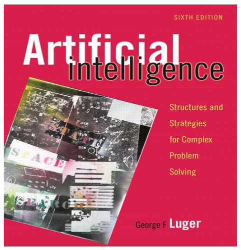
3 Structures and Strategies For Space State Search



George F Luger

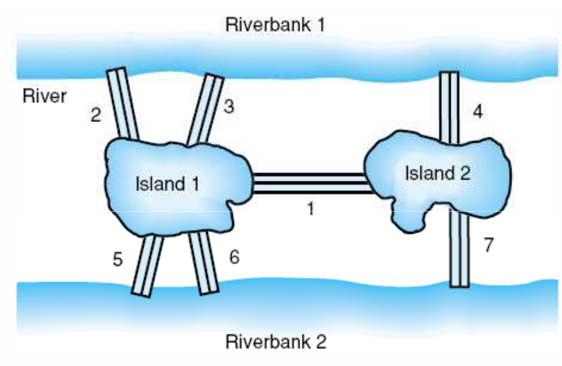
ARTIFICIAL INTELLIGENCE *6th edition*

Structures and Strategies for Complex Problem Solving

Introduction

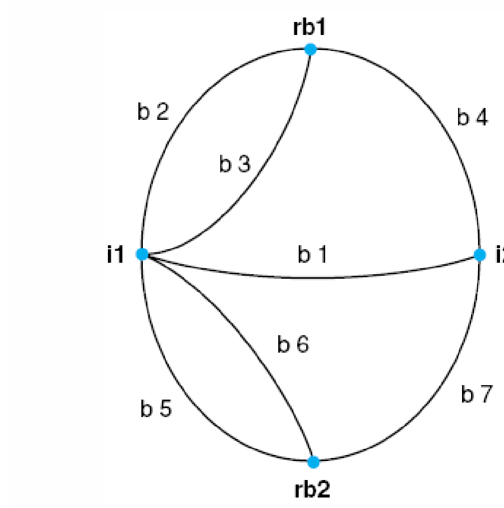
- Predicate calculus
  - provides a means to describe facts and relations in a problem domain mathematically
  - Uses rules to infer new knowledge
- The inference rules define a space that is searched to find a problem solution
- State space search theory provides a visual approach for finding a solution to the space search problem
  - Represent problem as a state graph
  - Use graph theory to analyze the structure and complexity of the problem and search procedure

Figure 3.1: The city of Königsberg.



Is there a walk around the city that crosses each bridge exactly once?

Figure 3.2: Graph of the Königsberg bridge system.



The graph preserves the structure of bridges, while ignoring extraneous features such as bridge lengths, distances, etc.

Bridges of Königsberg Problem

- Alternatively, the Königsberg bridge system can be represented using predicate calculus – each arc in the graph is represented by the connect predicate:
  - connect(i1, i2, b1)
  - connect(rb1, i1, b2)
  - connect(rb1, i1, b3)
  - connect(rb1, i2, b4)
  - connect(rb2, i1, b5)
  - connect(rb2, i1, b6)
  - connect(rb2, i2, b7)

Bridges of Königsberg Problem

- However the structure of the problem can be visualized more directly in the graph representation
- Euler noted that unless a graph contains either zero or two nodes of odd degree, the walk is impossible (the degree of a node is the number of arcs connecting the node)

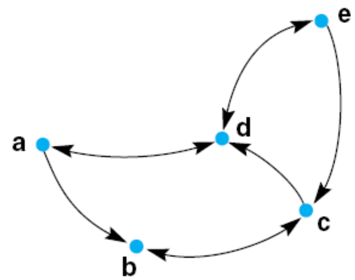
Graph Theory

- A graph is a set of nodes or states and a set of arcs that connect the nodes
- A labeled graph has one or more descriptors (labels) attached to each node
- In a state space graph, the descriptors identify states in a problem-solving process
- The arcs may also be labeled
- Arc labels indicate named relationships or attach weights to arcs

Graph Theory

- A graph is directed if arcs have directions (Fig. 3.3)
- A path through a graph connects a sequence of nodes through successive arcs ([a, b, c, d] in Fig. 3.3)
- A rooted graph has a unique node, root, such that there is a path from the root to all nodes within the graph (Fig. II.5)
- A tree is a graph in which two nodes have at most one path between them (Fig. 3.4)

Figure 3.3: A labeled directed graph.



Nodes = {a,b,c,d,e}

Arcs = {(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}

Figure II.5: Portion of the state space for tic-tac-toe.

