD1, MD5
4	Optimal Decisions in Games	1	23.08.2023		CO2	K3	MD1, MD5
5	Alpha - Beta Search	1	24.08.2023		CO2	K3	MD1, MD5
6	Monte-carlo Search for Games	1	25.08.2023		CO2	K3	MD1, MD5
7	Constraint Satisfaction Problems ,Constraint Propagation	1	26.08.2023		CO2	K3	MD1, MD5
8	Backtracking Search	1	28.08.2023		CO2	K3	MD1, MD5
9	Local Search for CSP Structure of CSP	1	29.08.2023		CO2	K3	MD1, MD5

LECTURE PLAN – UNIT II

ASSESSMENT COMPONENTS

-  AC 1. Unit Test
-  AC 2. Assignment
-  AC 3. Course Seminar
-  AC 4. Course Quiz
-  AC 5. Case Study
-  AC 6. Record
-  Work
-  AC 7. Lab / Mini Project
-  AC 8. Lab Model Exam
- AC 9. Project Review

MODE OF DELEIVERY

- MD 1. Oral presentation
- MD 2. Tutorial
- MD 3. Seminar
- MD 4 Hands On
- MD 5. Videos
- MD 6. Field Visit



R.M.K.
GROUP OF
INSTITUTIONS

8. ACTIVITY BASED LEARNING : UNIT – II

ACTIVITY 1: Constraint Satisfaction problem(SUDOKU)

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

Puzzle

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

Solution

9. LECTURE NOTES : UNIT – II

PROBLEM SOLVING

Syllabus:

Heuristics Search Strategies-Heuristics functions - Game Playing - Optimal Decisions in Games – Alpha - Beta Search – Monte Carlo Search for Games -Searching with Partial Observations - Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search for CSP- - Local Search for CSP- Structure of CSP

2.1 INFORMED(Heuristic) SEARCH STRATEGIES

This section shows how an informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy. The general approach we consider is called best-first search. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best- first graph search is identical to that for uniform-cost search, except for the use of f instead of g to order the priority queue.

The choice of f determines the search strategy. (For example, best-first tree search includes depth-first search as a special case.) Most best-first algorithms include as a component of f a heuristic function, denoted $h(n)$:

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

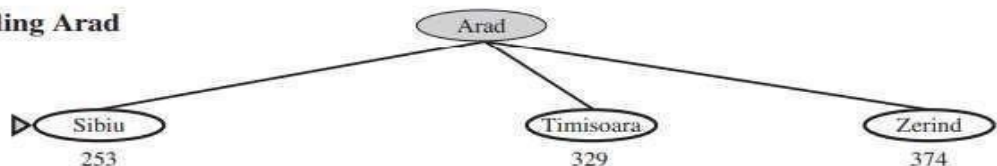
(Notice that $h(n)$ takes a node as input, but, unlike $g(n)$, it depends only on the state at that node.) For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest. Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm

2.1.1 Greedy best-first search

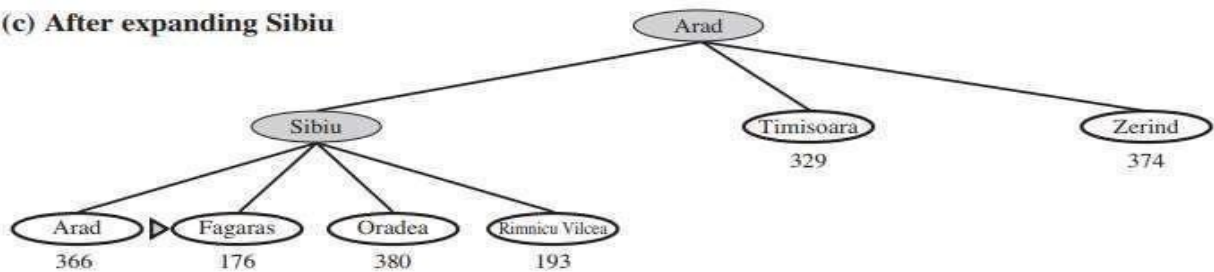
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras

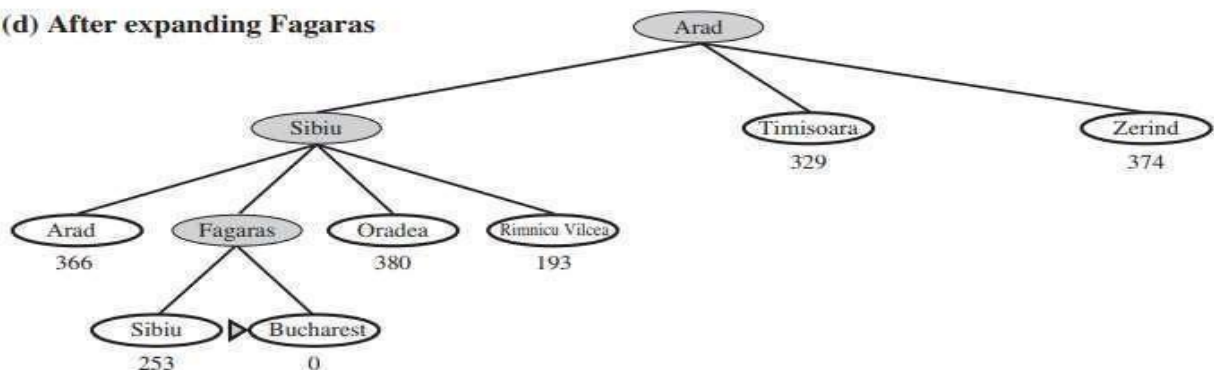


Figure 21 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD} . Nodes are labeled with their h -values.

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is,

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

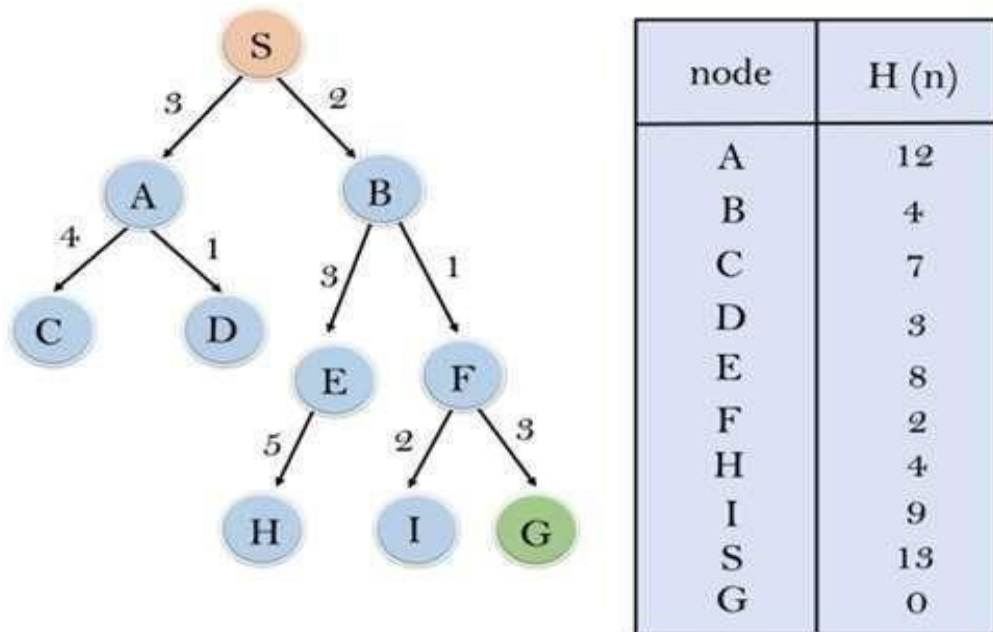
Let us see how this works for route-finding problems in Romania; we use the straightline distance heuristic, which we will call hSLD. If the goal is Bucharest, we need to know the straight-line distances to Bucharest. For example, $hSLD(In(Arad)) = 366$. Notice that the values of hSLD cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that hSLD is correlated with actual road distances and is, therefore, a useful heuristic.

Figure 10 shows the progress of a greedy best-first search using hSLD to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.

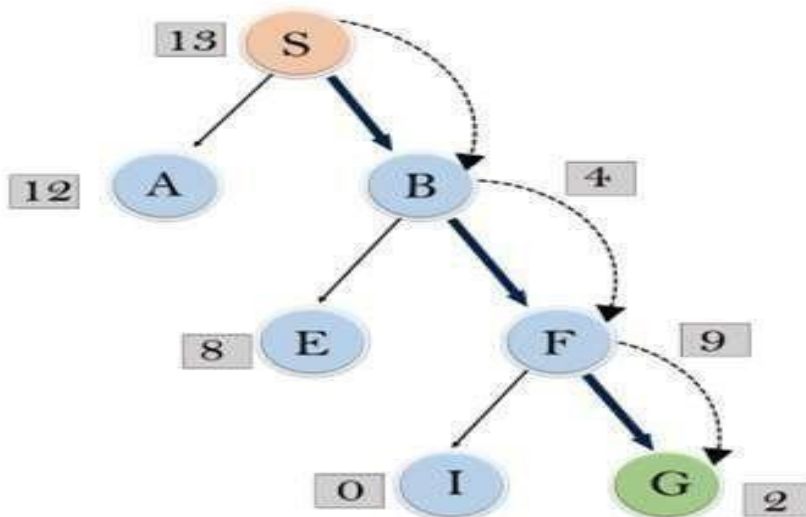
Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search. The worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function $f(n)=h(n)$, which is given in the below table. S initial state and G goal state.



Solution:



2.1.2 A* search: Minimizing the total estimated solution cost

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

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

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

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

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

(i) Conditions for optimality: Admissibility and consistency

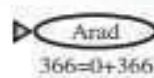
The first condition we require for optimality is that $h(n)$ be an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal. Because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, we have as an immediate consequence that $f(n)$ never overestimates the true cost of a solution along the current path through n . Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest.

A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A* to graph search. A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

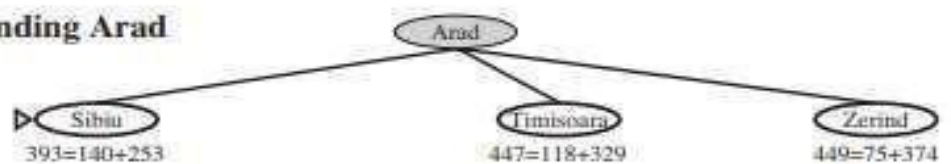
$$h(n) \leq c(n, a, n') + h(n') .$$

This is a form of the general triangle inequality, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n , n' and the goal G_n closest to n

