Artificial Intelligence

CS 165A

Apr 18, 2022

Instructor: Prof. Yu-Xiang Wang

Today

→ Problem Solving by Search

→ Search algorithms

Logistics

- Project 1 due this Thursday 11:59 pm
  - the bonus part of it has no deadline
  - Check piazza for announcements
- Additional instructor office hour at 2pm
  - Henley Hall 2013
- TA office hour at 2 pm Wednesday

Recap of the last lecture

- Three steps in modelling with Bayesian networks
- Inference with Bayesian networks using only CPTs
- Three equivalent ways of describing structures of a joint distribution
  - Factorization ⇔ DAG ⇔ the set of conditional independences
- Prove conditional independence by definition.

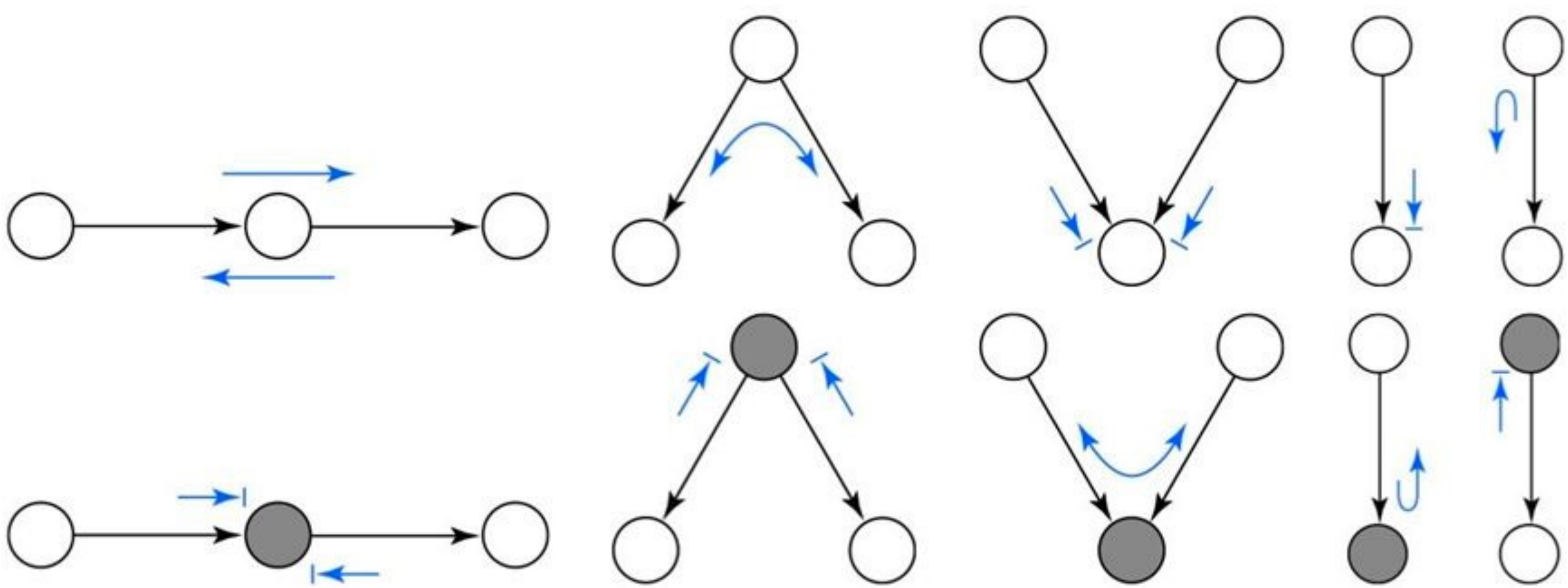
Recap of the last lecture

- Reading conditional independences from the DAG itself.

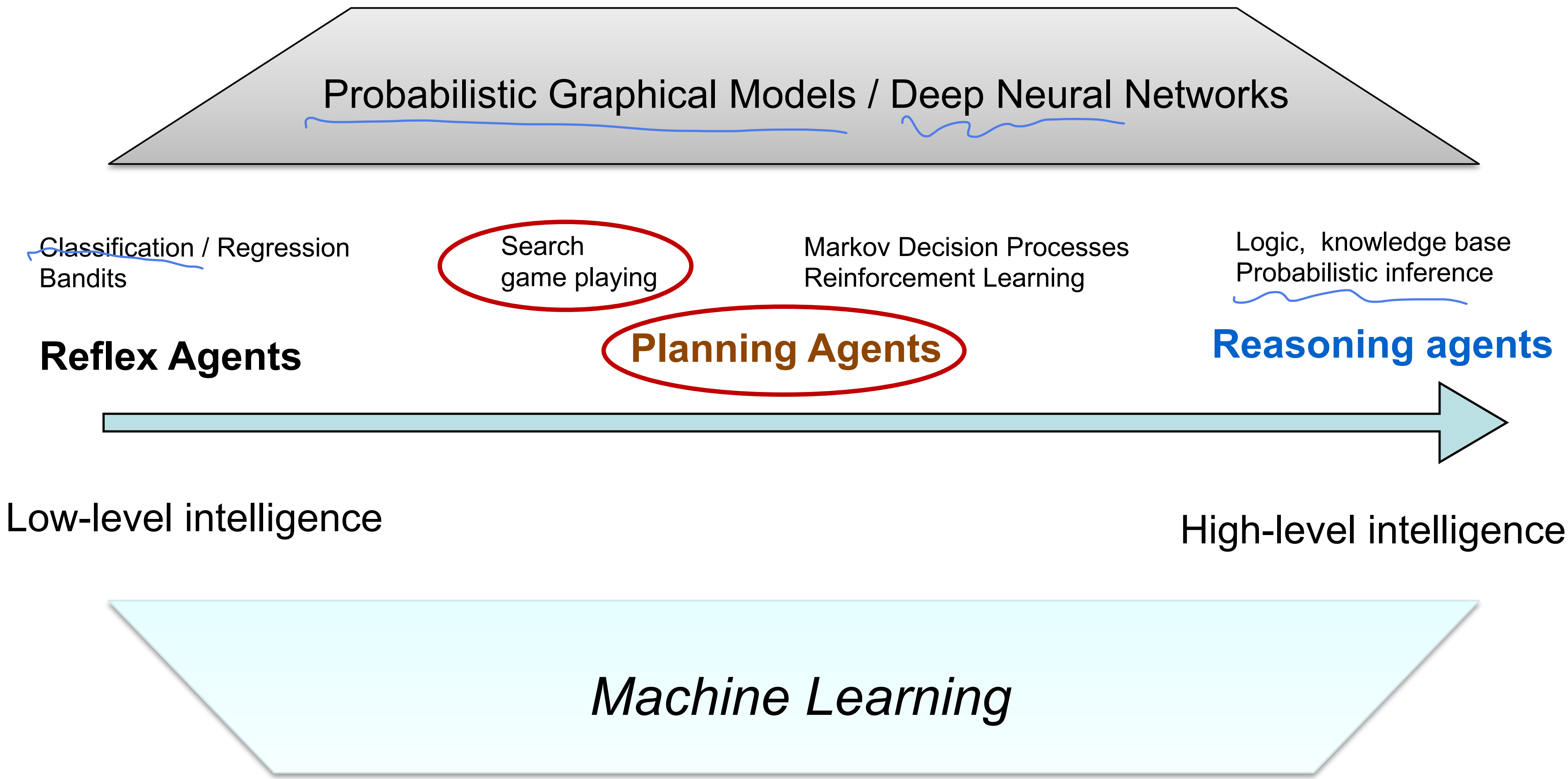
- d-separation
  - Three canonical graphs

- Bayes ball algorithm for determining whether $\mathbf{X} \perp \mathbf{Z} \mid \mathbf{Y}$
  - Bounce the ball from any node in X by following the ten rules
  - If any ball reaches any node in Z, then return "False"
  - Otherwise, return "True"

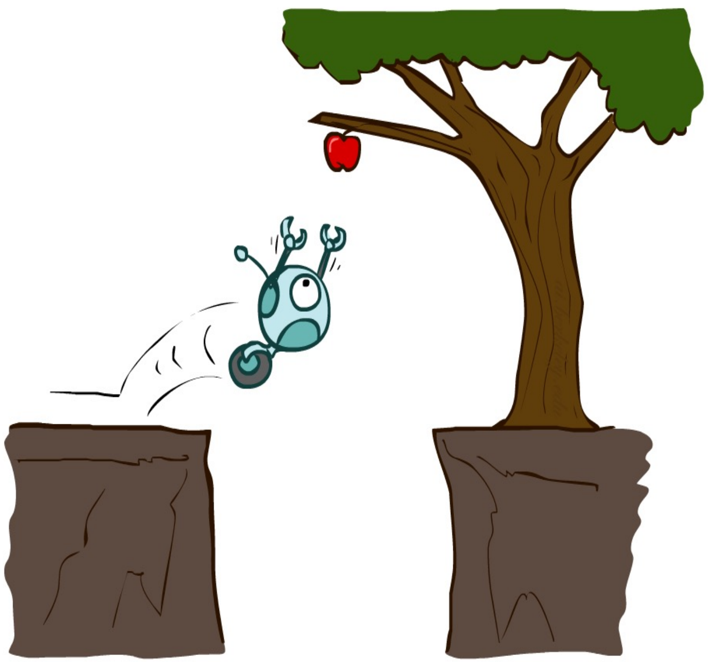
The Ten Rules of Bayes Ball Algorithm

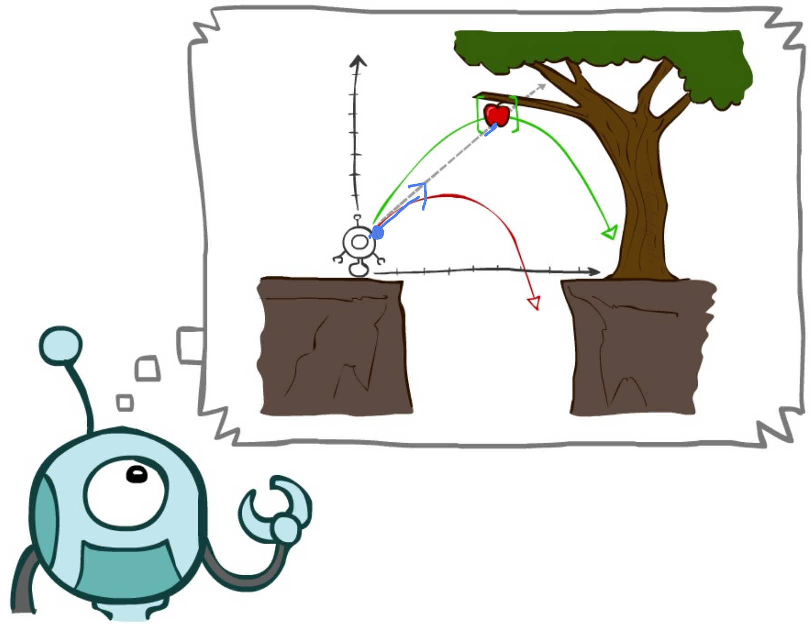


Structure of the course



Reflex Agents vs. Planning agent





(illustration credit: Dan Klein)

- Reflex agents act based on immediate observation / memory; often optimizes immediate reward.
- Planning agent looks further into the future and "try out" different sequences of actions --- in its mind --- before taking an action; optimizes long-term reward.

Modeling-Learning-Inference Paradigm

	Modeling	**Learning**	**Inference**
Classifier agent (Spam filter)	Feature engineering Hypothesis class	Minimize Error rate	Prediction on new data points
Probabilistic Inference agent (Sherlock)	Joint distribution Draw edges in BN Conditional independences	Fitting the CPTs to Data	Marginalization (conceptually easy)
Search agents	State-Space-diagram	Environment given (learn edge weights)	**Nontrivial search algorithms**

Search sequence of lectures

- Today: Problem Solving by Search + Search algorithms
- Apr 21: Search algorithms
- Apr 26: Minimax search and game playing
- Apr 28: Finish “search” + Midterm review.

- Recommended readings on search:
  - AIMA Ch 3.1 – 3.6, Ch 5.1-5.4

Remaining time today

- Formulating problems as search problems

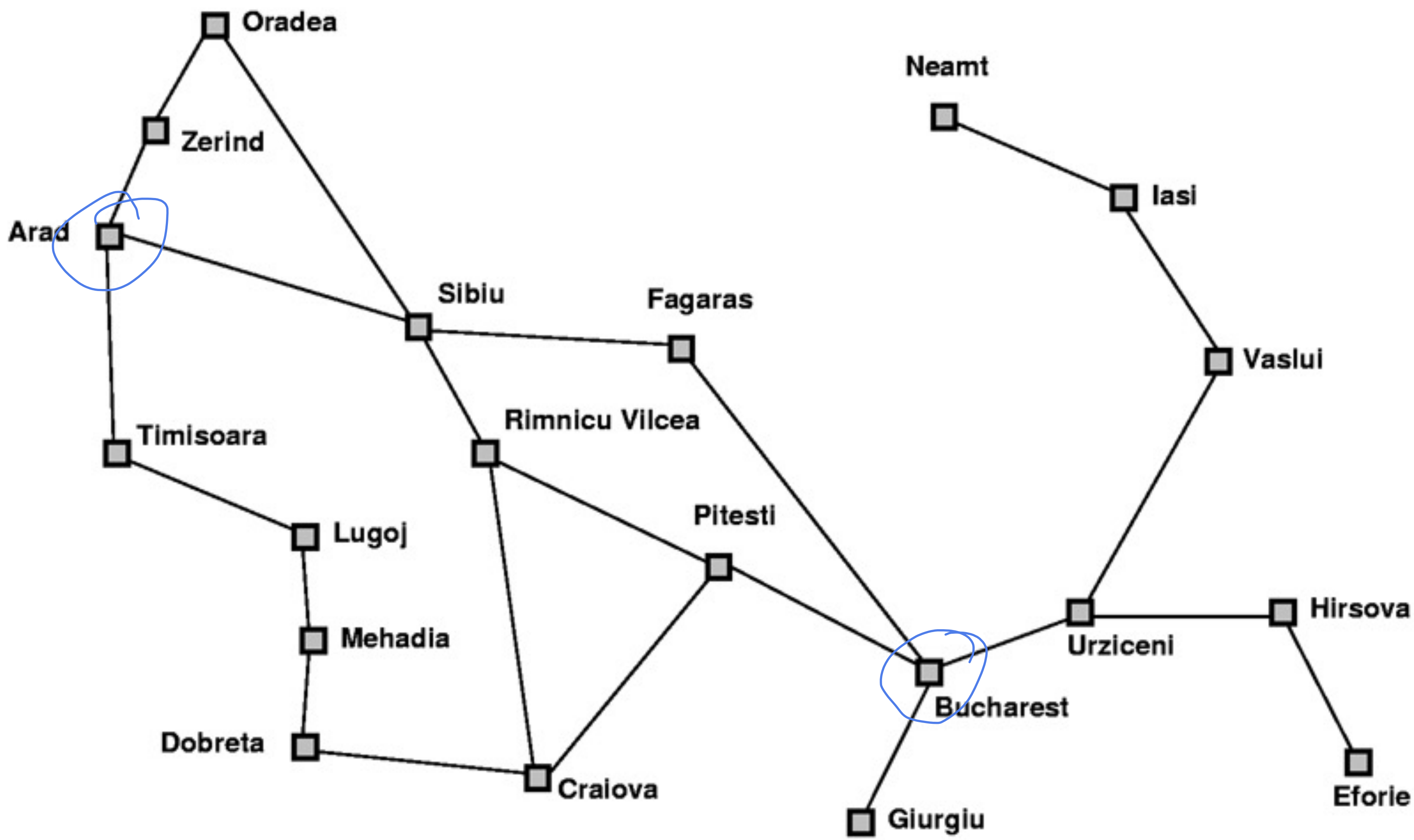
- Basic algorithms for search

Example: Romania

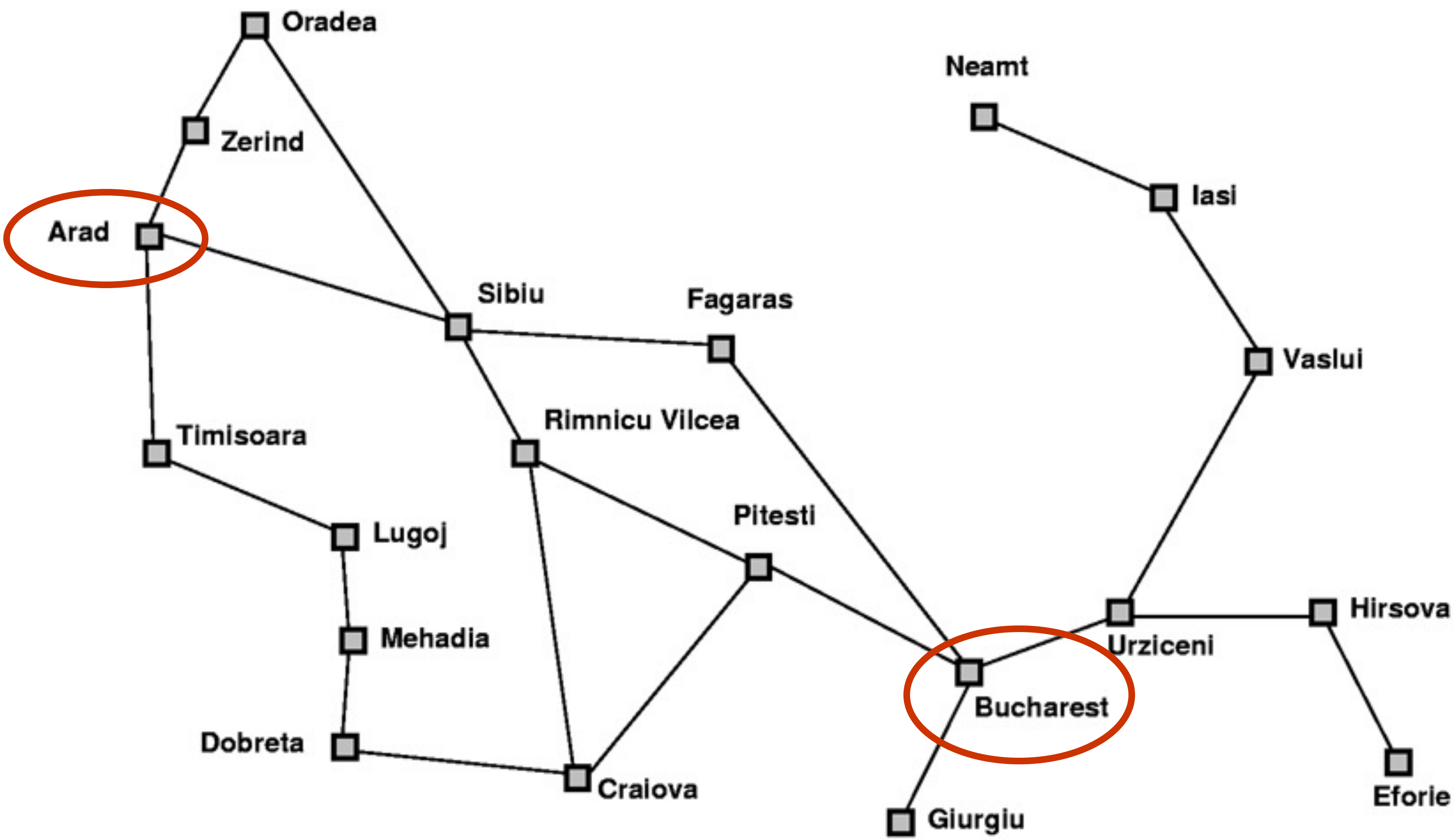
You're in Arad, Romania, and you need to get to Bucharest as quickly as possible to catch your flight.

- Formulate problem
  - States: Various cities
  - Operators: Drive between cities
- Formulate goal
  - Be in Bucharest before flight leaves
- Find solution
  - Actual sequence of cities from Arad to Bucharest
  - Minimize driving distance/time

Romania (cont.)



Romania (cont.)

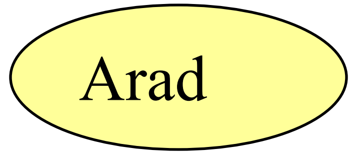


Romania (cont.)

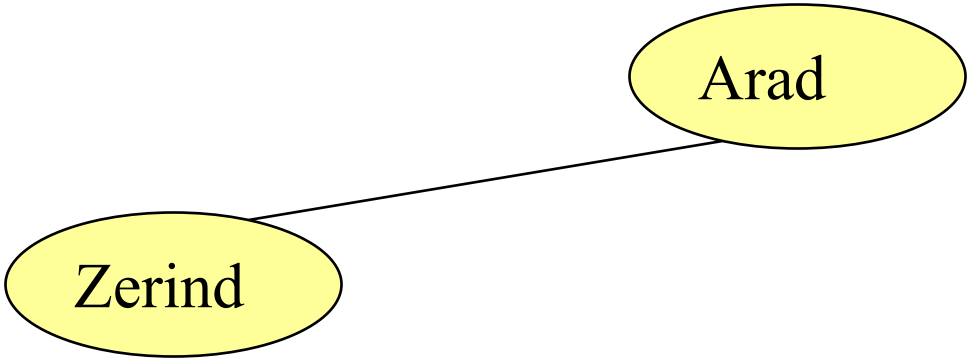
Problem description **<{S}, S_0, {S_G}, {O}, {g}>**

- **{S}** – cities (c_i)
- $\mathbf{S_0}$ – Arad
- $\mathbf{S_G}$ – Bucharest
  - G(S) – Is the current state (S) Bucharest?
- {O}: { $c_i \rightarrow c_j$, for some *i* and *j* }
- $\mathbf{g_{ij}}$
  - Driving distance between c_i and c_j?
  - Time to drive from c_i to c_j?
  - 1?