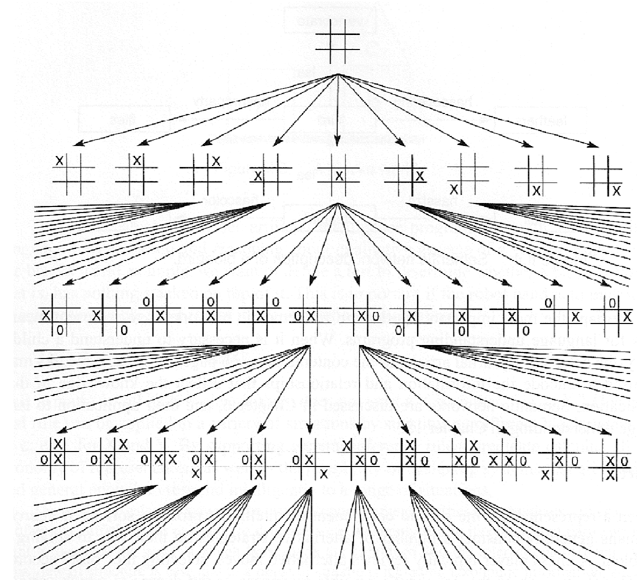
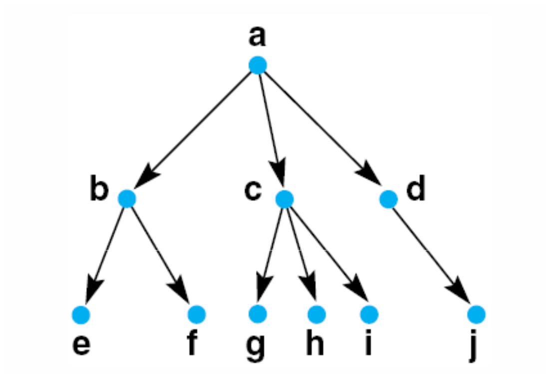


Figure 3.4: A rooted tree, exemplifying family relationships.



b is a parent of **e** and **f**
e and **f** are children of b and siblings of each other
a and **c** are ancestors of **g, h** and **i**
g, h and **i** are descendants of **a** and **c**

DEFINITION

GRAPH

A graph consists of:

A set of *nodes* $N_1, N_2, N_3, \ldots, N_n, \ldots$, which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc (N_3, N_4) connects node N_3 to node N_4. This indicates a direct connection from node N_3 to N_4 but not from N_4 to N_3, unless (N_4, N_3) is also an arc, and then the arc joining N_3 and N_4 is undirected.

If a directed arc connects N_j and N_k, then N_j is called the *parent* of N_k and N_k, the *child* of N_j. If the graph also contains an arc (N_j, N_l), then N_k and N_l are called *siblings*.

A *rooted* graph has a unique node N_S from which all paths in the graph originate. That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

An ordered sequence of nodes $[N_1, N_2, N_3, \ldots, N_n]$, where each pair N_i , N_{i+1} in the sequence represents an arc, i.e., (N_i, N_{i+1}), is called a *path* of length $n - 1$.

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

A path that contains any node more than once (some N_j in the definition of path above is repeated) is said to contain a *cycle* or *loop*.

A *tree* is a graph in which there is a unique path between every pair of nodes. (The paths in a tree, therefore, contain no cycles.)
The edges in a rooted tree are directed away from the root. Each node in a rooted tree has a unique parent.

Two nodes are said to be *connected* if a path exists that includes them both.

The Finite State Machine (FSM)

- It is a finite, directed, connected graph
- It has a set of states, a set of input values, and a state transition function describing the effect of input stream on the states of the graph
- It is primary used to recognize components of a formal language (often “words” made from characters of an “alphabet”)

DEFINITION

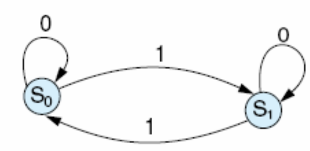
FINITE STATE MACHINE (FSM)

A *finite state machine* is an ordered triple (S, I, *F*), where:

S is a finite set of *states* in a connected graph $s_1, s_2, s_3, \ldots, s_n$.

I is a finite set of *input* values $i_1, i_2, i_3, \ldots, i_m$.

F is a state transition function that for any $i \in I$, describes its effect on the states S of the machine, thus $\forall\, i \in I, F_i : (S \rightarrow S)$. If the machine is in state s_j and input i occurs, the next state of the machine will be $F_i\,(s_j)$.



(a)

	0	1
S_0	S_0	S_1
S_1	S_1	S_0

(b)

Fig 3.5 (a) The finite state graph for a flip flop and (b) its transition matrix.

DEFINITION

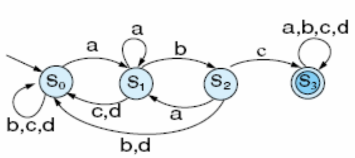
FINITE STATE ACCEPTOR (MOORE MACHINE)

A *finite state acceptor* is a finite state machine (S, I, *F*), where:

$\exists\, s_0 \in S$ such that the input stream starts at s_0, and

$\exists\, s_n \in S$, an *accept* state. The input stream is accepted if it terminates in that state. In fact, there may be a set of accept states.

The finite state acceptor is represented as $(S, s_0, \{s_n\}, I, F)$



(a)

	a	b	c	d
→ S_0	S_1	S_0	S_0	S_0
S_1	S_1	S_2	S_0	S_0
S_2	S_1	S_0	S_3	S_0
S_3	S_3	S_3	S_3	S_3

(b)

Fig 3.6 (a) The finite state graph and (b) the transition matrix for string (*abc*) recognition example

The State Space Representation of Problems

- A **state space** representation of a problem is a directed graph where the nodes correspond to partial problem solution states, and the arcs correspond to steps in a problem solving process
- State space search characterizes problem solving as a process of finding a solution path from a start state to a goal state

DEFINITION

STATE SPACE SEARCH

A *state space* is represented by a four-tuple [N,A,S,GD], where:

N is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.