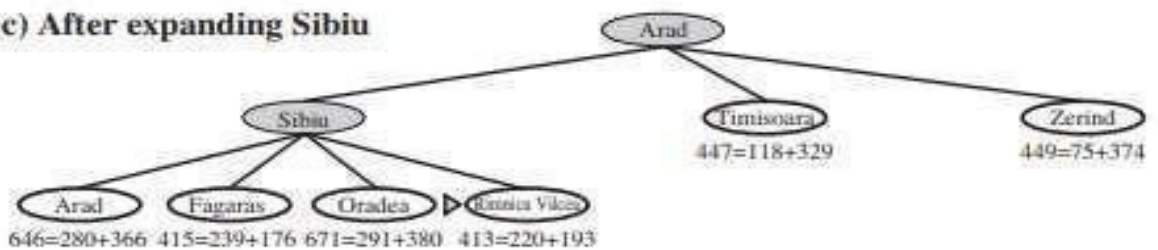
(a) The initial state



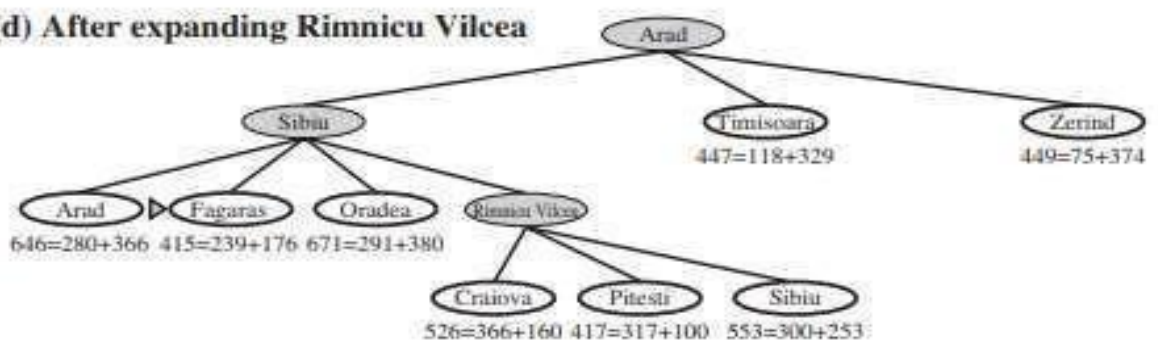
(b) After expanding Arad



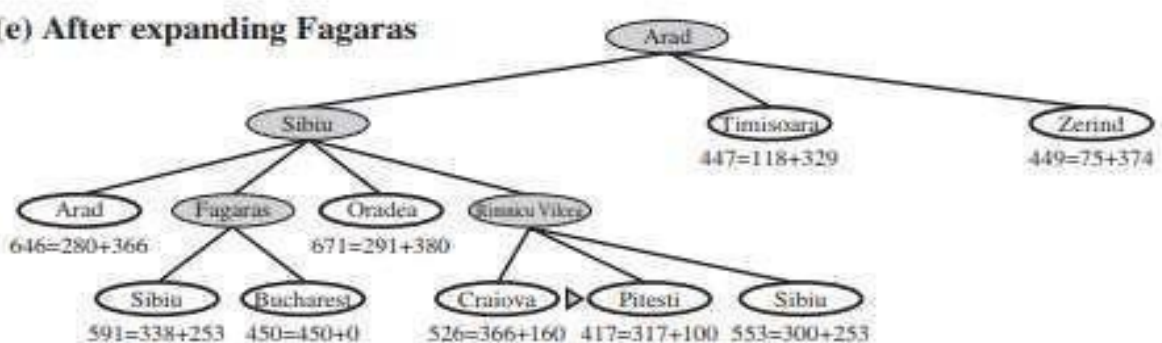
(c) After expanding Sibiu



(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti

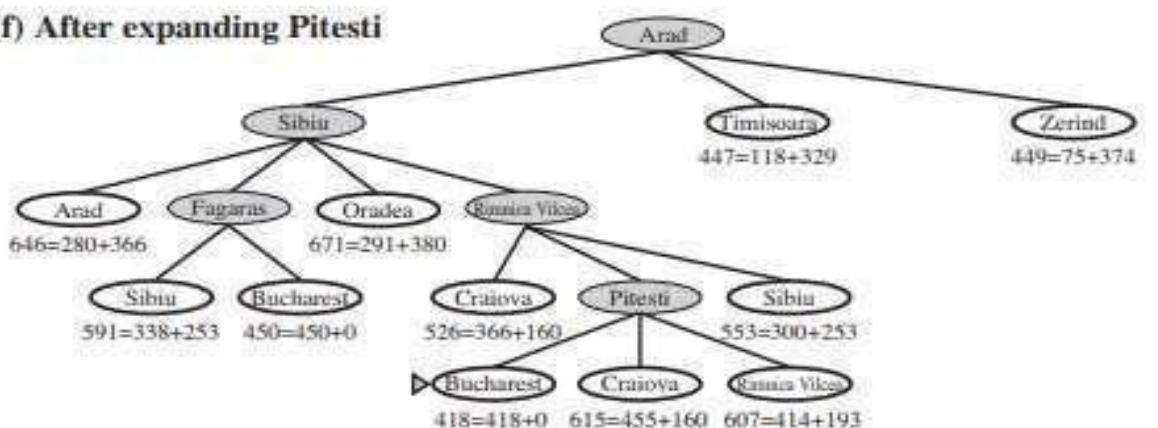


Figure 2. Stages in an A* search for Bucharest.

Nodes are labeled with $f = g + h$.

(ii) Optimality of A*

A* has the following properties: the tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent. The first step is to establish the following: if $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing. The proof follows directly from the definition of consistency. Suppose n' is a successor of n ; then $g(n') = g(n) + c(n, a, n')$ for some action a , and we have

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

The next step is to prove that whenever A* selects a node n for expansion, the optimal path to that node has been found. Were this not the case, there would have to be another frontier node n' on the optimal path from the start node to n , by the graph separation property, because f is nondecreasing along any path, n' would have lower f -cost than n and would have been selected first. From the two preceding observations, it follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have $h = 0$) and all later goal nodes will be at least as expensive.

If C^* is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < C^*$.
- A* might then expand some of the nodes right on the “goal contour” (where $f(n) = C^*$) before selecting a goal node.

Completeness requires that there be only finitely many nodes with cost less than or equal to C^* , a condition that is true if all step costs exceed some finite ϵ and if b is finite. One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root and use the same heuristic information—A* is optimally efficient for any given consistent heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A*. Figure 11 shows the stages of A* Search.

For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the absolute error or the relative error of the heuristic. The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and the relative error is defined as

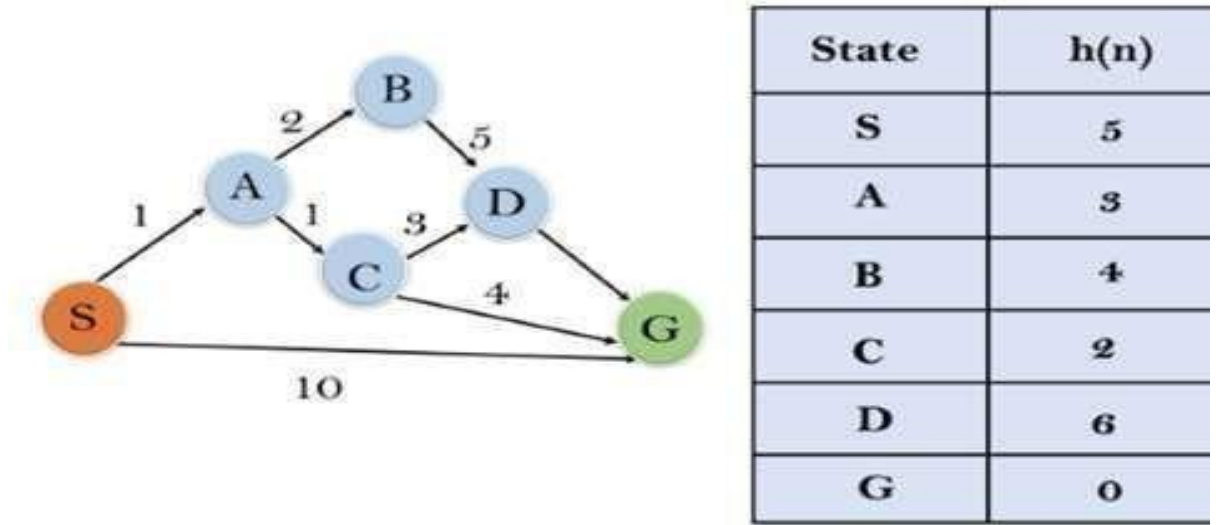
$$\varepsilon \equiv (h^* - h)/h^*.$$

The complexity of A^* often makes it impractical to insist on finding an optimal solution. One can use variants of A^* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible. Computation time is not, however, A^* 's main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A^* usually runs out of space long before it runs out of time. For this reason, A^* is not practical for many large-scale problems. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

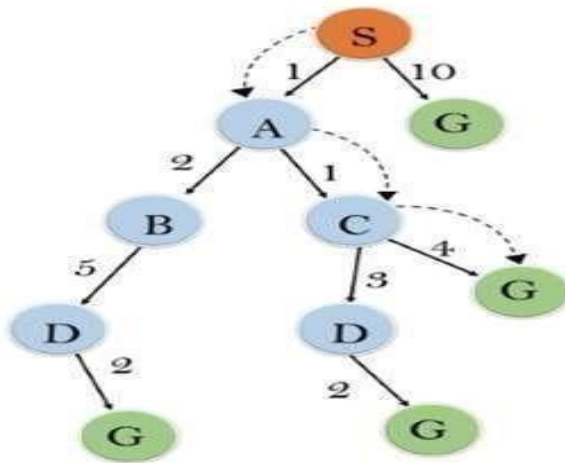
Example:

In this example, we will traverse the given graph using the A^* algorithm. The heuristic value of all states is given in the below table so we will calculate the $f(n)$ of each state using the formula $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach any node from start state.

S- Initial state; G-Goal state.



Solution:



2.1.3 Memory-bounded heuristic search

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A* (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f-limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children

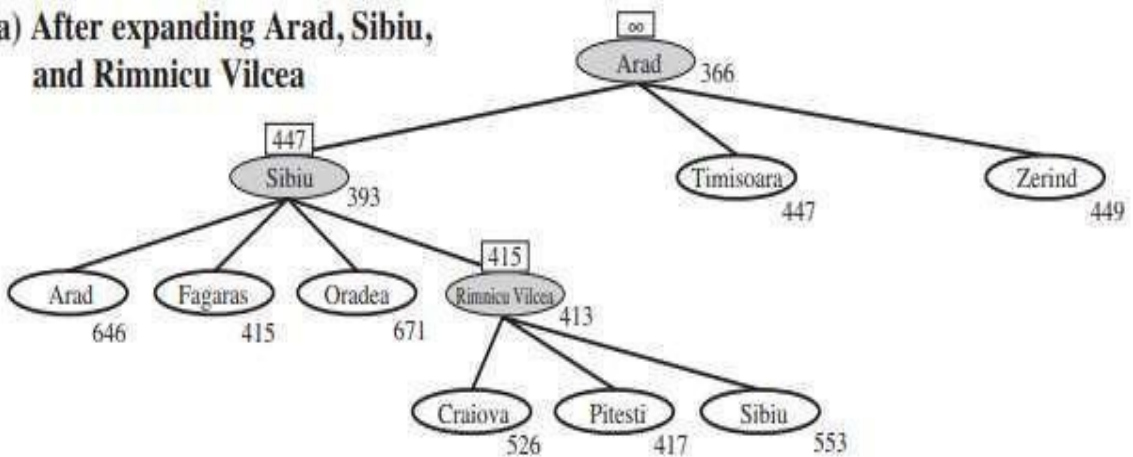
The algorithm for recursive best-first search

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE),  $\infty$ )