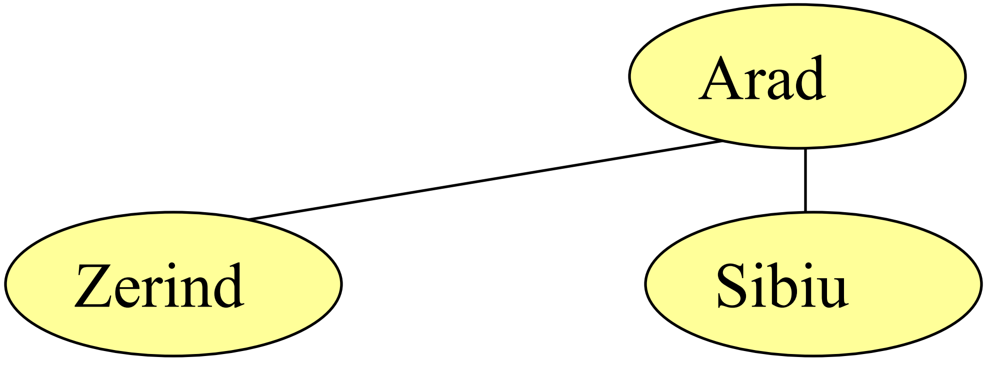
Possible paths



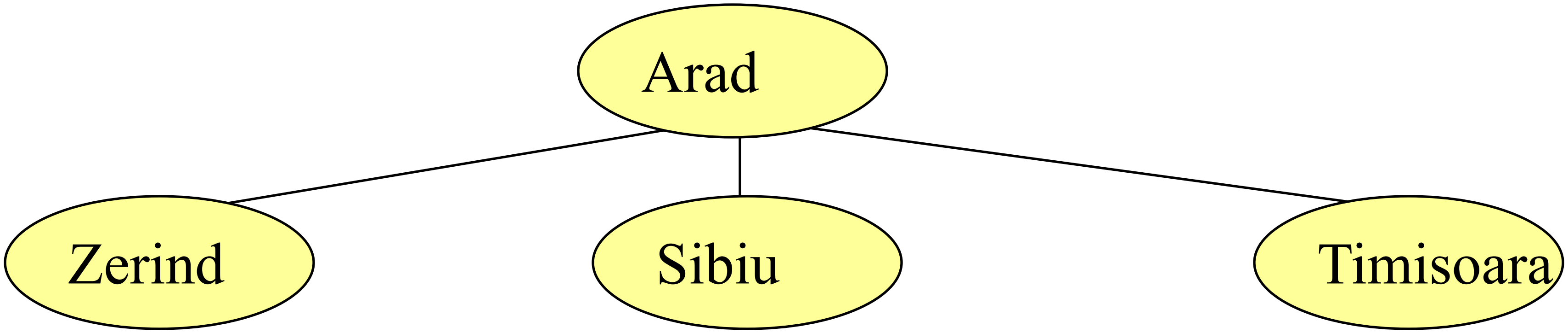
Possible paths



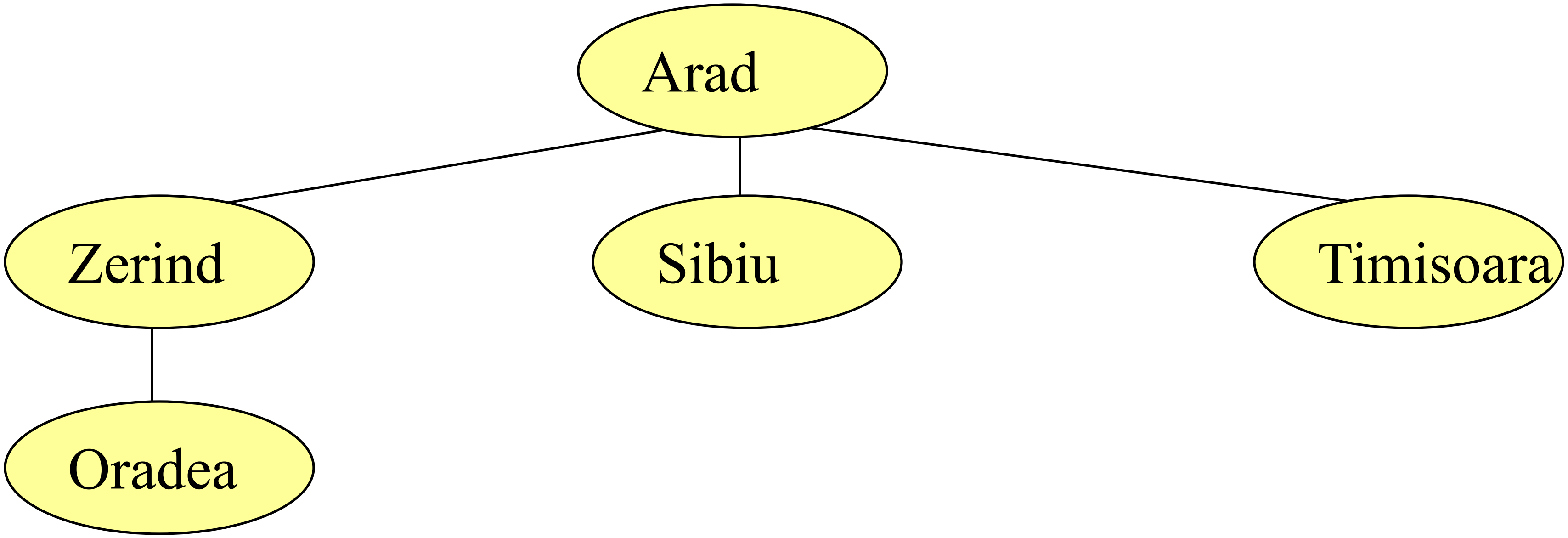
Possible paths



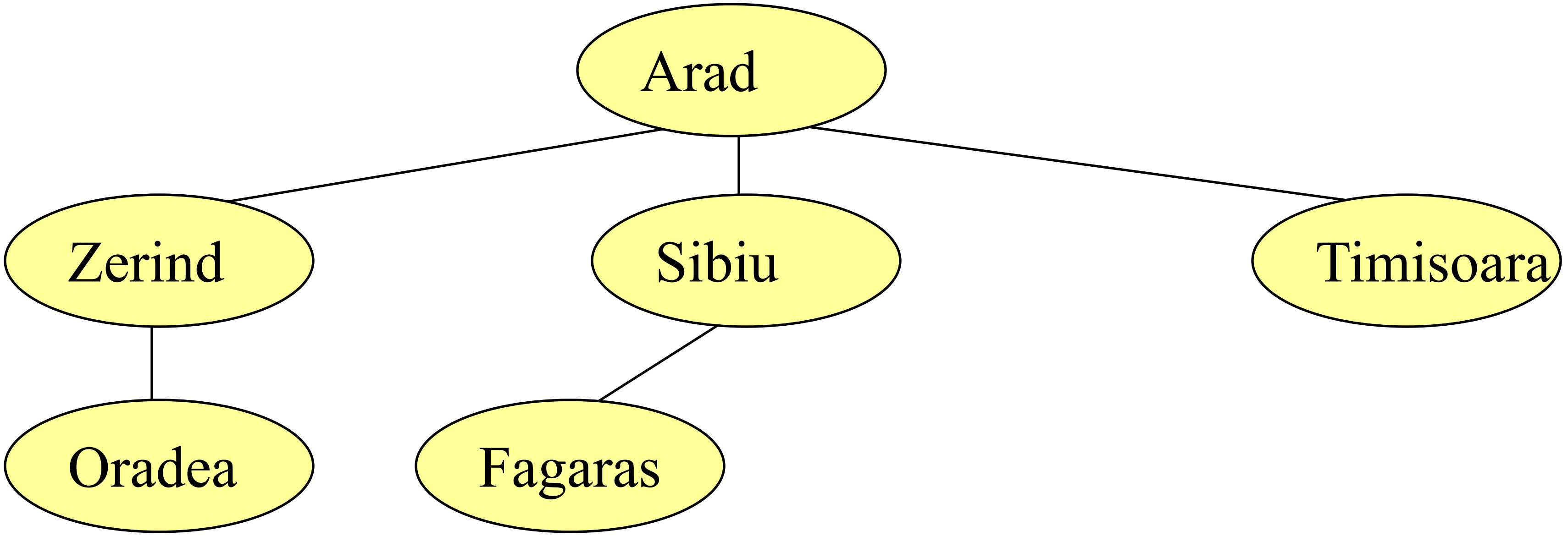
Possible paths



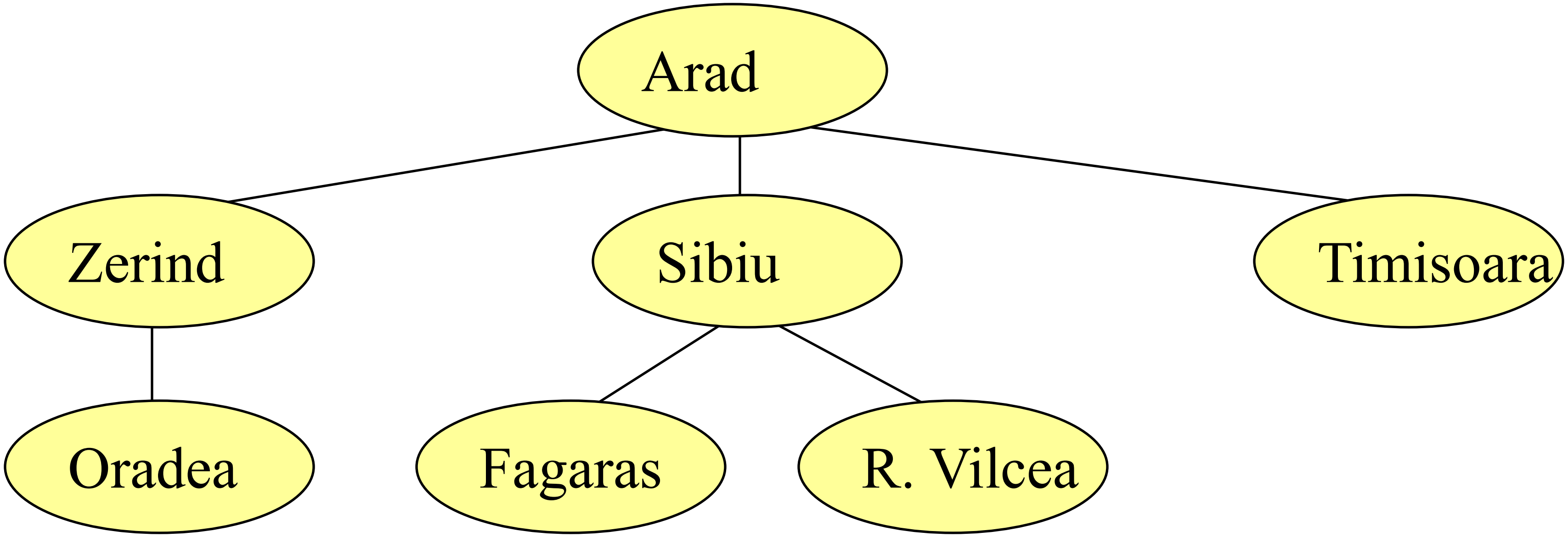
Possible paths



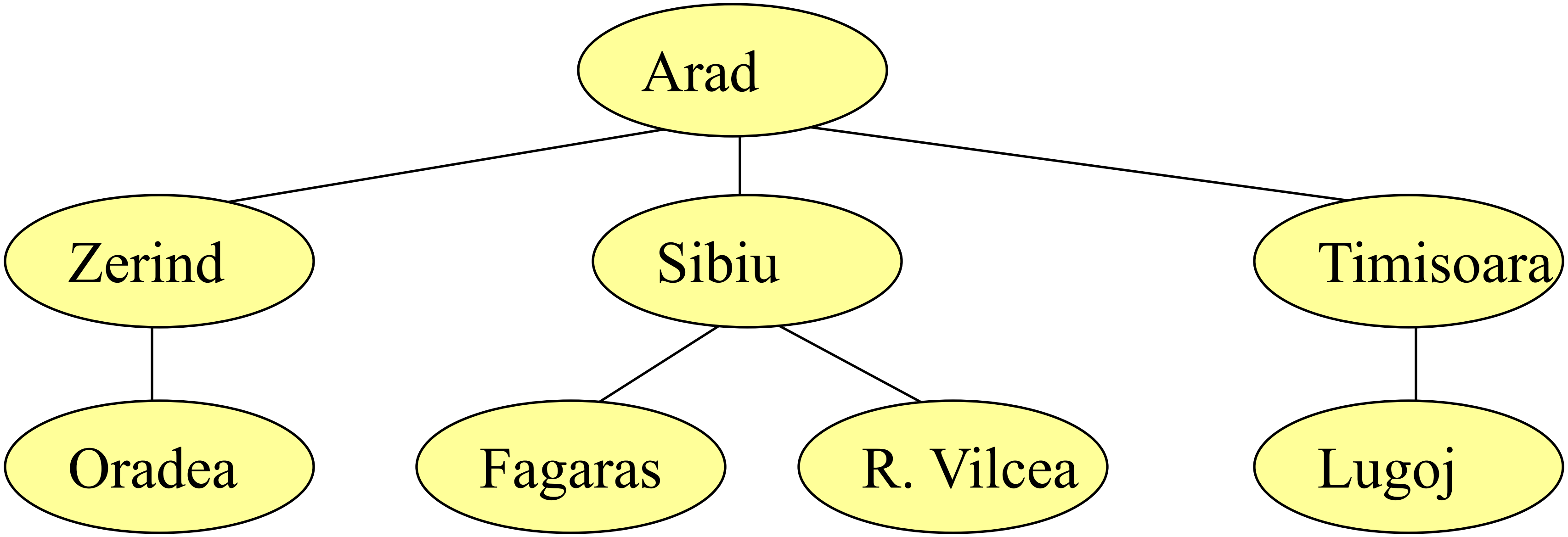
Possible paths



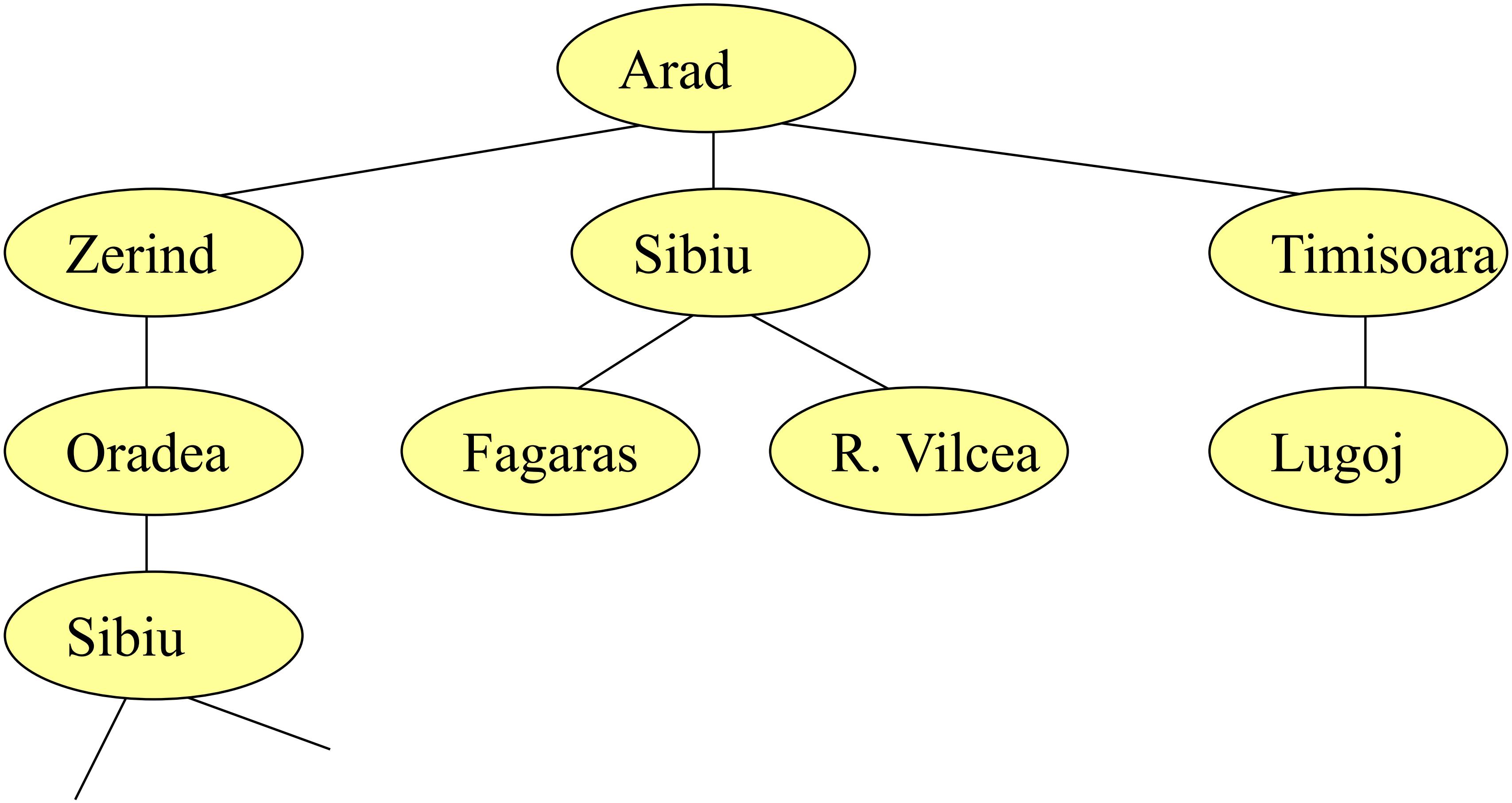
Possible paths



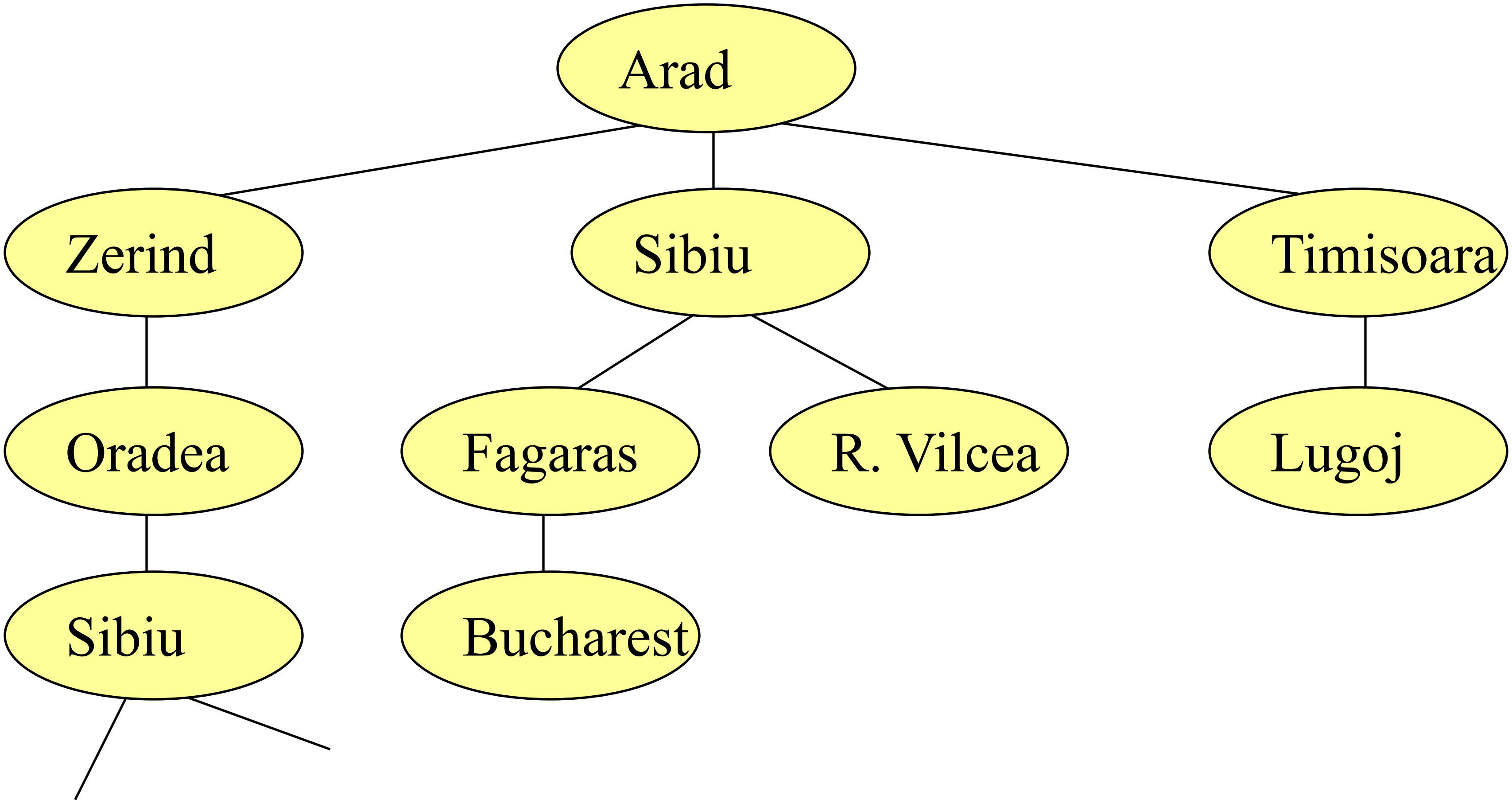
