Artificial Intelligence

CPSC 481

AI as Knowledge Representation and Search:

State Space Search for Problem Solving

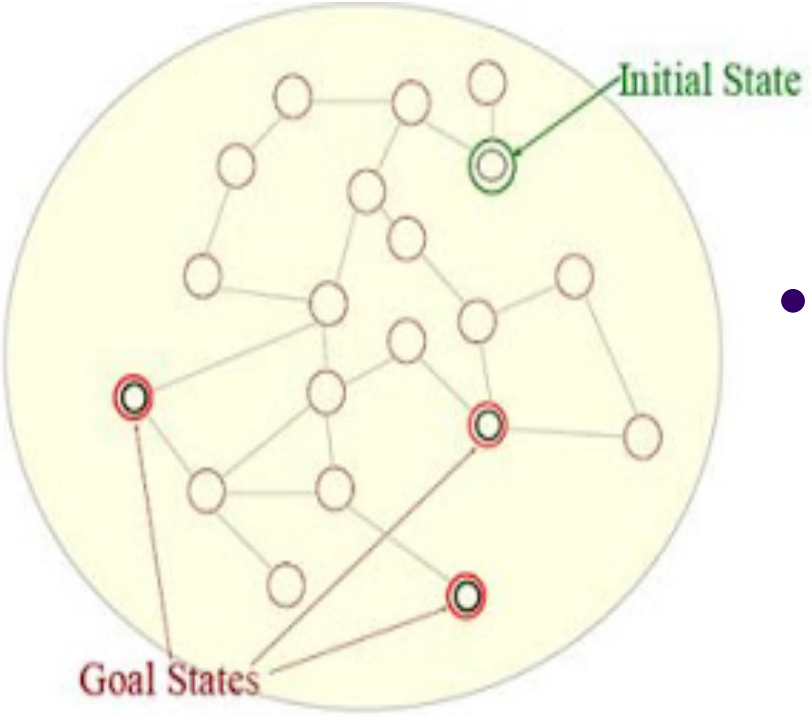
Lecture Overview

- States and state space
- States for knowledge/problem/solution representation
- Search in problem solving process
- General search approaches in state space
    - Depth-first search
    - Breadth-first search
    - Backtracking
- State space search as a general problem solving strategy

General Problem Solving Strategy

- How does human solve a problem in general?
  - Do we use thousands of algorithms to solve different problems or use only a few general method to solve all types of problems?
- Is there any general purpose process or framework to solve all types of problems?
  - Driverless car, Playing chess, Finding the cheapest car, Buying a ticket, etc.
- State space search as a general problem solving strategy

Human Problem Solving Process



- **Think about these problems:**
  - Playing chess (Tic-tac-toe or 8 puzzle) game,
  - Exit a maze
  - Ticket purchasing process,
  - Driverless car
- What do we do to solve a problem?
  - **Understand** the problem
    - solution/goal, constraints, states
  - **Define** a **state** for each step and find a **sequence of states (or steps)**.
    - A state can be a problem solving step or status (information and available methods), e.g., a state of object in Object-Oriented programming.
  - Use available **information** and **methods** to **move from one state to next state**.

State Space Search

- **State**, **State Space**, and **Search**:
  - A state is a representation for a problem solving step that involves **available information** and **methods**.
    - A **state** needs to capture the **essential** features of a problem domain and make the information accessible to a problem-solving procedure.
  - A state space of a problem is all possible states.
  - A search refers to a navigation method in a state space.
- State space search as a **general problem solving strategy** is **modeled** based on a strategy used by humans to solve difficult problems (those without algorithm solutions) or almost all problems if resources and time are unlimited!
  - AI was considered as a problem of state representation and search in early AI research.
  - State space search may be a candidate strategy for strong AI.

State Representation

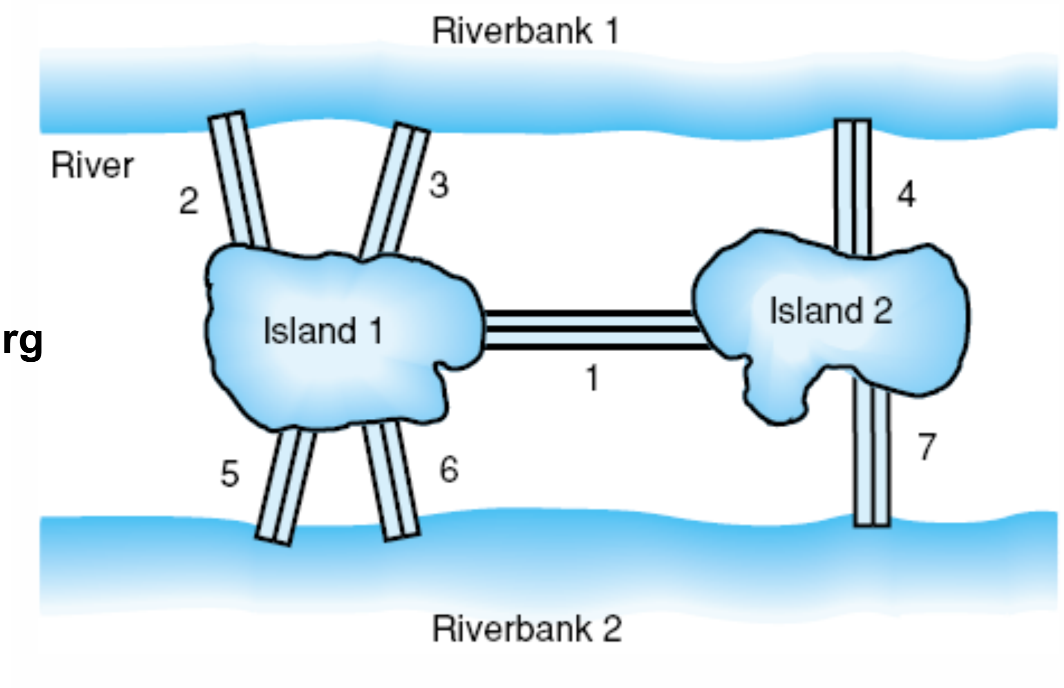
- *Expressiveness* and *efficiency* are the key factors.
  - Need to optimize the trade-off between **expressiveness** and **efficiency** (using methods, e.g., search, read/write/update, etc.)
  - Ultimately we need a *powerful representation scheme* to solve AI problems.

- Different levels of state representation:
  - **Conceptual** (or mental) representation,
    - State
  - ***Symbolic*** *representation*,
    - *Graph*
  - **Computer** representation (data structure)
    - Variable, array, record, object, table, list, tree, queue, etc.

Invention of Graph Theory

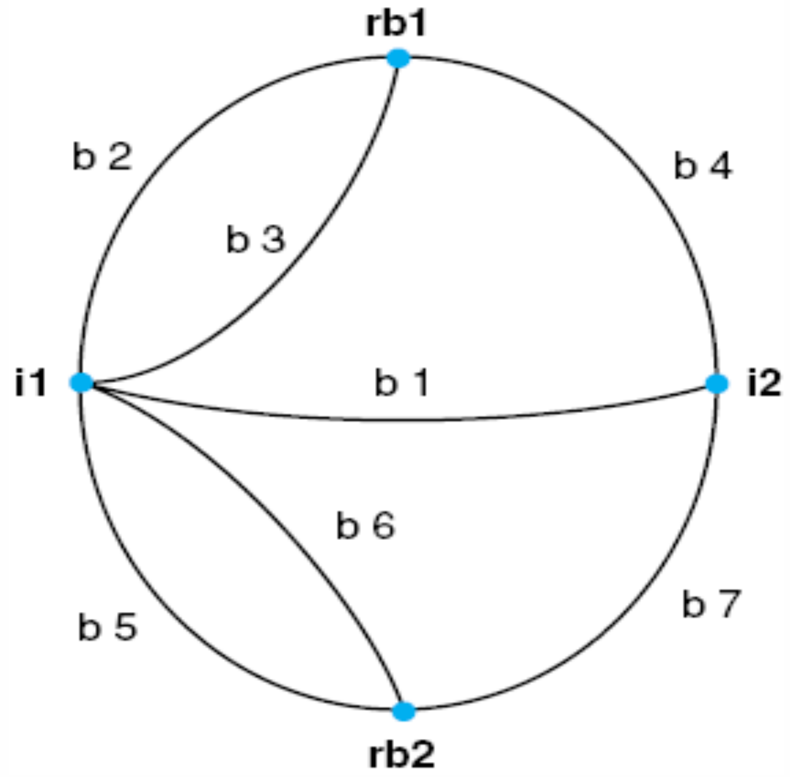
Is there a walk around the city that crosses each bridge exactly once?