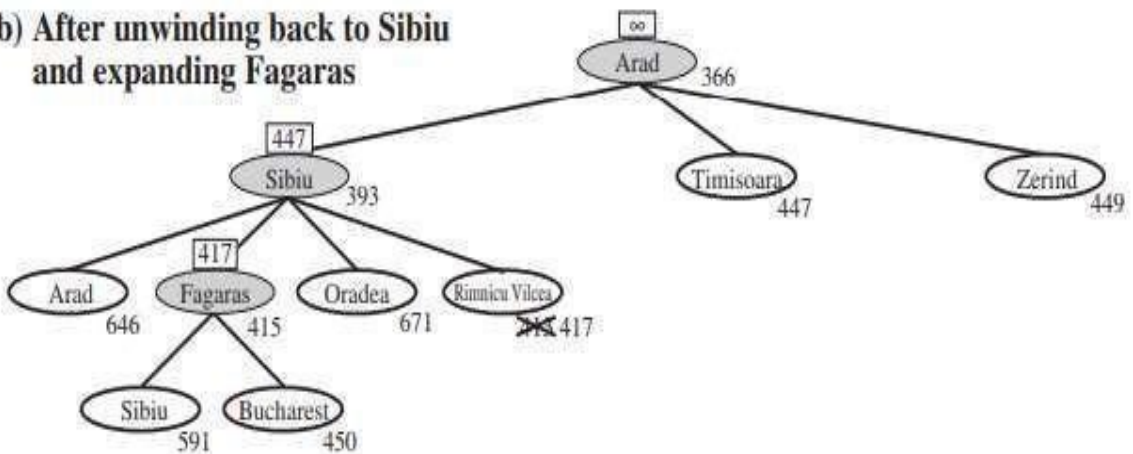
function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    successors  $\leftarrow []$ 
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure,  $\infty$ 
    for each s in successors do /* update f with value from previous search, if any */
        s.f  $\leftarrow \max(s.g + s.h, \text{node.f})$ 
    loop do
        best  $\leftarrow$  the lowest f-value node in successors
        if best.f > f_limit then return failure, best.f
        alternative  $\leftarrow$  the second-lowest f-value among successors
        result, best.f  $\leftarrow$  RBFS(problem, best, min(f_limit, alternative))
        if result  $\neq$  failure then return result
```

In figure 12, The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti

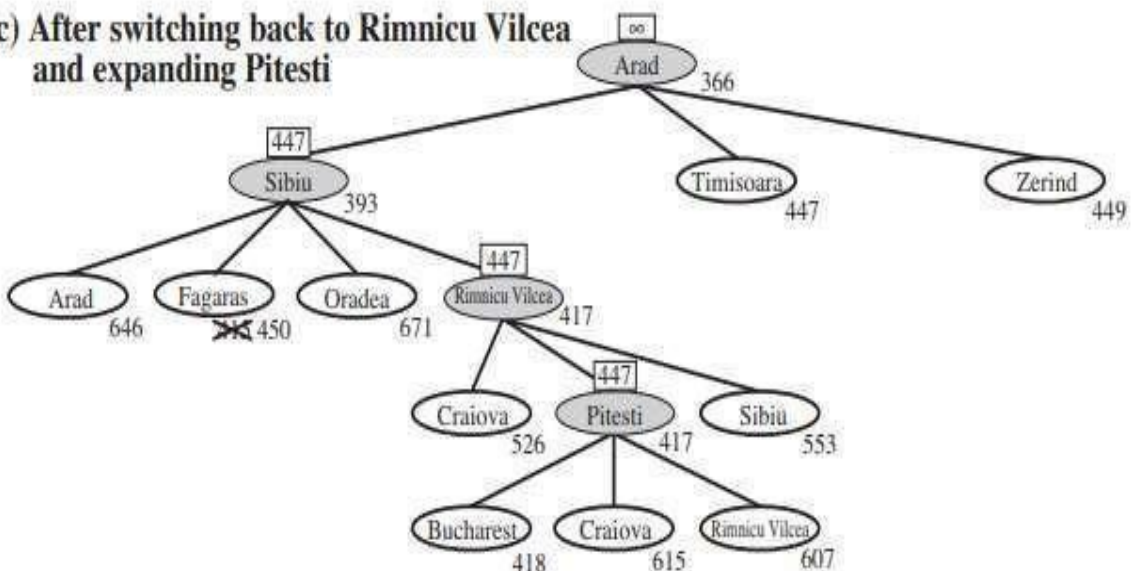
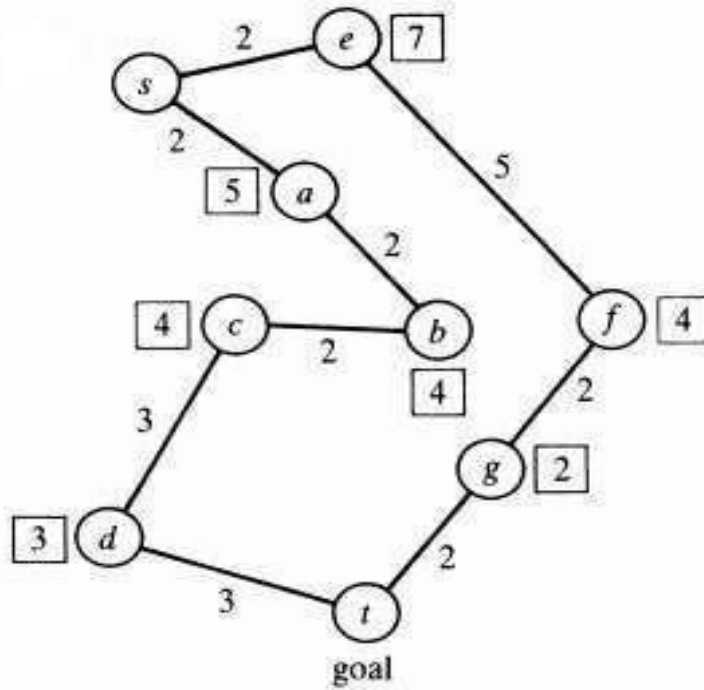
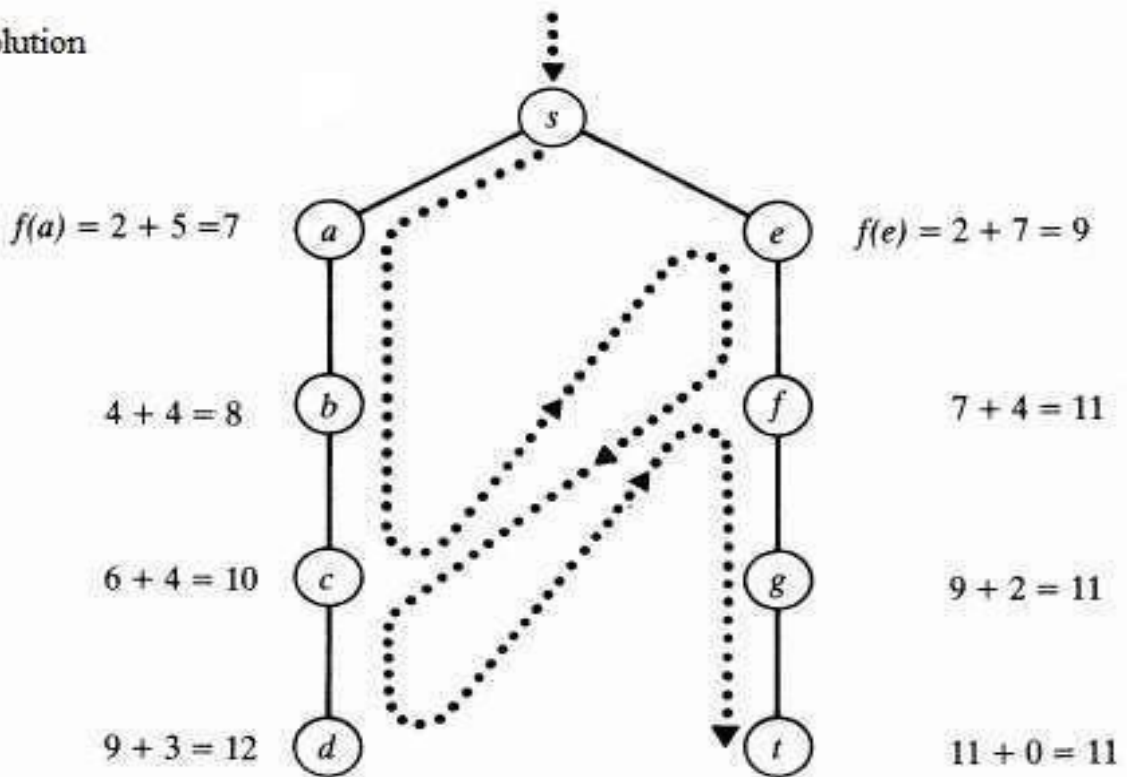


Figure 3 Stages in an RBFS search for the shortest route to Bucharest.

Example of RBFS



Solution



IDA* and RBFS suffer from using too little memory. Between iterations, IDA* retains only a single number: the current f-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up re-expanding the same states many times over. It seems sensible, therefore, to use all available memory. Two algorithms that do this are MA* (memory-bounded A*) and SMA* (simplified MA*). SMA* is—well—simpler, so we will describe it. SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the worst leaf node—the one with the highest f-value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent.

SMA* is complete if there is any reachable solution—that is, if d , the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the explored set. Memory limitations can make a problem intractable from the point of view of computation time. Although no current theory explains the tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

2.2. HEURISTIC FUNCTIONS

The 8-puzzle was one of the earliest heuristic search problems. The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means that an exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states.

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Figure 13. A typical instance of the 8-puzzle. The solution is 26 steps long.

Two commonly used candidates:

1. h_1 = the number of misplaced tiles. For Figure 13, all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
2. h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. h_2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

As expected, neither of these overestimates the true solution cost, which is 26.

2.2.1 The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristic is the effective branching factor b^* . If the total number of nodes generated by A* for a particular problem is N and the solution depth is d , then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes. It is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

2.2.2 Generating admissible heuristics from relaxed problems

We have seen that both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h_2 is better. h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h_1 would give the exact number of steps in the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a relaxed problem. The state-space graph of the relaxed problem is a super graph of the original state space because the removal of restrictions creates added edges in the graph.

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically. For example, if the 8-puzzle actions are described as

A tile can move from square A to square B if

A is horizontally or vertically adjacent to B and B is blank, we can generate three relaxed problems by removing one or both of the conditions:

- (a) A tile can move from square A to square B if A is adjacent to B.
- (b) A tile can move from square A to square B if B is blank.
- (c) A tile can move from square A to square B.

From (a), we can derive h_2 (Manhattan distance). The reasoning is that h_2 would be the proper score if we moved each tile in turn to its destination. From (c), we can derive h_1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step. Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially without search, because the relaxed rules allow the problem to be decomposed into eight independent subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain. A program called ABSOLVER can generate heuristics automatically from problem definitions, using the “relaxed problem” method and various other techniques

2.2.3 Generating admissible heuristics from subproblems: Pattern databases

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. (The locations of the other four tiles are irrelevant for the purposes of solving the subproblem, but moves of those tiles do count toward the cost.) Then we compute an admissible heuristic h_{DB} for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database.

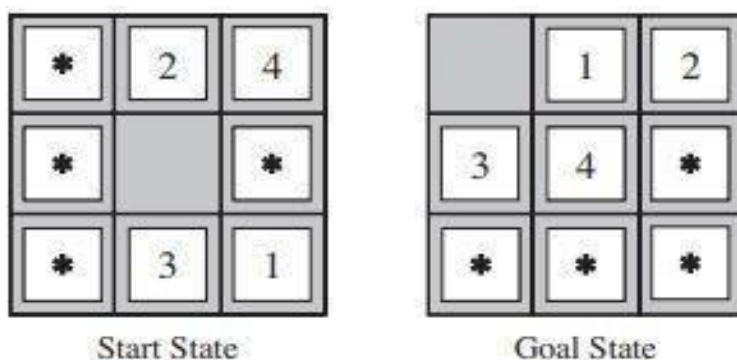


Figure 14 A subproblem of the 8-puzzle instance given in figure 13

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate than the Manhattan distance; the number of nodes generated when solving random 15-puzzles can be reduced by a factor of 1000. One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be added, since the two subproblems seem not to overlap.

Would this still give an admissible heuristic? The answer is no, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves—it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa. But what if we don't count those moves?

That is, we record not the total cost of solving the 1-2- 3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem. This is the idea behind disjoint pattern databases. Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time.

2.2.4 Learning heuristics from experience

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n . One solution was given in the preceding sections—namely, to devise relaxed problems for which an optimal solution can be found easily. Another solution is to learn from experience. “Experience” here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides examples from which $h(n)$ can be learned. Each example consists of a state from the solution path and the actual cost of the solution from that point. From these examples, a learning algorithm can be used to construct a function $h(n)$ that can (with luck) predict solution costs for other states that arise during search.

2.3 Game Playing

Mathematical **game theory**, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive. In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games of perfect information** (such as chess). Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed.

For AI researchers, the abstract nature of games makes them an appealing subject for study. The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules. With the exception of robot soccer, these physical games have not attracted much interest in the AI community. A game can be formally defined as a kind of search problem with

the following elements:

- S_0 : The initial state, which specifies how the game is set up at the start.
- $\text{PLAYER}(s)$: Defines which player has the move in a state.
- $\text{ACTIONS}(s)$: Returns the set of legal moves in a state.
- $\text{RESULT}(s, a)$: The transition model, which defines the result of a move.
- $\text{TERMINAL-TEST}(s)$: A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- $\text{UTILITY}(s, p)$: A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p . In chess, the outcome is a win, loss, or draw, with values $+1$, 0 , or $\frac{1}{2}$. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to $+192$. A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$. “Constant-sum” would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves. Figure 27 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX’s placing an X and MIN’s placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names). A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

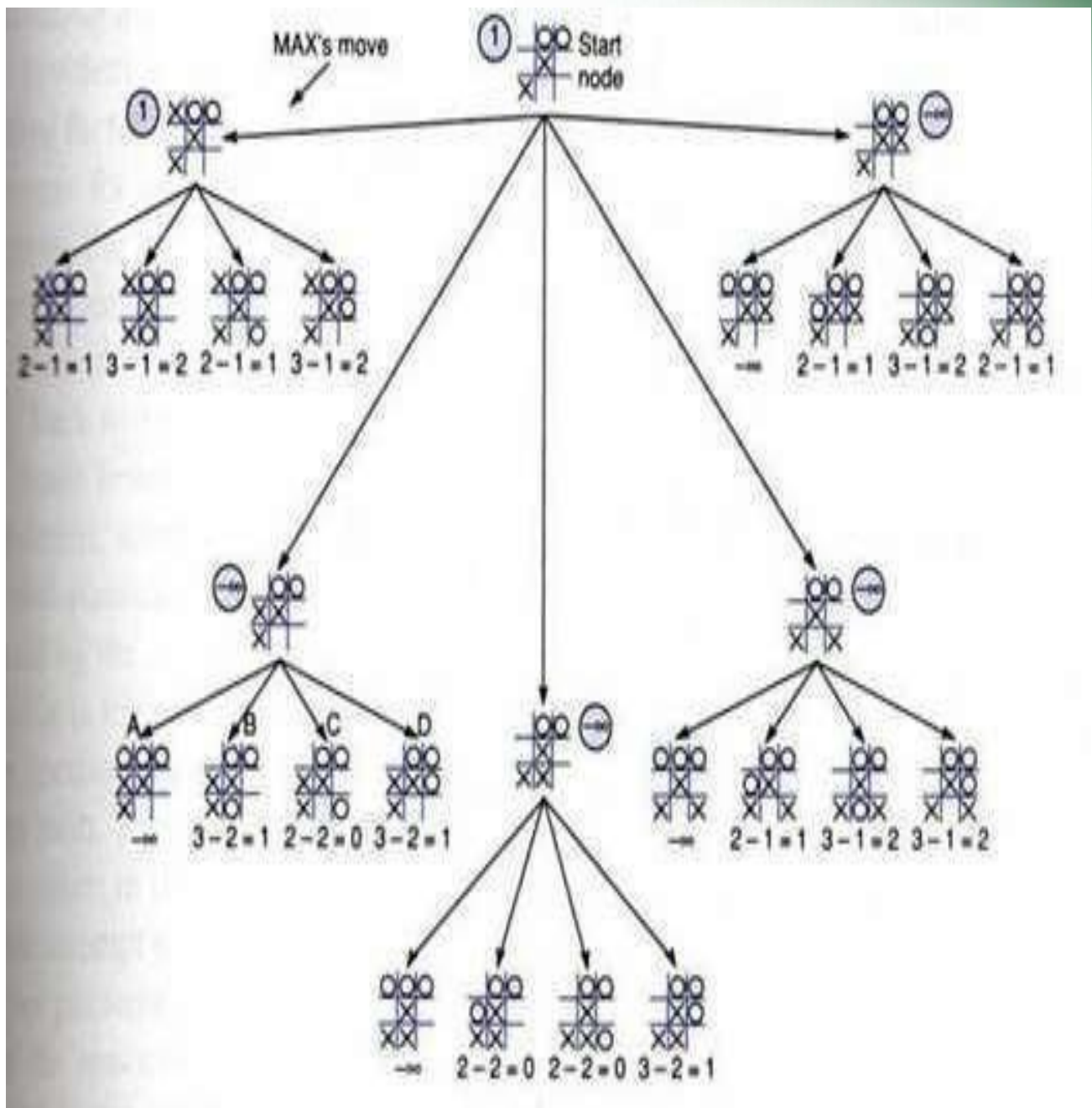


Figure 4 tic-tac-toe game

OPTIMAL DECISIONS IN GAMES

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.

This is exactly analogous to the AND–OR search algorithm with MAX playing the role of OR and MIN equivalent to AND. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent. We begin by showing how to find this optimal strategy. Given a game tree, the optimal strategy can be determined from the **mini max value** of each node, which we write as MINIMAX(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game.

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

2.3.1 The minimax algorithm

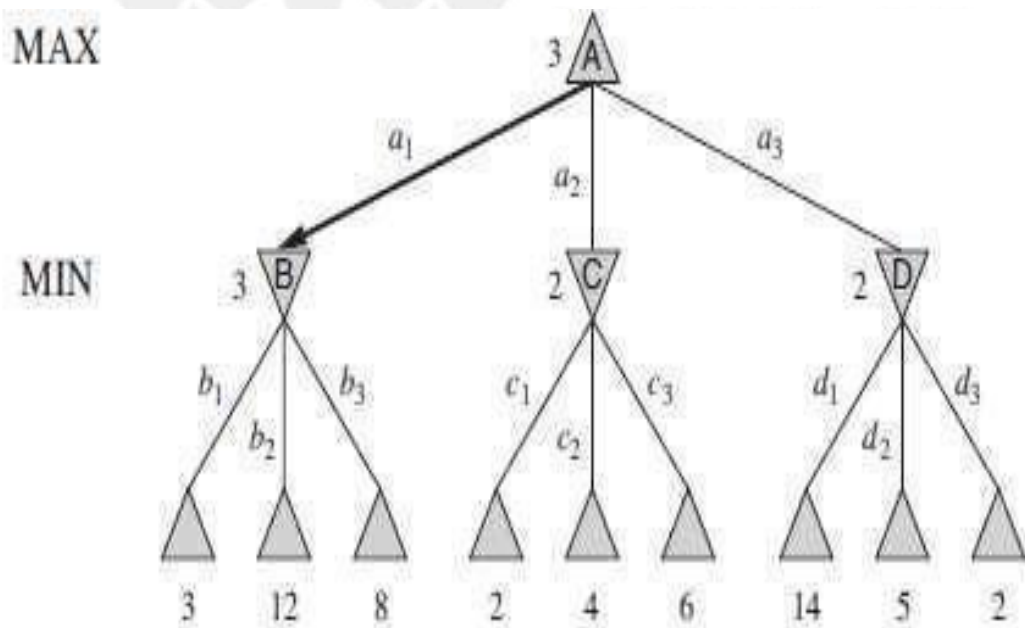


Figure 5 A two-ply game tree.

The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in Figure 28, the algorithm first recurses down to the three bottomleft nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D.

Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(bm)$. The space complexity is $O(bm)$ for an algorithm that generates all actions at once, or $O(m)$ for an algorithm that generates actions one at a time (see page 87). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

2.3.2 Optimal decisions in multiplayer games

Some interesting new conceptual issues. First, we need to replace the single value for each node with a vector of values. For example, in a three-player game with players A, B, and C, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities. Now we have to consider nonterminal states.

Consider the node marked X in the game tree shown in Figure 29. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached,

subsequent play will lead to a terminal state with utilities

$\langle v_A = 1, v_B = 2, v_C = 6 \rangle$

Hence, the backed-up value of X is this vector. The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n . Anyone who plays multiplayer games, such as Diplomacy, quickly becomes aware that much more is going on than in two-player games. Multiplayer ALLIANCE games usually involve **alliances**, whether formal or informal, among the players.

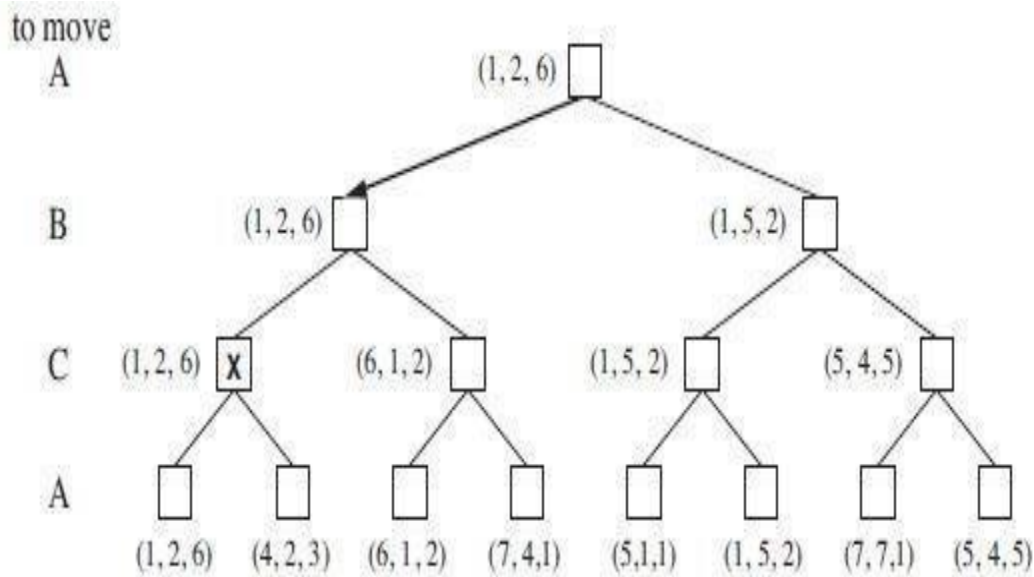


Figure 6 The first three plies of a game tree with three players

Algorithm

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

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for each  $a$  in  $\text{ACTIONS}(\text{state})$  do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
  return  $v$ 
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$ 
  for each  $a$  in  $\text{ACTIONS}(\text{state})$  do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
  return  $v$ 
```