Possible paths



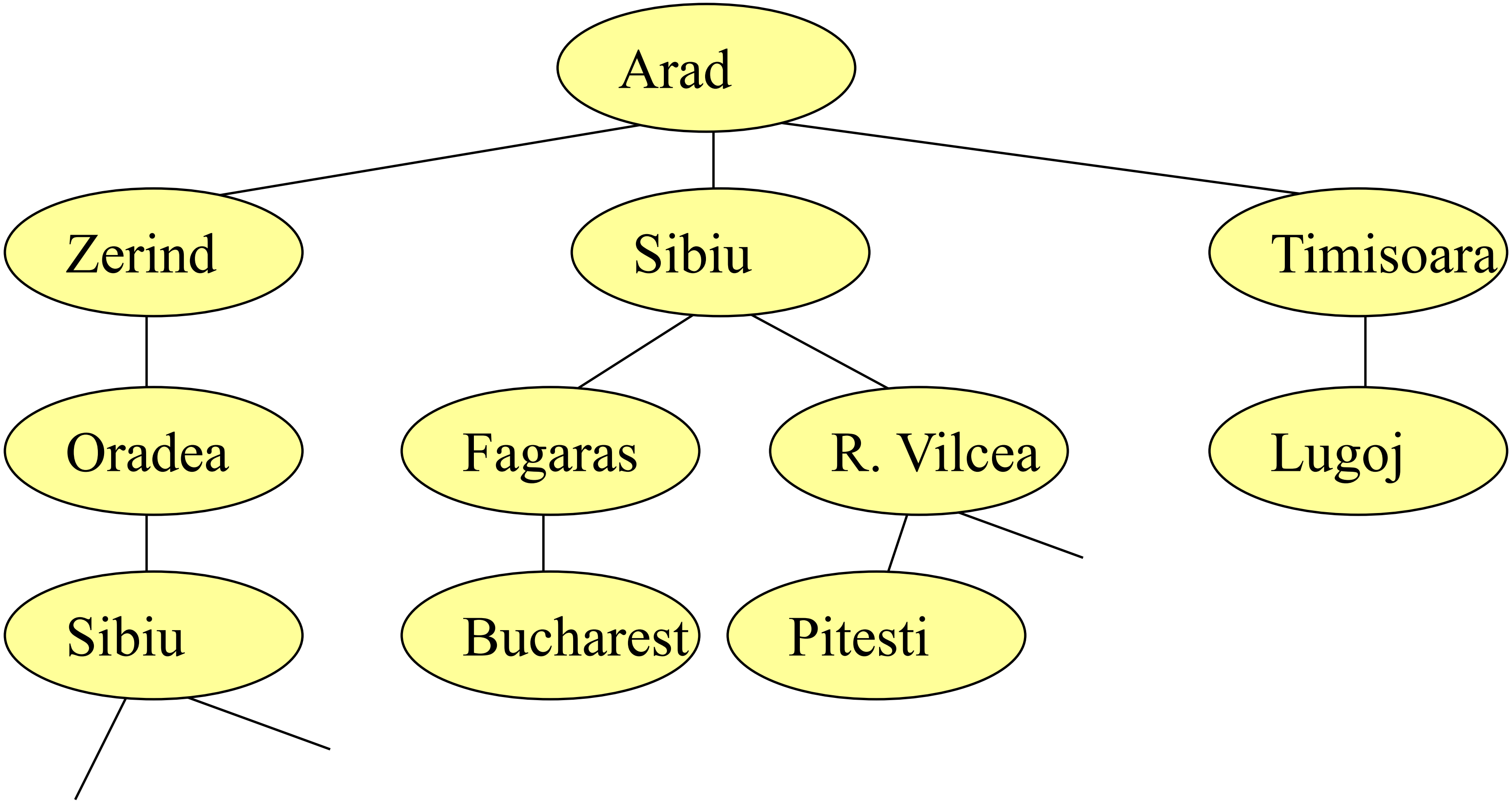
Possible paths



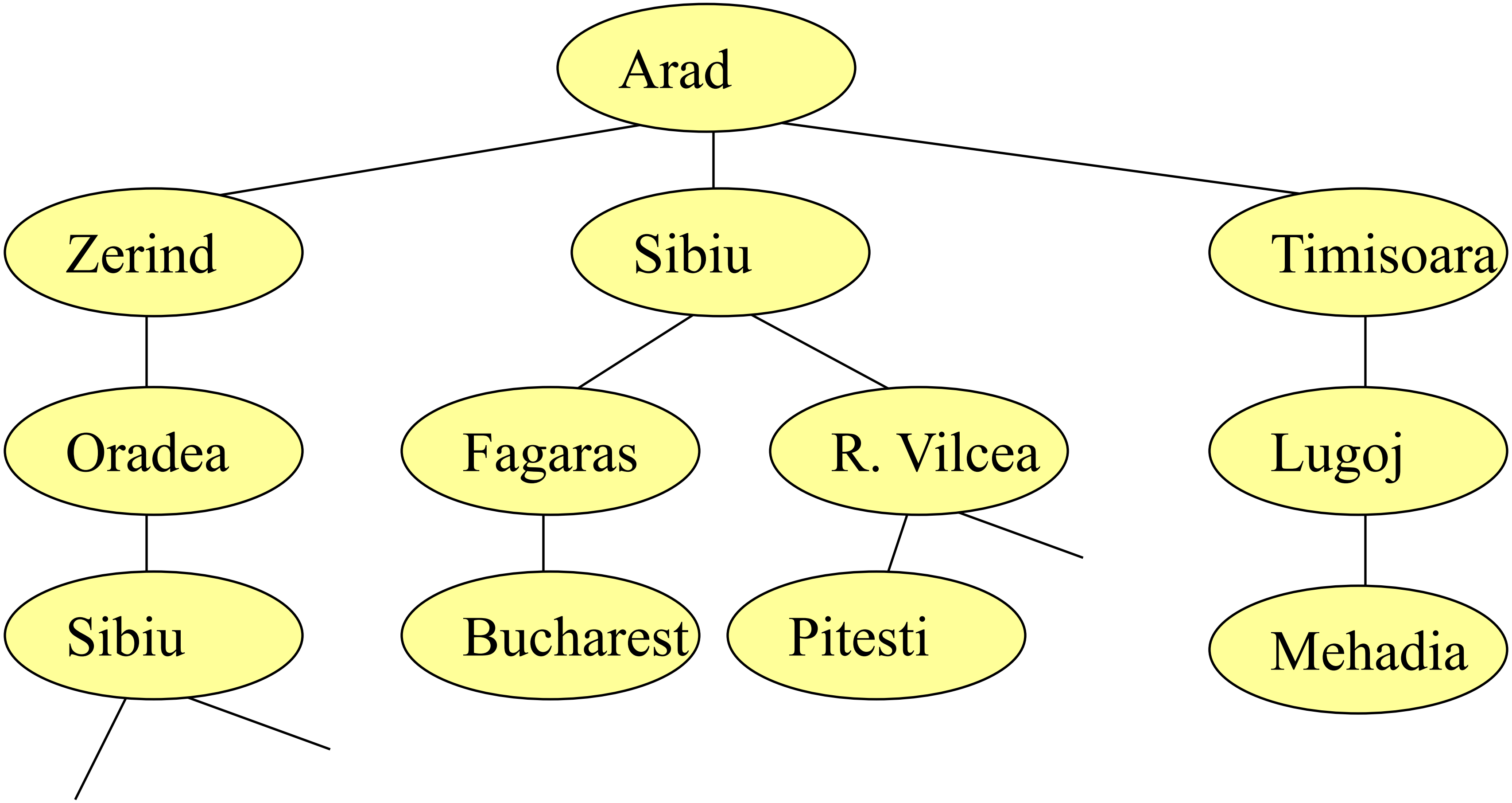
Possible paths



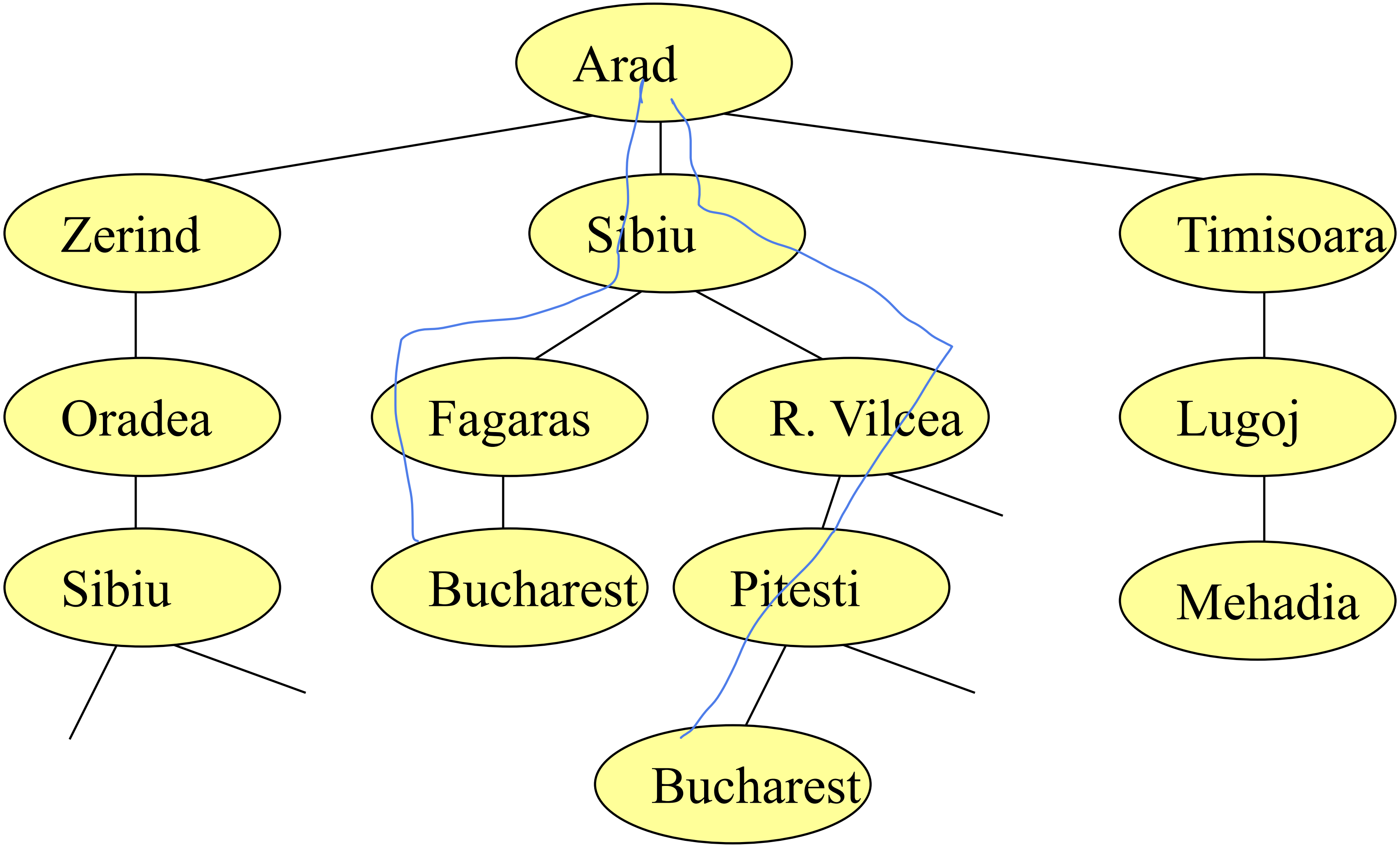
Possible paths



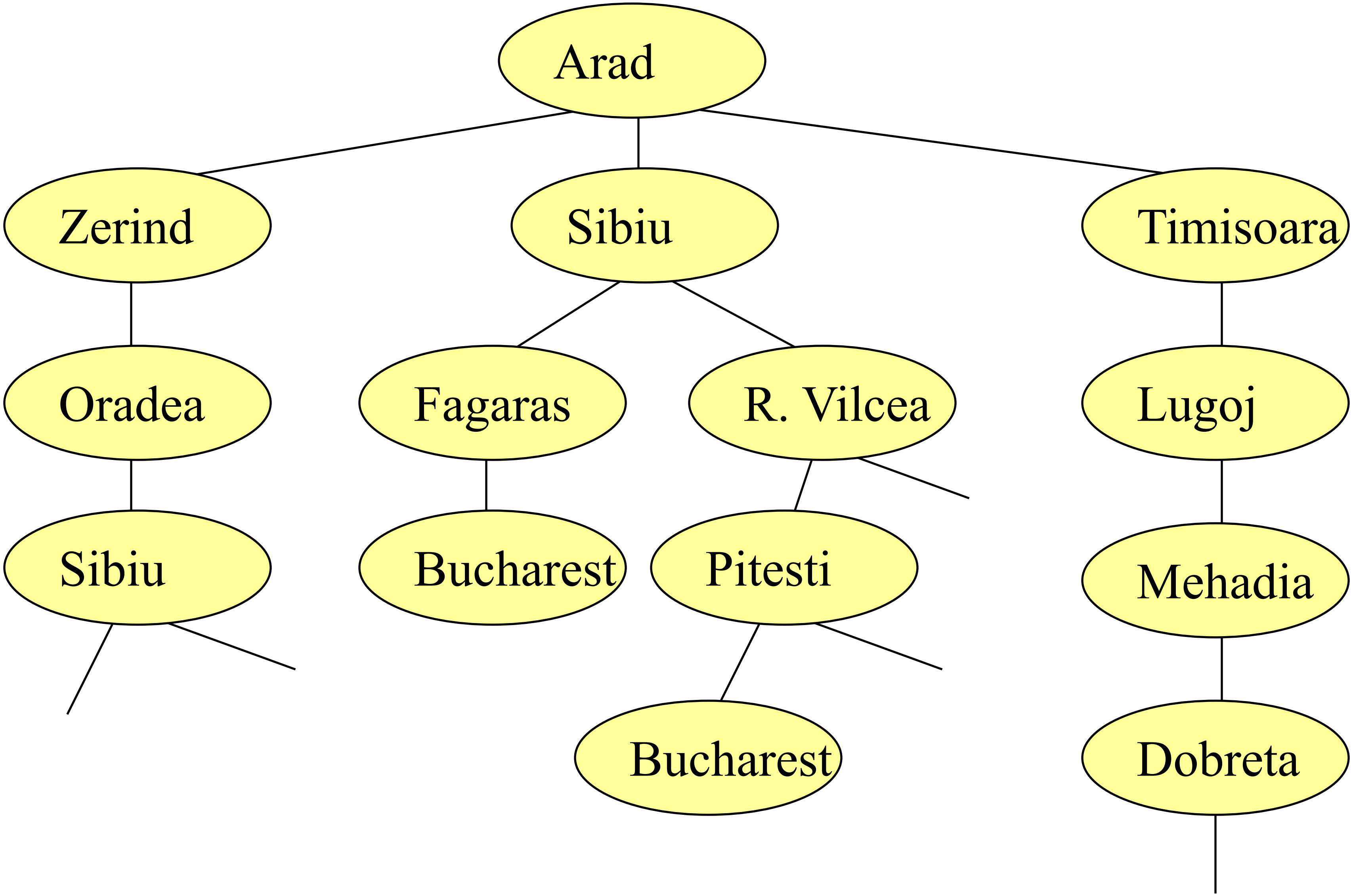
Possible paths



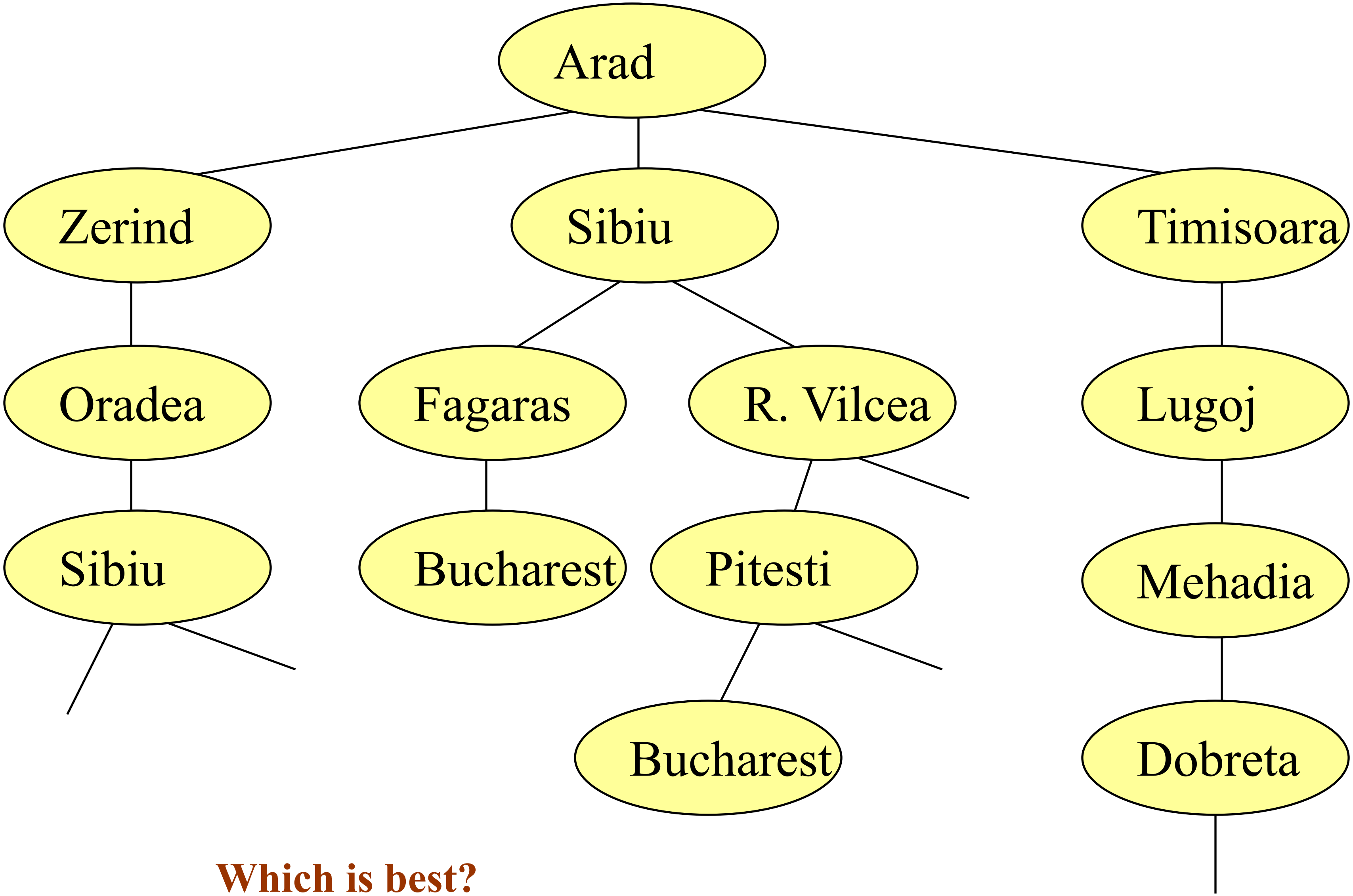
Possible paths



Possible paths

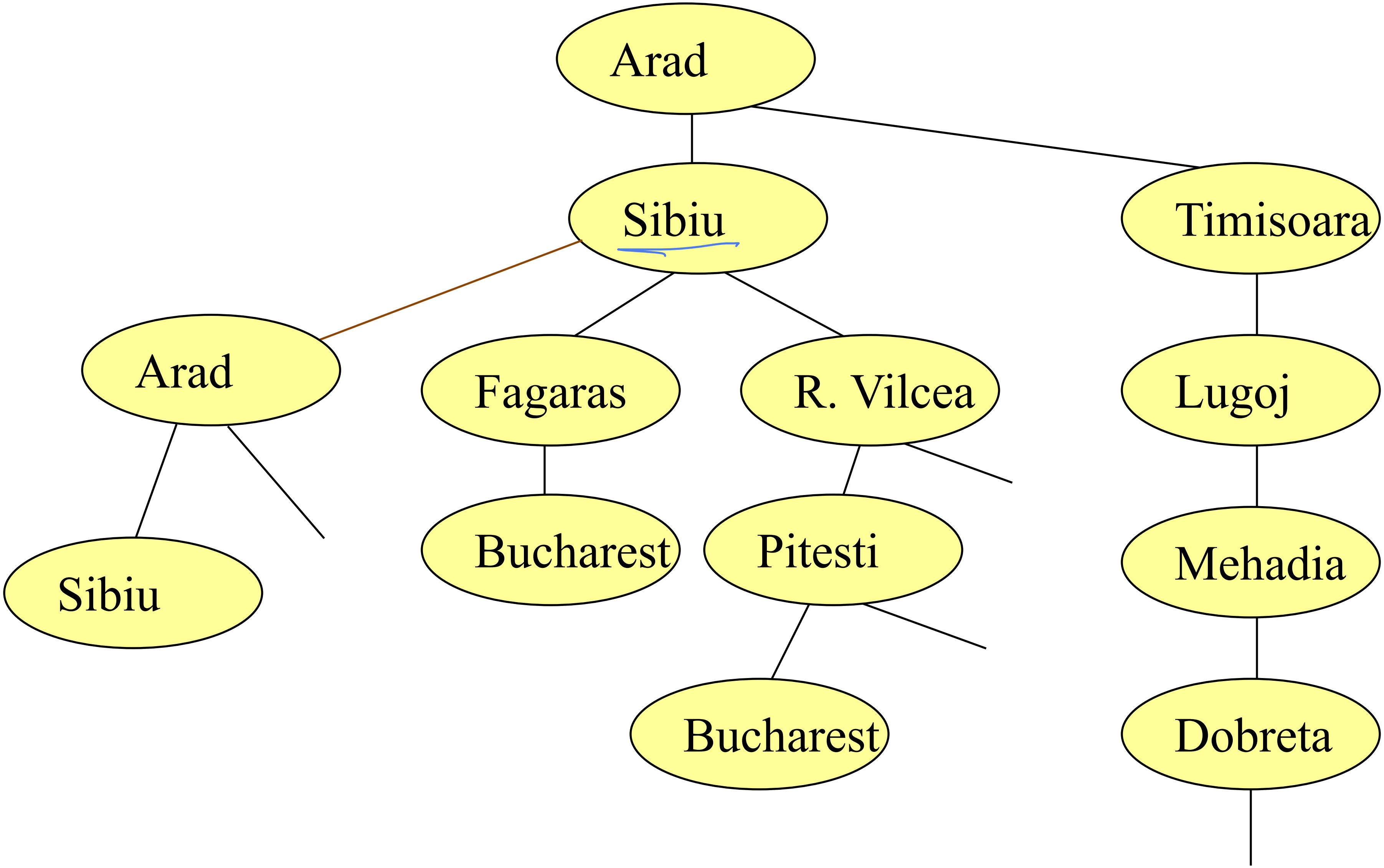


Possible paths

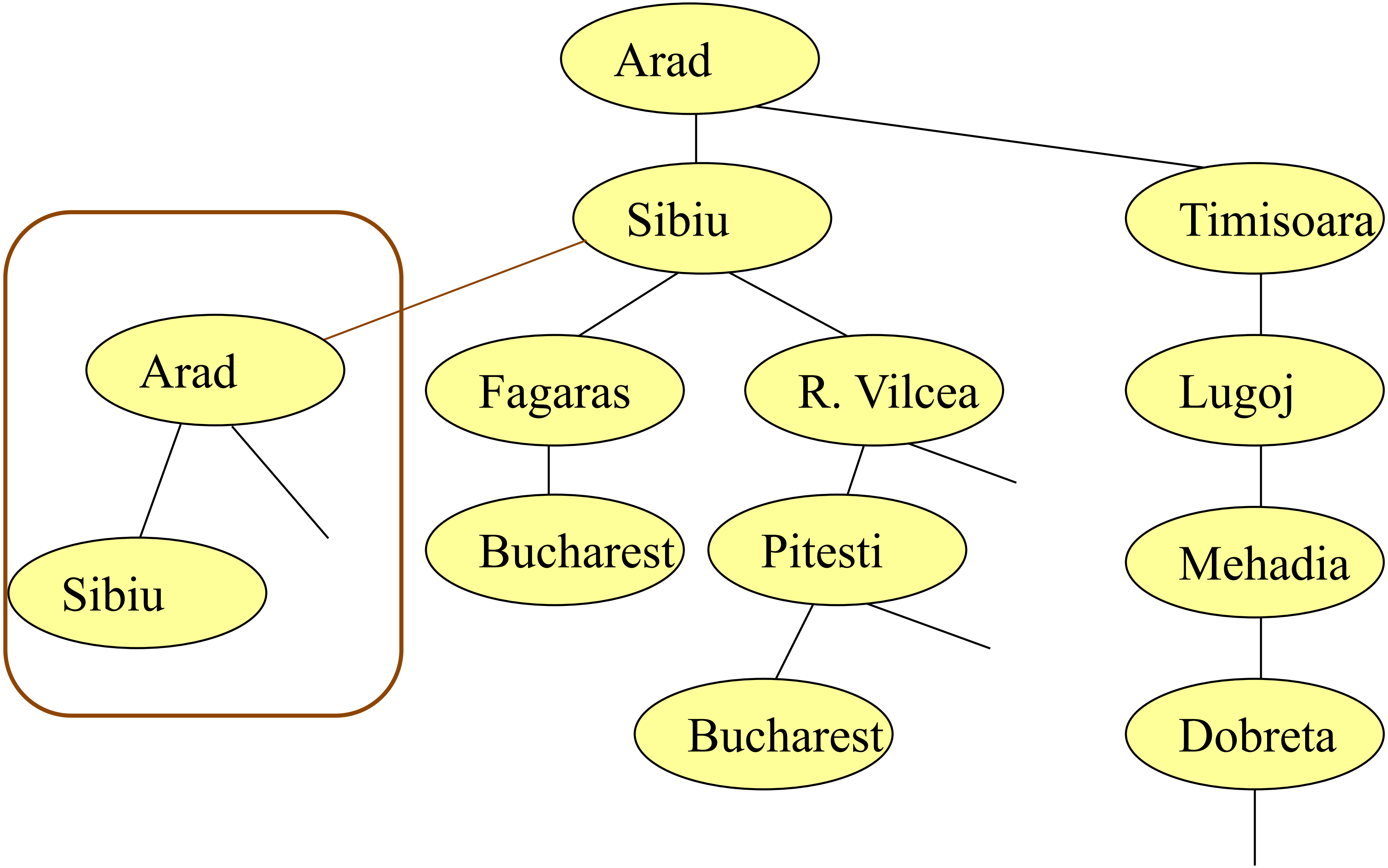


Which is best?