The city of Königsberg



Euler invented **graph theory** to solve this problem.

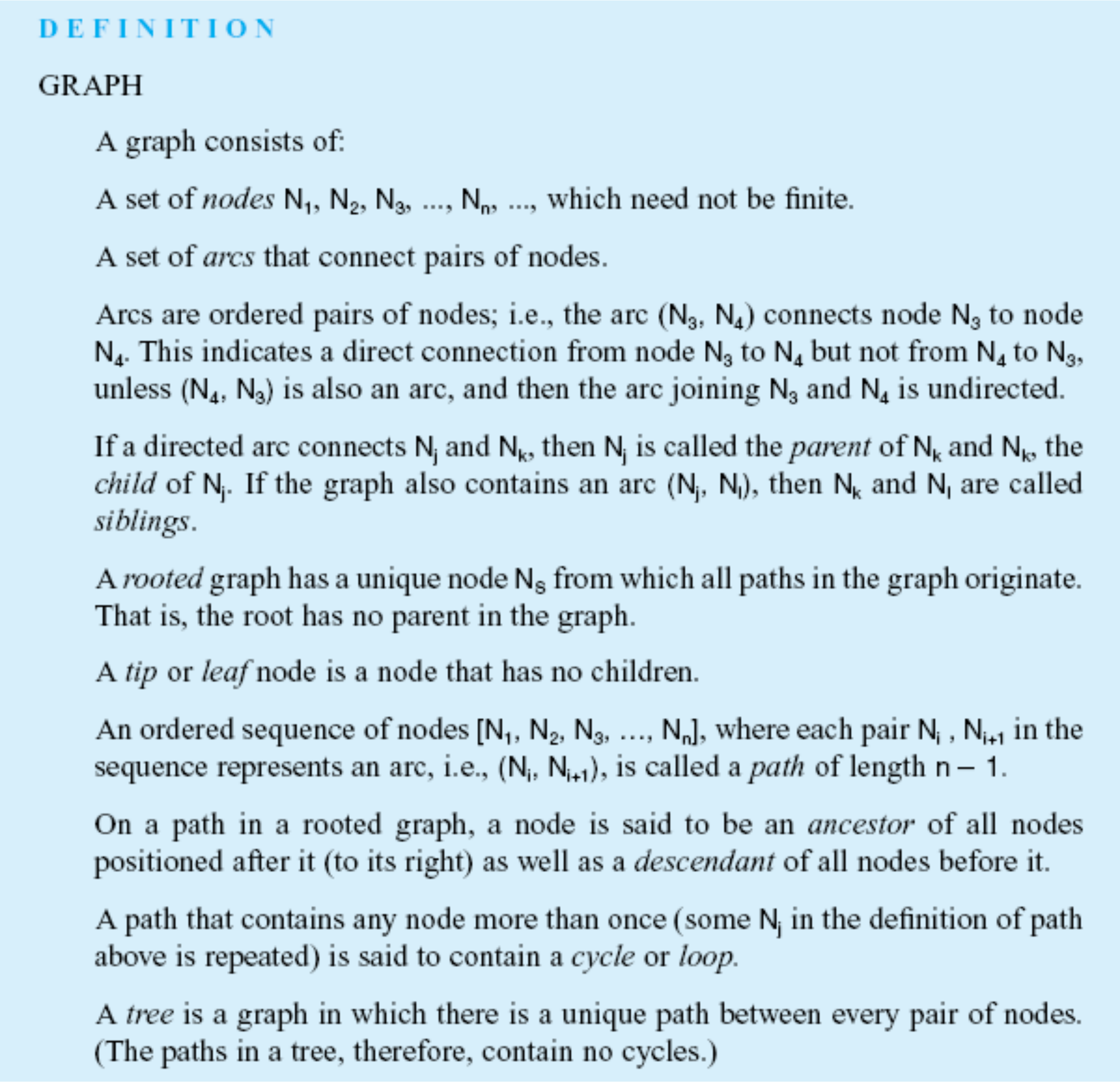
Graph of the Königsberg Bridge System



*__Euler__ proved the problem:
Unless a graph contained either exactly zero or two nodes of odd degree, the walk is impossible.

Many other real-world problems can be thought (conceptually) as graph problems – abstract thinking.

***State and state space can be represented using the Graph Theory.**



DEFINITION

GRAPH

A graph consists of:

A set of *nodes* N_1, N_2, N_3, ..., N_n, ..., 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.

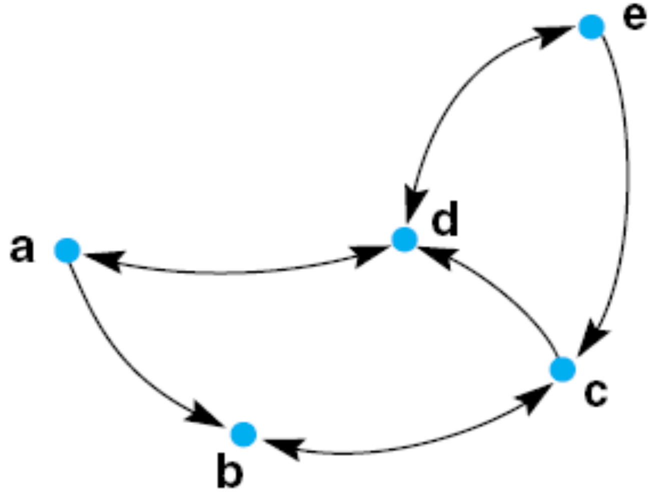
An ordered sequence of nodes [N_1, N_2, N_3, ..., N_n], where each pair N_i , N_{i+1} in the sequence represents an arc, i.e., (N_i, N_{i+1}), is called a *path* of length $n - 1$.

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.)

A Labeled Directed Graph

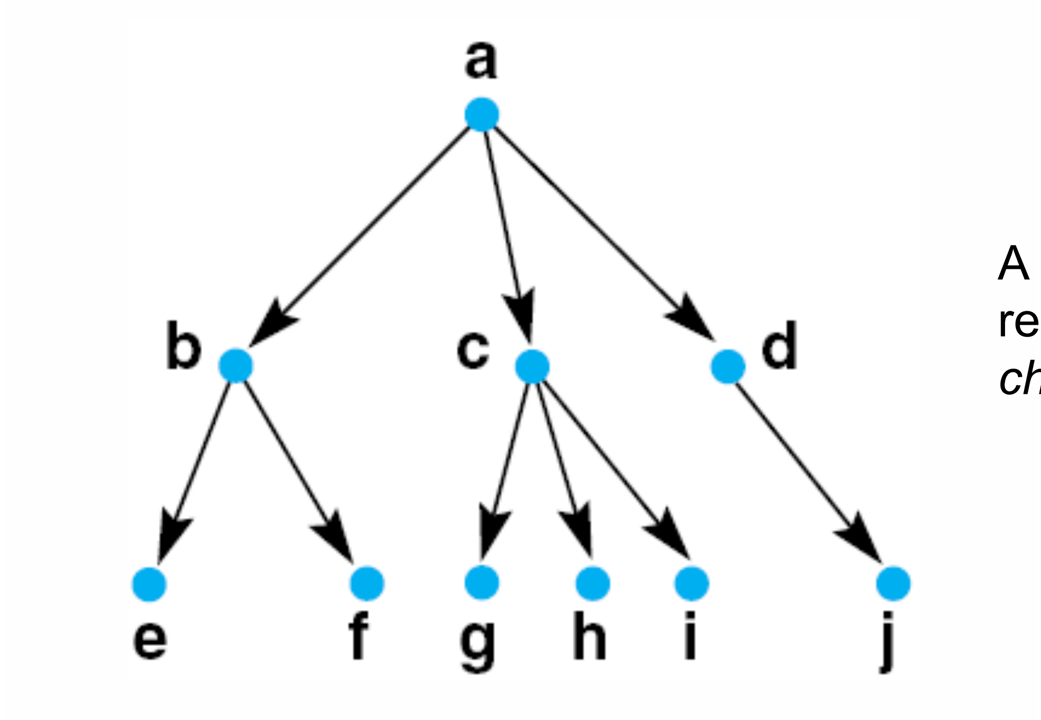


Nodes = {a,b,c,d,e}

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

***Directed graph**: A graph is directed if arcs have a direction.
***Path**: a sequence of nodes through successive arcs, e.g., (a, b, c, d)

A Tree is A Rooted Graph

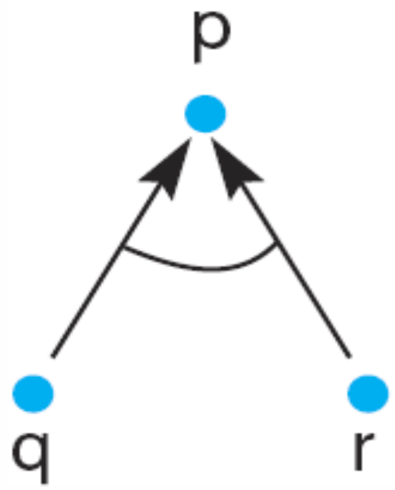


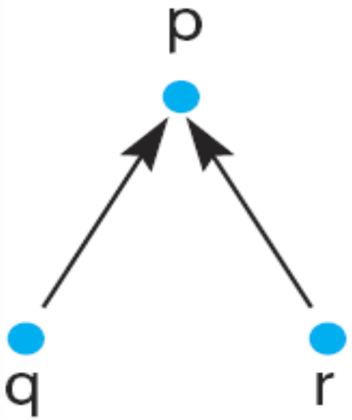
A **tree** showing a family relationships, *parent* and *children*

*****Tree**: has a root that has path from the root to all nodes and every path is unique without cycle.

AND/OR Graph to Represents Subproblems and Alternative Paths

q and r are subproblems to be solved.