2.3.3 ALPHA BETA SEARCH

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** to eliminate large parts of the tree from consideration. The particular technique we examine is called **alpha-beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

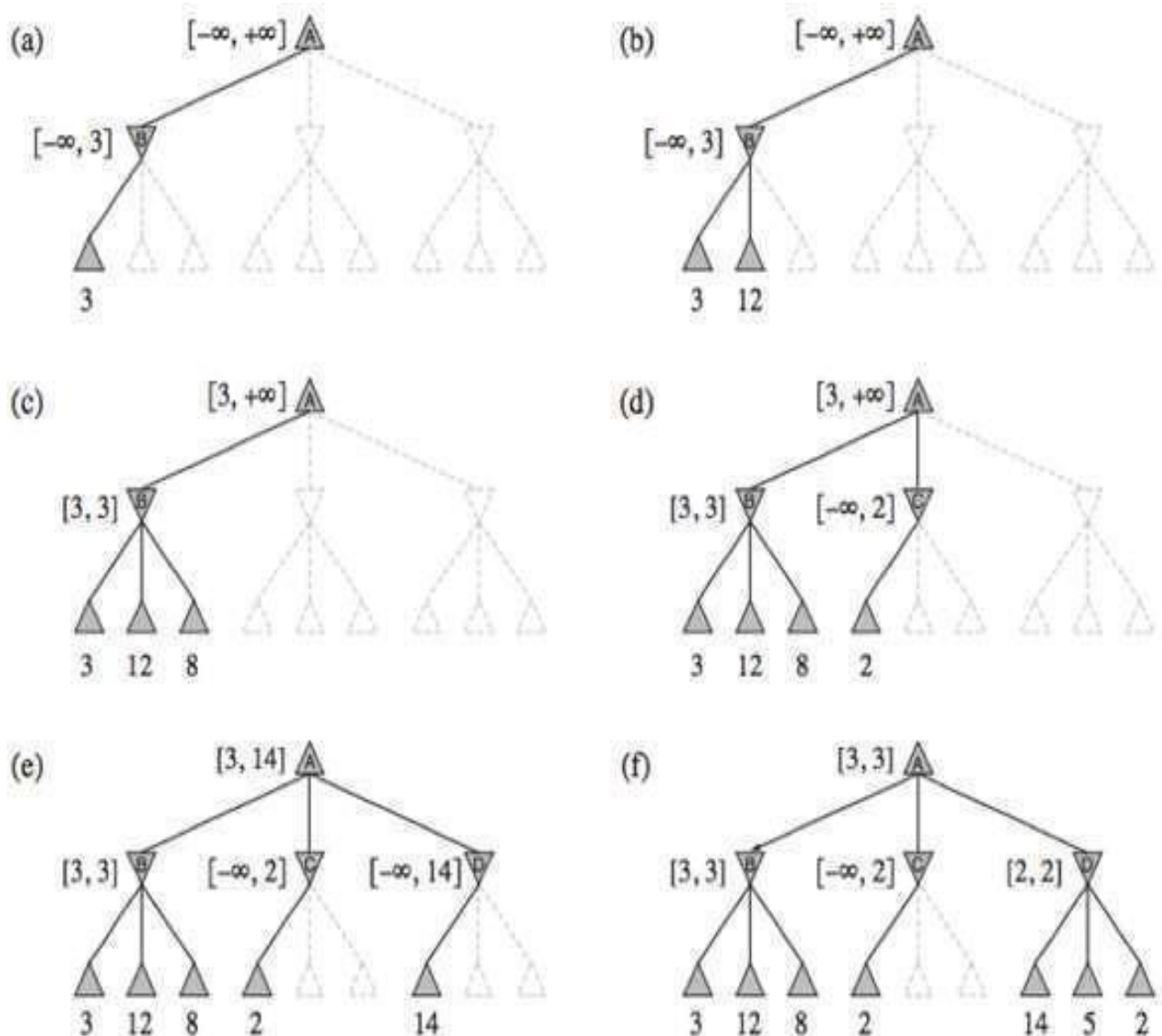


Figure 7 Stages in the calculation of the optimal decision for the game tree

Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node C in Figure 30 have values x and y . Then the value of the root node is given by

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

Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

Steps:

Stages in the calculation of the optimal decision for the game tree in Figure 28 At each point, we show the range of possible values for each node.

- (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.
- (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.
- (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.
- (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning..

- (e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.
- (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move. One way to gain information from the current move is with iterative deepening search. First, search 1 ply deep and record the best path of moves. Then search 1 ply deeper, but use the recorded path to inform move ordering. Iterative deepening on an exponential game tree adds only a constant fraction to the total search time, which can be more than made up from better move ordering. The best moves are often called **killer moves** and to try them first is called the killer move heuristic.

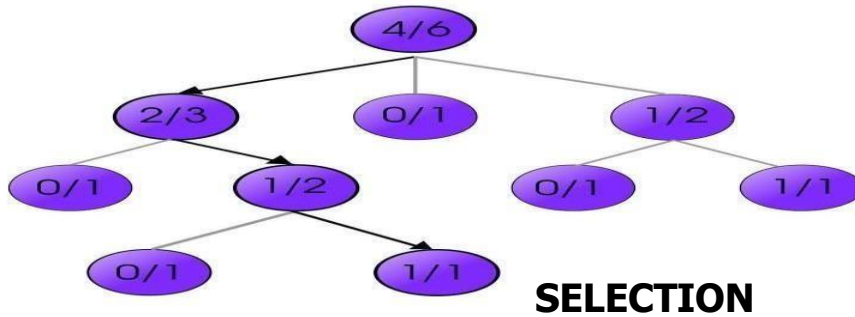
2.3.4 MONTE-CARLO TREE SEARCH

What is Monte-carlo Tree Search?

MCTS is an algorithm that figures out the best move out of a set of moves by Selecting → Expanding → Simulating → Updating the nodes in tree to find the final solution. This method is repeated until it reaches the solution and learns the policy of the game.

How does Monte Carlo Tree Search Work?

Let's look at parts of the loop one-by-one.

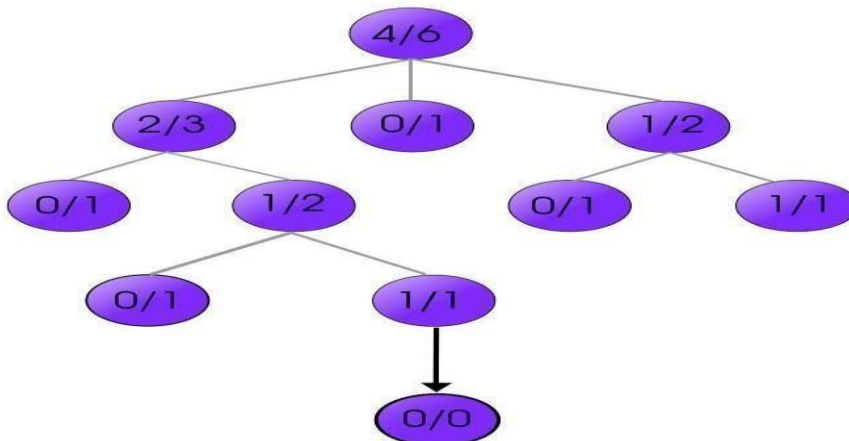


Selecting 🖐️ | This process is used to select a node on the tree that has the highest possibility of winning.