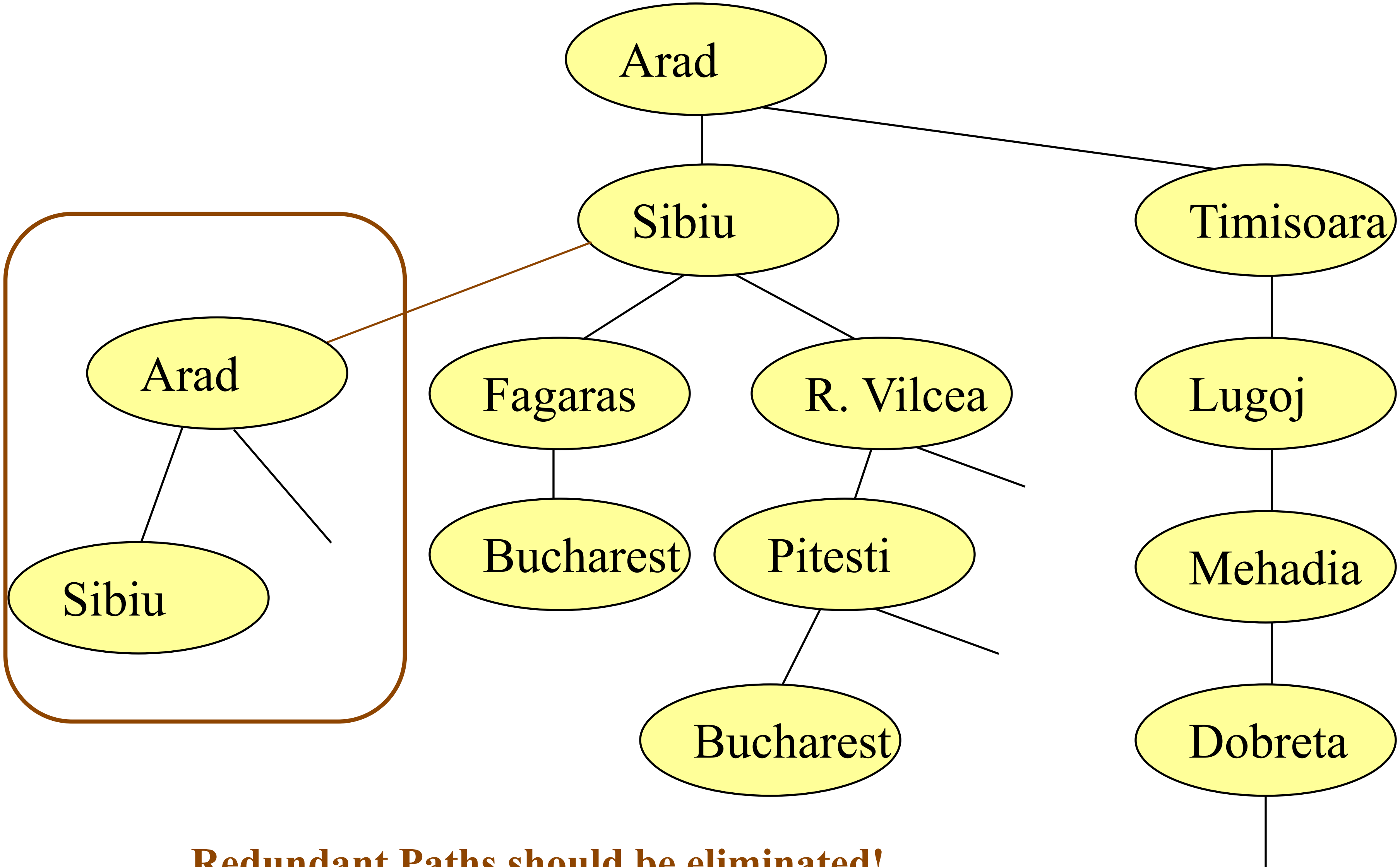
Should we consider cycles?



Should we consider cycles?



Should we consider cycles?



Redundant Paths should be eliminated!

Branching Factor and Depth

- If there are *b* possible choices at each state, then the **branching factor** is *b*

Branching Factor and Depth

- If there are b possible choices at each state, then the **branching factor** is b
- If it takes d steps (state transitions) to get to the goal state, then it may be the case that $O(b^d)$ states have to be checked

Branching Factor and Depth

- If there are b possible choices at each state, then the **branching factor** is b
- If it takes d steps (state transitions) to get to the goal state, then it may be the case that $O(b^d)$ states have to be checked
  - $b = 3, d = 5 \rightarrow b^d = 243$

Branching Factor and Depth

- If there are b possible choices at each state, then the **branching factor** is b
- If it takes d steps (state transitions) to get to the goal state, then it may be the case that $O(b^d)$ states have to be checked
  - $b = 3$, $d = 5 \rightarrow b^d = 243$
  - $b = 5$, $d = 10 \rightarrow b^d = 9{,}765{,}625$

Branching Factor and Depth

- If there are b possible choices at each state, then the **branching factor** is b
- If it takes d steps (state transitions) to get to the goal state, then it may be the case that $O(b^d)$ states have to be checked
  - $b = 3$, $d = 5 \rightarrow b^d = 243$
  - $b = 5$, $d = 10 \rightarrow b^d = 9,765,625$
  - $b = 8$, $d = 15 \rightarrow b^d = 35,184,372,088,832$

Branching Factor and Depth

- If there are b possible choices at each state, then the **branching factor** is b
- If it takes d steps (state transitions) to get to the goal state, then it may be the case that $O(b^d)$ states have to be checked
  - $b = 3, d = 5 \rightarrow b^d = 243$
  - $b = 5, d = 10 \rightarrow b^d = 9{,}765{,}625$
  - $b = 8, d = 15 \rightarrow b^d = 35{,}184{,}372{,}088{,}832$

- Ouch…. Combinatorial explosion!

Abstraction

- The real world is highly complex!
  - The state space must be *abstracted* for problem-solving
    - Simplify and aggregate
      - Can't represent all the details

- Choosing a good abstraction
  - Keep only those relevant for the problem
  - Remove as much detail as possible *while retaining validity*

Problem Solving Agents

Problem Solving Agents

- Task: Find a sequence of actions that leads to desirable (goal) states
  - Must define *problem* and *solution*

Problem Solving Agents

- Task: Find a sequence of actions that leads to desirable (goal) states
  - Must define *problem* and *solution*
- Finding a solution is typically a *search process* in the problem space
  - Solution = A path through the state space from the initial state to a goal state
  - <u>Optimal</u> search find the *least-cost* solution

Problem Solving Agents

- Task: Find a sequence of actions that leads to desirable (goal) states
  - Must define *problem* and *solution*
- Finding a solution is typically a *search process* in the problem space
  - Solution = A path through the state space from the initial state to a goal state
  - <u>Optimal</u> search find the *least-cost* solution
- Search algorithm
  - Input: Problem statement (incl. goal)
  - Output: Sequence of actions that leads to a solution

Problem Solving Agents

- Task: Find a sequence of actions that leads to desirable (goal) states
  - Must define *problem* and *solution*
- Finding a solution is typically a *search process* in the problem space
  - Solution = A path through the state space from the initial state to a goal state
  - <u>Optimal</u> search find the *least-cost* solution
- Search algorithm
  - Input: Problem statement (incl. goal)
  - Output: Sequence of actions that leads to a solution
- Formulate, search, execute (action)

Problem Formulation and Search

Problem Formulation and Search

- Problem formulation

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states
    - **S_0**: Initial state of the agent

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states
    - **S_0**: Initial state of the agent
    - **S_G**: Goal state(s)
      - Or equivalently, a goal test **G(S)**

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, $\mathbf{S_0}$, {$\mathbf{S_G}$}, {O}, {g} >**
    - **S**: Possible states
    - $\mathbf{S_0}$: Initial state of the agent
    - $\mathbf{S_G}$: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states
    - **S_0**: Initial state of the agent
    - **S_G**: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent
    - **g**: Path cost function, assigns a cost to a path/action

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states
    - $\mathbf{S_0}$: Initial state of the agent
    - $\mathbf{S_G}$: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent
    - **g**: Path cost function, assigns a cost to a path/action
- At any given time, which possible action $\boldsymbol{O_i}$ is best?

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states
    - **S_0**: Initial state of the agent
    - **S_G**: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent
    - **g**: Path cost function, assigns a cost to a path/action
- At any given time, which possible action $\boldsymbol{O_i}$ is best?
  - Depends on the goal, the path cost function, the future sequence of actions….

Problem Formulation and Search

- Problem formulation
  - State-space description **< {S}, S_0, {S_G}, {O}, {g} >**
    - **S**: Possible states
    - $\mathbf{S_0}$: Initial state of the agent
    - $\mathbf{S_G}$: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent
    - **g**: Path cost function, assigns a cost to a path/action
- At any given time, which possible action $\boldsymbol{O_i}$ is best?
  - Depends on the goal, the path cost function, the future sequence of actions….
- Agent's strategy: Formulate, Search, and Execute

Problem Formulation and Search

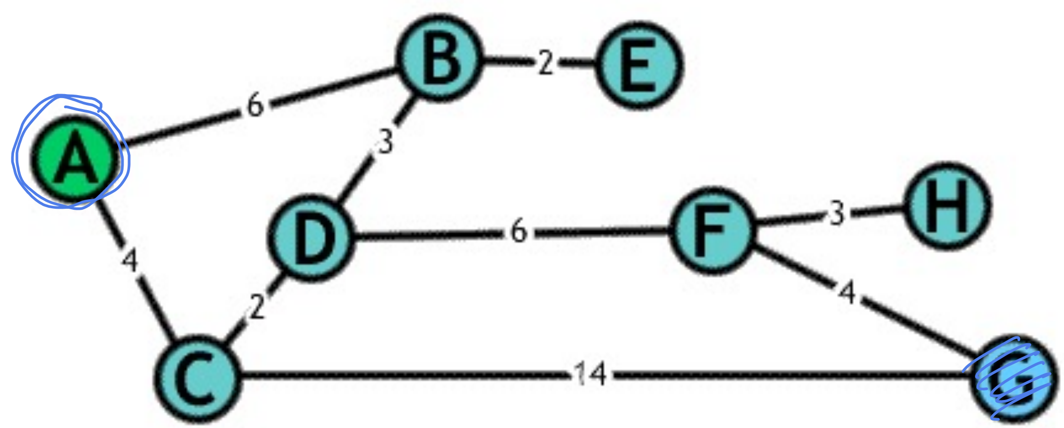
- Problem formulation
  - State-space description **< {S}, $\mathbf{S_0}$, {$\mathbf{S_G}$}, {O}, {g} >**
    - **S**: Possible states
    - $\mathbf{S_0}$: Initial state of the agent
    - $\mathbf{S_G}$: Goal state(s)
      - Or equivalently, a goal test **G(S)**
    - **O**: Operators O: {S} => {S}
      - Describes the possible actions of the agent
    - **g**: Path cost function, assigns a cost to a path/action
- At any given time, which possible action $\boldsymbol{O_i}$ is best?
  - Depends on the goal, the path cost function, the future sequence of actions….
- Agent's strategy: Formulate, Search, and Execute
  - This is *offline* problem solving

State-Space Diagrams

- State-space description can be represented by a state-space diagram, which shows
  - States (incl. initial and goal)
  - Operators/actions (state transitions)
  - Path costs

State-Space Diagrams

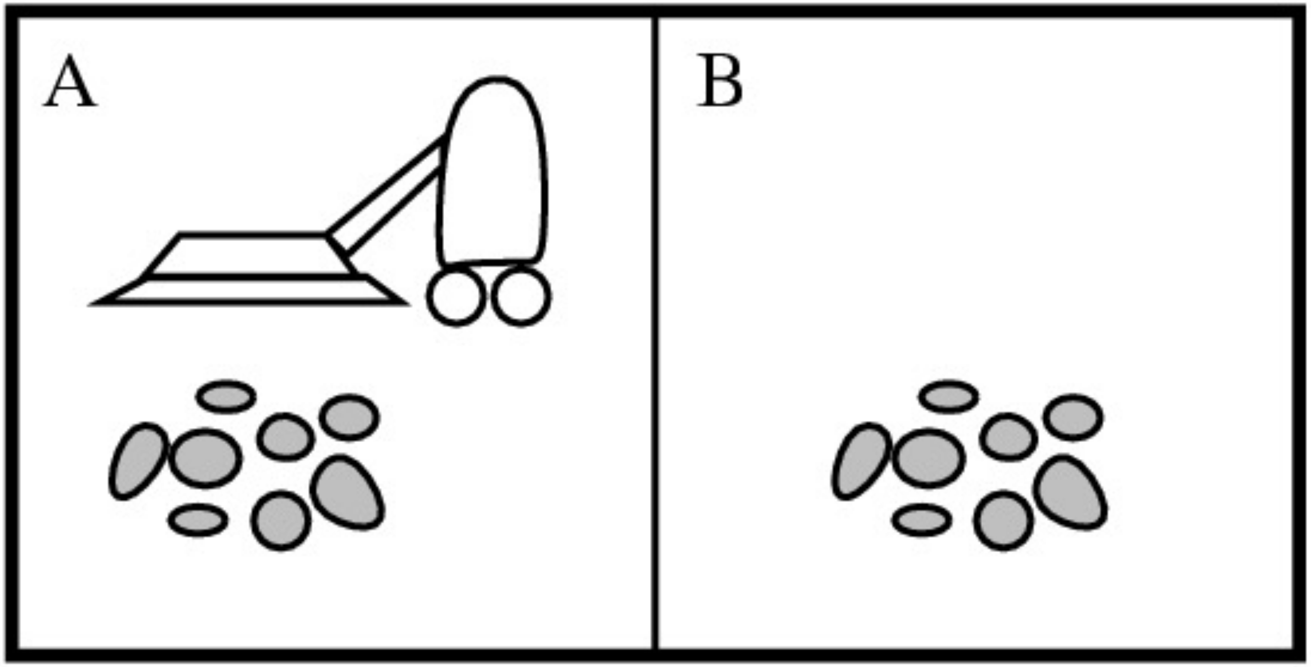
- State-space description can be represented by a state-space diagram, which shows
  - States (incl. initial and goal)
  - Operators/actions (state transitions)
  - Path costs



Typical assumptions

- Environment is observable
- Environment is static
- Environment is discrete
- Environment is deterministic

Example: The Vacuum World



The Vacuum World

- Simplified world: 2 grids

The Vacuum World

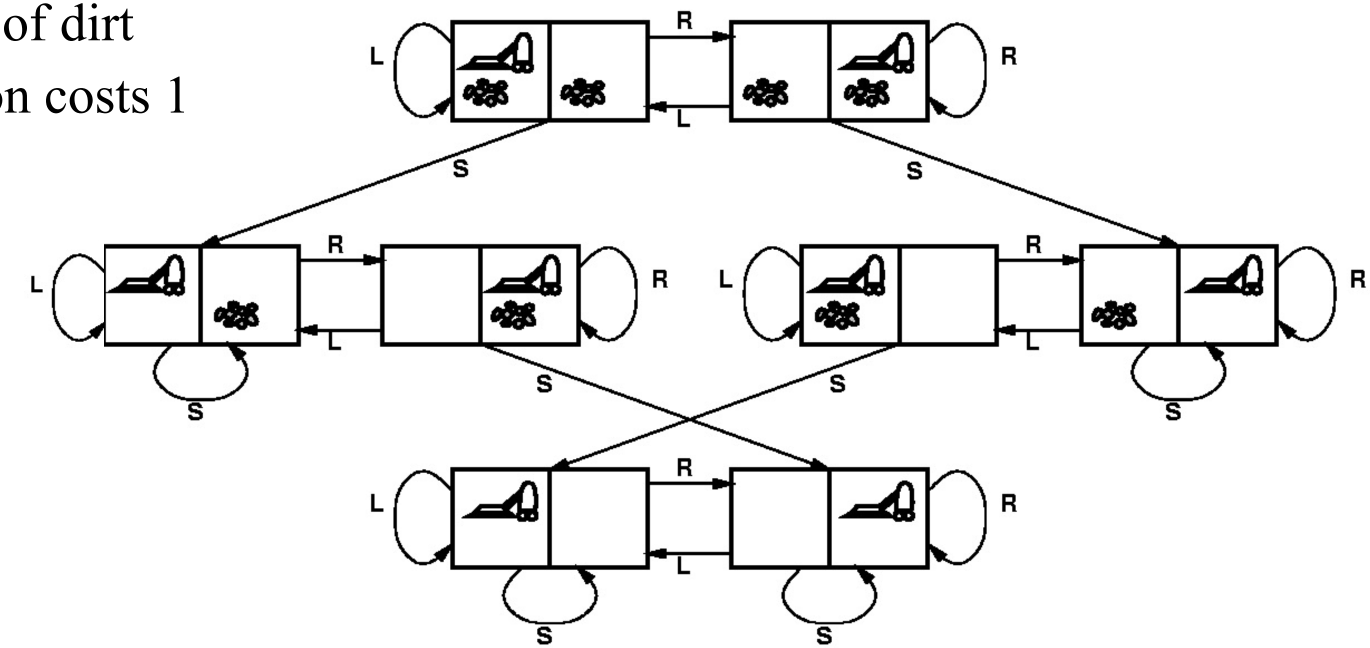
- Simplified world: 2 grids

States: Location of vacuum, dirt in grids

Operators: Move left, move right, suck dirt

Goal test: Grids free of dirt

Path cost: Each action costs 1



The Vacuum World

- Simplified world: 2 grids

3^n

n^3 ?

$2^2 \times 2$

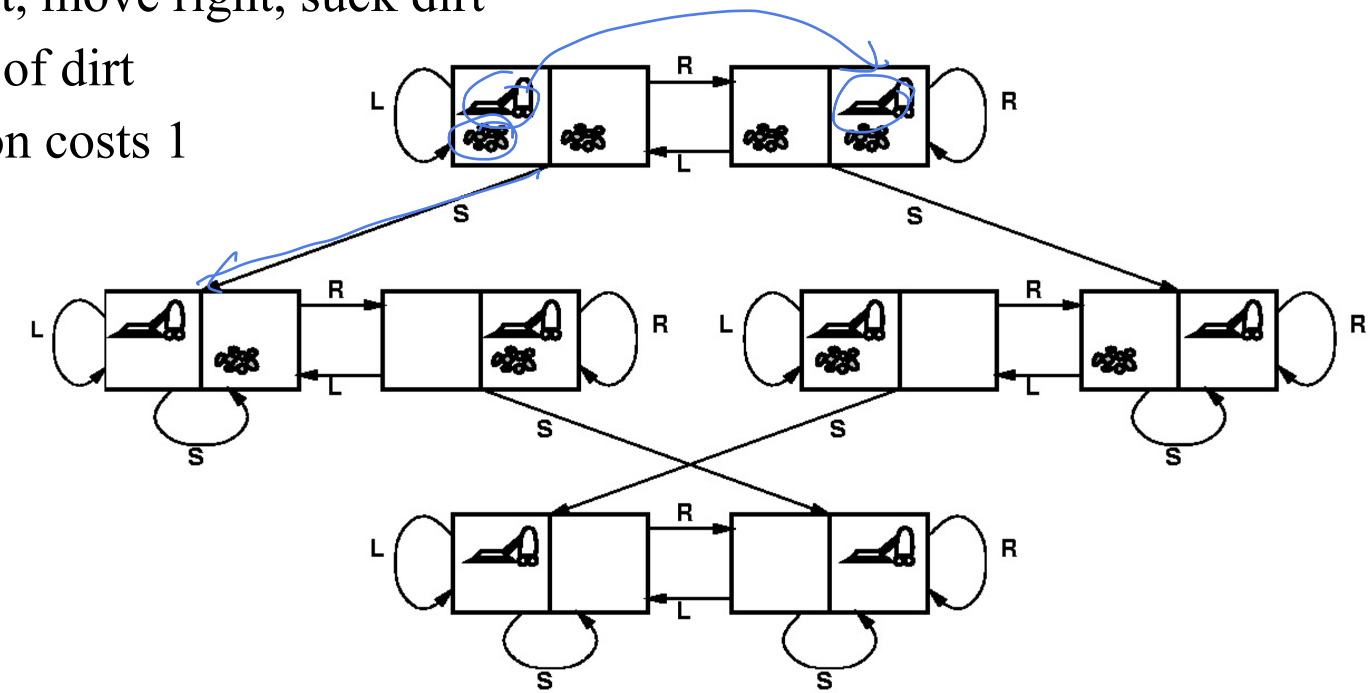
$2^n \times n$

States: Location of vacuum, dirt in grids

Operators: Move left, move right, suck dirt

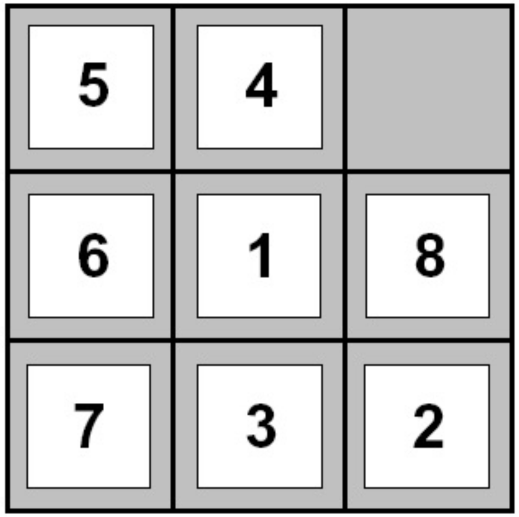
Goal test: Grids free of dirt

Path cost: Each action costs 1

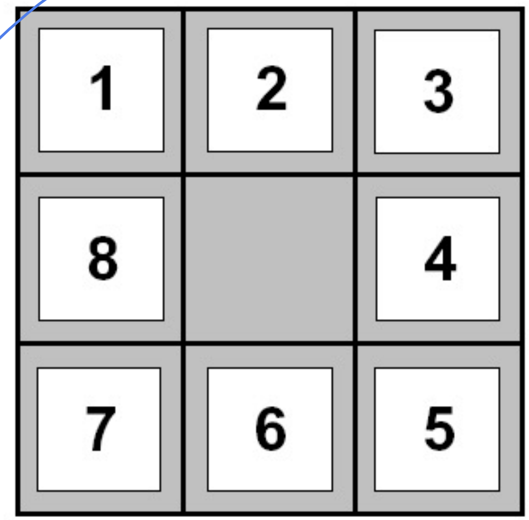


How many states for n grids?