A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.

S, a nonempty subset of N, contains the start state(s) of the problem.

GD, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:

1. A measurable property of the states encountered in the search.
2. A property of the path developed in the search, for example, the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

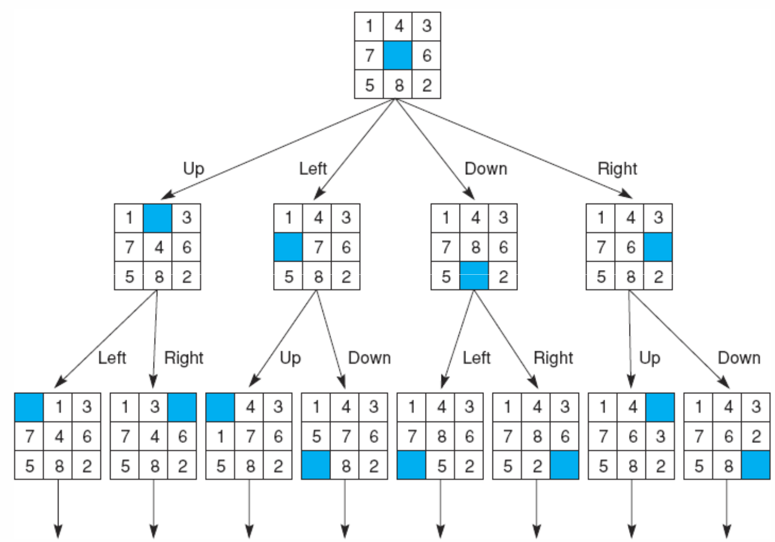
Tic-Tac-Toe Example

- The start state (**S**) is an empty board (Figure II.5)
- The goal description (**GD**) is a board state having three Xs in a row, column, or diagonal
- The path from the start state to a goal state gives the series of moves in a winning game
- The states (**N**) of the space are all the different configurations of Xs and Os the game can have (3^9)
- Arcs (**A**) corresponds to legal moves of the game, alternating between placing an X and an O in an unused location
- Total number of paths = 9!

The 8-Puzzle Example

- 8 differently numbered tiles are fitted into 9 spaces. One space is left blank so that tiles can be moved around to form different patterns
- The goal is to find a series of moves of tiles to place the board in a goal configuration
- Number of states of the space = 9!

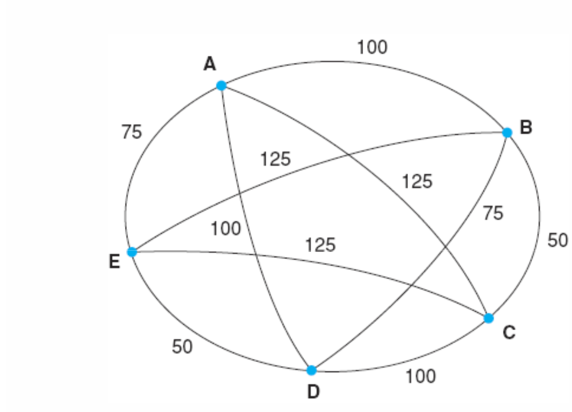
Fig 3.8 State space of the 8-puzzle generated by "move blank" operations



The Traveling Salesperson Example

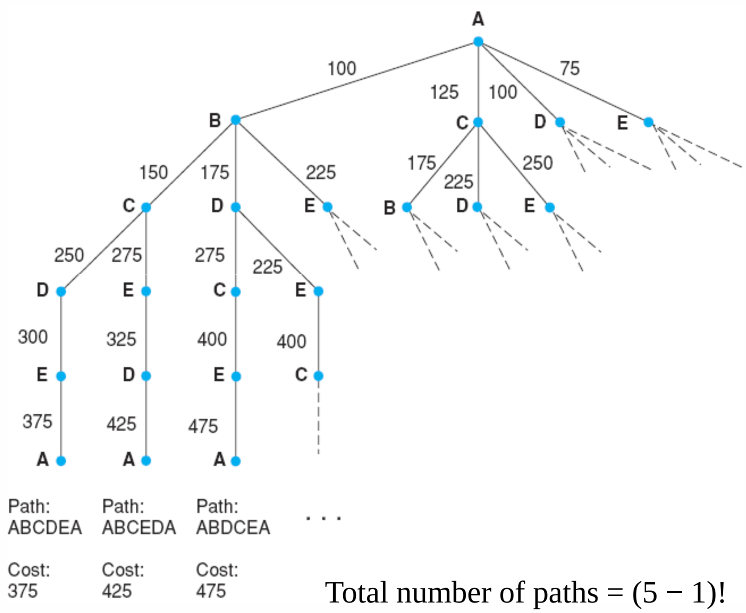
- A salesperson needs to visit five cities and then return home
- The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and return to the starting city