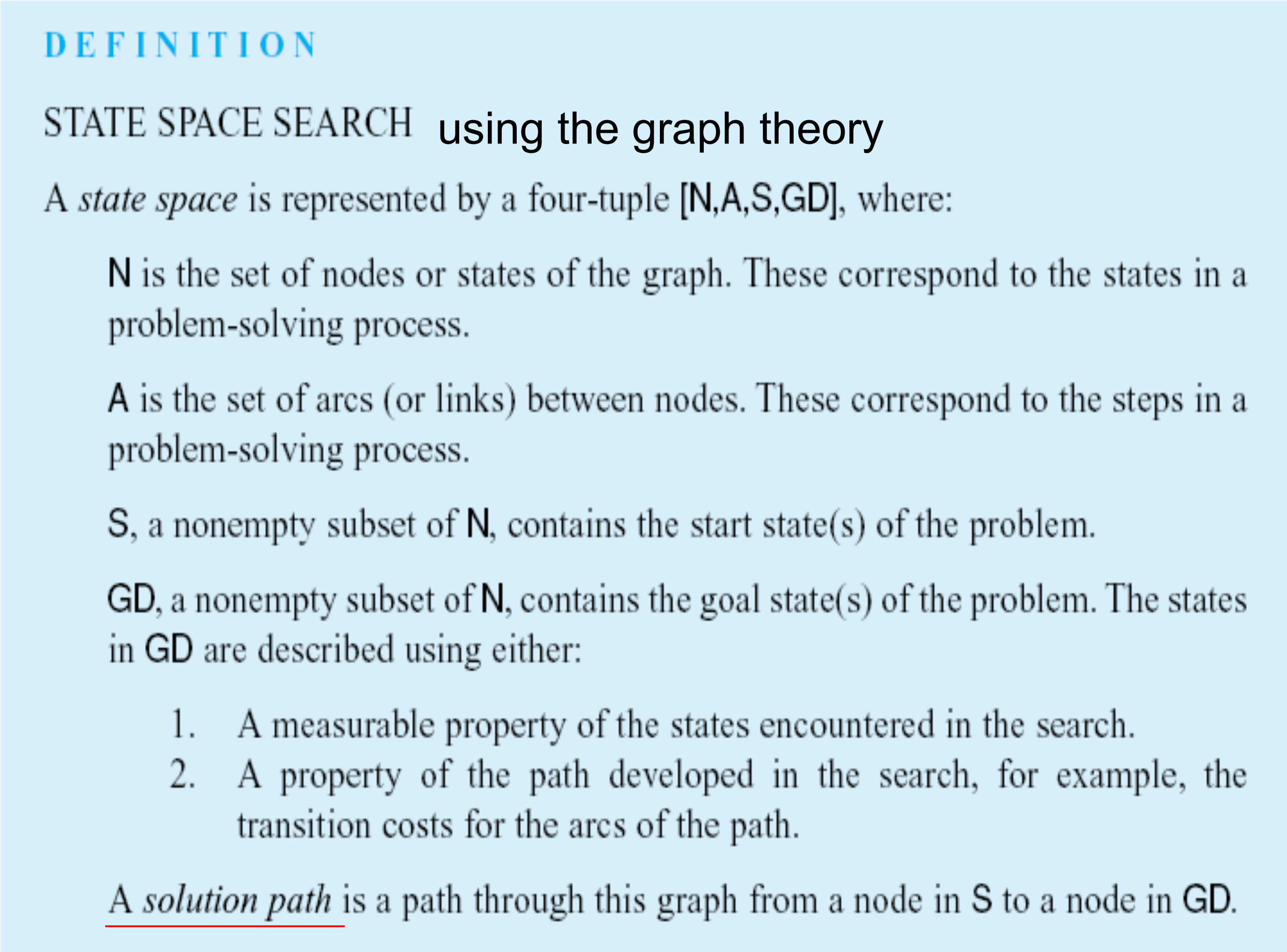
q and r are alternative paths.

$(q \wedge r) \rightarrow p$

AND/OR graph can be also represented by implication in logic.

$(q \vee r) \rightarrow p$

∧ (**AND**) operator indicates a problem decomposition (as subproblems to be solved).
v (**OR**) operator indicates alternative solution paths.
→ (**edge**) operator indicates IF Then, implication, or dependency relationship.



DEFINITION

STATE SPACE SEARCH using the graph theory

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.

+State space search is a method to find a solution in the state space.
+**Solution** can be a state (containing the solution), path, or both

+**State space** can be also used as a means of determining the problem complexity e.g., search space (all possible moves) for Chess.

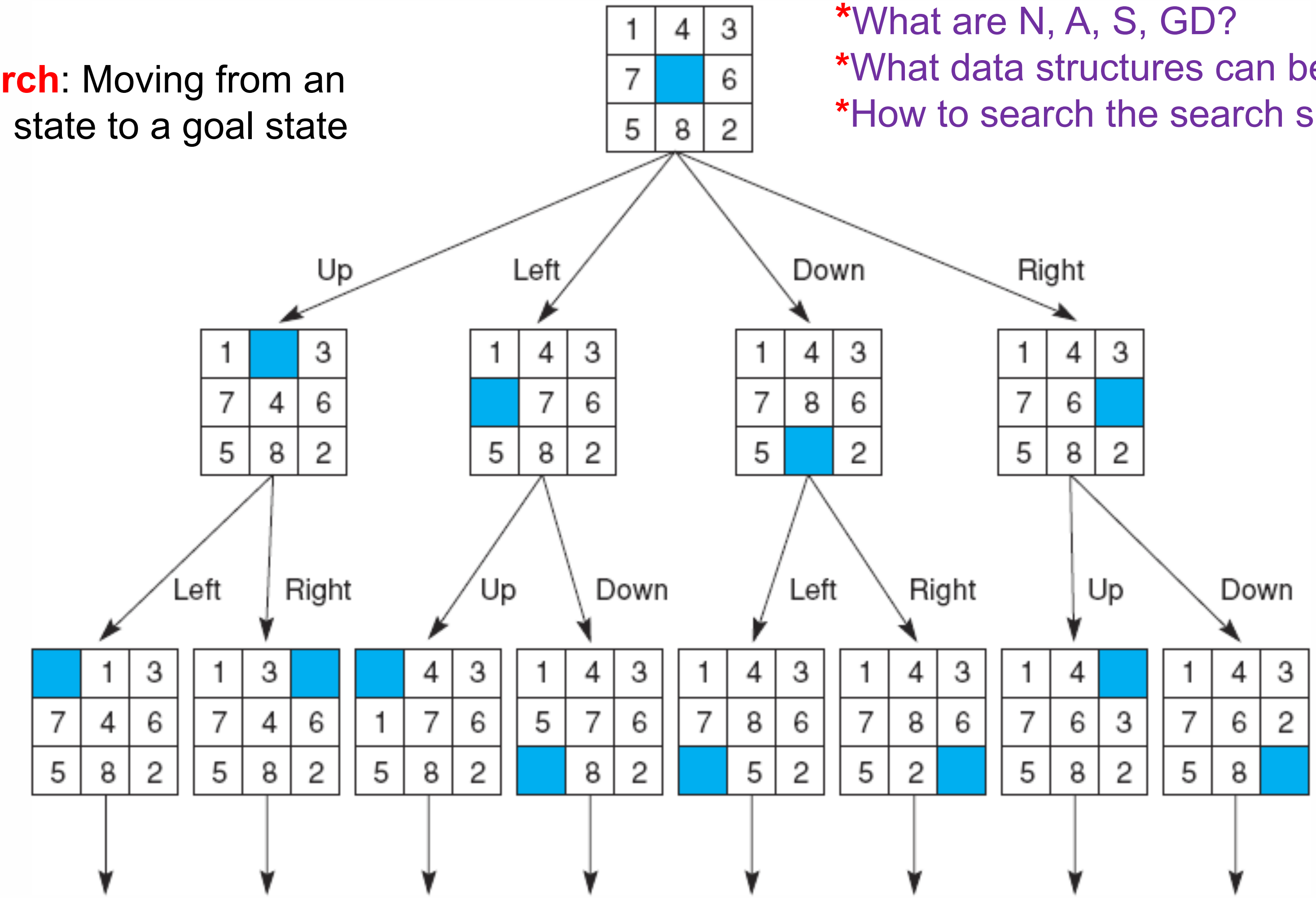
A State Space Graph for the 8-puzzle Generated by “move blank” Operations

***Search**: Moving from an initial state to a goal state

*What are N, A, S, GD?