Example Problem: 8-Puzzle



Start State



Goal State

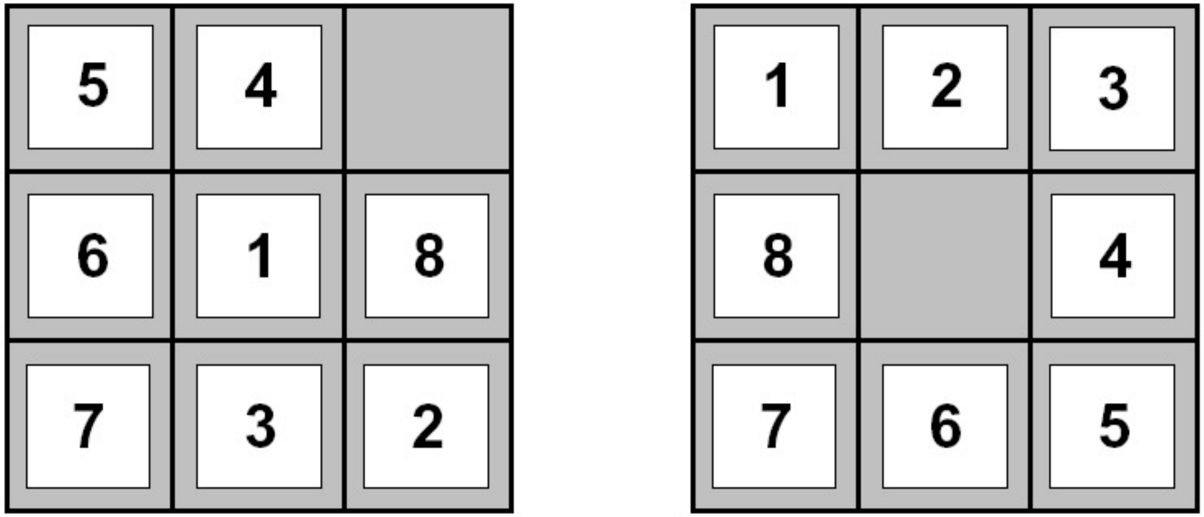
States: 9!

Operators: 4 ↑ → ↓ ←

goal:

Cost:

Example Problem: 8-Puzzle



Start State　　　　Goal State

States: Various configurations of the puzzle

Operators: Movements of the blank

Goal test: Goal configuration

Path cost: Each move costs 1

Example Problem: 8-Puzzle

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

States: Various configurations of the puzzle

Operators: Movements of the blank

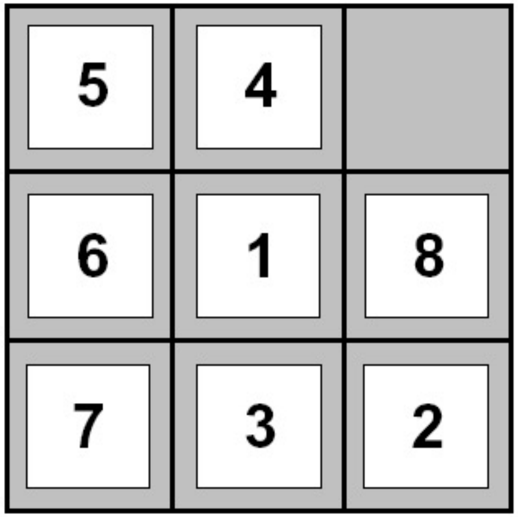
Goal test: Goal configuration

Path cost: Each move costs 1

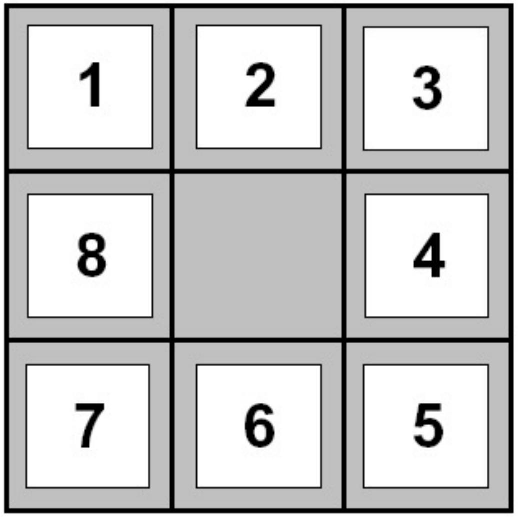
How many states are there?

9!

Example Problem: 8-Puzzle



Start State



Goal State

States: Various configurations of the puzzle
Operators: Movements of the blank
Goal test: Goal configuration
Path cost: Each move costs 1

How many states are there?

9! = 362,880

8-Puzzle is hard (by definition)!

- Optimal solution of the N-puzzle family of problems is NP-complete
  - Likely exponential increase in computation with N
  - Uninformed search will do very poorly

8-Puzzle is hard (by definition)!

- Optimal solution of the N-puzzle family of problems is NP-complete
  – Likely exponential increase in computation with N
  – Uninformed search will do very poorly

- Ditto for the Traveling Salesman Problem (TSP)
  – Start and end in Bucharest, visit every city at least once
  – Find the shortest tour

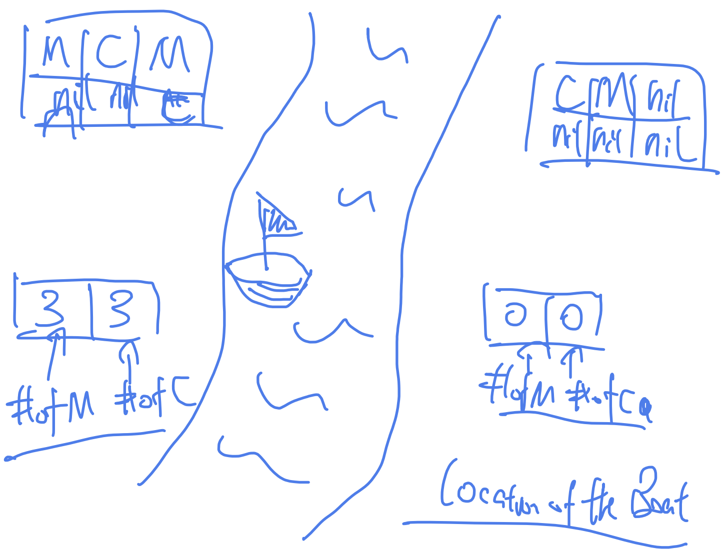
8-Puzzle is hard (by definition)!

- Optimal solution of the N-puzzle family of problems is NP-complete
  - Likely exponential increase in computation with N
  - Uninformed search will do very poorly
- Ditto for the Traveling Salesman Problem (TSP)
  - Start and end in Bucharest, visit every city at least once
  - Find the shortest tour
- Ditto for lots of interesting problems!

Example: Missionaries and Cannibals (3 min discussion)

Problem: Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place

- States, operators, goal test, path cost?



LHS RHS Boat

| 3 3 | 0 0 | 0 |

↓ 2 Ms ferrying

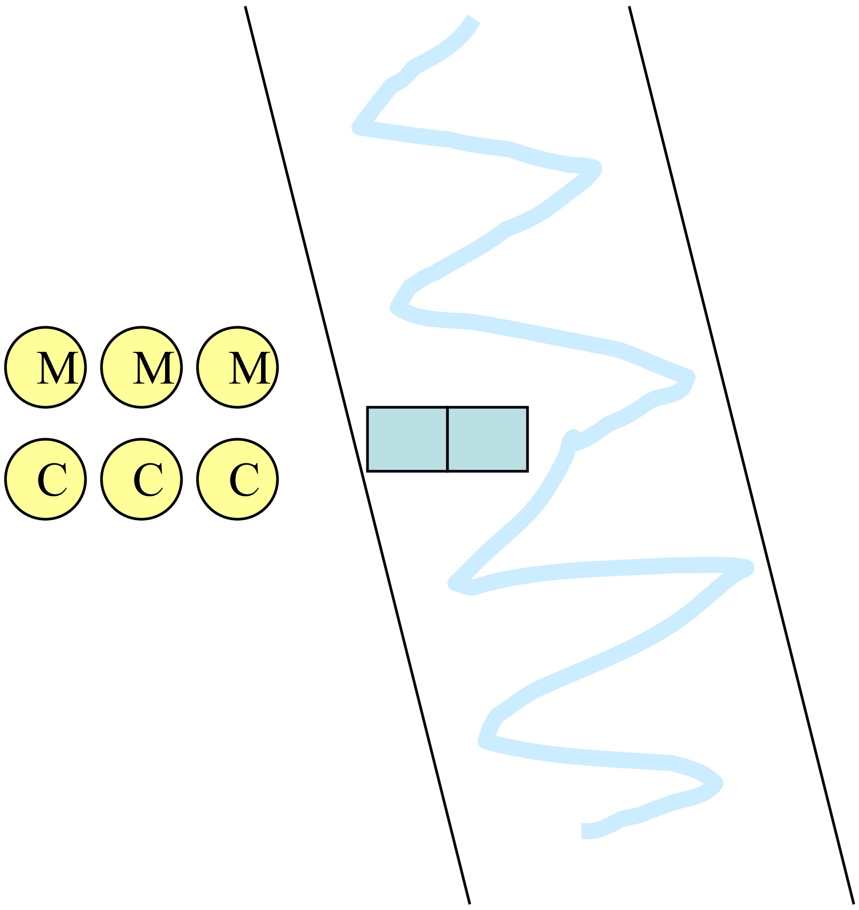
| 1 3 | 2 0 | 1 |

M&C (cont.)

- Initial state
- Goal state

M&C (cont.)

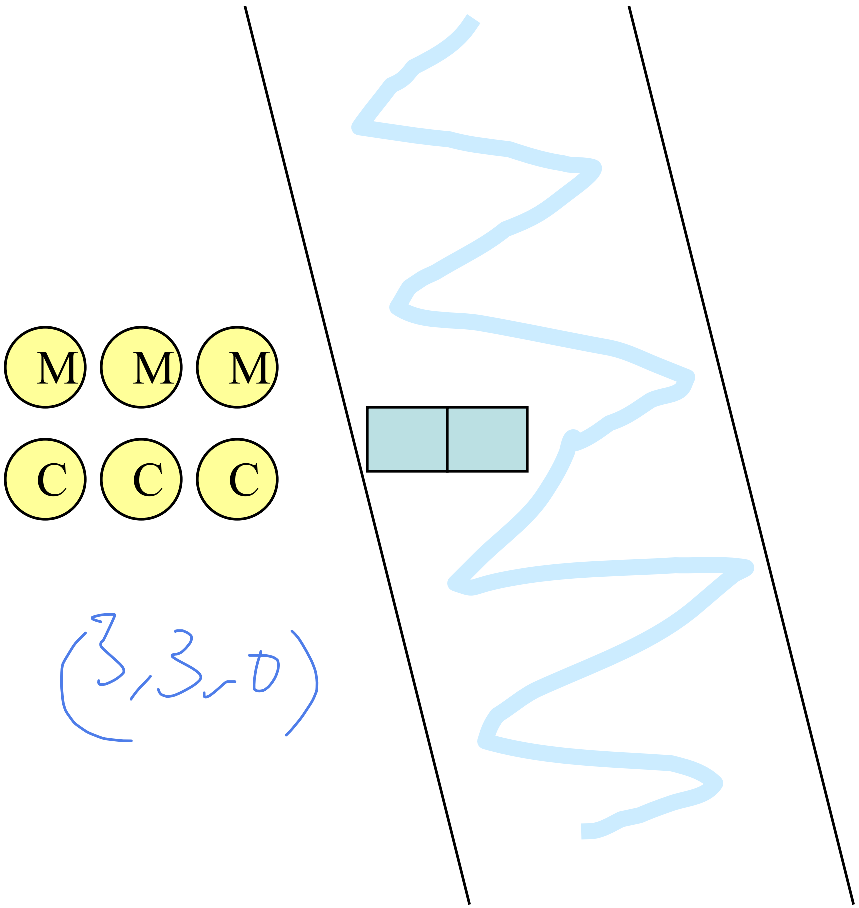
- Initial state



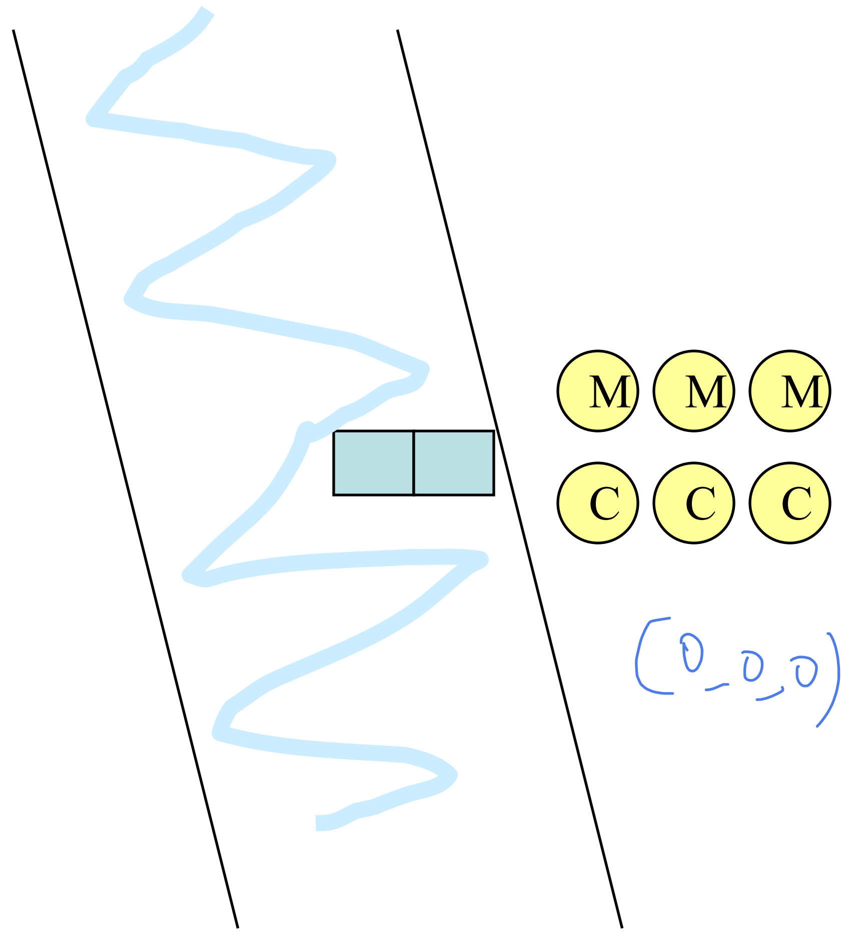
- Goal state

M&C (cont.)

- Initial state

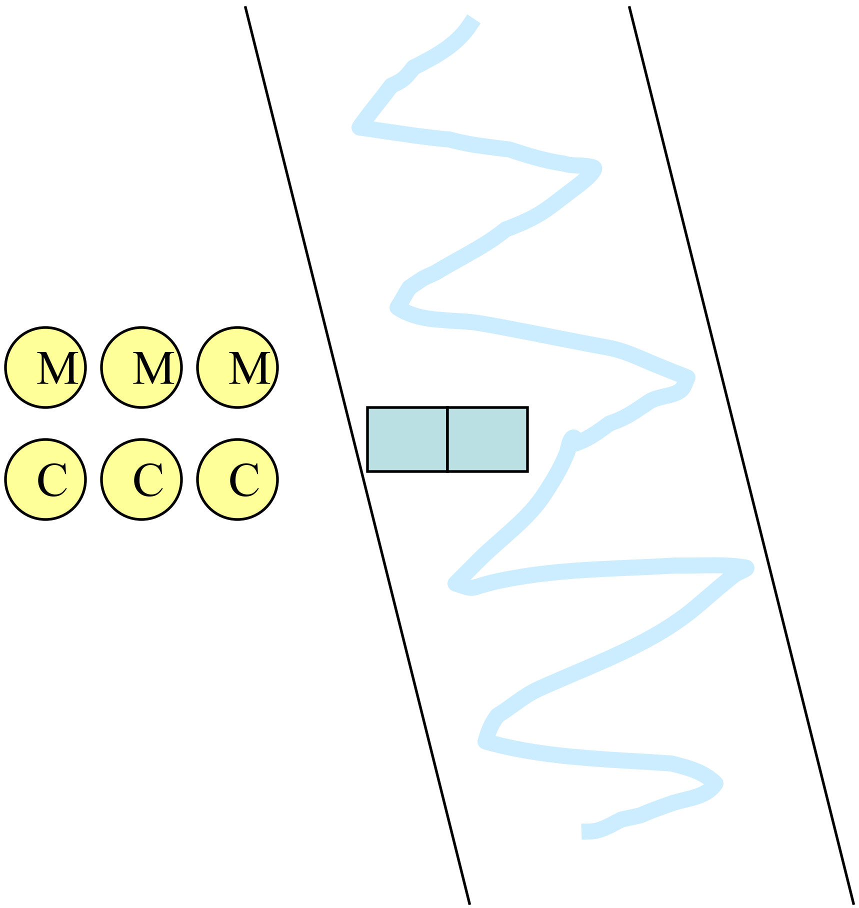


- Goal state

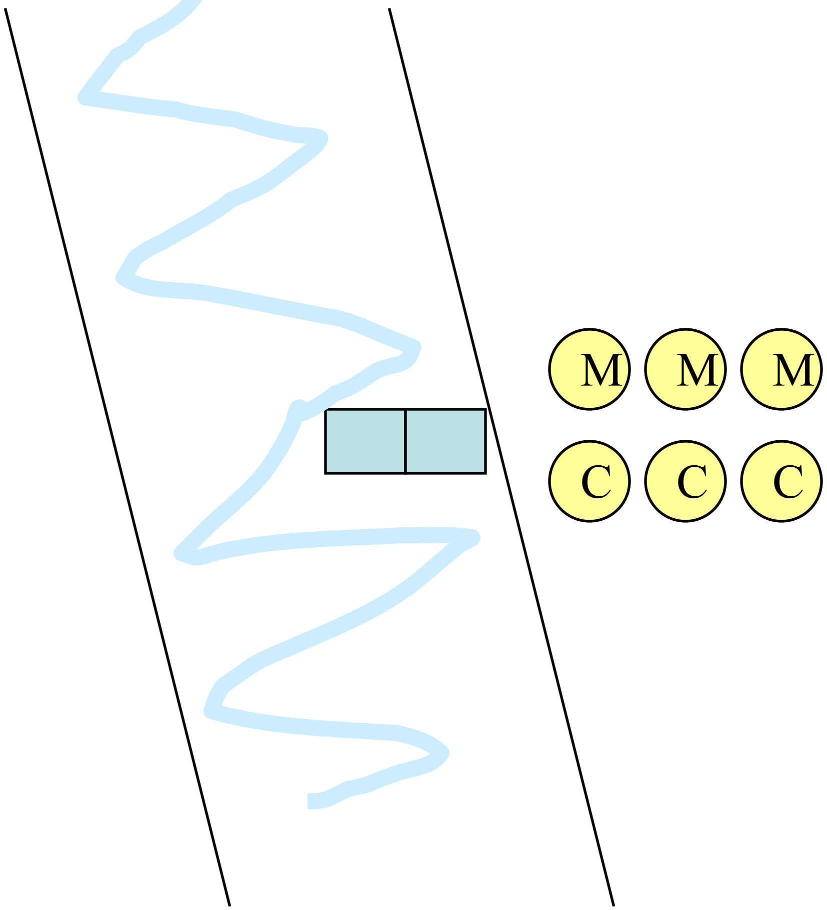


M&C (cont.)

- Initial state



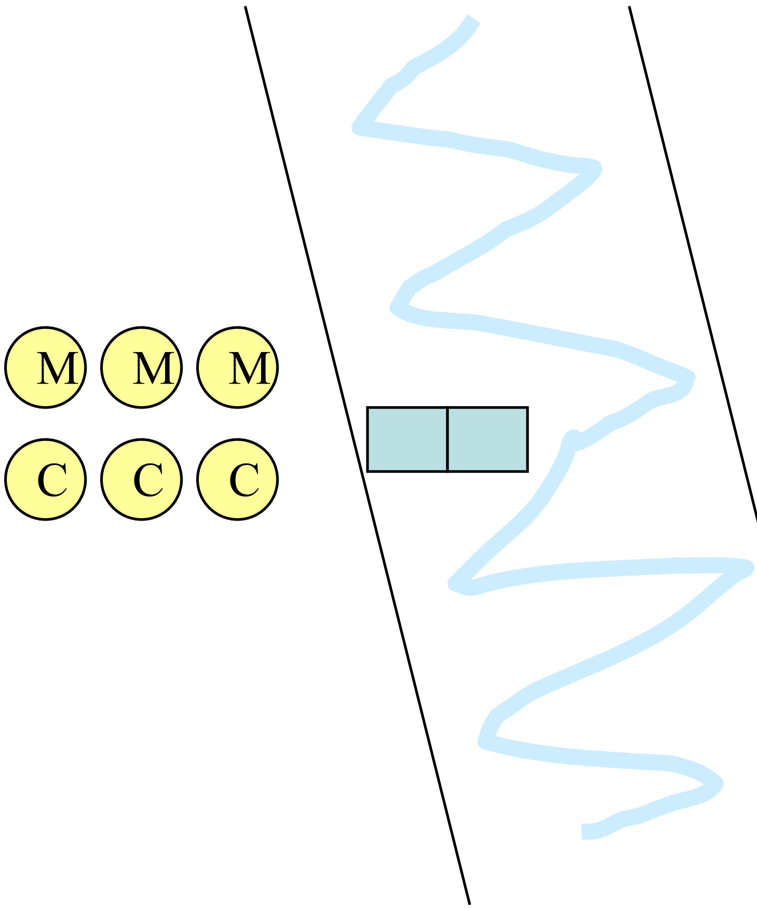
- Goal state



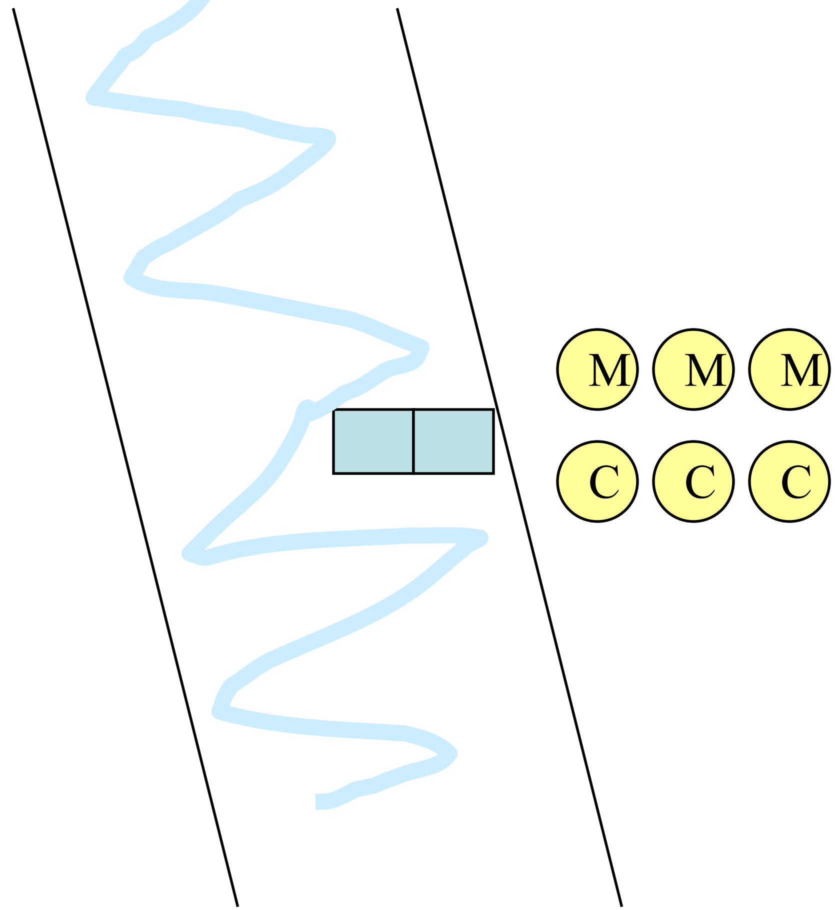
$(M_L \;\; C_L \;\; B_L)$

M&C (cont.)