Fig 3.9 An instance of the travelling salesperson problem



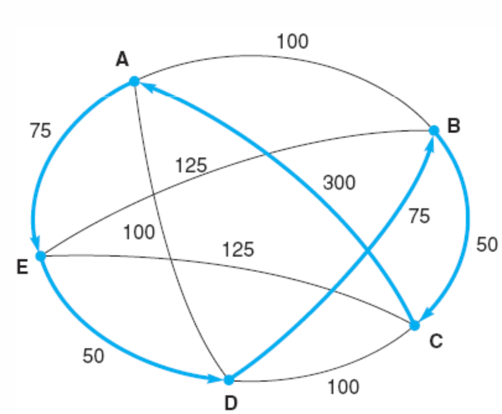
- The nodes of the graph represent the cities.
- The labels on the arcs represent the distances.
- Assume the salesperson lives in city A.

Fig 3.10 Search for the travelling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.



Total number of paths = (5 − 1)!

Fig 3.11:
An instance of the travelling salesperson problem with the nearest neighbor path in bold. Note this path (A, E, D, B, C, A), at a cost of 550, is not the shortest path. The comparatively high cost of arc (C, A) defeated the heuristic. (**The rule is "go to the closest unvisited city".**)



Strategies for State Space Search

- A state space may be searched in two directions
  - From the given data of a problem toward a goal (**data-driven** search or forward chaining)
  - From a goal back to the data (**goal-driven** reasoning or backward chaining)

Strategies for State Space Search

- Data-driven reasoning
  - Takes the facts and applies the rules or legal moves to produce new facts
  - New facts are used by the rules to generate more new facts
  - This process continues until it generates a path that leads to a goal condition

Strategies for State Space Search

- Goal-driven reasoning
  - Takes the goal, finds the rules that produce the goal and determines what conditions must be true to use them
  - The conditions become the new goal for search
  - Search continues backward through successive rules and subgoals until it works back to the facts of the problem

Comparison of Search Strategies

- Both data-driven and goal-driven problem solver search the same state space graph
- The order and actual number of states searched can be different
- The preferred strategy is determined by the properties of the problem itself
- For example:
  - Prove "I am a descendent of Thomas Jefferson."

Fig 3.12 State space in which goal-directed search effectively prunes extraneous search paths.

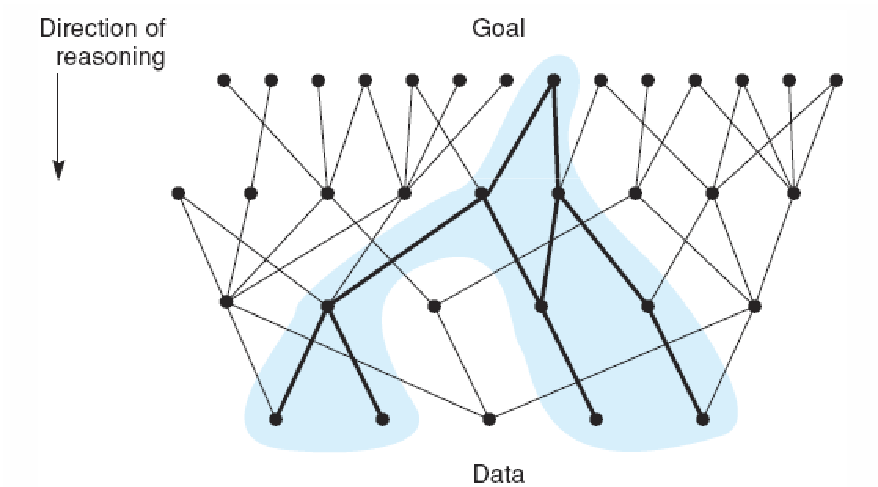
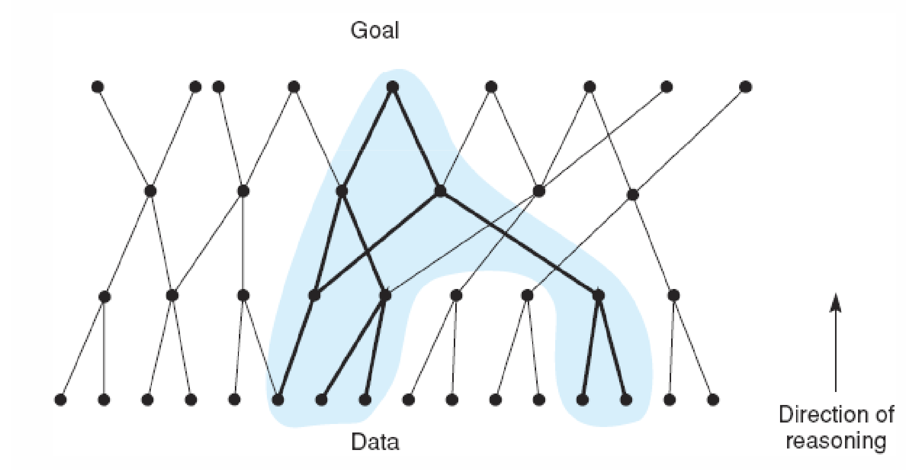