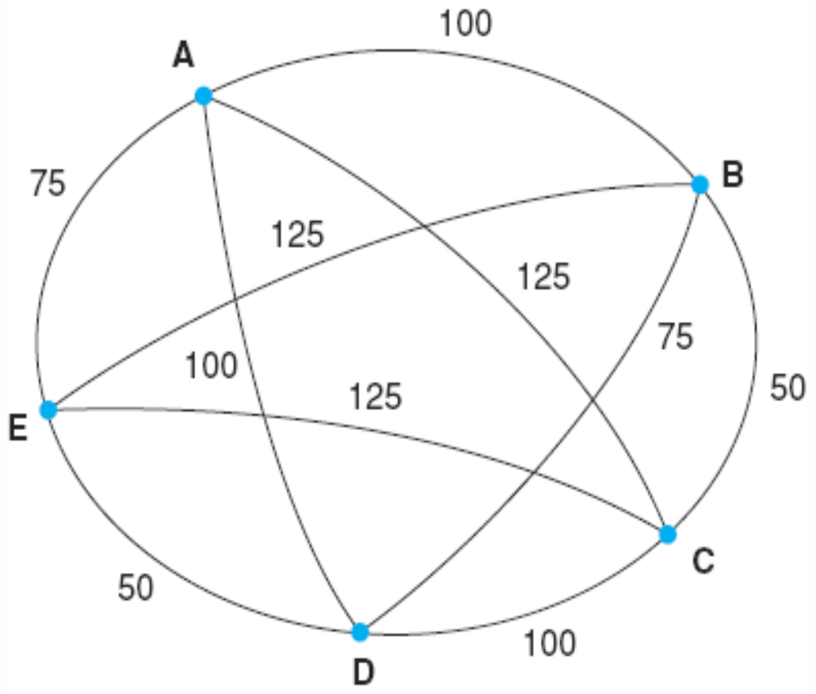
*What data structures can be used?

*How to search the search space?



A State Space Graph for the Traveling Salesman Problem

*__Goal:__ Find the *shortest path* for the salesman to travel, visiting each city and return to the starting city.

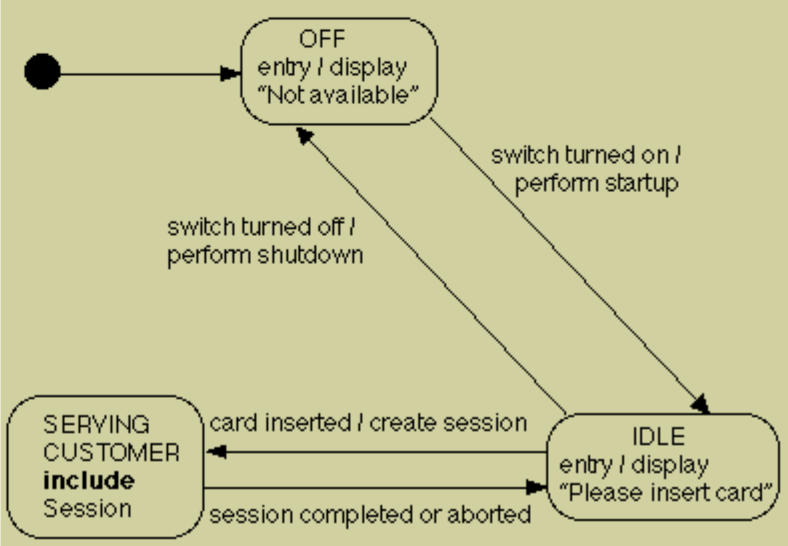


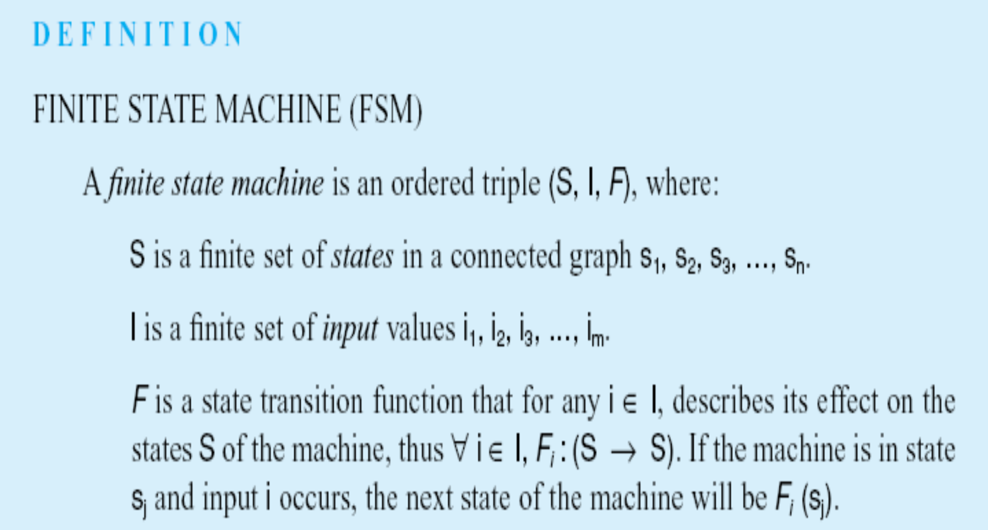
*An instance of the travelling salesperson problem:
A –D—C—B—E—A with **450** miles

*Is it optimum solution (with the minimum cost)?

A State Space Graph for Finite State Machines

State Chart for ATM Machine





DEFINITION

FINITE STATE MACHINE (FSM)

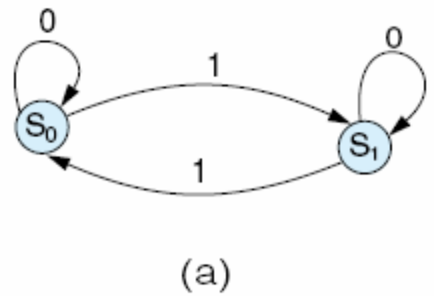
A *finite state machine* is an ordered triple (S, I, F), where:

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 $i_1, i_2, i_3, \ldots, i_m$.

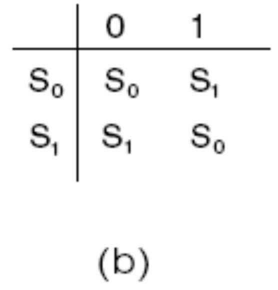
F is a state transition function that for any $i \in I$, describes its effect on the states S of the machine, thus $\forall i \in I, F_i : (S \to S)$. If the machine is in state s_j and input i occurs, the next state of the machine will be $F_i(s_j)$.

*Difference between State Chart and FSM?