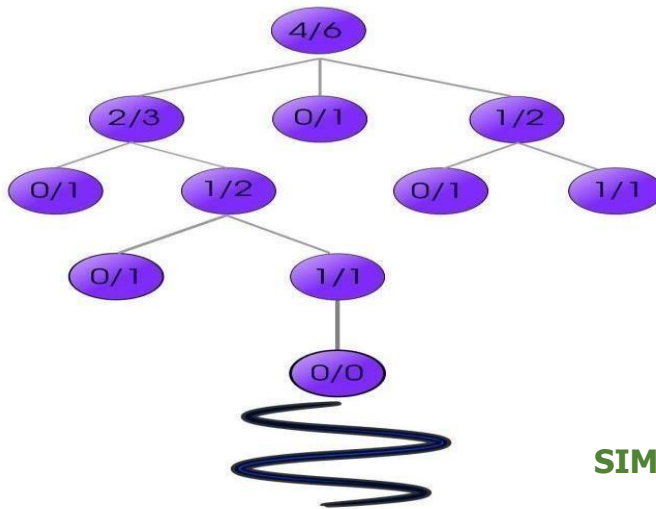
For Example — Consider the moves with winning possibility $2/3$, $0/1$ & $1/2$ after the first move $4/6$, the node $2/3$ has the highest possibility of winning.

The node selected is searched from the current state of the tree and selected node is located at the end of the branch.

Since the selected node has the highest possibility of winning — that path is also most likely to reach the solution faster than other path in the tree.



Expanding — After selecting the right node. Expanding is used to increase the options further in the game by expanding the selected node and creating many children nodes. We are using only one children node in this case. These children nodes are the future moves that can be played in the game. The nodes that are not expanded further for the time being are known as leaves.



SIMULATION

SIMULATING | EXPLORING Since nobody knows which node is the best children/ leaf from the group. The move which will perform best and lead to the correct answer down the tree. How do we find the best children which will lead us to the correct solution? We use Reinforcement Learning to make random decisions in the game further down from every children node. Then, reward is given to every children node — by calculating how close the output of their random decision was from the final output that we need to win the game.

For example: In the game of Tic-Tac-Toe. Does the random decision to make cross(X) next to previous cross(X) in the game results in three consecutive crosses(X-X-X) that are needed to win the game?

The simulation is done for every children node is followed by their individual rewards.

2.4 CONSTRAINT SATISFACTION PROBLEM

A constraint satisfaction problem consists of three components, X, D , and C :

X is a set of variables, $\{X_1, \dots, X_n\}$.

D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.

C is a set of constraints that specify allowable combinations of values.

Each domain D_i consists of a set of allowable values, $\{v_1, \dots, v_k\}$ for variable X_i . Each constraint C_i consists of a pair $\langle \text{scope}, \text{rel} \rangle$, 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. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.

For example, if X_1 and X_2 both have the domain $\{A, B\}$, then the constraint saying the two variables must have different values can be written as $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ or as $\langle (X_1, X_2), X_1 \neq X_2 \rangle$. To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \dots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that assigns values to only some of the variables.

1. Example problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions

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

The domain of each variable is the set $D_i = \{\text{red}, \text{green}, \text{blue}\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

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

Here we are using abbreviations, $SA \neq WA$ is a shorthand for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$\{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), (\text{green}, \text{blue}), (\text{blue}, \text{red}), (\text{blue}, \text{green})\}$.

There are many possible solutions to this problem, such as

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

It can be helpful to visualize a CSP as a **constraint graph**, as shown in Figure 24. The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint. In figure 24(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. In figure 24 (b), The map-coloring problem represented as a constraint graph. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

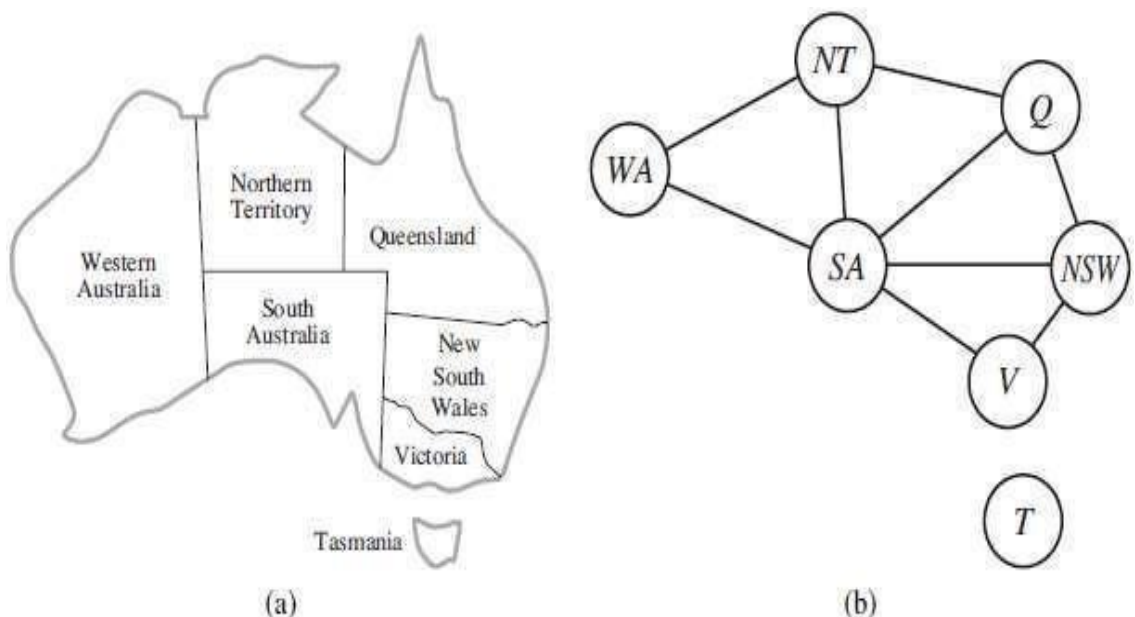


Figure 8 Example for map colouring

2.Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

$$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$$

The value of each variable is the time that the task starts. Next we represent precedence constraints between individual tasks. Whenever a task T1 must occur before task T2, and task T1 takes duration d1 to complete, we add an arithmetic constraint of the form

$$T1 + d1 \leq T2$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$\begin{aligned} Axle_F + 10 &\leq Wheel_{RF}; & Axle_F + 10 &\leq Wheel_{LF}; \\ Axle_B + 10 &\leq Wheel_{RB}; & Axle_B + 10 &\leq Wheel_{LB}. \end{aligned}$$

Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$\begin{aligned} Wheel_{RF} + 1 &\leq Nuts_{RF}; & Nuts_{RF} + 2 &\leq Cap_{RF}; \\ Wheel_{LF} + 1 &\leq Nuts_{LF}; & Nuts_{LF} + 2 &\leq Cap_{LF}; \\ Wheel_{RB} + 1 &\leq Nuts_{RB}; & Nuts_{RB} + 2 &\leq Cap_{RB}; \\ Wheel_{LB} + 1 &\leq Nuts_{LB}; & Nuts_{LB} + 2 &\leq Cap_{LB}. \end{aligned}$$

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does: This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_B$ can take on. We also need to assert that the inspection comes last and takes 3 minutes. For every variable except $Inspect$ we add a constraint of the form $X + d_x \leq Inspect$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i = \{1, 2, 3, \dots, 27\}.$$

This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used later.

2.7.3 Variations on the CSP formalism

The simplest kind of CSP involves variables that have **discrete, finite domains**. Map-coloring problems and scheduling with time limits are both of this kind. A discrete domain can be **infinite**, such as the set of integers or strings. (If we didn't put a deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.) With infinite domains, it is no longer possible to

describe constraints by enumerating all allowed combinations of values.

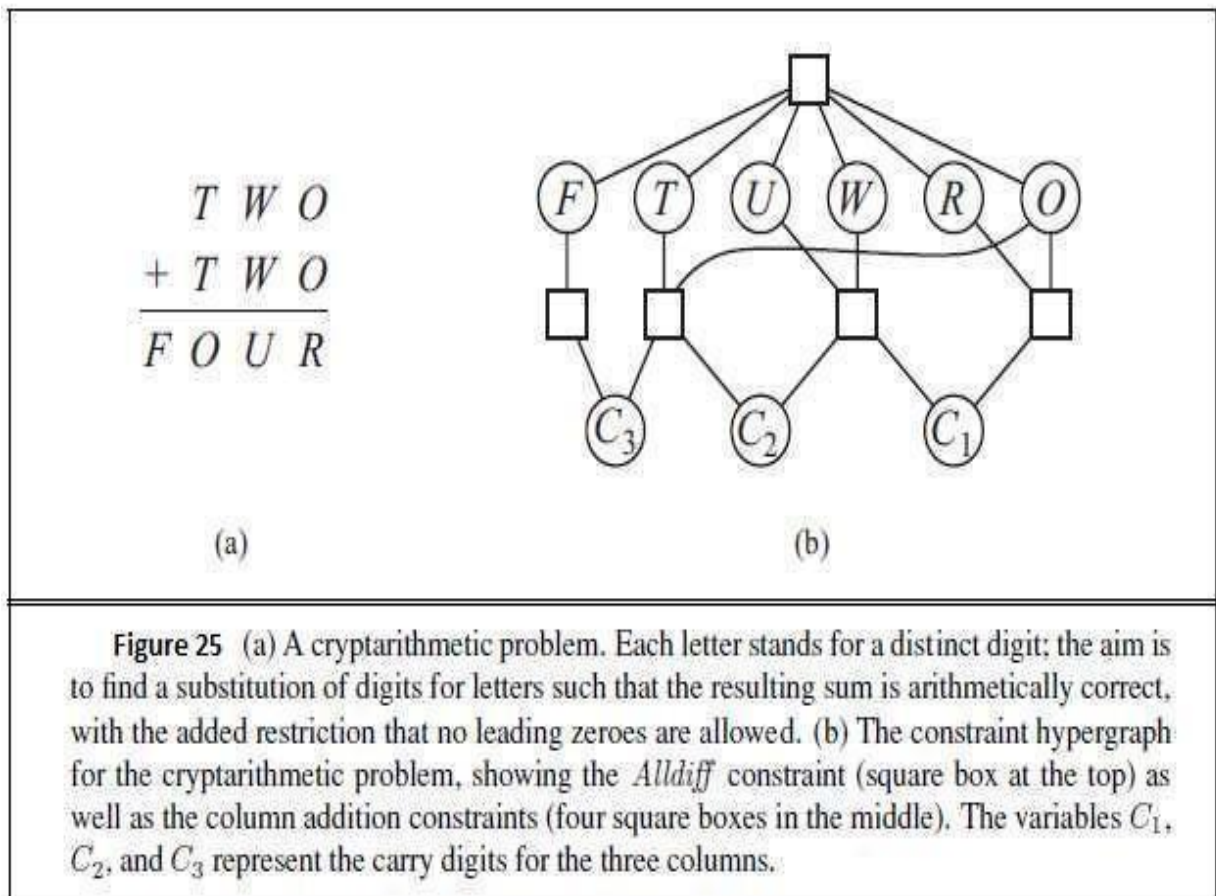
Instead, a **constraint language** must be used that understands constraints such as $T1 + d1 \leq T2$ directly, without enumerating the set of pairs of allowable values for $(T1, T2)$. Special solution algorithms (which we do not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables.

Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because it need not involve all the variables in a problem). One of the most common global constraints is **Alldiff**, which says that all of the variables involved in the constraint must have different values. In Sudoku problems, all variables in a row or column must satisfy an **Alldiff** constraint. An other example is provided by **cryptarithmic** puzzles. (See Figure 25(a)) Each letter in a cryptarithmic puzzle represents a different digit. For the case in Figure 25(a), this would be represented as the global constraint **Alldiff** (F, T, U, W, R, O). The addition constraints on the four columns of the puzzle can be written as the following n -ary constraints:

$$\begin{aligned} O + O &= R + 10 \cdot C_{10} \\ C_{10} + W + W &= U + 10 \cdot C_{100} \\ C_{100} + T + T &= O + 10 \cdot C_{1000} \\ C_{1000} &= F, \end{aligned}$$

where C10, C100, and C1000 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 25(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.



The constraints we have described so far have all been absolute constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference constraints** indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one.

Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constraint optimization problem**, or COP. Linear programming problems do this kind of optimization.

2.4.1 CONSTRAINT PROPAGATION : INFERENCE IN CSP

In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of **inference** called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. The key idea is **local consistency**. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

Node consistency

A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem where South Australians dislike green, the variable SA starts with domain {red, green, blue}, and we can make it node consistent by eliminating green, leaving SA with the reduced domain {red, blue}. We say that a network is node-consistent if every variable in the network is node-consistent. It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j) . A network is arc-consistent if every variable is arc-consistent with every other variable. For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of digits. We can write this constraint explicitly as

$$\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9)\} \rangle .$$

To make X arc-consistent with respect to Y , we reduce X 's domain to $\{0, 1, 2, 3\}$. If we also make Y arc-consistent with respect to X , then Y 's domain becomes $\{0, 1, 4, 9\}$ and the whole CSP is arc-consistent.