- Initial state



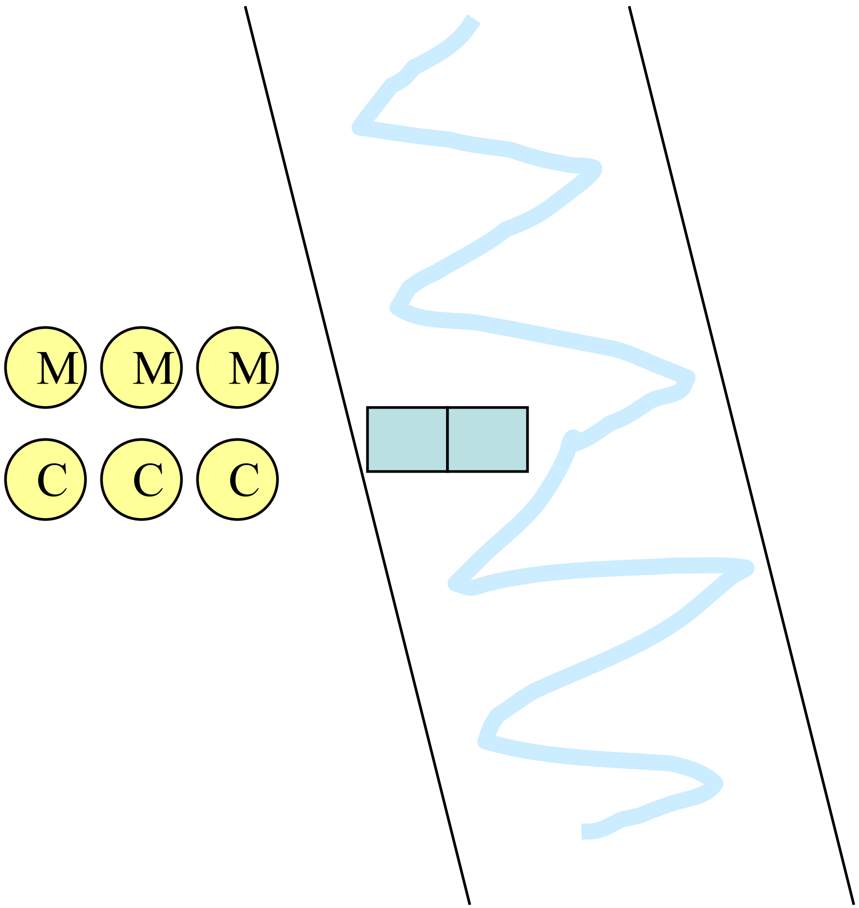
- Goal state

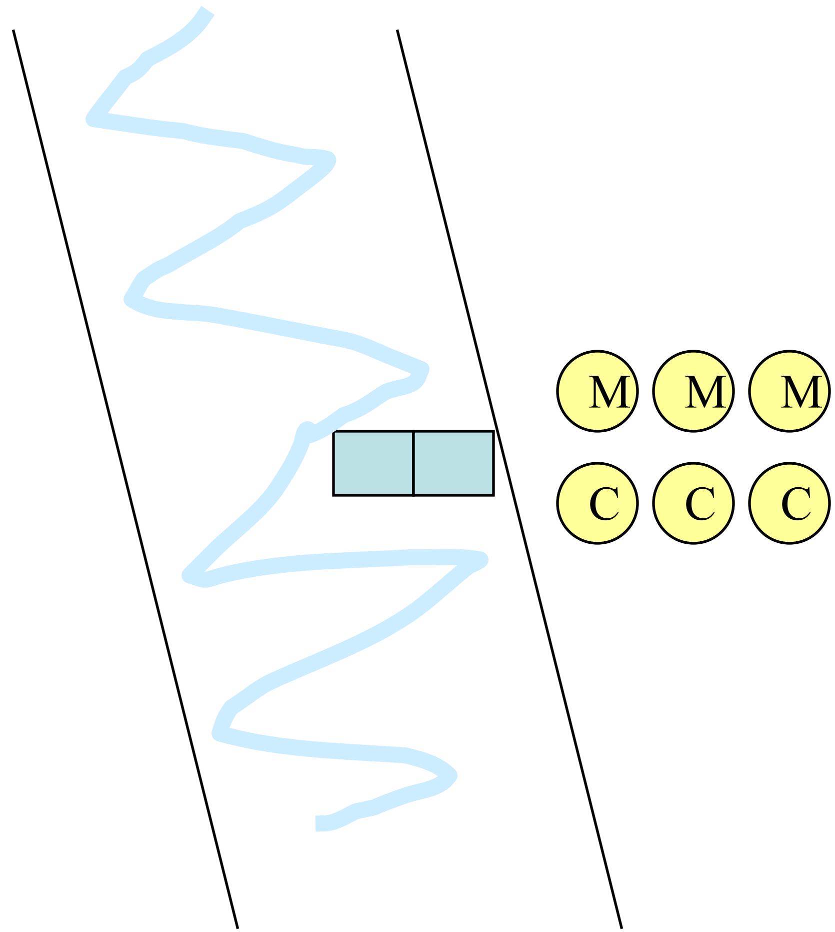


(3 3 1) $(M_L \ \ C_L \ \ B_L)$

M&C (cont.)

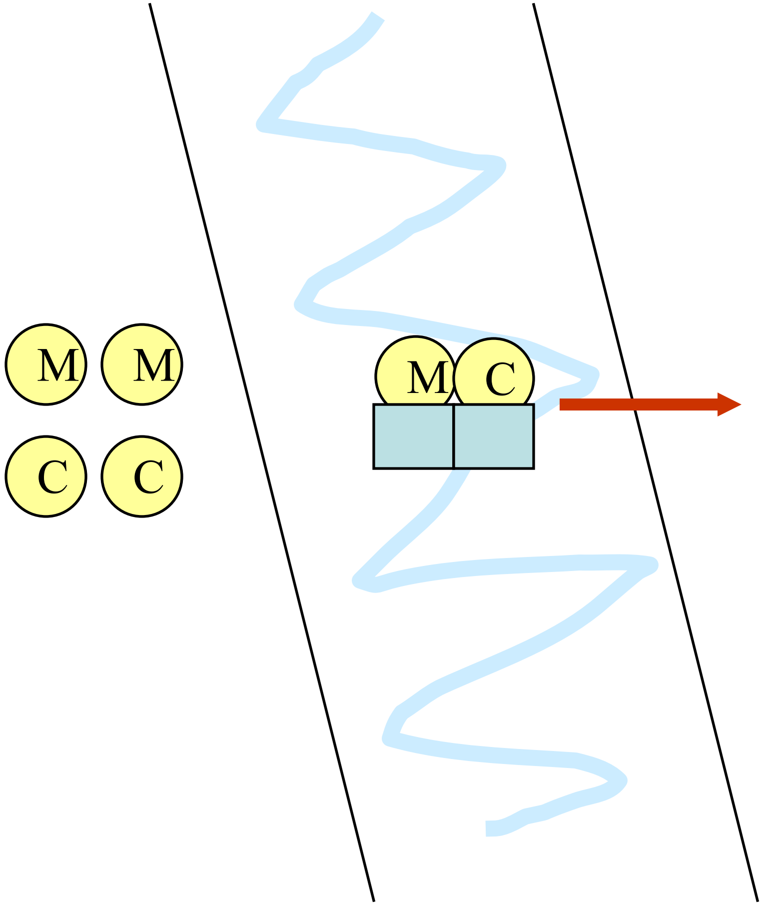
- Initial state
- Goal state



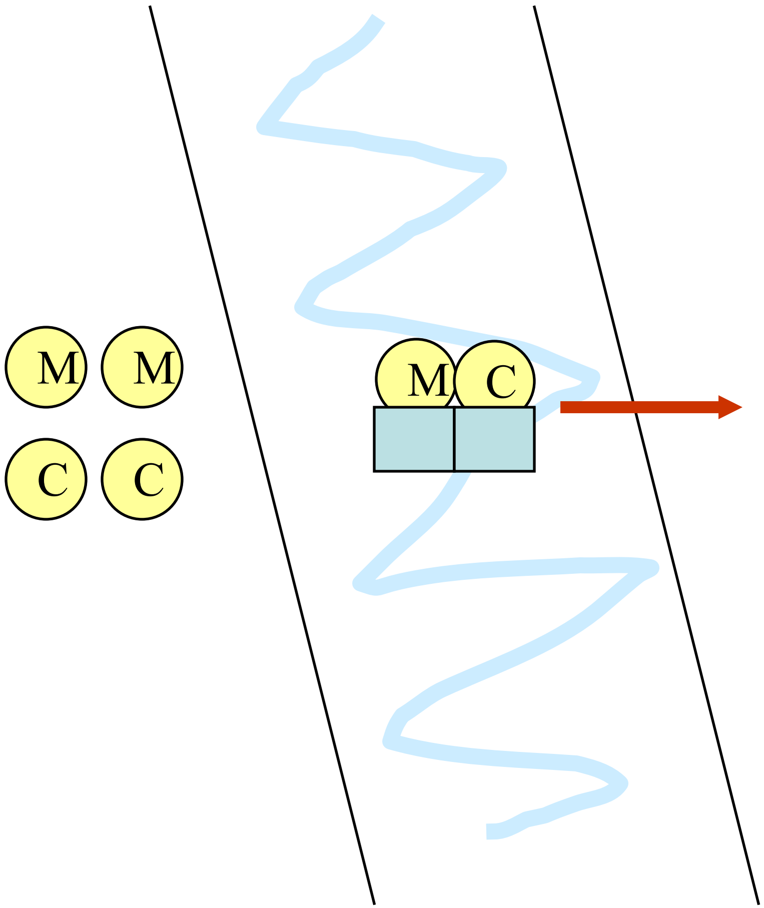


(3 3 1) $(M_L \; C_L \; B_L)$ (0 0 0)

M&C (cont.)



M&C (cont.)



(2 2 0)

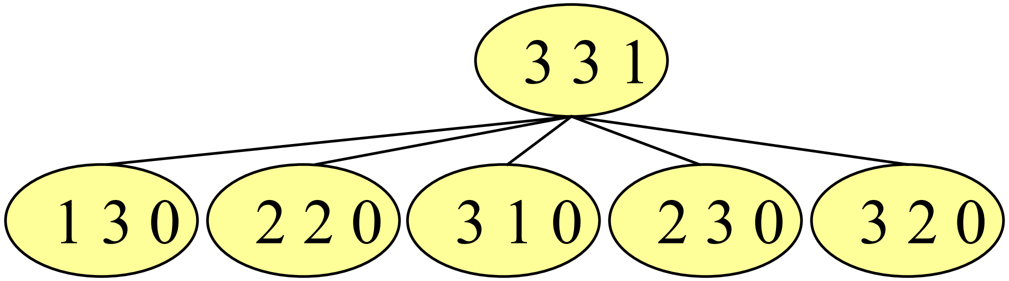
M&C (cont.)

- Problem description **<{S}, S_0, {S_{G_j}}, {O_i}, {g_i}>**
- **{S}** : { ({0,1,2,3} {0,1,2,3} {0,1}) }
- **S_0** : (3 3 1)
- **S_G** : (0 0 0)
- **g** = 1
- **{O}** : { (x y b) $\rightarrow$ (x' y' b') }
- Safe state: (x y b) is safe iff
  - x > 0 implies x $\geq$ y and x < 3 implies y $\geq$ x
  - Can be restated as (x = 1 or x = 2) implies (x = y)

Operators:

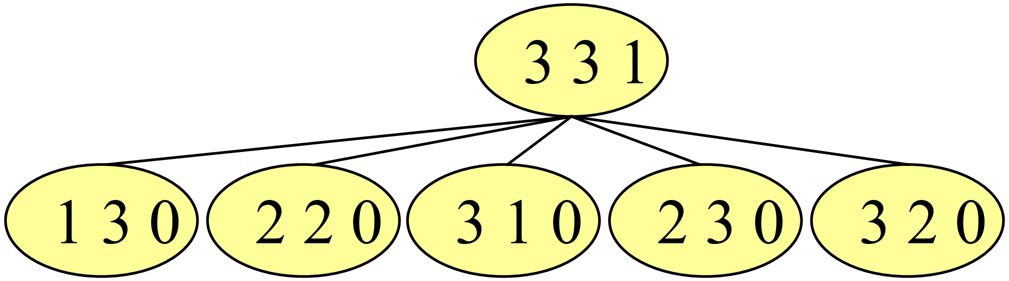
(x y 1) $\rightarrow$ (x-2 y 0)
(x y 1) $\rightarrow$ (x-1 y-1 0)
(x y 1) $\rightarrow$ (x y-2 0)
(x y 1) $\rightarrow$ (x-1 y 0)
(x y 1) $\rightarrow$ (x y-1 0)
(x y 0) $\rightarrow$ (x+2 y 1)
(x y 0) $\rightarrow$ (x+1 y+1 1)
(x y 0) $\rightarrow$ (x y+2 1)
(x y 0) $\rightarrow$ (x+1 y 1)
(x y 0) $\rightarrow$ (x y+1 1)

M&C (cont.)



- 11 steps
- 5^{11} = 48 million states to explore

M&C (cont.)



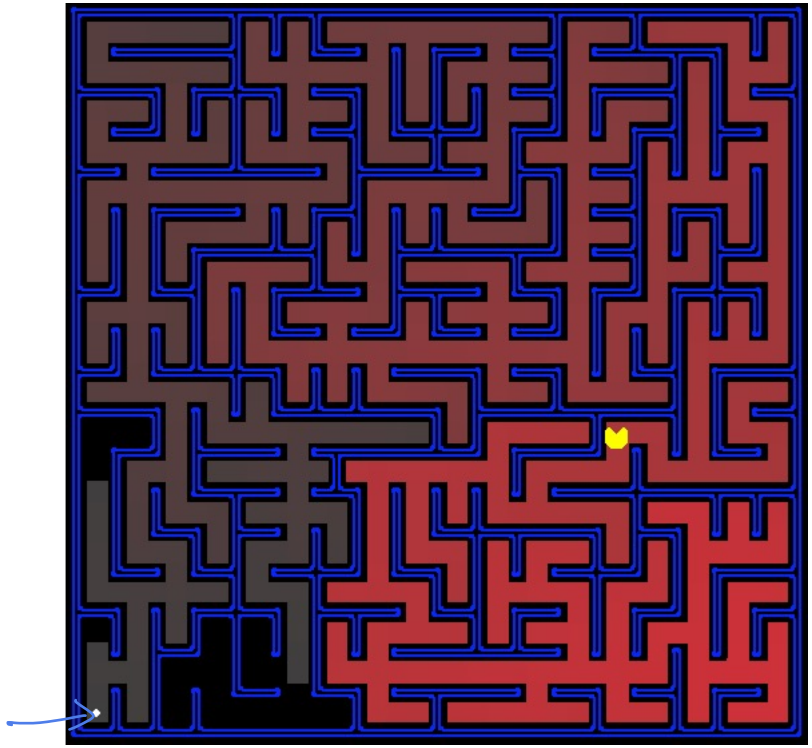
- 11 steps
- 5^{11} = 48 million states to explore

One solution path:

(3 3 1)
(2 2 0)
(3 2 1)
(3 0 0)
(3 1 1)
(1 1 0)
(2 2 1)
(0 2 0)
(0 3 1)
(0 1 0)
(0 2 1)
(0 0 0)

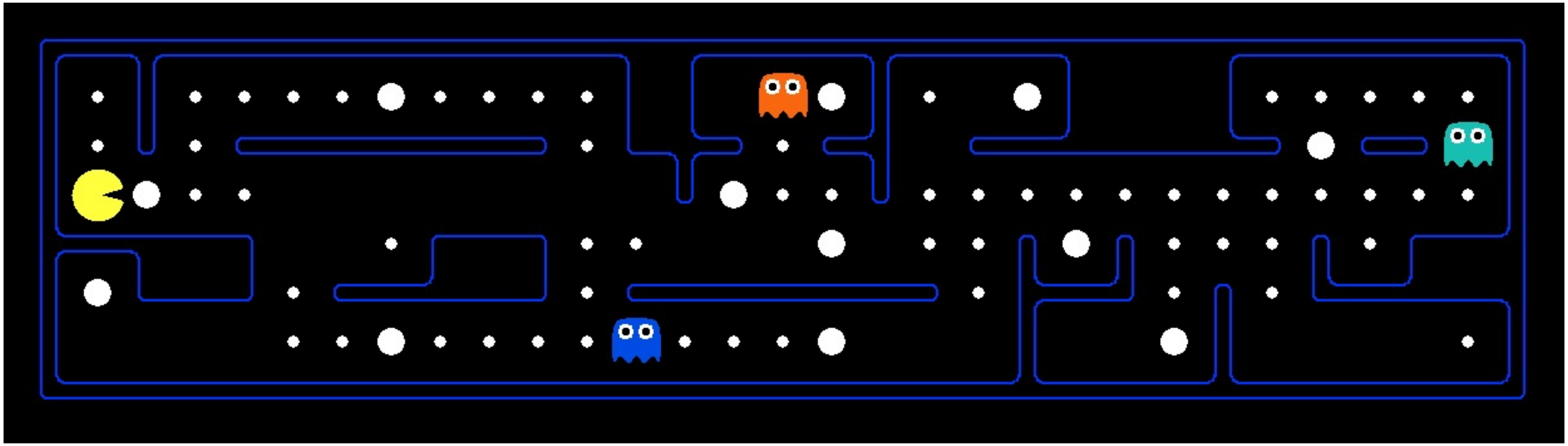
More quizzes: PACMAN

- The goal of a simplified PACMAN is to get to the pellet as quick as possible.
  - For a grid of size 30*30. Everything static.
  - What is a reasonable representation of the State, Operators, Goal test and Path cost?



More quizzes: PACMAN with static ghosts

- The goal is to eat all pellets as quickly as possible while staying alive. Eating the “Power pellet” will allow the pacman to eat the ghost.



- Think about how to formulate this problem. We will revisit it in the next lecture.

Quick summary on problem formulation

- Formulate problems as a search problem
  - Decide your level of abstraction. State, Action, Goal, Cost.
  - Represented by a state-diagram
  - Required solution: A sequence of actions
  - Optimal solution: A sequence of actions with minimum cost.

- Caveats:
  - Might not be a finite graph
  - Might not have a solution
  - Often takes exponential time to find the optimal solution

Quick summary on problem formulation

- Formulate problems as a search problem
  - Decide your level of abstraction. State, Action, Goal, Cost.
  - Represented by a state-diagram
  - Required solution: A sequence of actions
  - Optimal solution: A sequence of actions with minimum cost.

- Caveats:
  - Might not be a finite graph
  - Might not have a solution
  - Often takes exponential time to find the optimal solution

Let’s try solving it anyways!
- Do we need an exact optimal solution?
- Are problems in practice worst case?