(a)

The finite state graph for a flip flop in visualized representation



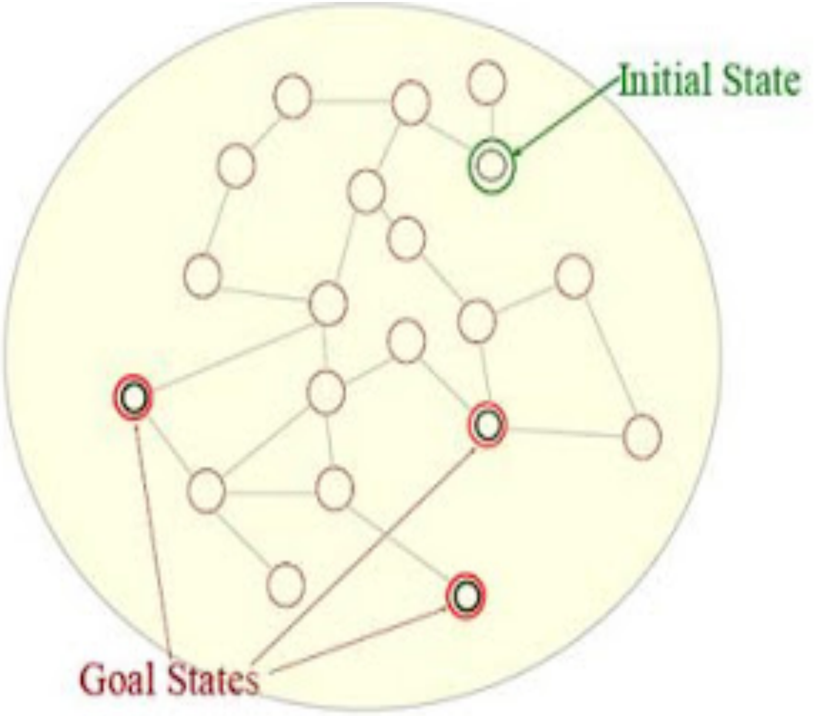
	0	1
S_0	S_0	S_1
S_1	S_1	S_0

(b)

Transition matrix in compact data structure

Example of FSM:
Natural language processing
I was/am …
I are
I I I was/am …

Searching a Graph



- **Types of solutions**
  - A goal state containing a solution, e.g., theorem proving
  - A path from initial to goal state, e.g., finding the shortest path
  - Both a goal state and path
- **Search directions:**
  - From Initial to goal
  - From Goal to Initial
- **Search method**
  - How to search?

Search Directions in a State Space

- ***Data-driven (or forward)***
  - Use the knowledge and constraints found in each state of the problem to guide search by applying rules/methods to produce new states until it finds a goal state/solution.
    - Most problems can be solved via data-driven approach.
- ***Goal-driven (or backward)***
  - Use knowledge of the goal to guide the search by checking what rules/methods can be used to generate this goal and determine what conditions must be true to use them.
  - These **conditions** become the new goals/subgoals, and continue working backward until it works back to the **facts** of the problem.
    - Diagnosis, theorem proving, answering some multiple-choice questions, etc.
- **Note**: **Both approaches explore the same problem space**.
  - Preferred strategy is chosen by the properties of the problem.
  - **Factors** to consider: **complexity** and implementation **difficulty**, and **search space** (estimated by **branching factor**)

When is the Data-driven Search Better?

- When all or most of the data are given in the initial problem statement.
  - For many interpretation problems by presenting a collection of data and asking the system to provide a high-level interpretation
    - Systems analyze data (e.g., interpreting geological data to find minerals, PROSPECTOR)
- When there are a large number of potential goals, but there are only a few ways to use the facts and given information of a particular problem instance.
  - DENDRAL expert system finds the molecular structure of organic compounds based on their formula, mass.
- When it is difficult to formulate a goal or hypothesis.

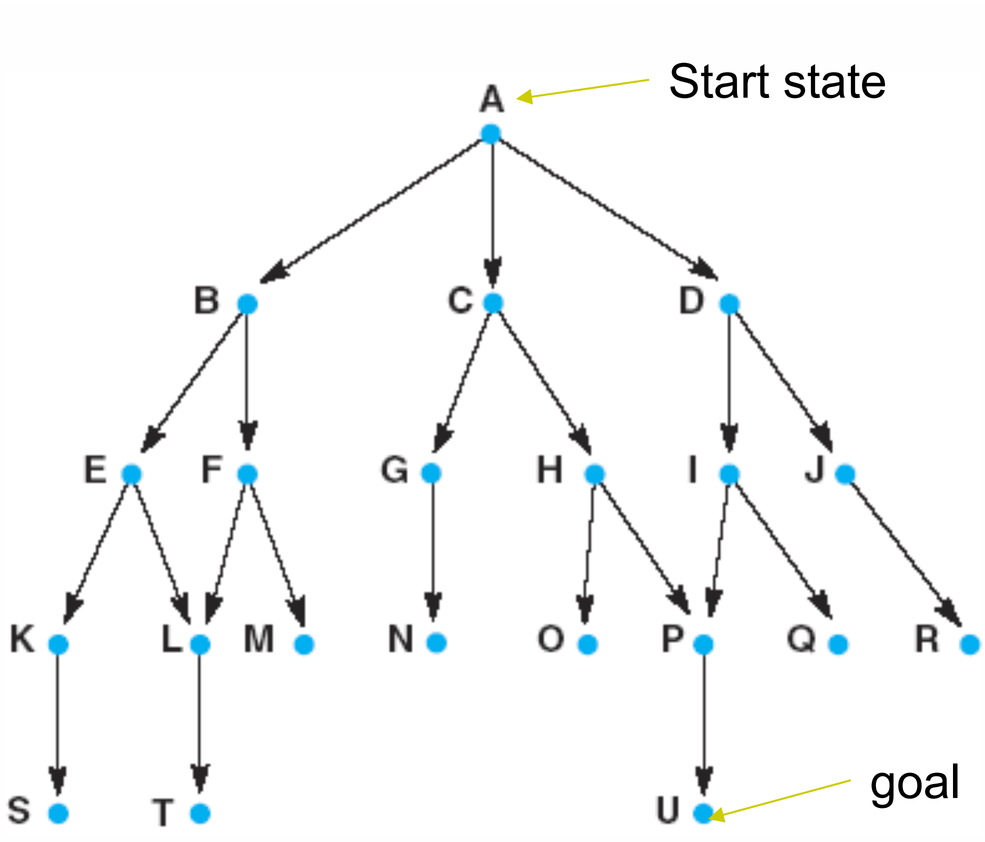
When is the Goal-driven Search Better?

- Useful when the goal/hypothesis is already known or easily formulated and finding causes when something is already happened.
  - Theorem proving (goal is the theorem to prove), question answering in expert systems (questions are goals).
- Problem data are not given but must be acquired by the problem solver.
  - Finding causes, e.g., medical diagnosis problem, doctor orders only those that are necessary to confirm or deny a particular hypothesis.
- When there are a large number of rules that match the states of the problem and thus produce an increasing number of conclusions (for reduced search space).
  - Prove a statement “I am a descendant of Thomas Jefferson.”

General (Graph) Search Methods

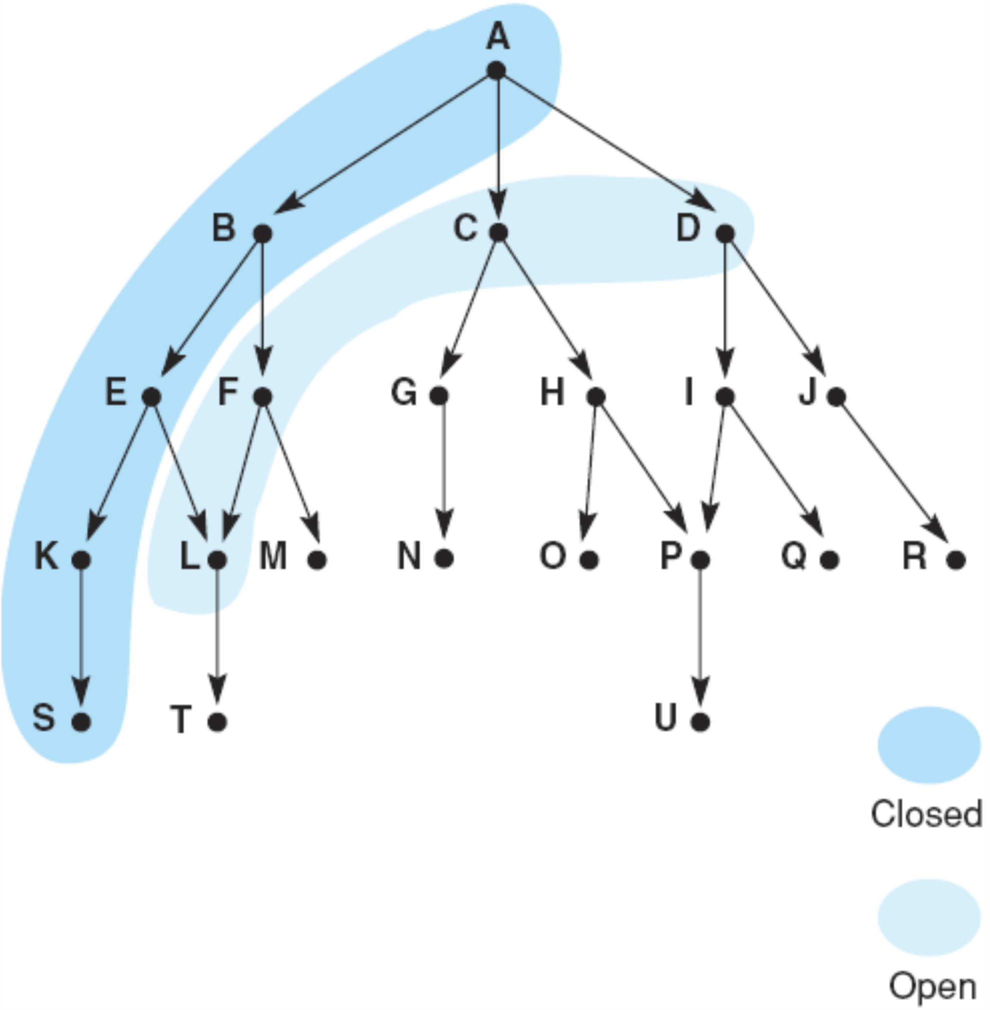
- **Depth-First Search** (DFS)
  - When a state is examined, all of its children and their descendant are examined before any of its siblings
  - Goes deeper and deeper into the search space, stop only when no other descendants or goal is found
- **Breadth-First Search** (BFS)
  - Explores the space in a level-by-level fashion. Only stop when there are no more states to be explored at a given level and move to the next level until it finds a goal
- **Backtracking search**
  - Works like DFS except that it is allowed to backtrack to previous node based on the cost computed for current node to a different path.