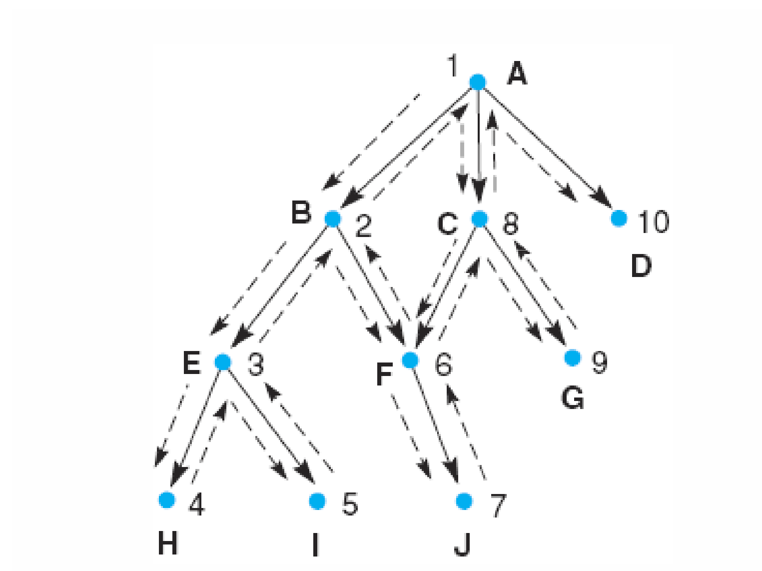
Fig 3.13 State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.



Implementing Graph Search

- The problem solver must find a path from a start state to a goal state through the state space graph
- The sequence of arcs in this path correspond to the ordered steps of the solution
- **Backtracking** is a technique for systematically trying all paths through a state space
  - Begins at a start state and pursues a path until it reaches either a goal or a “dead end”
  - If it reaches a dead end, it backtracks to the most recent node on the path with unexamined siblings and continues searching

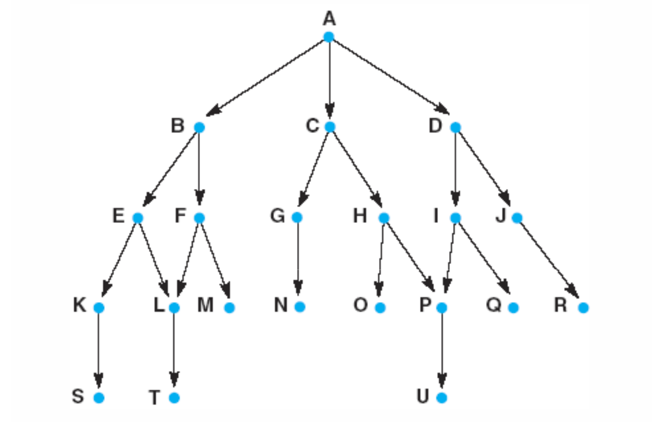
Fig 3.14 Backtracking search of a hypothetical state space.



Depth-First and Breath-First Search

- Data-driven and goal-driven specifies the search direction; **depth-first** and **breath-first** determines the search order
- Depth-first search examines the children and the descendants of a state before examining the siblings; it goes deeper whenever possible
- Breath-first search explores the space in a level-by-level fashion. It moves to the next level only when no more states can be explored

Fig 3.15 Graph for breadth - and depth - first search examples



Depth-first: A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R

Breath-first: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U

Fig 3.17 Breadth-first search of the 8-puzzle, showing order in which states were searched.