On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (SA, WA) :

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable. The complexity of AC-3 can be analyzed as follows. Assume a CSP with n variables, each with domain size at most d , and with c binary constraints (arcs). Each arc (X_k, X_i) can be inserted in the queue only d times because X_i has at most d values to delete. Checking consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.

It is possible to extend the notion of arc consistency to handle n -ary rather than just binary constraints; this is called generalized arc consistency or sometimes hyperarc consistency, depending on the author. A variable X_i is **generalized arc consistent** with respect to an n -ary constraint if for every value v in the domain of X_i there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its X_i component equal to v .

Path consistency

Arc consistency can go a long way toward reducing the domains of variables, sometimes finding a solution (by reducing every domain to size 1) and sometimes finding that the CSP cannot be solved (by reducing some domain to size 0). But for other networks, arc consistency fails to make enough inferences. Consider the map-coloring problem on Australia, but with only two colors allowed, red and blue.

Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa). But clearly there is no solution to the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colors for them alone. A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. This is called path consistency because one can think of it as looking at a path from X_i to X_j with X_m in the middle

K-consistency

Stronger forms of propagation can be defined with the notion of **k-consistency**. A CSP is k-consistent if, for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint networks, 3-consistency is the same as path consistency. A CSP is **strongly k-consistent** if it is k-consistent and is also $(k - 1)$ -consistent, $(k - 2)$ -consistent, all the way down to 1-consistent.

Global constraints

Remember that a **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far.

Sudoku example

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box (see Figure 26). A row, column, or box is called a **unit**.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

(a) A Sudoku puzzle and (b) its solution.

Figure 10 Example of Sudoku Puzzle

The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second. A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the prefilled squares have a domain consisting of a single value. In addition, there are 27 different AllDiff constraints: one for each row, column, and box of 9 squares.

Alldiff (A1,A2,A3,A4,A5,A6, A7, A8, A9)

Alldiff (B1,B2,B3,B4,B5,B6,B7,B8,B9)

■ . .
Alldiff (A1,B1,C1,D1,E1, F1,G1,H1, I1)

Alldiff (A2,B2,C2,D2,E2, F2,G2,H2, I2)

■ . .
Alldiff (A1,A2,A3,B1,B2,B3,C1,C2,C3)

Alldiff (A4,A5,A6,B4,B5,B6,C4,C5,C6)

■ . .

But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of Alldiff constraints and has nothing to do with Sudoku per se. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

2.4.2 BACKTRACKING SEARCH FOR CSPs

Sudoku problems are designed to be solved by inference over constraints. But many other CSPs cannot be solved by inference alone; there comes a time when we must search for a solution. In this section we look at backtracking search algorithms that work on partial assignments; in the next section we look at local search algorithms over complete assignments.

We could apply a standard depth-limited search. A state would be a partial assignment, and an action would be adding var = value to the assignment. But for a CSP with n variables of domain size d , we quickly notice something terrible: the branching factor at the top level is nd because any of d values can be assigned to any of n variables. Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order.

By varying the functions `SELECT-UNASSIGNED-VARIABLE` and `ORDER-DOMAIN-VALUES`, we can implement the general-purpose heuristics discussed in the text. The function `INFERENCE` can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by `INFERENCE` or by `BACKTRACK`), then value assignments (including those made by `INFERENCE`) are removed from the current assignment and a new value is tried.

A simple backtracking algorithm for constraint satisfaction problems

```

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

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

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then `BACKTRACK` returns failure, causing the previous call to try another value.

Variable and value ordering

The backtracking algorithm contains the line

$\text{var} \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(\text{csp})$

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, $\{X_1, X_2, \dots\}$. This static variable ordering seldom results in the most efficient search. Choosing the variable with the fewest “legal” values—is called the **minimum remaining-values** (MRV) heuristic. It also has been called the “most constrained variable” or “fail-first” heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables. In this case, the **degree heuristic** comes in handy.

It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 24, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has degree 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose any consistent color at each choice point and still arrive at a solution with no backtracking. The minimum-remaining values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker. Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

Interleaving search and inference

So far we have seen how AC-3 and other algorithms can infer reductions in the domain of variables before we begin the search. But inference can be even more powerful in the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables. One of the simplest forms of inference is called **forward checking**. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent with the value chosen for X . Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Figure 26 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after $WA=red$ and $Q=green$ are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q . A second point to notice is that after $V=blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA=red, Q=green, V=blue\}$ is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately. For many problems the search will be more effective if we combine the MRV heuristic with forward checking.

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After $WA=red$	(R)	G B	R G B	R G B	R G B	G B	R G B
After $Q=green$	(R)	B	(G)	R B	R G B	B	R G B
After $V=blue$	(R)	B	(G)	R	(B)		R G B

Figure 26 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.

Consider Figure 26 after assigning {WA=red}. Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those variables next, and then all the other variables will fall into place. That's exactly what happens with MRV: NT and SA have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job. The problem is that it makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent. The algorithm called MAC (for **Maintaining Arc Consistency (MAC)**) detects this inconsistency.

Intelligent backtracking: Looking backward

Consider what happens when we apply simple backtracking in Figure 24 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment {Q=red, NSW=green, V=blue, T=red}. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly resolve the problem with South Australia.

A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of SA impossible. To do this, we will keep track of a set of assignments that are in conflict with some value for SA. The set (in this case {Q=red, NSW=green, V=blue, }), is called the **conflict set** for SA. The **backjumping** method backtracks to the most recent assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator. Consider again the partial assignment {WA=red, NSW=red} (which, from our earlier discussion, is inconsistent). Suppose we try T=red next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT.

Now, the question is, where to backtrack? Backjumping cannot work, because NT does have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, taken together, failed because of a set of preceding variables, which must be those variables that directly conflict with the four.

This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, together with any subsequent variables, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**. **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

2.4.3 LOCAL SEARCH FOR CSPS

Local search algorithms for CSPs use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. The min-conflicts heuristic: In choosing a new value for a variable, select the value that results in the minimum number of conflicts with other variables.

Constraint weighting:

A technique that can help concentrate the search on the important constraints. Each constraint is given a numeric weight W_i , initially all 1.

At each step, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.

The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

Local search can be used in an online setting when the problem changes, this is particularly important in scheduling problems.

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 \leftarrow an initial complete assignment for *csp*

for *i* = 1 to *max_steps* **do**

if *current* is a solution for *csp* **then return** *current*

var \leftarrow a randomly chosen conflicted variable from *csp*.VARIABLES

value \leftarrow the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)

 set *var* = *value* in *current*

return *failure*

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. 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.

2.4.4 THE STRUCTURE OF PROBLEM:

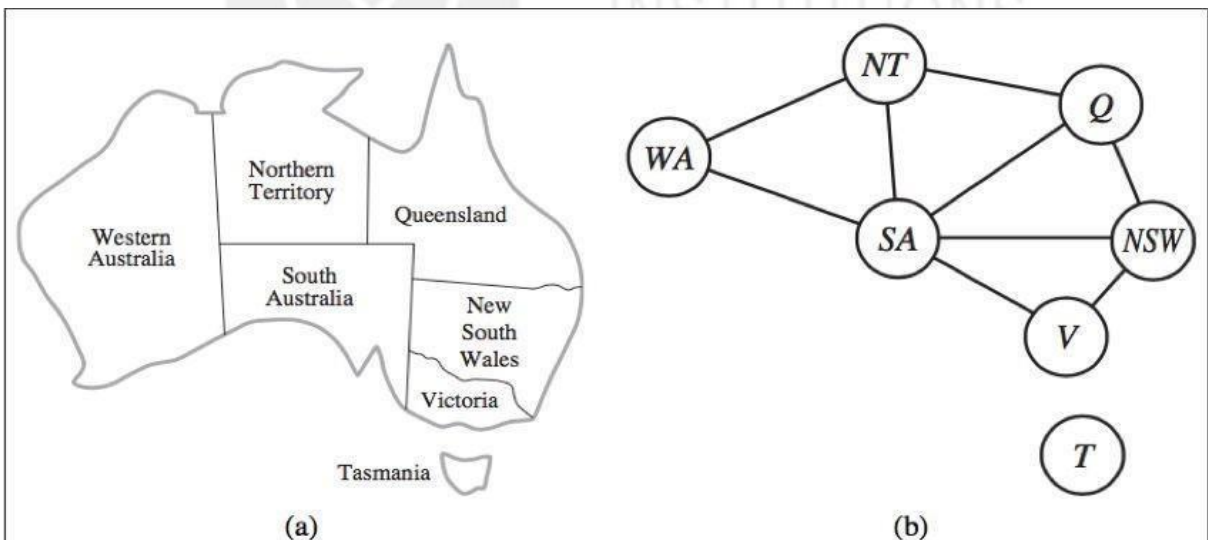


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

i) The structure of constraint graph

The structure of the problem as represented by the constraint graph can be used to find solution quickly.

e.g. The problem can be decomposed into 2 independent subproblems: Coloring T and coloring the mainland.

Tree: A constraint graph is a tree when any two variable are connected by only one path.

Directed arc consistency (DAC): A CSP is defined to be directed arc-consistent under an ordering of variables X_1, X_2, \dots, X_n if and only if every X_i is arc-consistent with each X_j for $j > i$.

By using DAC, any tree-structured CSP can be solved in time linear in the number of variables.

How to solve a tree-structure CSP:

- Pick any variable to be the root of the tree;
- Choose an ordering of the variable such that each variable appears after its parent in the tree. (topological sort)
- Any tree with n nodes has $n-1$ arcs, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for 2 variables, for a total time of $O(nd^2)$.
- Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value.
- Since each link from a parent to its child is arc consistent, we won't have to backtrack, and can move linearly through the variables.

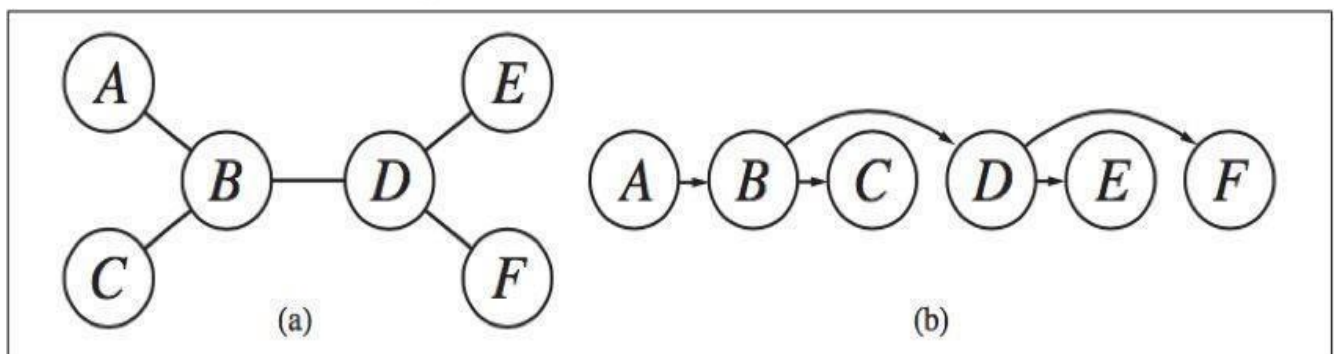


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

```

function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components  $X$ ,  $D$ ,  $C$ 