Example Graph and Search by DFS and BFS

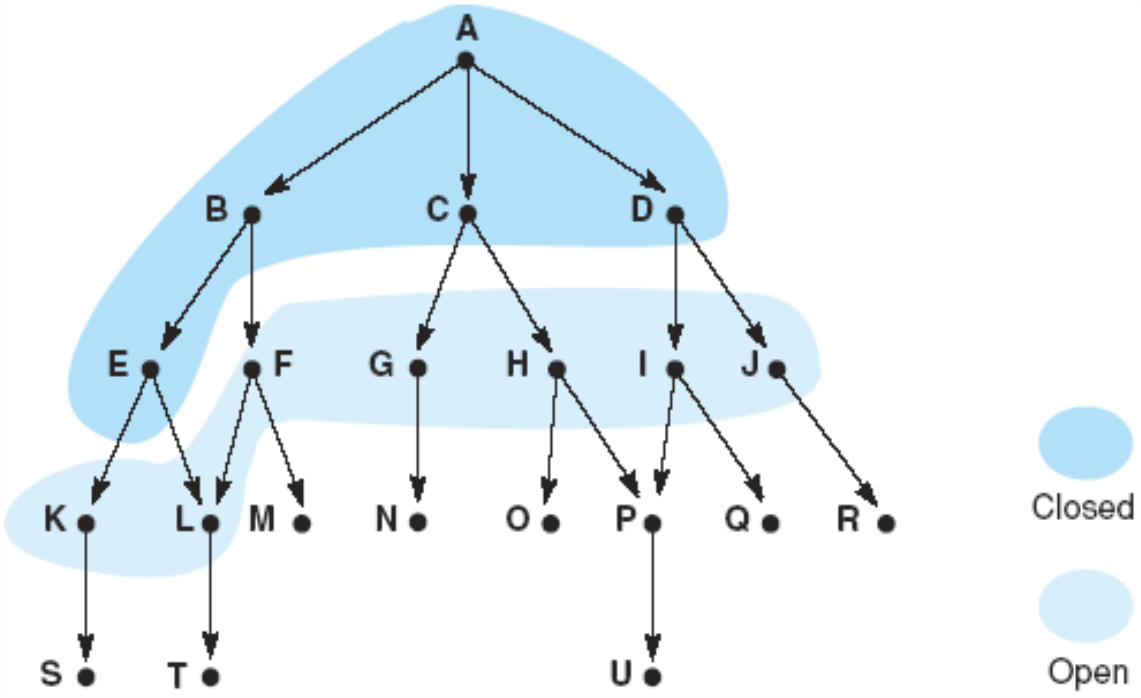


Note: In actual problem solving process, this type of search tree is NOT given, instead we must explore it until it finds a solution.

States at Iteration 6 of DFS



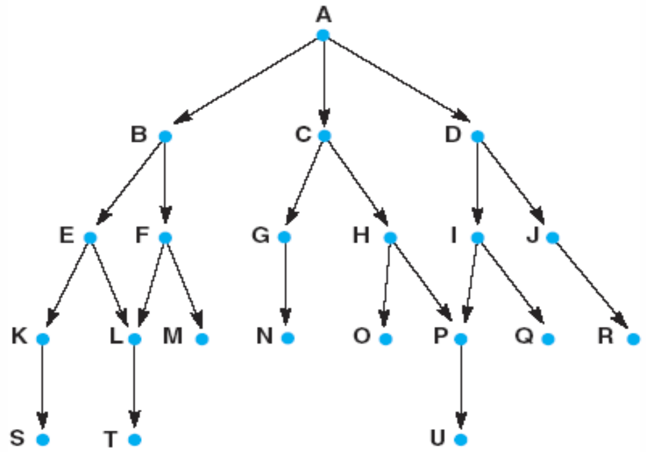
States at Iteration 6 of BFS



A Trace of DFS on the Graph

Note: This graph is **not** given. Instead we must **explore** it from the **initial state** **A** by **DFS**.

We need only **two queues**, **Open** and **Closed**.



Assuming the graph is search space.

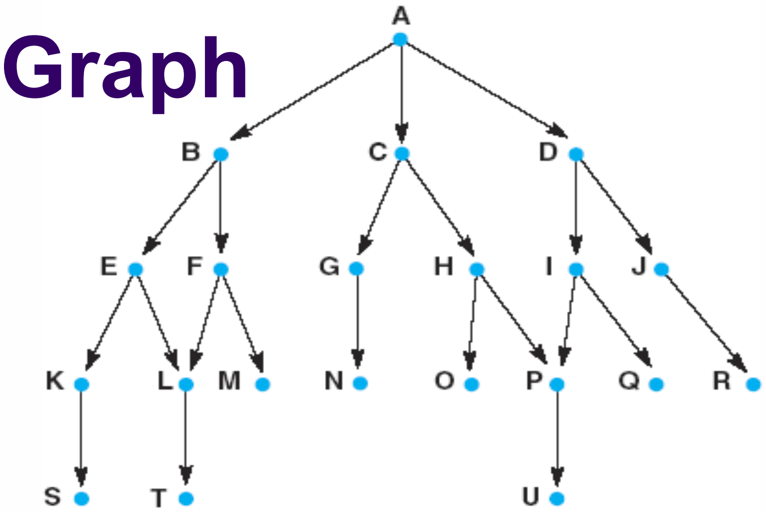
1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [E,F,C,D]; closed = [B,A]**
4. **open = [K,L,F,C,D]; closed = [E,B,A]**
5. **open = [S,L,F,C,D]; closed = [K,E,B,A]**
6. **open = [L,F,C,D]; closed = [S,K,E,B,A]**
7. **open = [T,F,C,D]; closed = [L,S,K,E,B,A]**
8. **open = [F,C,D]; closed = [T,L,S,K,E,B,A]**
9. **open = [M,C,D]**, as L is already on **closed**; **closed = [F,T,L,S,K,E,B,A]**
10. **open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]**
11. **open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]**

In order to maintain a path we need additional data structure.

DFS Algorithm

```
begin
    open := [Start];                                                          % initialize
    closed := [ ];
    while open ≠ [ ] do                                                       % states remain
        begin
            remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                                % goal found
                else begin
                    generate children of X;
                    put X on closed;
                    discard children of X if already on open or closed;       % loop check
                    put remaining children on left end of open                % stack
                end
        end;
    return FAIL                                                               % no states left
end.
```

A Trace of BFS on the Graph



Note: This graph is not given. Instead we must explore it from the **initial state** **A** by **BFS**.

We need only **two queues**, **Open** and **Closed**.

Assuming the graph is search space.