Searching for Solutions

- Finding a solution is done by searching through the state space
  - While maintaining a set of partial solution sequences
- The *search strategy* determines which states should be expanded first

Searching for Solutions

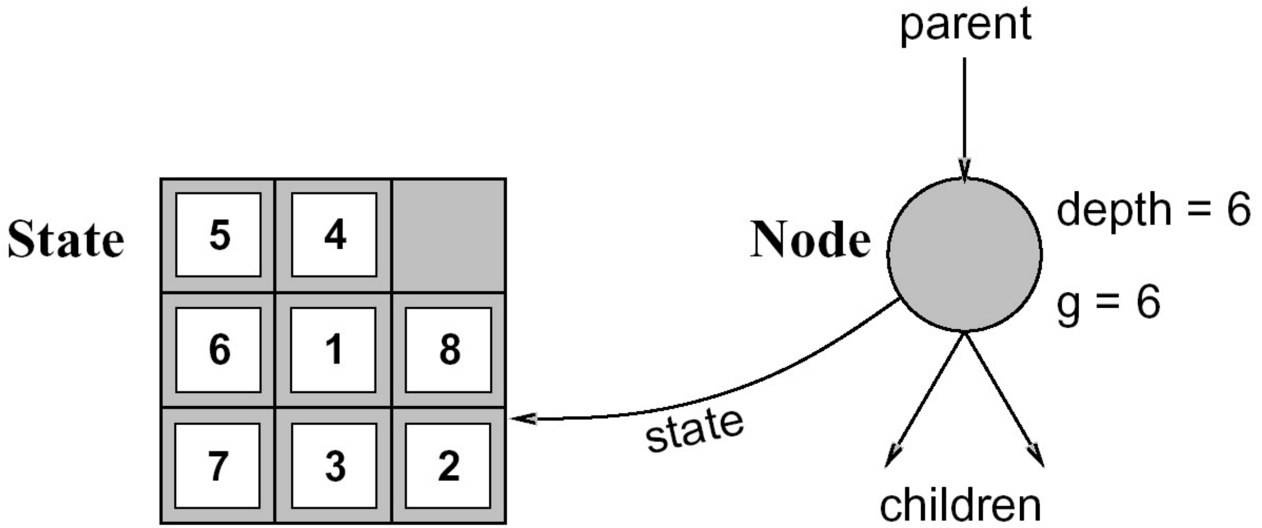
- Finding a solution is done by searching through the state space
  - While maintaining a set of partial solution sequences
- The *search strategy* determines which states should be expanded first
  - **Expand** a state = Applying the operators to the current state and thereby generating a new set of successor states

Searching for Solutions

- Finding a solution is done by searching through the state space
  - While maintaining a set of partial solution sequences
- The *search strategy* determines which states should be expanded first
  - **Expand** a state = Applying the operators to the current state and thereby generating a new set of successor states
- Conceptually, the search process builds up a *search tree* that is superimposed over the state space
  - Root node of the tree ↔ Initial state
  - Leaves of the tree ↔ States to be expanded (or expanded to null)
  - At each step, the search algorithm chooses a leaf to expand

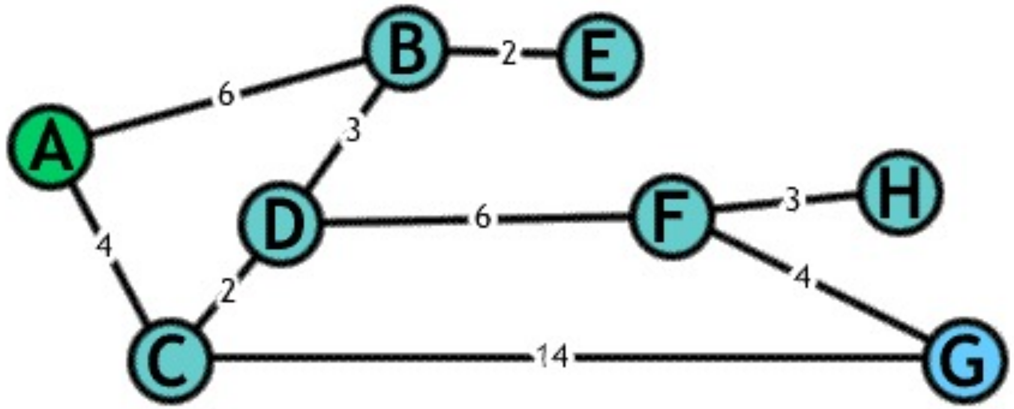
State Space vs. Search Tree

- The **state space** and the **search tree** are not the same thing!
  - A *state* represents a (possibly physical) configuration
  - A *search tree node* is a data structure which includes:
    - { parent, children, depth, path cost }
  - States do not have parents, children, depths, path costs
  - Number of states ≠ number of nodes in the search tree



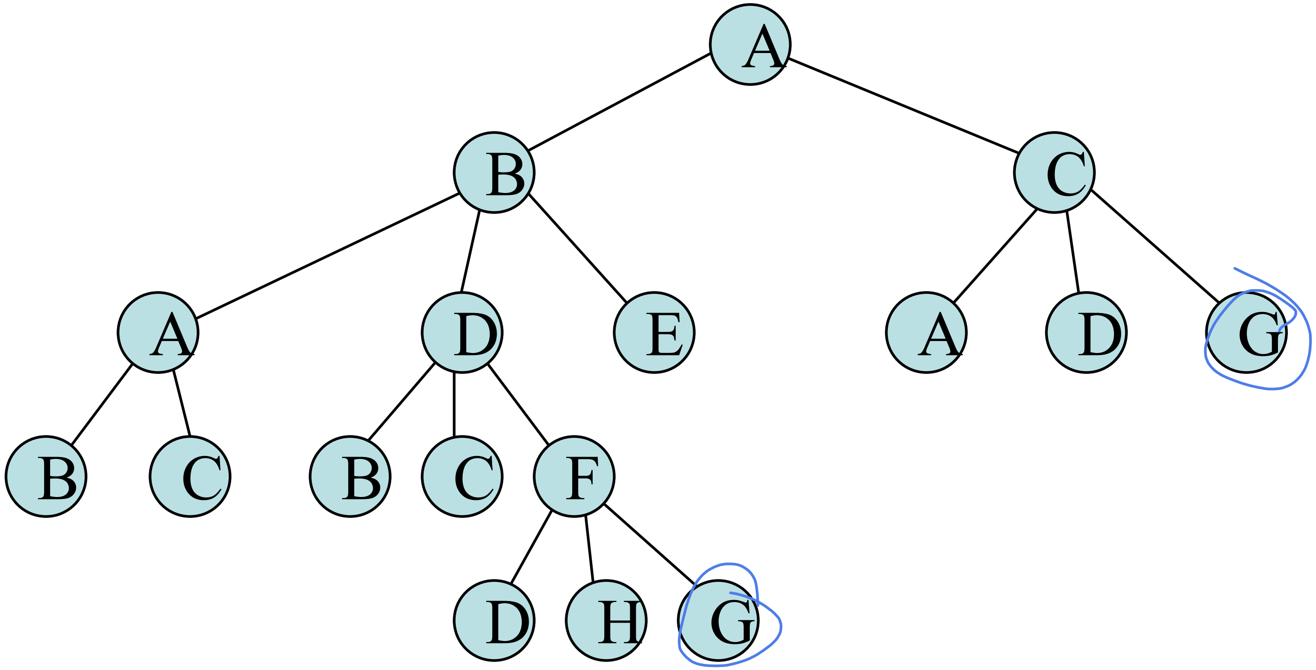
State Space vs. Search Tree (cont.)

State space: 8 states



State Space vs. Search Tree (cont.)

Search tree (partially expanded)



Search Strategies

- Uninformed (blind) search
  - Can only distinguish goal state from non-goal state

- Informed (heuristic) search
  - Can evaluate states

Uninformed ("Blind") Search Strategies

- No information is available other than
  - The current state
    - Its parent (perhaps complete path from initial state)
    - Its operators (to produce successors)
  - The goal test
  - The current path cost (cost from start state to current state)

Uninformed (“Blind”) Search Strategies

- No information is available other than
  - The current state
    - Its parent (perhaps complete path from initial state)
    - Its operators (to produce successors)
  - The goal test
  - The current path cost (cost from start state to current state)

- Blind search strategies
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

General Search Algorithm (Version 1)

- Various strategies are merely variations of the following function:

General Search Algorithm (Version 1)

- Various strategies are merely variations of the following function:

```
function GENERAL-SEARCH(problem, strategy) returns a solution or failure

 initialize the search tree using the initial state of problem
 loop do
   if there are no candidates for expansion then return failure
   choose a leaf node for expansion according to strategy
   if the node contains a goal state then return the corresponding solution
   else expand the node and add the resulting nodes to the search tree
 end
```

(Called "Tree-Search" in the textbook)

General Search Algorithm (Version 2)

- Uses a queue (a list) and a **queuing function** to implement a *search strategy*
  - **Queuing-Fn(*queue*, *elements*)** inserts a set of elements into the queue and <u>determines the order of node expansion</u>

General Search Algorithm (Version 2)

- Uses a queue (a list) and a **queuing function** to implement a *search strategy*
  - **Queuing-Fn(*queue*, *elements*)** inserts a set of elements into the queue and determines the order of node expansion

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution or failure

 nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
 loop do
   if nodes is empty then return failure
   node ← REMOVE-FRONT(nodes)
   if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
   nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
 end
```

General Search Algorithm (Version 2)

- Uses a queue (a list) and a **queuing function** to implement a *search strategy*
  - **Queuing-Fn(*queue*, *elements*)** inserts a set of elements into the queue and determines the order of node expansion

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution or failure

 nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
 loop do
   if nodes is empty then return failure
   node ← REMOVE-FRONT(nodes)
   if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
   nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
 end
```

"Nodes" is also known as a **"frontier"** --- the set of states we haven't yet explored/expanded.
"EXPAND" is known as the "**successor function**" --- the set of all states that you could expand on.

How do we evaluate a search algorithm?

How do we evaluate a search algorithm?

- Primary criteria to evaluate search strategies
  - **Completeness**
    - Is it guaranteed to find a solution (if one exists)?
  - **Optimality**
    - Does it find the "best" solution (if there are more than one)?
  - **Time complexity**
    - Number of nodes generated/expanded
    - (How long does it take to find a solution?)
  - **Space complexity**
    - How much memory does it require?

How do we evaluate a search algorithm?

- Primary criteria to evaluate search strategies
  - **Completeness**
    - Is it guaranteed to find a solution (if one exists)?
  - **Optimality**
    - Does it find the “best” solution (if there are more than one)?
  - **Time complexity**
    - Number of nodes generated/expanded
    - (How long does it take to find a solution?)
  - **Space complexity**
    - How much memory does it require?

- Some performance measures
  - Best case
  - Worst case
  - Average case
  - Real-world case

How do we evaluate a search algorithm?

- Primary criteria to evaluate search strategies
  - **Completeness**
    - Is it guaranteed to find a solution (if one exists)?
  - **Optimality**
    - Does it find the "best" solution (if there are more than one)?
  - **Time complexity**
    - Number of nodes generated/expanded
    - (How long does it take to find a solution?)
  - **Space complexity**
    - How much memory does it require?

- Some performance measures
  - Best case
  - Worst case
  - Average case
  - Real-world case

How do we evaluate a search algorithm?

- Primary criteria to evaluate search strategies
  - **Completeness**
    - Is it guaranteed to find a solution (if one exists)?
  - **Optimality** *Note that this is not saying it's space/time complexity is optimal.
    - Does it find the "best" solution (if there are more than one)?
  - **Time complexity**
    - Number of nodes generated/expanded
    - (How long does it take to find a solution?)
  - **Space complexity**
    - How much memory does it require?

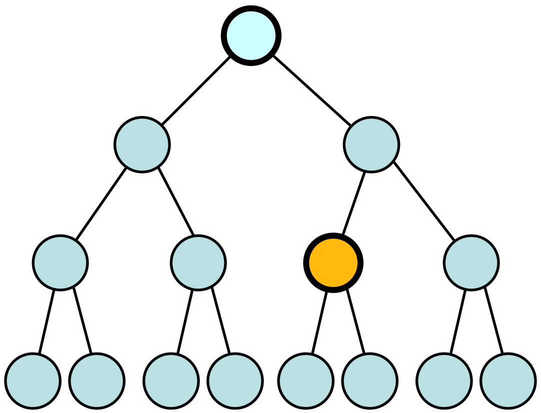
- Some performance measures
  - Best case
  - Worst case
  - Average case
  - Real-world case

How do we evaluate a search algorithm?

- Complexity analysis and O( ) notation (see Appendix A)
  - b = Maximum branching factor of the search tree
  - d = Depth of an optimal solution (may be more than one)
  - m = maximum depth of the search tree (may be infinite)

- Examples
  - O($b^3 d^2$) – polynomial time
  - O(b^d) – exponential time

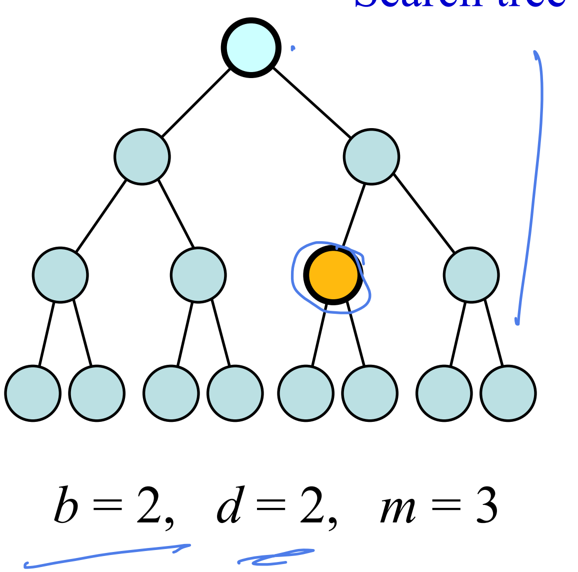
Search tree



How do we evaluate a search algorithm?

- Complexity analysis and O( ) notation (see Appendix A)
  - b = Maximum branching factor of the search tree
  - d = Depth of an optimal solution (may be more than one)
  - m = maximum depth of the search tree (may be infinite)

- Examples
  - $O(b^3 d^2)$ – polynomial time
  - $O(b^d)$ – exponential time

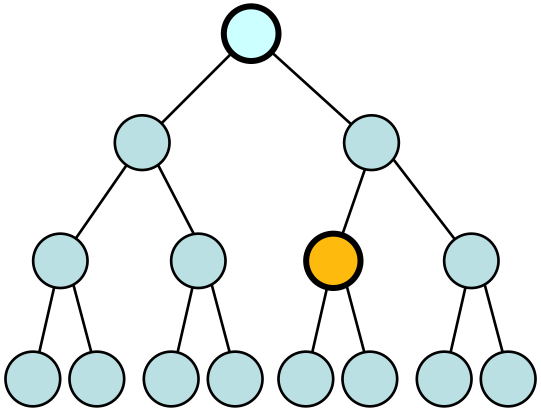


How do we evaluate a search algorithm?

- Complexity analysis and O( ) notation (see Appendix A)
  - b = Maximum branching factor of the search tree
  - d = Depth of an optimal solution (may be more than one)
  - m = maximum depth of the search tree (may be infinite)

- Examples
  - O($b^3 d^2$) – polynomial time
  - O(b^d) – exponential time

Search tree



For chess, b_{ave}= 35

$b = 2, \quad d = 2, \quad m = 3$

Breadth-First Search

Breadth-First Search

- All nodes at depth *d* in the search tree are expanded before any nodes at depth *d+1*
  - First consider all paths of length N, then all paths of length N+1, etc.
- Doesn't consider path cost – finds the solution with the shortest path
- Uses FIFO queue

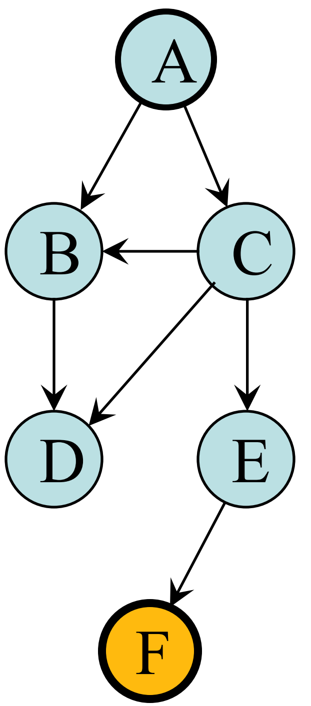
Breadth-First Search

- All nodes at depth *d* in the search tree are expanded before any nodes at depth *d+1*
  - First consider all paths of length N, then all paths of length N+1, etc.
- Doesn't consider path cost – finds the solution with the shortest path
- Uses FIFO queue