   $n \leftarrow$  number of variables in  $X$ 
  assignment  $\leftarrow$  an empty assignment
  root  $\leftarrow$  any variable in  $X$ 
   $X \leftarrow \text{TOPOLOGICALSORT}(X, \text{root})$ 
  for  $j = n$  down to 2 do
    MAKE-ARC-CONSISTENT(PARENT( $X_j$ ),  $X_j$ )
    if it cannot be made consistent then return failure
  for  $i = 1$  to  $n$  do
    assignment[ $X_i$ ]  $\leftarrow$  any consistent value from  $D_i$ 
    if there is no consistent value then return failure
  return assignment

```

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

There are 2 primary ways to reduce more general constraint graphs to trees:

1. Based on removing nodes;

The general algorithm:

1) Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S . S is called a cycle cutset.

2) For each possible assignment to the variables in S that satisfies all constraints on S ,

(a) remove from the domain of the remaining variables any values that are inconsistent with the assignment for S , and

(b) If the remaining CSP has a solution, return it together with the assignment for S .

Time complexity: $O(d \cdot c \cdot (n - c) d^2)$, c is the size of the cycle cut set.

Cutset conditioning: The overall algorithmic approach of efficient approximation algorithms to find the smallest cycle cutset.

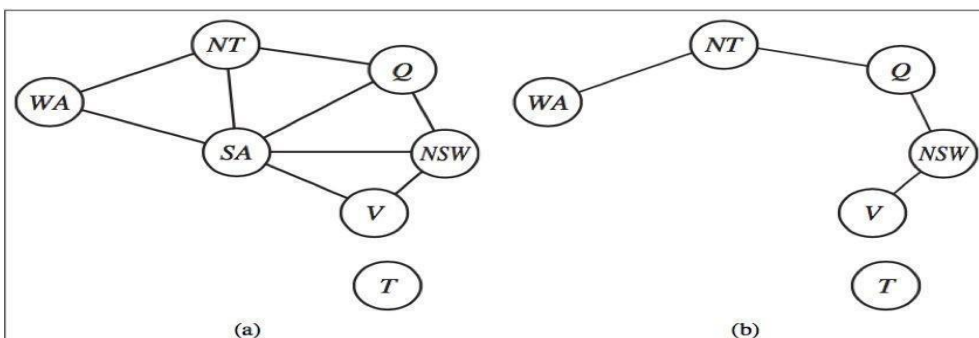


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of SA .

2. Based on collapsing nodes together

Tree decomposition: construct a tree decomposition of the constraint graph into a set of connected subproblems, each subproblem is solved independently, and the resulting solutions are then combined

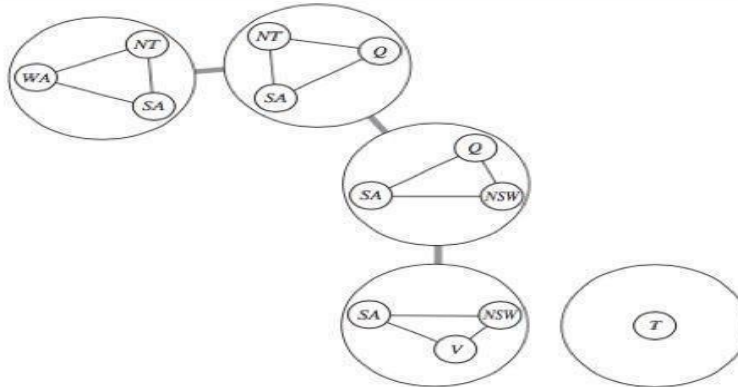


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

A tree decomposition must satisfy 3 requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If 2 variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in 2 subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

We solve each subproblem independently. If any one has no solution, the entire problem has no solution. If we can solve all the subproblems, then construct a global solution as follows:

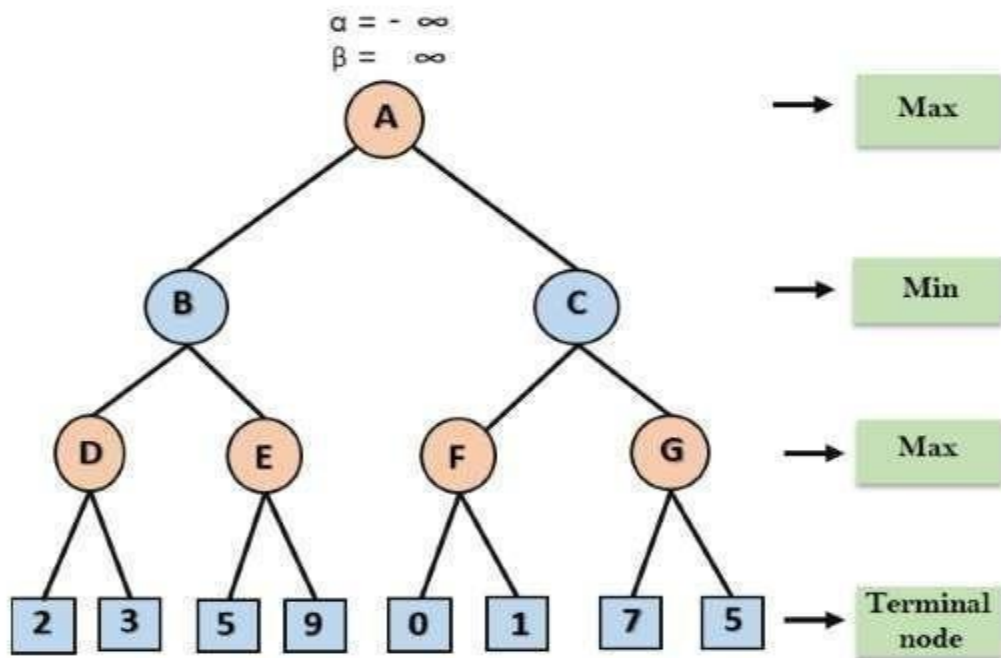
First, view each subproblem as a “mega-variable” whose domain is the set of all solutions for the sub-problem. Then, solve the constraints connecting the subproblems using the efficient algorithm for trees. A given constraint graph admits many tree decompositions;

In choosing a decomposition, the aim is to make the subproblems as small as possible.

- Tree width:
- The tree width of a tree decomposition of a graph is one less than the size of the largest subproblems.
- The tree width of the graph itself is the minimum tree width among all its tree decompositions.
- Time complexity: $O(ndw+1)$, w is the tree width of the graph.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. Cutset conditioning can reduce a general CSP to a tree-structured one and is quite efficient if a small cut set can be found. Tree decomposition techniques transform the CSP into a tree of subproblems and are efficient if the tree width of constraint graph is small.
- **2. The structure in the values of variables**
- By introducing a symmetry-breaking constraint, we can break the value symmetry and reduce the search space by a factor of $n!$.
- e.g. Consider the map-coloring problems with n colors, for every consistent solution, there is actually a set of $n!$ solutions formed by permuting the color names.(value symmetry)
- On the Australia map, WA, NT and SA must all have different colors, so there are $3!=6$ ways to assign.
- We can impose an arbitrary ordering constraint $NT < SA < WA$ that requires the 3 values to be in alphabetical order. This constraint ensures that only one of the $n!$ solution is possible: {NT=blue, SA=green, WA=red}. (symmetry-breaking constraint)

10.ASSIGNMENT 1- UNIT II

1. Solve Alpha-Beta Pruning



2. Solve tic-tac-toe using Backtracking search for CSP

11. PART A Q & A (WITH K LEVEL AND CO) UNIT 2

1. Define Problem Formulation. K1,C02

- Problem Formulation is the process of deciding what actions and states to consider, given a goal. Eg. Assume that the agent consider actions at the level of driving from one major town to another. Each state therefore corresponds to being a particular town.

2.What are the four components to define a problem? Define them.

(May/June 13) K2 ,C02

A problem can be defined by five components:

Initial state: that the agent starts in.

- Actions: description of the possible actions available to the agent
- Transition model: a description of what each action does.
- Goal test: determines whether a given state is a goal state.
- Path cost: function that assigns numeric cost to each path.

3. What is heuristic function? [Nov/Dec 2016] K1,C02

The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal. There is nothing magical about a heuristic function. It must use only information that can be readily obtained about a node.

4. What is the use of heuristic functions?[Apr/May 2008] K1,C02

A heuristic is a function, $h(n)$ defined on the nodes of a search tree, which serves as an estimate of the cost of the cheapest path from that node to the goal node. Heuristics are used by informed search algorithms such as Greedy best-first search and A^*

5. How does one characterize the quality of a heuristic?[May/Jun 2009] K2 ,C02

The performance of heuristic search algorithms depends on the quality of the heuristic function.

Good heuristic can sometimes be constructed by relaxing the problem definition, by pre computing solution costs for sub problems in a pattern database, or by learning from experience with the problem class.

6. Define a graph and a path. K2 ,C02

The state space forms a directed network or graph in which nodes are states and the links between the nodes are actions. A path in the state space is a sequence of states connected by a sequence of actions.

7. What is an optimal solution? K2 ,C02

A solution to a problem is an action sequence that leads from the initial state to the goal state. A solution quality is measured by the path cost function and an optimal solution has the lowest path cost among all solutions.

8. Indicate the role of Heuristics in guiding a search.[Nov/Dec 2008] K1,C02

The path cost from the current state to goal state is calculated, to select the minimum path cost as the next state.

9. Define abstraction (May/June 12) K1 ,C02

The process of removing detail from a representation is called abstraction. The abstraction is valid if any abstract solution can be expanded into a solution in the more detailed world. The abstraction is useful if carrying out each of the actions in the solution is easier than the original problem. The choice of good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

10. Define the effect of heuristic accuracy on performance. (Nov/Dec 13) K2 ,C02

One way to characterize the quality of the heuristic is the effective branching factor b^* . If the total number of nodes generated by A^* for a particular problem is N , and the solution depth is d , then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N+1$ nodes. Thus $N+1=1+b^*+(b^*)^2+\dots$

$+ (b^*)^d$.

11. How does the operation of an off-line search differ from that of an on-line search?[Nov/Dec 2008] K2 ,C02

❁ **Offline search:** They compute a complete solution before setting foot in the real world, and then execute the solution without recourse to their percepts.

❁ **Online search:** Agents operate by interleaving computation and action: first it takes an action, and then it observes the environment and computes the next action.

12. What is the difference between uninformed and informed search strategies?[Nov/Dec 2008] K1 ,C02

UNINFORMEDSEARCH(BLIND SEARCH)	INFORMED SEARCH(HEURISTIC SEARCH)
No information about the number of steps(or) path cost from the current state to goal state	The path cost from the current state to goal state is calculated, to select the minimum path cost as the next state
Less effective in search method	More effective
Problem to be solved with the given information	Additional information can be added as assumption to solve the problem
E.g. a) Breadth first search b) Uniform cost search c) Depth first search d) Depth limited search e) Interactive deepening search f) Bi-directional search	E.g. a) Best first search b) Greedy search c) A* search

13. State the significance of using heuristic functions?[Nov/Dec 2011] K2 ,C02

❁ The path cost from the current state to goal state is calculated, to select the minimum path cost as the next state.

❁ Find the shortest solution using heuristic function that never over estimates the number of steps to the goal.

14. List the criteria to measure the performance of search strategies.[May/Jun 2014] K2,C02

The criteria to measure the performance of search strategies are:

- ✿ Completeness: is the algorithm guaranteed to find a solution when there is one?
- ✿ Optimality: does the strategy find the optimal solution?
- ✿ Time complexity: how long does it take to find a solution?
- ✿ Space complexity: how much memory is needed to perform the search?

15. How to improve the effectiveness of a search-based problem-solving technique?[Apr/May 2008] K1,C02

- ✿ Goal formulation
- ✿ Problem formulation
- ✿ Search
- ✿ Solution
- ✿ Execution phase

16. Will Breadth-First Search always find the minimal solution. Why?[April/May 2018] K2,C02

Yes, It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors, This method provides shortest path to the solution.

17. What is a constraint satisfaction problem?[Nov/Dec 2015][Apr/May 2008] K2,C02

A Constraint Satisfaction problem (or CSP) is defined by a set of variables X_1, X_2, \dots, X_n , and a set of constraints, C_1, C_2, \dots, C_m . Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \dots\}$. A solution to a CSP is a complete assignment that satisfies all the constraints.

18. What is the use of online search agents in unknown environment?[Nov/Dec 2007] K3,C02

Online search agents operate by interleaving computation and action: first it takes an action, and then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains and stochastic domains. Online search is a necessary idea for an exploration problem, where the states and actions are unknown to the agent.

19. Formally define Game as a kind of search problems.[May/Jun 2009] K1 ,C02

Consider of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state.

20. List some of the uninformed search techniques. [APRIL/MAY 2017] K2 ,C02

- ✿ Uninformed Search Techniques:
Depth-first Search
- ✿ Breadth-first Search
- ✿ Iterative Deepening
- ✿

21. What are optimization problems? K2 ,C02

In optimization problems, the aim is to find the best state according to an objective function the optimization problem is then: Find values of the variables that minimize or maximize the objective function while satisfying the constraints.