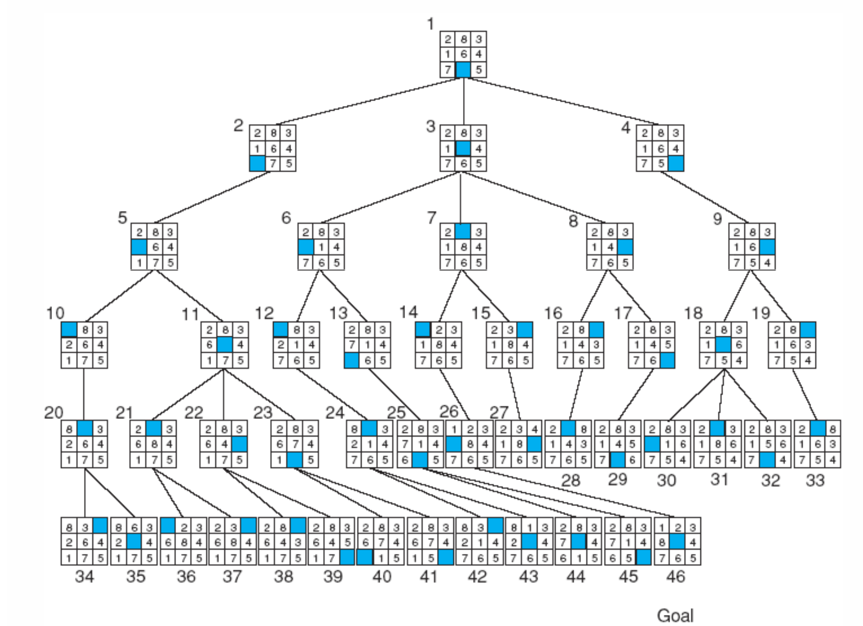
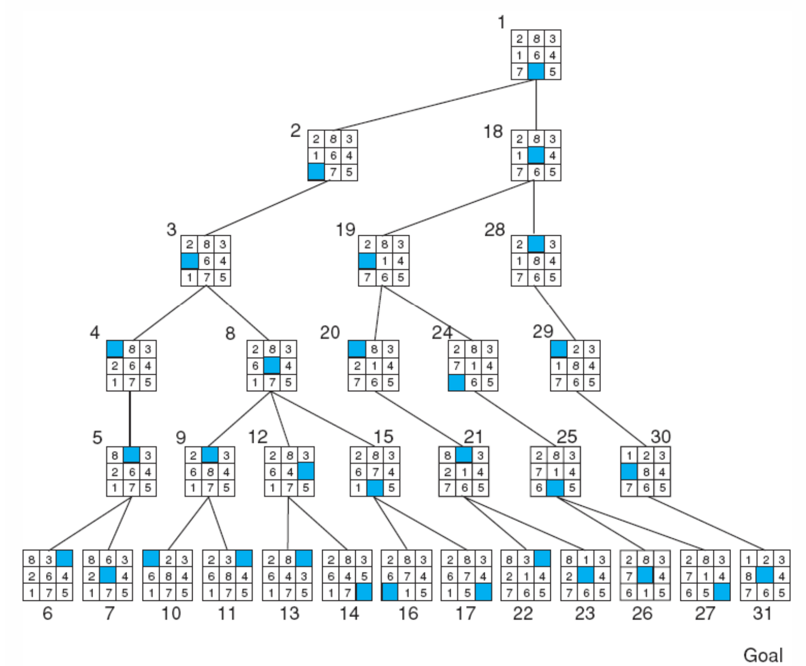


Fig 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.



Depth-First and Breath-First Search

- Breath-first search
  - Because it always examines all nodes at level n before proceeding to level n+1, it always finds the shortest path to a goal node
  - All unexpanded nodes for each level of search must be kept in memory
  - If states have a high average number of children, the combinatorial explosion of states may prevent the algorithm from finding a solution using available memory

Depth-First and Breath-First Search

- Depth-first search
  - If the solution path is long, it will not waste time searching a large number of “shallow” states in the graph
  - Can get “lost” deep in a graph, missing shorter paths to a goal or even becoming stuck in an infinitely long path that does not lead to a goal
  - Much more efficient use of search spaces because it does not need to keep all the nodes at a given level on the memory (it retains only the children of a single state)

Depth-First Search with Iterative Deepening

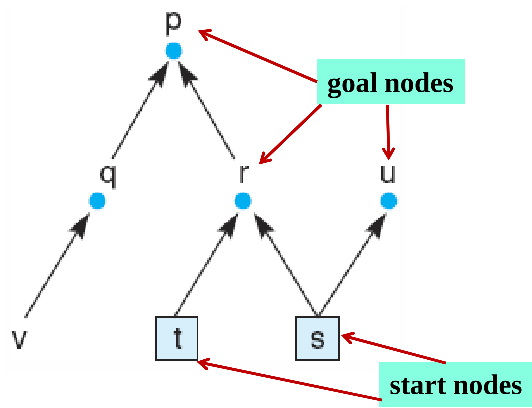
- A nice compromise on the trade-offs of the depth-first and breadth-first search
- Performs a depth-first search of the space with depth bound of 1
- If it fails to find a goal, it performs another depth search with a depth bound of 2
- This continues until a goal is found
- It is guaranteed to find a shortest path to a goal
- It uses less memory space for storing the states

State Space Description of a Logic System

- Symbols and predicates in propositional and predicate calculus can be represented using the nodes of a state space graph
- Inference rules can be described by the arcs between states
- Problems in the predicate calculus may be solved by searching the state space

Fig 3.20 State space graph of a set of implications in the propositional calculus.

$q \rightarrow p$
$r \rightarrow p$
$v \rightarrow q$
$s \rightarrow r$
$t \rightarrow r$
$s \rightarrow u$
s
t



Determining if a proposition is true requires the finding of a path from a start node to the proposition

And/Or Graph

- In previous example all assertions were in the form of $q \rightarrow p$
- And/Or graph is an extension to the basic graph (see Graph Theory in the beginning)
- It allows logic operators **or** and **and** to be represented in the graph
  - **and** node: $q \wedge r$
  - **or** node: $q \vee r$

Fig 3.21 And/or graph of the expression $q \wedge r \rightarrow p$
Both q and r must be true for p to be true.