1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [C,D,E,F]; closed = [B,A]**
4. **open = [D,E,F,G,H]; closed = [C,B,A]**
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**
6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M]** (as L is already on open); **closed = [F,E,D,C,B,A]**
8. **open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]**
9. and so on until either U is found or **open** = [ ]

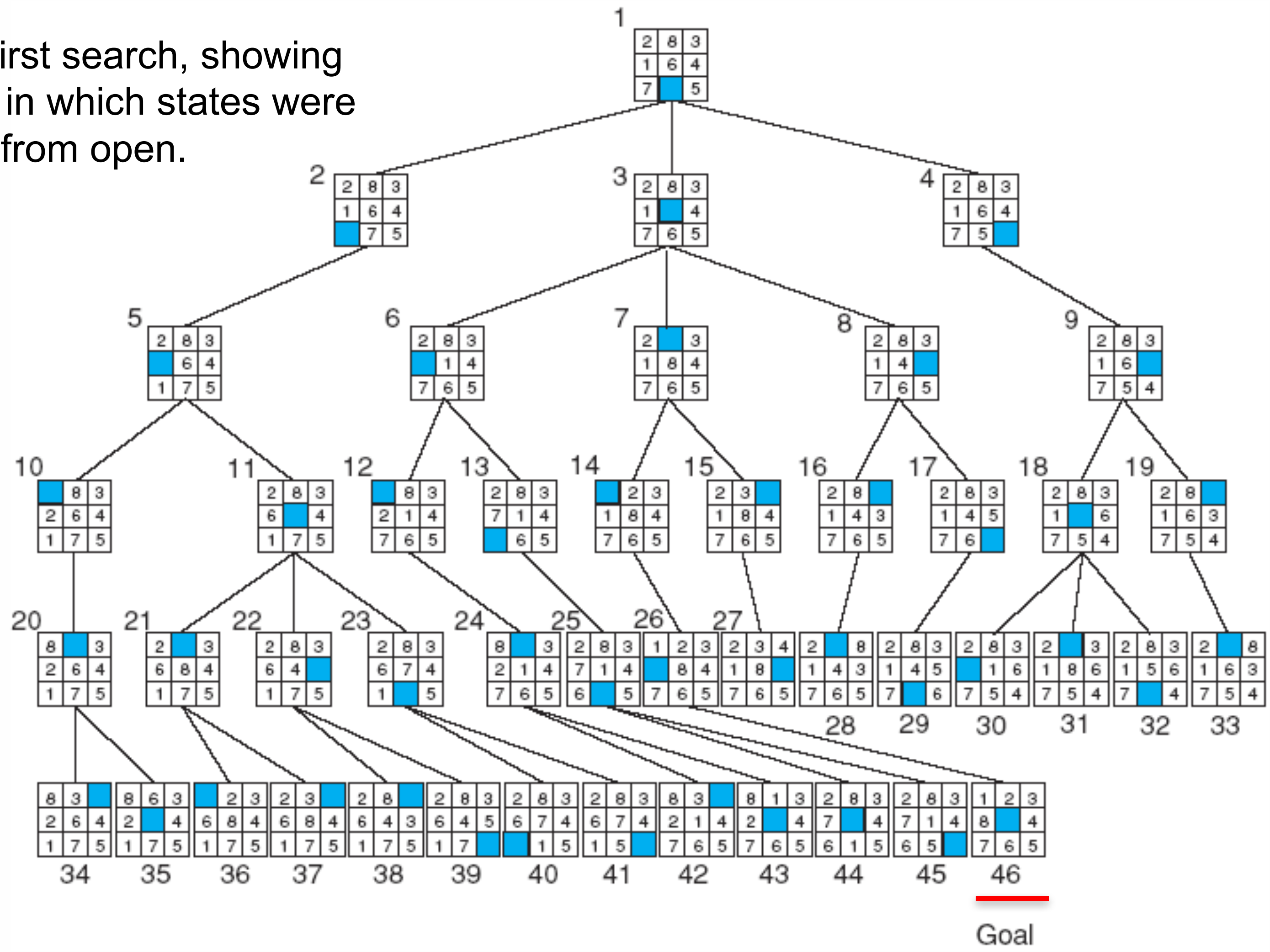
BFS Algorithm

```
function breadth_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                         % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;  % loop check
                  put remaining children on right end of open          % queue
               end
      end
   return FAIL                                                         % no states left
end.
```

BFS of the 8-Puzzle Problem

Breadth-first search, showing the order in which states were removed from open.



Breadth-first vs. Depth-first

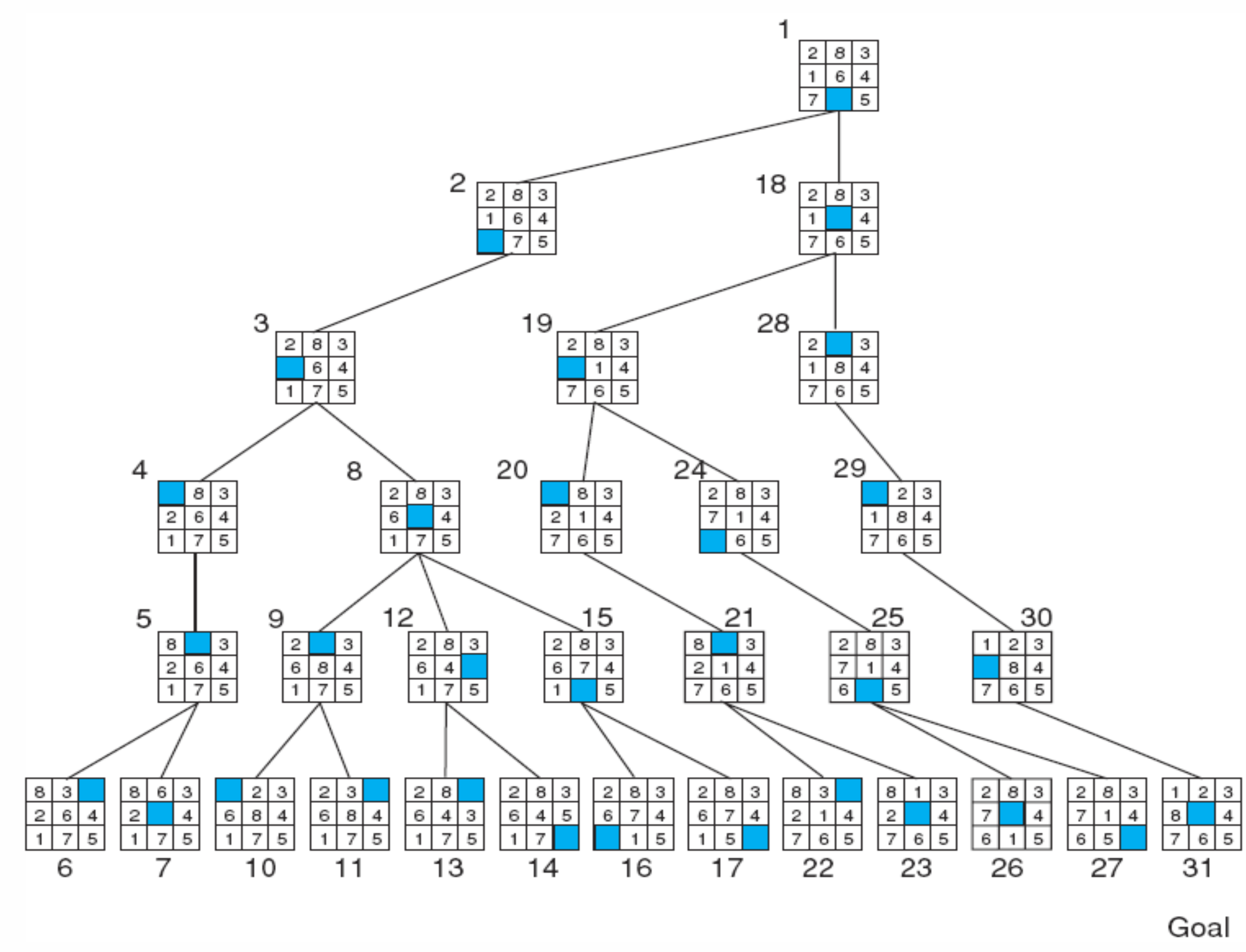
- Breadth-first search
  - Always examine all the nodes at level n before proceeding to level n+1.
    - It may need a large amount of memory in many cases.
  - Appropriate for a problem with small search space but a problem with large space can be intractable.
- Depth-first search
  - Can be efficient for a problem with many branches. If solution path is long, it may find it quickly without wasting other branches.
    - Space usage is good (may need less memory needed than BFS in general).
  - Can be lost deep in the graph, possibly missing shorter paths in other branches.
- Which approach is better?
  - The decision should be based on the property of the problem.
- How to improve DFS or combine DFS and BFS?

Variations of DFS (Improved DFSs)

- DFS with bound
  - At each iteration, it performs a complete DFS to the specified level (**bound**).
  - Once it gets below a certain level (or time), assume a failure on a search path and go for another path, e.g., in chess play in a limited time.
  - May handle some problems of both DFS and BFS.
- DFS with deepening
  - At each iteration, it performs a complete DFS to the current depth bound. This continues, increasing the depth bound by one at each iteration.
- **DFS with bound and deepening** has the advantages over both DFS and BFS, but space usage: B x n, (B = avg. # of children, n = level), complexity O(B^n), *still exponential*.