22. What's the difference between a world state, a state description, and a search node?Why is this distinction useful? K21,C02

A world state is what the world looks like, while a state description tells us about the state in every detail, and a search node is a data representation of the search. So the world state is the state itself, the state description is information on it, and the search node is the search data.

23. Define Monotonicity K1,C02

Monotonicity (consistency): In search tree any path from the root, the f-cost never decreases. This condition is true for almost all admissible heuristics. A heuristic which satisfies this property is called monotonicity.

24. What is a ridge?[May/June 2016] K2,C02

Ridges are a challenging problem for hill climbers that optimize in continuous spaces. Because hill climbers only adjust one element in the vector at a time, each step will move in an axis-aligned direction.

25. What are the advantages of Breadth First Search? [NOV/DEC 2017, APR/MAY 2018] K2,C02

BFS will not get trapped exploring a blind alley. This contrast to the DFS which may follow a single unfruitful path for a very long time, perhaps forever before the path actually terminates in a state that has no successors.

If there is a solution, then BFS is guaranteed to find it. Furthermore, if there are multiple solutions then a minimal solution will be found.

26. Write Generate and Test algorithm. [MAY / JUNE 2016] K23,C02

Generate a possible solution. For some problems this means generating a particular point in the problem space. For others, it means generating a path from a start state.

Test to see if this is actually a solution by comparing the chosen point or the end point of the chosen path to the set of acceptable goal states.

If a solution has been found, quit otherwise return step1

27. What is the difference between Simple Hill Generate and Test algorithm Climbing [MAY/ JUNE 2016] K23,C02

The key difference between Simple Hill Climbing and Generate and Test algorithm is the use of an evaluation function as a way to inject task-specific knowledge into the control process.

28. What is A* search? K1 ,C02

A * search is the most widely-known form of best-first search. It evaluates the nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal: $f(n) = g(n) + h(n)$

Where $f(n)$ = estimated cost of the cheapest solution through n . $g(n)$ is the path cost from the start node to node n .

$h(n)$ = heuristic function

A * search is both complete and optimal.

29. Give a classification of CSP with respect to constraints. K2

,C02 The classification of CSP with respect to constraints are as follows

- Unary constraint CSP restricts the value of a single variable.
- Binary constraint CSP relates two variables
- Global constraint CSP involves an arbitrary number of variables.

30. List out the types of assignment in CSP problem and explain each. K2

,C02

Consistent or legal assignment an assignment that does not violate any

constraints

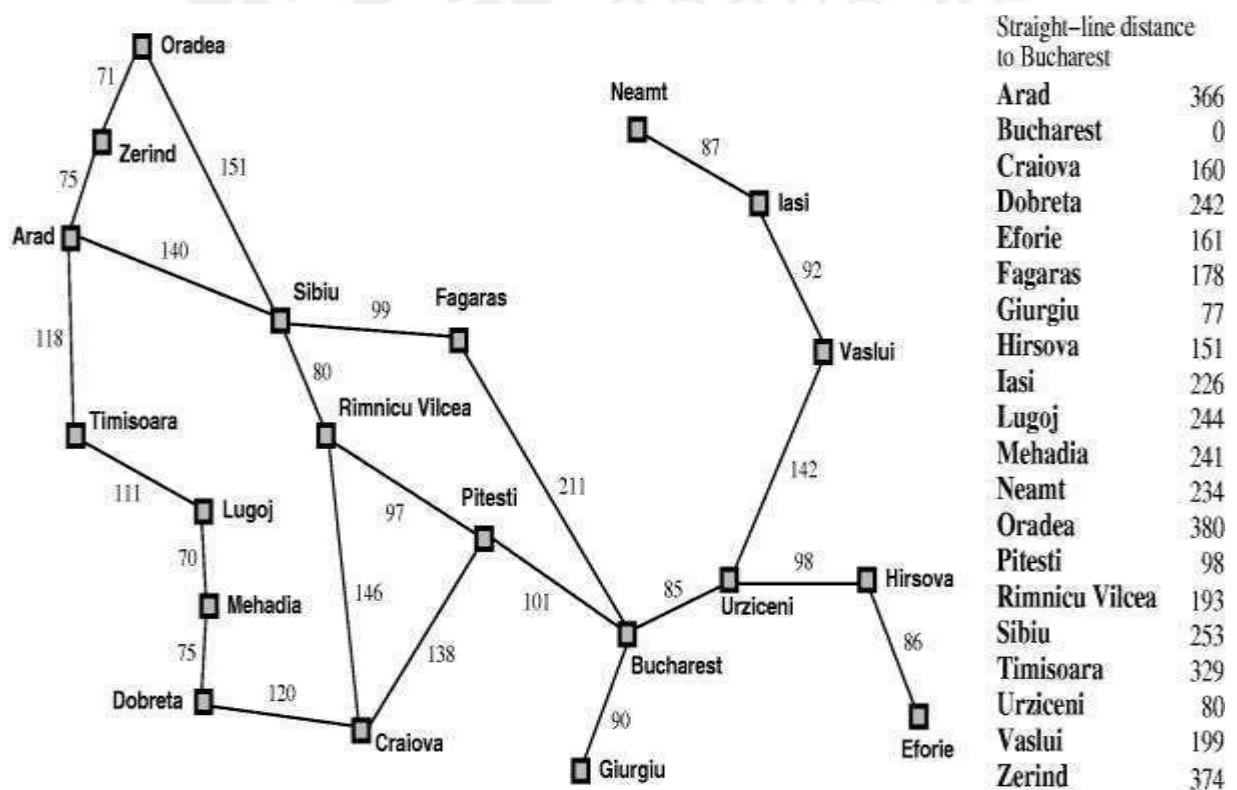
Complete assignment is one in which every variable is assigned and a solution

to a CSP is consistent.

Partial assignment assigns values to only some of the variables.

12. PART B Q s (WITH K LEVEL AND CO) UNIT 2

1. a. Explain Hil climbing in detail. (8) K2 ,C02
b. Explain A* search in detail. (8) (13) K2 ,C02
2. a. Explain simulated annealing search in detail. (8) K2 ,C02
b. Explain Memory bounded heuristic search in detail. (8) K1 ,C02
3. Define informed search? Explain BFS, A*, RBFS with the folowing example map and distance. (13)(15) K2 ,C02
4. Elaborate in detail about alpha beta pruning with example (13) K1 ,C02
5. a. Explain MinMax search in gaming with example (8) K1 ,C02
b. Write in detail the learning of an agent in online search method(8) K2 ,C02
7. What is CSP? Explain CSP and constraint propagation with graph coloring problem.(13) K1 ,C02
8. Define CSP. Elaborate in detail about use of constraint propagation and backtracking in CSP.(13) K1 ,C02
9. Elaborate in detail about different hil climbing algorithms.(8)(13) K3 ,C02



13. Supportive online Certification courses

1. Udacity: **Artificial Intelligence**

<https://www.udacity.com/course/ai-artificial-intelligence-nanodegree--nd898>

2. NPTEL: **Artificial Intelligence**

<https://nptel.ac.in/courses/106/102/106102220/>



14.Real time Applications in day to day life and to Industry

AI Search Algorithms Implementations-A* search

- ✿ In Naboo Planet the R2-D2 droid is serving her Queen Amidala and is successful in stealing some important documents containing secrets of the castle of Dark Lord Darth Vader located on the volcanic planet of Mustafar.
- ✿ As soon as Dark Lord finds this out about this, he sends his army after R2D2 to recover the documents from him.
- ✿ Fearing the Darth's Army, R2D2 hides in a Cave. While entering the cave R2D2 has found a map of the cave and It knows that it is at grid location 0 and needs to reach grid 61 to go out of the Cave.

	0	1	2	3	4	5	6	7
0	START 0	8	16	24	32	40	48	56
1	1	9	17	25	33	41	49	57
2	2	10	18	26	34	42	50	58
3	3	11	19	27	35	43	51	59
4	4	12	20	28	36	44	52	60
5	5	13	21	29	37	45	53	FINISH 61
6	6	14	22	30	38	46	54	62
7	7	15	23	31	39	47	55	63

❁ Darth's Army knows that R2D2 is hiding in the cave so they have set up the explosives in the cave that will go off after a certain time. Let us use our knowledge of AI and the AI search algorithms and help R2D2 to search his path out of the Cave and successfully get the stolen documents to his queen.

❁ R2D2 has to follow the following rules for Searching his path out of the grid. This logic is hardcoded in his memory.

❁ The (x, y) coordinates of each node are defined by the column and the row shown at the top and left of the maze, respectively. For example, node 13 has (x, y) coordinates (1, 5).

❁ Process neighbours in increasing order. For example, if processing the neighbours of node 13, first process 12, then 14, then 21.

❁ Use a priority queue for frontier. Add tuples of (priority, node) to the frontier. For example, when performing UCS and processing node 13, add (15, 12) to the frontier; then (15, 14), then (15, 21), where 15 is the distance (or cost) to each node.

❁ When removing nodes from the frontier (or popping off the queue), break ties by taking the node that comes first lexicographically. For example, if deciding between (15, 12), (15, 14) and (15, 21) from above, choose (15, 12) first

(because $12 < 14 < 21$).

❁ A node is considered visited when it is removed from the frontier (or popped off the queue). Only valid moves are going horizontal and vertical (not diagonal).

❁ It takes 1 minute to explore a single node. The time to escape the maze will be the sum of all nodes explored, not just the length of the final path. All edges have cost 1.

❁ A* Search algorithm is a "smart" search algorithm and works faster as compared to other conventional search algorithms.

❁ The code used in this article and the complete working example can be found in the git repository below:

<https://github.com/fakemonk1/AI-Search-Algorithms-Implementations>

15. ASSESSMENT SCHEDULE

Tentative schedule for the Assessment During 2022-2023 ODD semester

S.NO	Name of the Assessment	Start Date	End Date	Portion
1	Unit Test 1			UNIT 1
2	IAT 1	09.09.2023	15.09.2023	UNIT 1 & 2
3	Unit Test 2			UNIT 3
4	IAT 2	26.10.2023	01.11.2023	UNIT 3 & 4
5	Revision 1			UNIT 5 , 1 & 2
6	Revision 2			UNIT 3 & 4
7	Model	15.11.2023	25.11.2023	ALL 5 UNITS

16. PRESCRIBED TEXT BOOKS & REFERENCE BOOKS

TEXT BOOKS:

1. Peter Norvig and Stuart Russel, Artificial Intelligence: A Modern Approach, Pearson, Fourth Edition, 2020.
2. Bratko, Prolog: Programming for Artificial Intelligence, Fourth edition, Addison-Wesley Educational Publishers Inc., 2011.

REFERENCES:

1. Elaine Rich, Kevin Knight and B. Nair, Artificial Intelligence 3rd Edition, McGraw Hill, 2017.
2. Melanie Mitchell, Artificial Intelligence: A Guide for Thinking Humans. Series : Pelican Books, 2020
3. Ernest Friedman-Hill, Jess in action, Rule-Based Systems in Java, Manning Publications, 2003
4. Nils J. Nilsson, The Quest for Artificial Intelligence, Cambridge University Press, 2009
5. Dan W. Patterson Introduction to Artificial Intelligence and expert systems, 1st Edition by Patterson, Pearson, India, 2015

17. MINI PROJECT SUGGESTION

Word Break Problem using Backtracking

Given a valid sentence without any spaces between the words and a dictionary of valid English words, find all possible ways to break the sentence in individual dictionary words.

Example

Consider the following dictionary

{ i, like, sam, sung, samsung, mobile, ice, and, cream, icecream, man, go, mango }

Input: "ilikesamsungmobile"

Output: i like sam sung mobile

i like samsung mobile

Input: "ilikeicecreamandmango"

Output: i like ice cream and man go

i like ice cream and mango

i like icecream and man go

i like icecream and mango

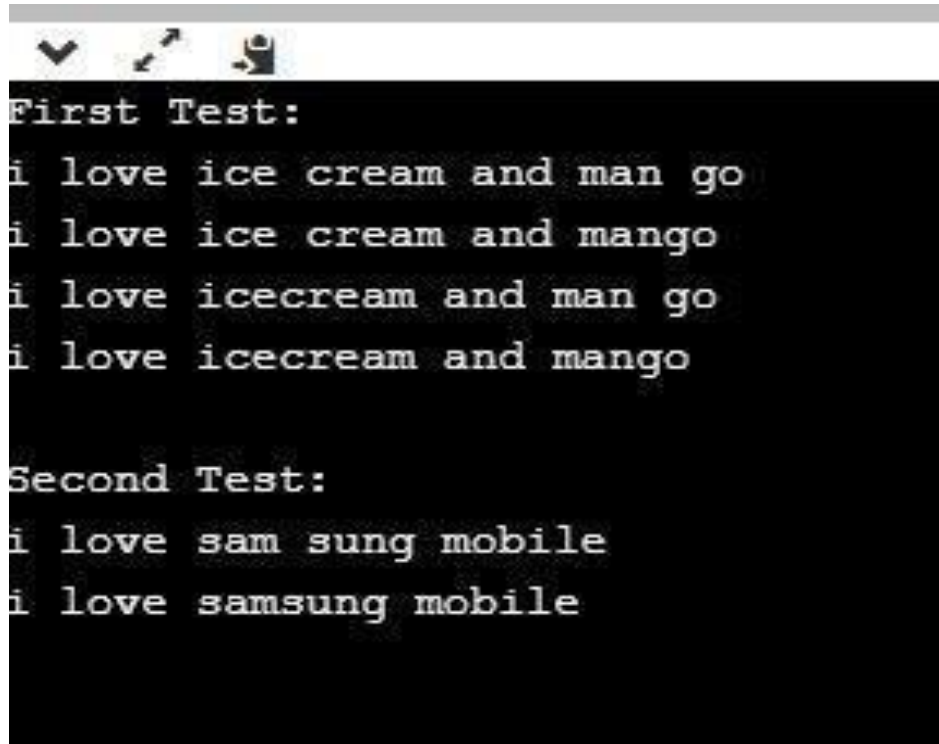
The Dynamic Programming solution only finds whether it is possible to break a word or not. Here we need to print all possible word breaks. We start scanning the sentence from left. As we find a valid word, we need to check whether the rest of the sentence can make valid words or not. Because in some situations the first found word from the left side can leave a remaining portion which is not further separable.

So, in that case, we should come back and leave the currently found word and keep on searching for the next word. And this process is recursive because to find out whether the right portion is separable or not, we need the same logic.

We will use recursion and backtracking to solve this problem. To keep track of the found words we will use a stack. Whenever the right portion of the string does not make valid words, we pop the top string from the stack and continue finding.

This concept can also implemented using C++,Python code.

OUTPUT



```
First Test:
i love ice cream and man go
i love ice cream and mango
i love icecream and man go
i love icecream and mango

Second Test:
i love sam sung mobile
i love samsung mobile
```



Thank you

Disclaimer:

This document is confidential and intended solely for the educational purpose of RMK Group of Educational Institutions. If you have received this document through email in error, please notify the system manager. This document contains proprietary information and is intended only to the respective group / learning community as intended. If you are not the addressee you should not disseminate, distribute or copy through e-mail. Please notify the sender immediately by e-mail if you have received this document by mistake and delete this document from your system. If you are not the intended recipient you are notified that disclosing, copying, distributing or taking any action in reliance on the contents of this information is strictly prohibited.