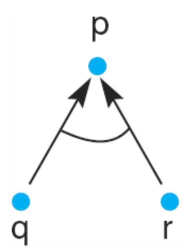
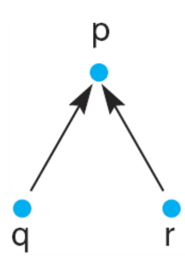
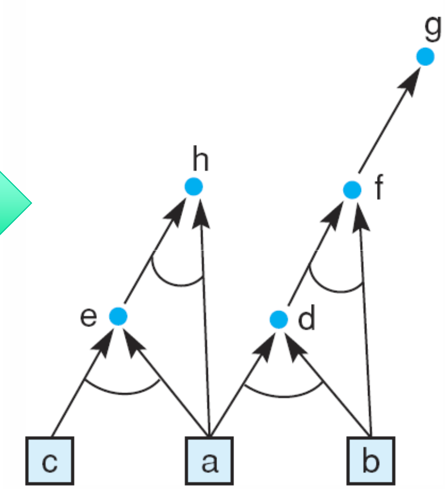


Fig 3.22 And/or graph of the expression $q \vee r \rightarrow p$
The truth of either q or r is sufficient to prove p is true.



And/Or Graph Search Example

a
b
c
$a \wedge b \rightarrow d$
$a \wedge c \rightarrow e$
$b \wedge d \rightarrow f$
$f \rightarrow g$
$a \wedge e \rightarrow h$

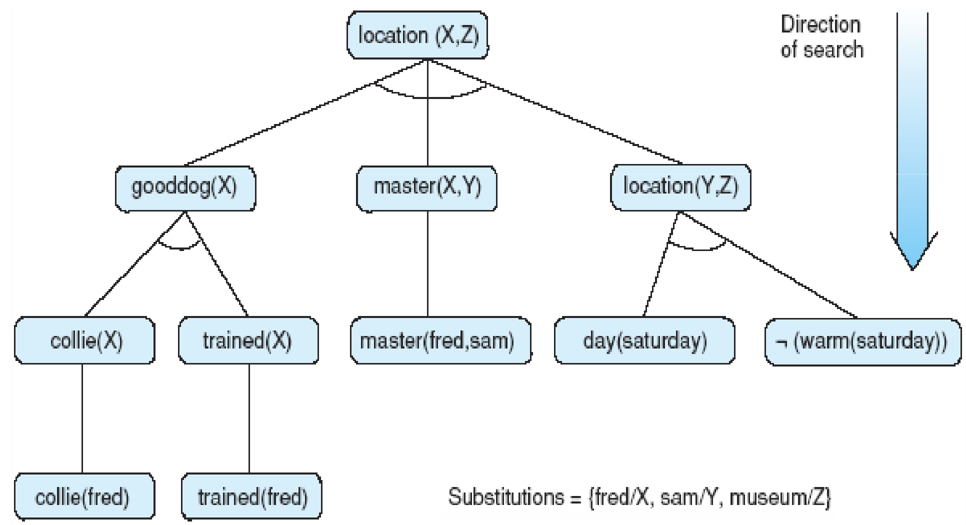


1. Is **h** true?
2. Is **h** true if **b** is no longer true?
3. What is the shortest path to show **h** is true?

Fred and Sam Example

1. Fred is a collie.
   collie(fred).
2. Sam is Fred's master.
   master(fred,sam).
3. The day is Saturday.
   day(saturday).
4. It is cold on Saturday.
   ¬ (warm(saturday)).
5. Fred is trained.
   trained(fred).
6. Spaniels are good dogs and so are trained collies.
   ∀ X[spaniel(X) ∨ (collie(X) ∧ trained(X)) → gooddog(X)]
7. If a dog is a good dog and has a master then he will be with his master.
   ∀ (X,Y,Z) [gooddog(X) ∧ master(X,Y) ∧ location(Y,Z) → location(X,Z)]
8. If it is Saturday and warm, then Sam is at the park.
   (day(saturday) ∧ warm(saturday)) → location(sam,park).
9. If it is Saturday and not warm, then Sam is at the museum.
   (day(saturday) ∧ ¬ (warm(saturday))) → location(sam,museum).

- Goal expression: **∃X location(fred, X)**, or "where is Fred?"
- Assume the problem solver tries rules in order
- The solution **subgraph** shows that Fred is at the museum.



The Financial Advisor Revisited

- Assume:
  - # of dependents = 2
  - Amount of saving = $20,000
  - A steady income of $30,000
  - minsavings(X) ≡ 5000 × X
  - minincome(X) ≡ 15000 + (4000 × X)
- Two ways to obtain the above facts:
  - Add facts as predicates to the database
  - Run the program first and let the program ask the user to enter the facts as needed

1. **savings_account(inadequate) → investment(savings).**
2. **savings_account(adequate) ∧ income(adequate) → investment(stocks).**
3. **savings_account(adequate) ∧ income(inadequate)**
   → investment(combination).
4. **∀ amount_saved(X) ∧ ∃ Y (dependents(Y) ∧**
   greater(X, minsavings(Y))) → savings_account(adequate).
5. **∀ X amount_saved(X) ∧ ∃ Y (dependents(Y) ∧**
   ¬ greater(X, minsavings(Y))) → savings_account(inadequate).
6. **∀ X earnings(X, steady) ∧ ∃ Y (dependents (Y) ∧**
   greater(X, minincome(Y))) → income(adequate).
7. **∀ X earnings(X, steady) ∧ ∃ Y (dependents(Y) ∧**
   ¬ greater(X, minincome(Y))) → income(inadequate).
8. **∀ X earnings(X, unsteady) → income(inadequate).**
9. **amount_saved(20000).**
10. **earnings(30000, steady).**
11. **dependents(2).**