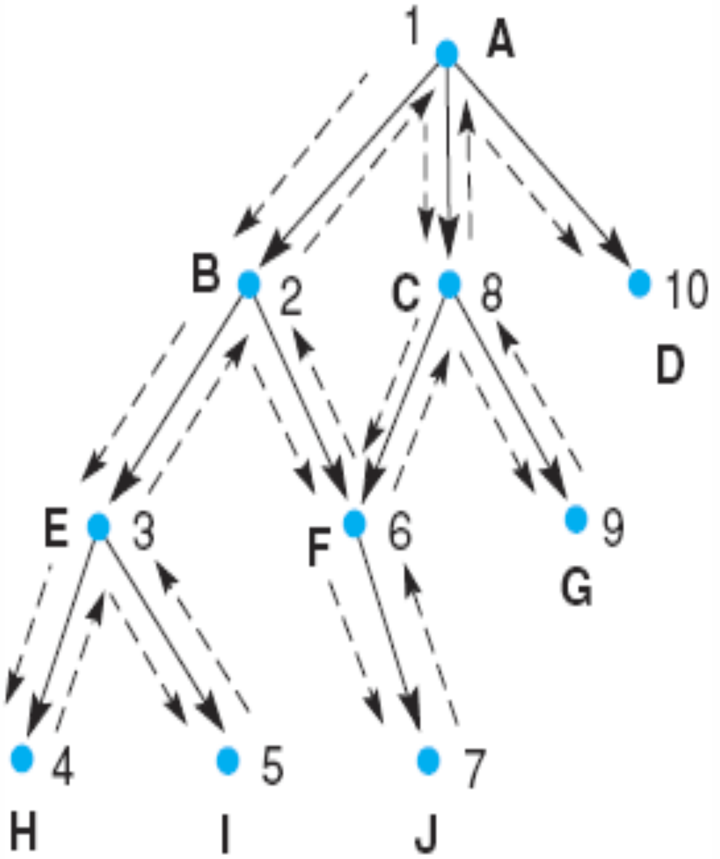
Movie: http://movingai.com/dfid.html

DFS of the 8-Puzzle with a Depth Bound of 5



Backtracking Search Algorithm

- One of the *first search algorithms*, earlier than DFS and BFS
- **Algorithm sketch**
  - Search begins at the start state and pursues a path until it finds a goal or dead end.
  - If the goal is found, return goal, if dead end or the current path is more expensive, backtrack to the most recent state on the path and continue other paths.
- Works very similarly to DFS but unlike DFS



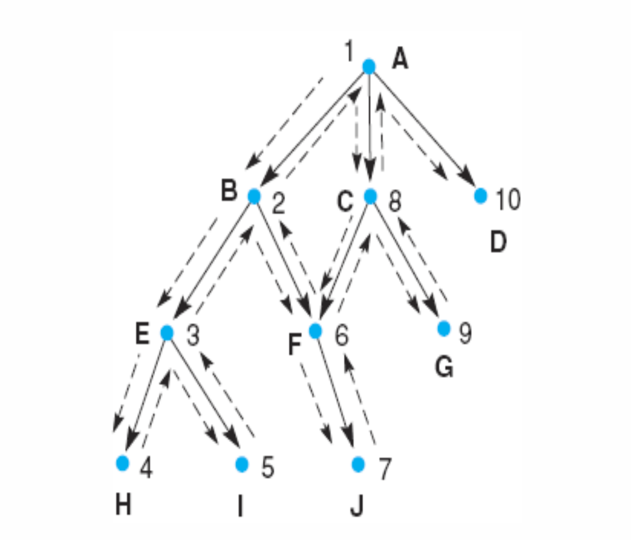
A sate space by backtracking search

A Trace of Backtrack on the Graph

SL(state list), NSL(new state list), DE(dead ends), CS(current state)

Initialize: SL = [A]; NSL = [A]; DE = [ ]; CS = A;

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[ ]
1	B	[B A]	[B C D A]	[ ]
2	E	[E B A]	[E F B C D A]	[ ]
3	H	[H E B A]	[H I E F B C D A]	[ ]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]
6	J	[J F B A]	[J F B C D A]	[E I H]
7	C	[C A]	[C D A]	[B F J E I H]
8	G	[G C A]	[G C D A]	[B F J E I H]



Note: No actual backtracking like tracing backward to root node, is needed. Instead, by maintaining ancestors information, we get the same search result as backtracking.

Backtracking Algorithm

NSL: like open queue,
DE: like closed queue,
SL: Current path,
CS: Current state

```
function backtrack;
SL(state list), NSL(new state list), DE(dead ends), CS(current state)
begin
  SL := [Start];  NSL := [Start];  DE := [ ];  CS := Start;            % initialize:
  while NSL ≠ [ ] do                               % while there are states to be tried
    begin
      if CS = goal (or meets goal description)
        then return SL;               % on success, return list of states in path.
      if CS has no children (excluding nodes already on DE, SL, and NSL)
        then begin
          while SL is not empty and CS = the first element of SL do
            begin
              add CS to DE;                              % record state as dead end
              remove first element from SL;                               %backtrack
              remove first element from NSL;
              CS := first element of NSL;
            end
          add CS to SL;
        end
        else begin
          place children of CS (except nodes already on DE, SL, or NSL) on NSL;
          CS := first element of NSL;
          add CS to SL
        end
    end;
    return FAIL;
end.
```

This condition is needed for backtracking.
Note: We've seen any state in SL.

A General Problem Solving Process using State Space Search Strategy

A **problem** to be solved

Represent the problem in **State Space** (conceptual)

Represent the State Space represented problem in **Graph** (symbolic)

Represent the Graph represented problem using **Data Structures and Search Algorithm** (implementation using computer)

Solution

Review Questions

- How does human solve a problem in general? Do we use a general purpose problem solving strategy?
- What is state space search strategy? What are state, state space, and search?
- What are the key elements to maintain for each state?
- What is the role of search algorithm in state space search strategy?
- Do you agree that human uses the state space search strategy when solving problems?
- What is search space?
- What is initial state, goal state, path?
- What are different forms of a solution in a problem solving based on the state space search?
- Describe the process of problem solving using the state space search strategy.
- To think about applications, try to describe the process of solving various complex problems such as driverless car, playing a board game like chess, go, Sudoku, etc., using the state space search strategy?

- How can a state be represented? What data structure(s) can be used to represent a state and a state space?
- What are the basic elements of a graph? What’s the benefit of using graph theory?
- How can the graph theory be used for problem solving based on state space search strategy?
- Describe the process of problem solving using graph? For applications of graph theory, try to describe the process of solving various problems such as 8-puzzle game, tic-tac-toe game, chess, buying a ticket, solving a math problem, traveling sales man problem, etc. using graph theory.
- What are the important factors to consider in estimating/determining the search space or complexity of a problem? Try to estimate the search space for various problems.
- What is data-driven (or forward) search? For what types of problems do we want to use data-driven search?
- What is goal-driven (or backward) search? For what types of problems do we want to use goal-driven search?
- What’s the purpose of choosing the search direction?

- How does Depth-first search (DFS) work? How can we find a solution using DFS?
- How does Breadth-first search (BFS) work?
- Why do we call DFS and BFS brute force search methods?
- How does Backtracking search work?
- What’s the primary difference between DFS and Backtracking search?
- Why is Backtracking search considered as an informed search method?
- Try some graph search examples by DFS, BFS, and Backtracking to fully understand these algorithms.
- What are the primary benefits and limitations of using BFS, DFS, and Backtracking search?
- What data structures can be used to implement a graph?
- What data structures can we use to implement DFS, BFS, and Backtracking? Try some examples with the data structures to implement these search methods.
- Describe a problem solving process with the state space search starting from conceptual level to implementation level using specific data structures.

Most Important Points to Remember

- Can you explain the concept of state space search strategy?
- Why is graph theory important for state space search strategy?
- For a given **complex problem**:
  - Can you describe the problem solving process using the state space strategy?
  - Can you describe the problem solving process using the graph theory?
  - Can you implement DFS algorithm?
  - Can you implement BFS algorithm?
  - Can you implement Backtracking algorithm?
- Do you understand that DFS and BFS are brute force algorithms?
- Do you understand that although Backtracking is considered an informed search, it is still based on the brute force search?

References

- George Fluger, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, 6th edition, **Chapters 3,** Addison Wesley, 2009.