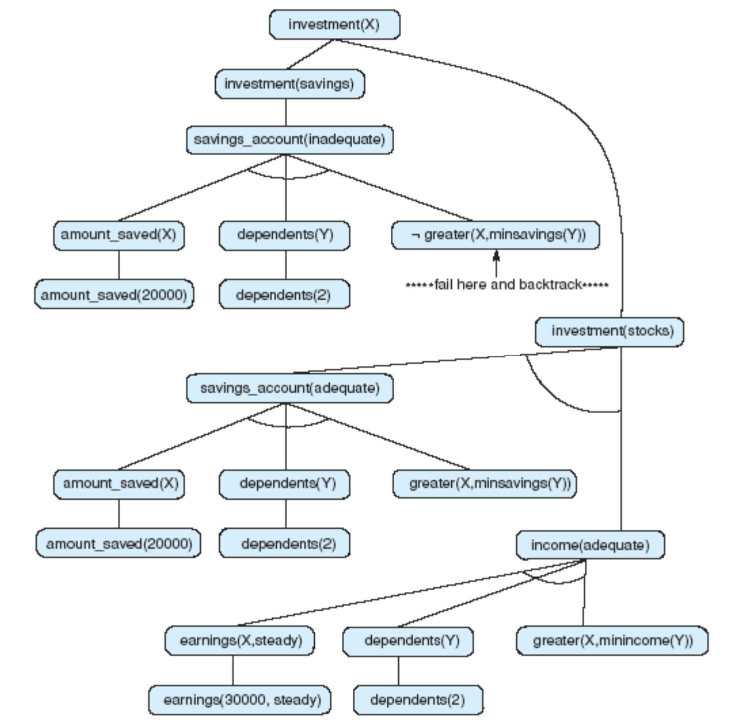


Fig 3.26 And/or graph searched by the financial advisor.

An English Language Parser and Sentence Generator

- A set of rewrite rules for parsing sentences in a subset of English grammar
- Used to determine if a sequence of words is a well-formed sentence (or grammatically correct)
- An expression is well formed in a grammar if it consists of entirely of terminal symbols (words from a dictionary) and can be reduced to the **sentence** symbol through a series of substitutions using the rewrite rules

Five rules for a simple subset of English grammar and some terminals:

1. **A sentence is a noun phrase followed by a verb phrase.**
   sentence ↔ np vp
2. **A noun phrase is a noun.**
   np ↔ n
3. **A noun phrase is an article followed by a noun.**
   np ↔ art n
4. **A verb phrase is a verb.**
   vp ↔ v
5. **A verb phrase is a verb followed by a noun phrase.**
   vp ↔ v np
6. **art ↔ a**
7. **art ↔ the**
   ("a" and "the" are articles)
8. **n ↔ man**
9. **n ↔ dog**
   ("man" and "dog" are nouns)
10. **v ↔ likes**
11. **v ↔ bites**
    ("likes" and "bites" are verbs)

Fig 3.27 And/or graph for the grammar of Example 3.3.6. Some of the nodes (np, art, etc) have been written more than once to simplify drawing the graph.

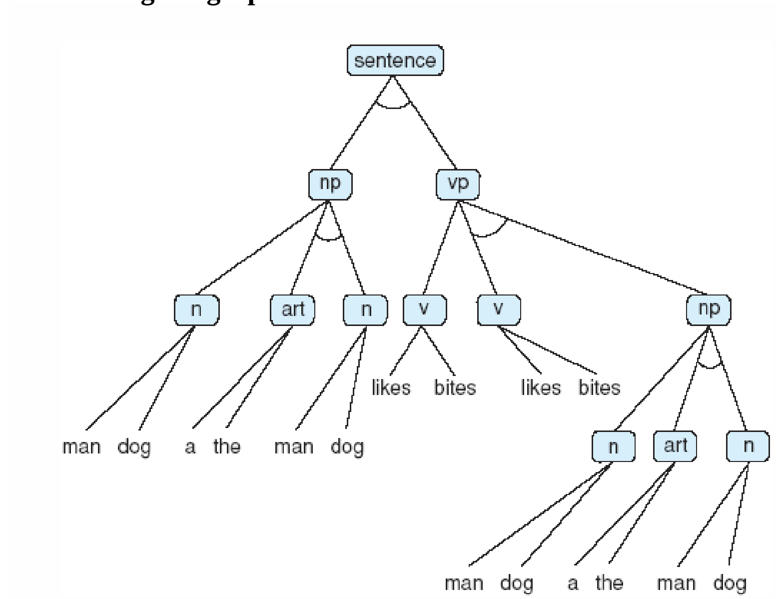


Fig 3.28 Parse tree for the sentence “The dog bites the man.” Note this is a subtree of the graph of fig 3.27.