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure
 return GENERAL-SEARCH(problem, ENQUEUE-AT-END)
```

Example

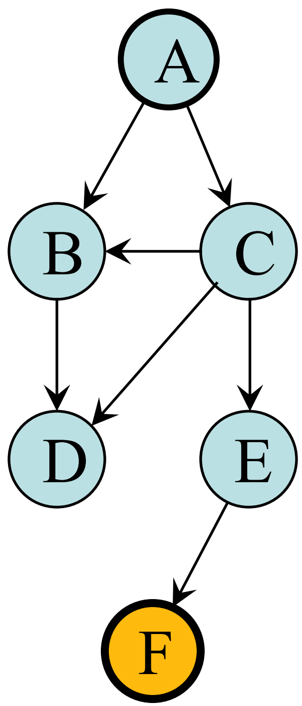
State space graph



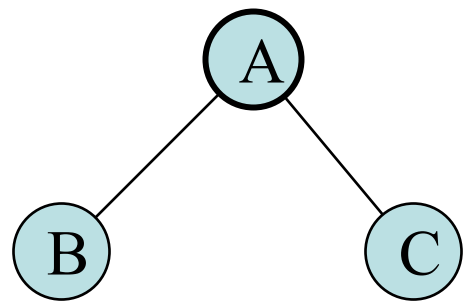
Search tree

Example

State space graph

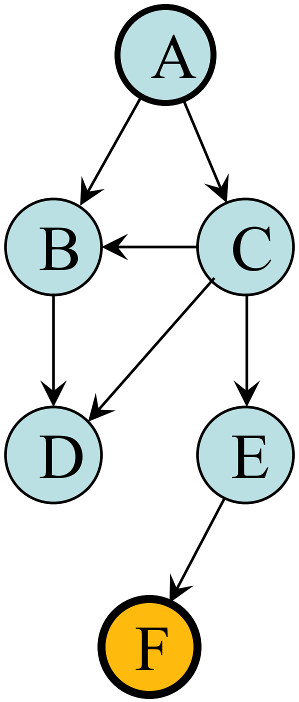


Search tree

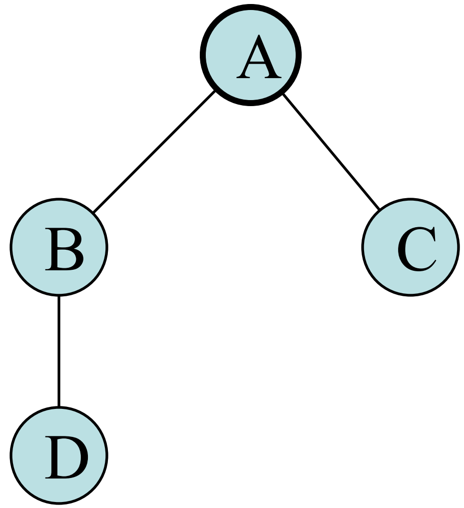


Example

State space graph

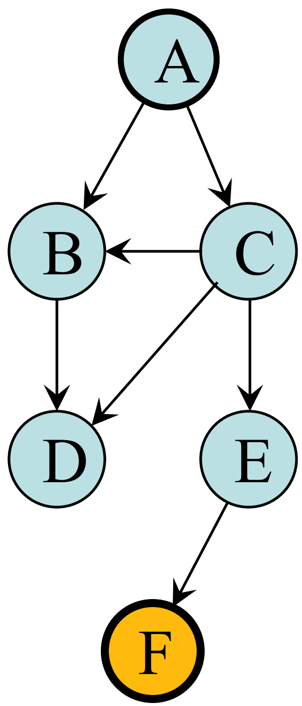


Search tree

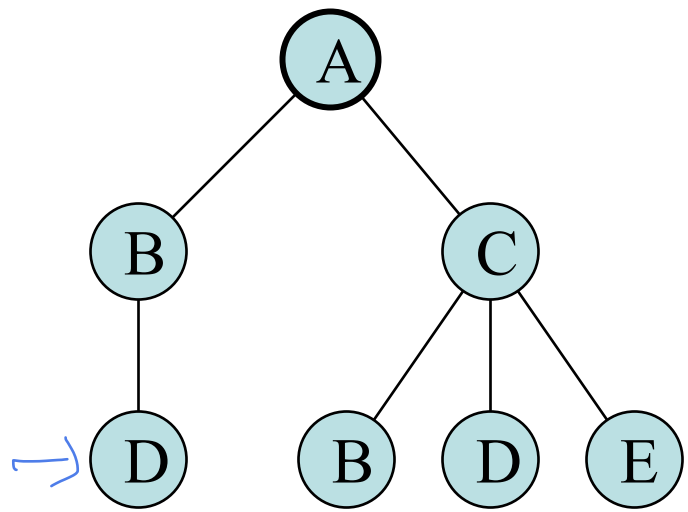


Example

State space graph

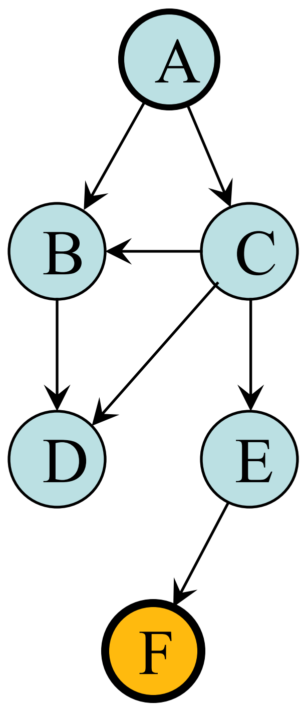


Search tree

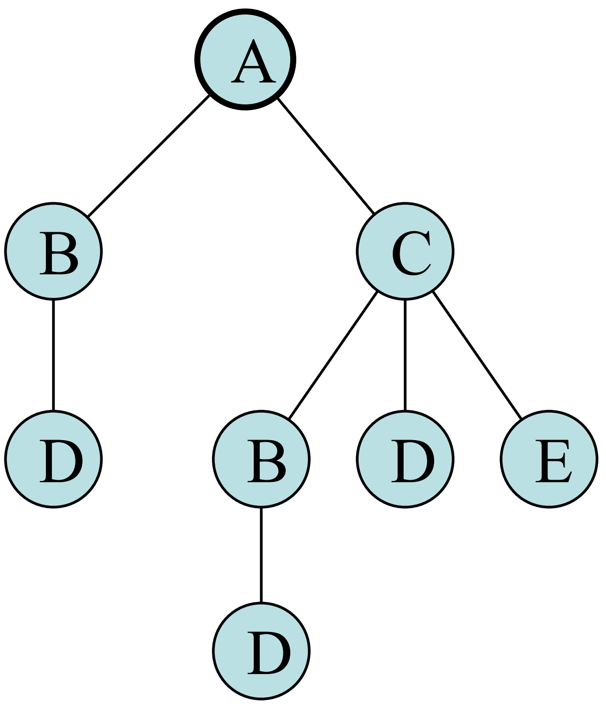


Example

State space graph

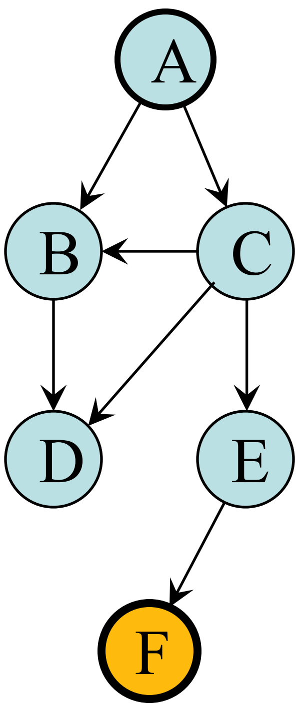


Search tree

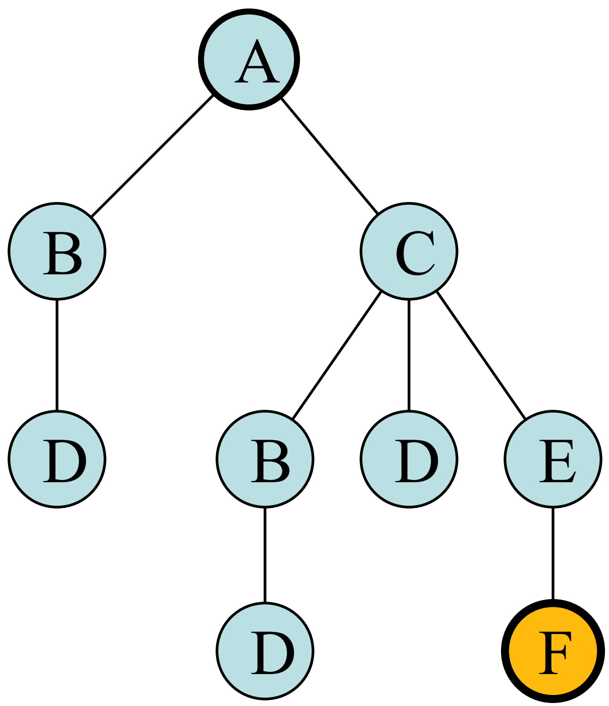


Example

State space graph

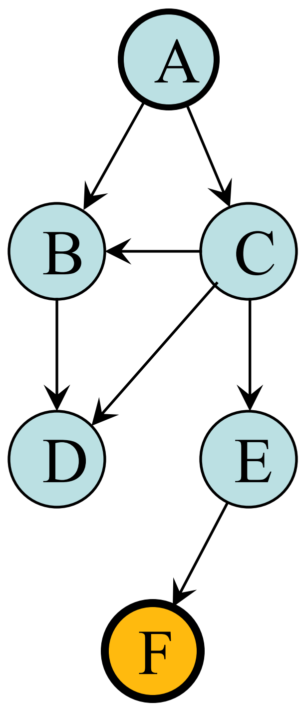


Search tree

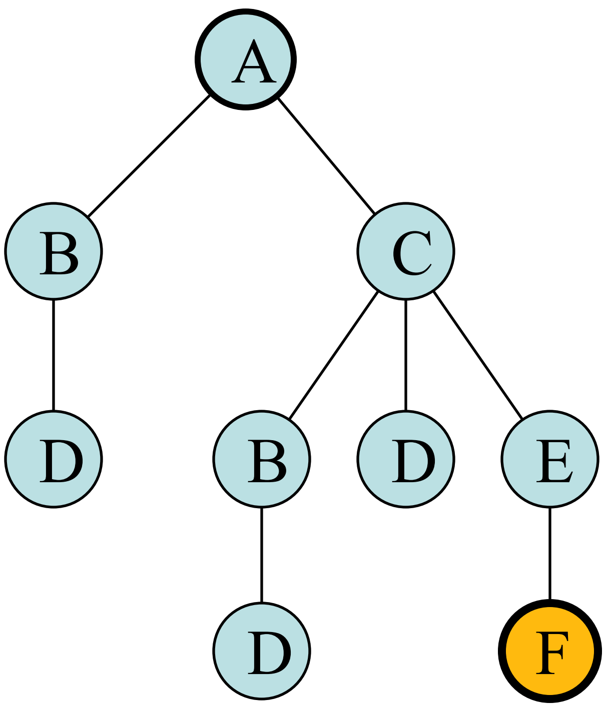


Example

State space graph



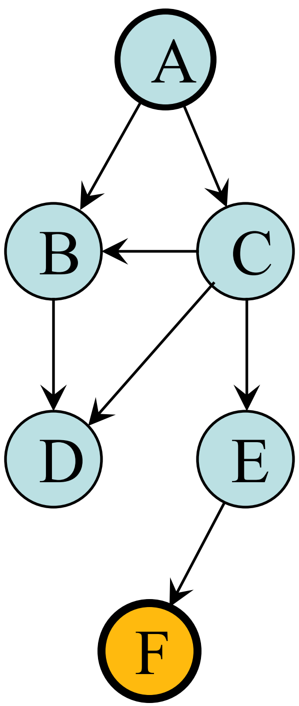
Search tree



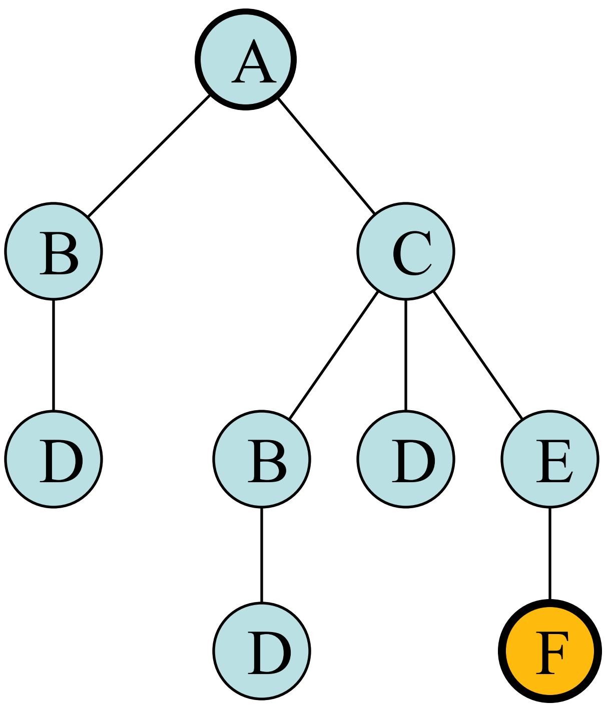
Queue

Example

State space graph



Search tree

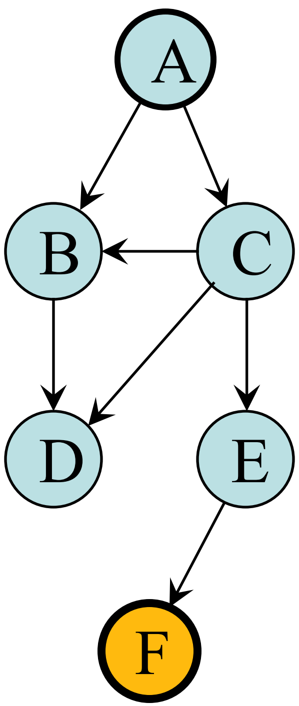


Queue

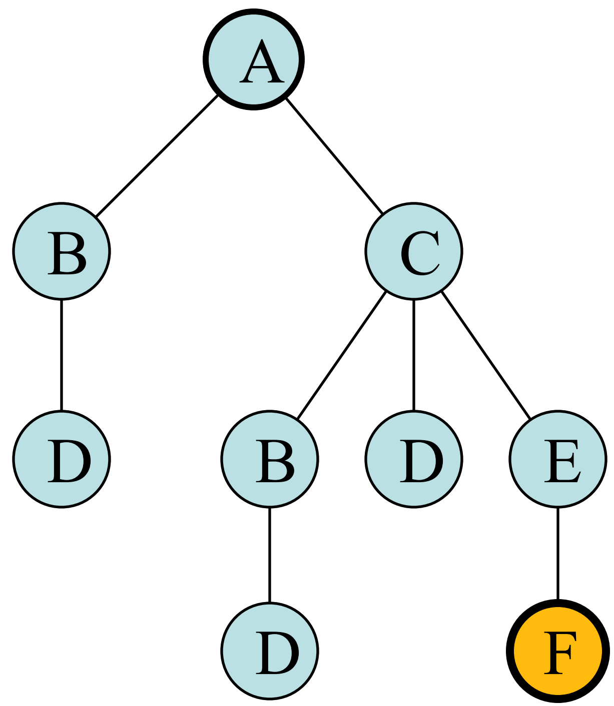
(A)

Example

State space graph



Search tree



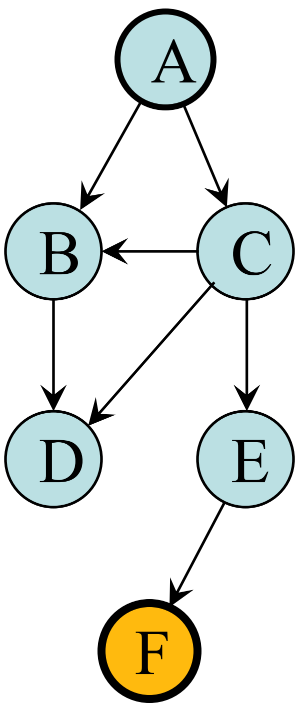
Queue

(A)

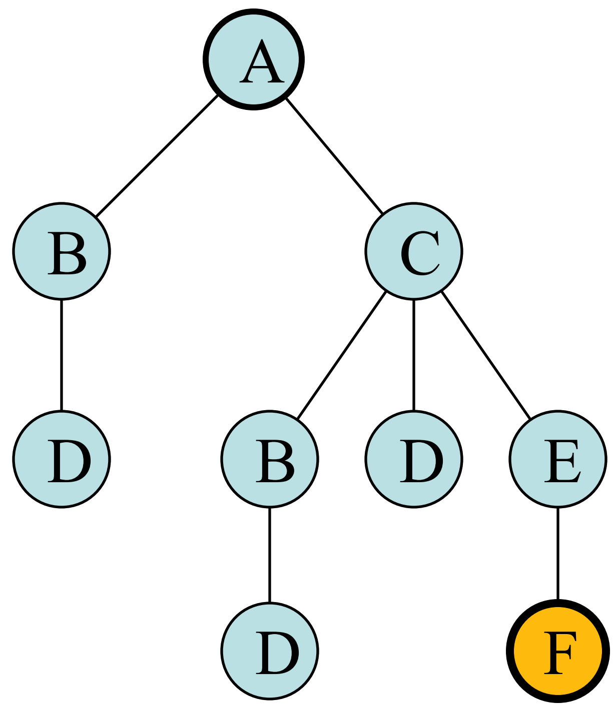
(B C)

Example

State space graph



Search tree



Queue

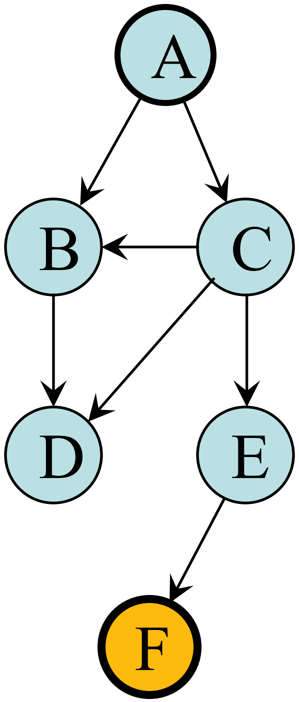
(A)

(B C)

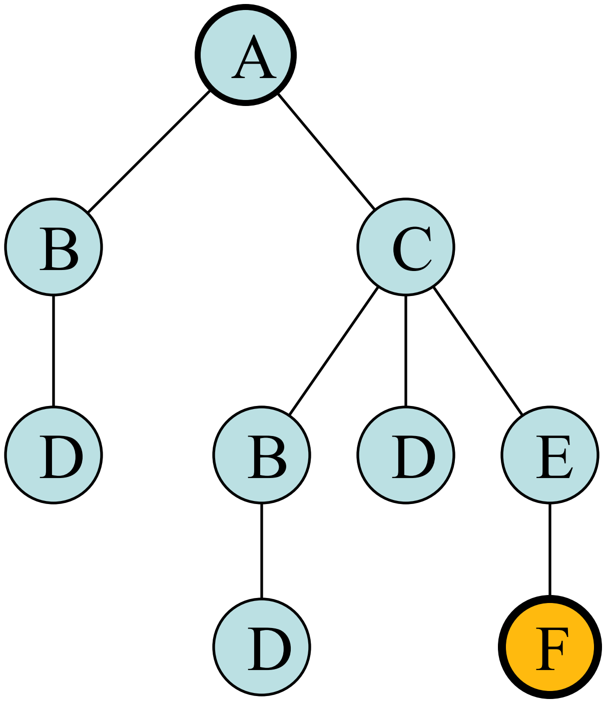
(C D)

Example

State space graph



Search tree



Queue

(A)

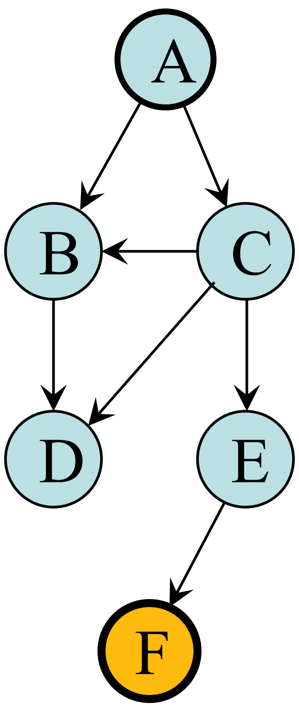
(B C)

(C D)

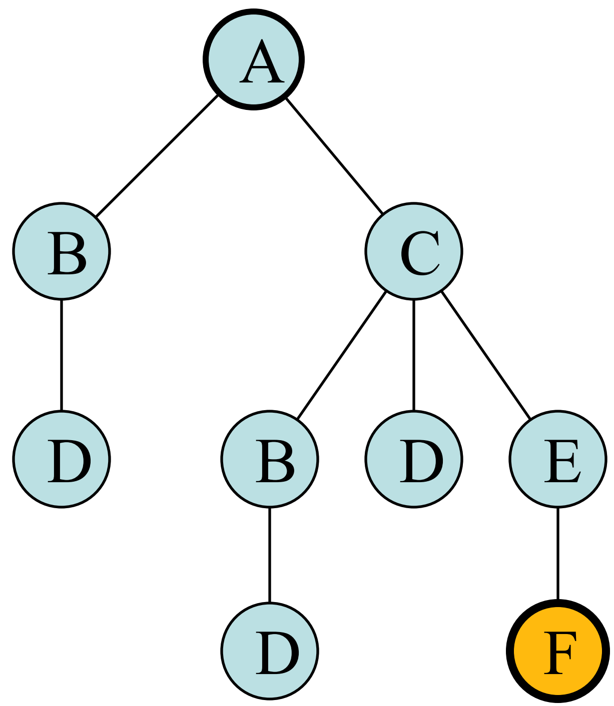
(D B D E)

Example

State space graph



Search tree



Queue

(A)

(B C)

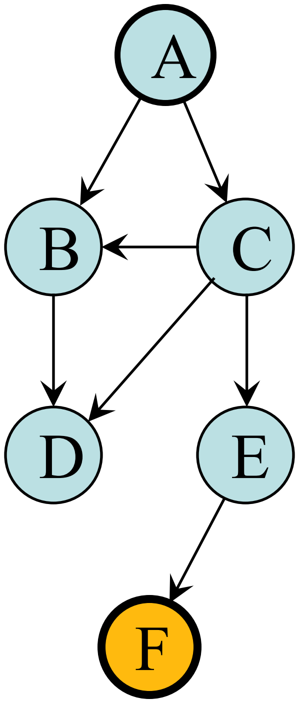
(C D)

(D B D E)

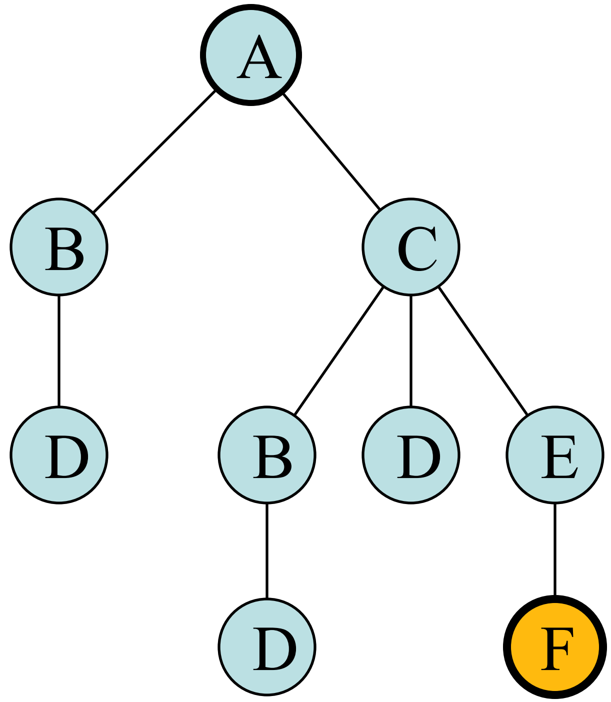
(B D E)

Example

State space graph



Search tree



Queue

(A)

(B C)

(C D)

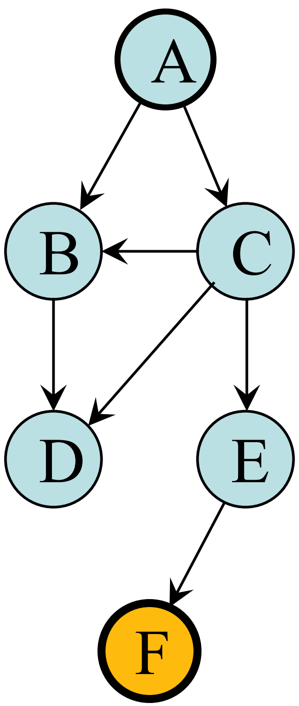
(D B D E)

(B D E)

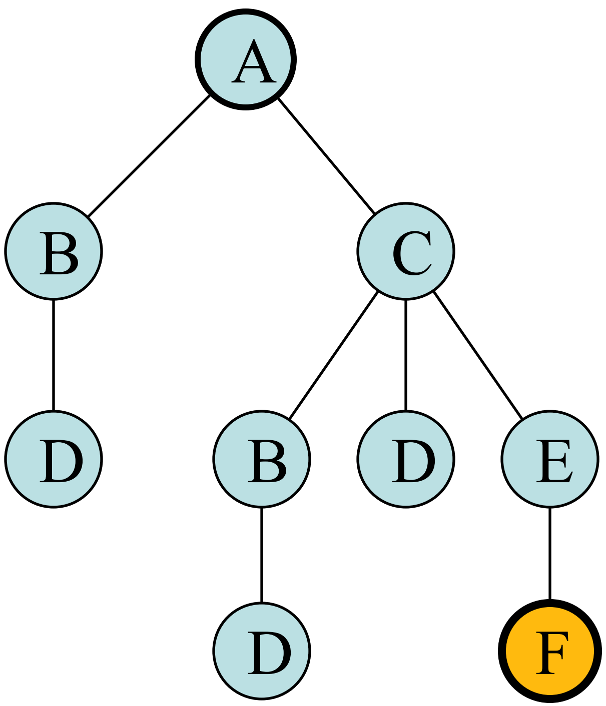
(D E D)

Example

State space graph



Search tree



Queue

(A)

(B C)

(C D)

(D B D E)

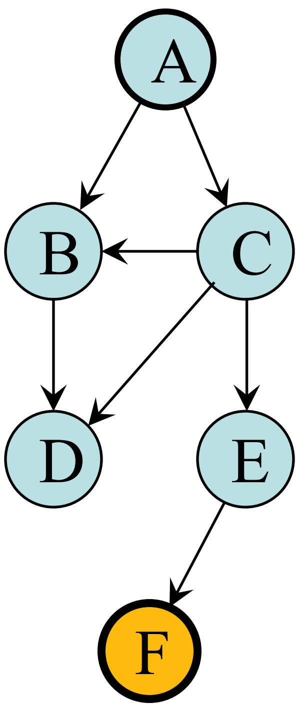
(B D E)

(D E D)

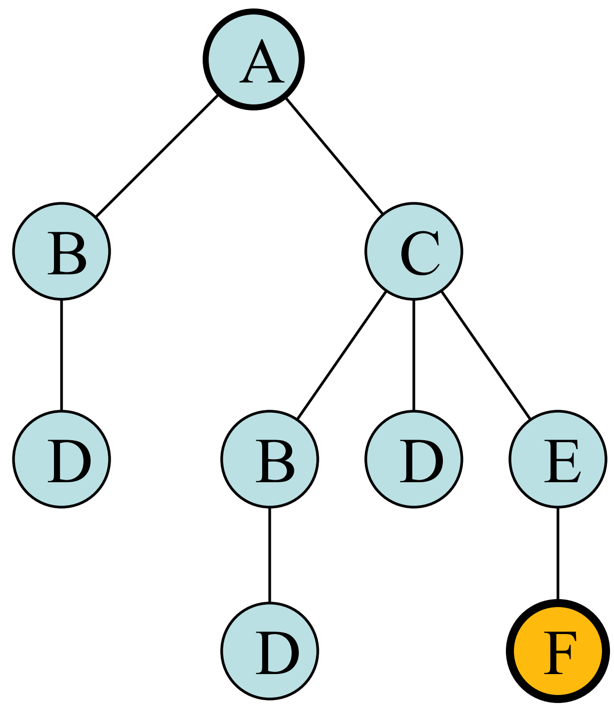
(E D)

Example

State space graph



Search tree



Queue

(A)

(B C)

(C D)

(D B D E)

(B D E)

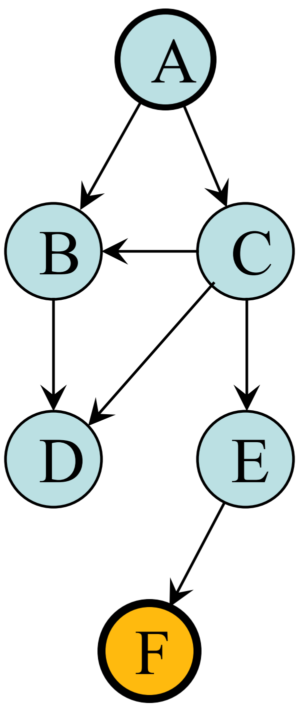
(D E D)

(E D)

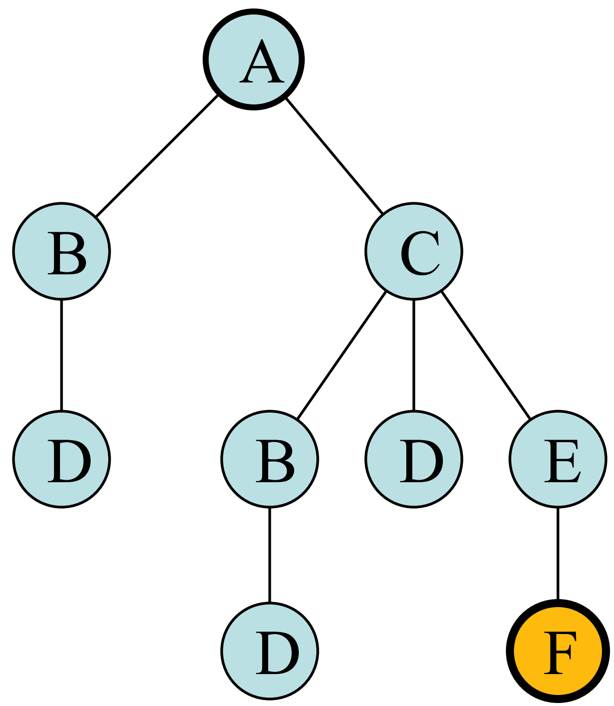
(D F)

Example

State space graph



Search tree



Queue

(A)

(B C)

(C D)

(D B D E)

(B D E)

(D E D)

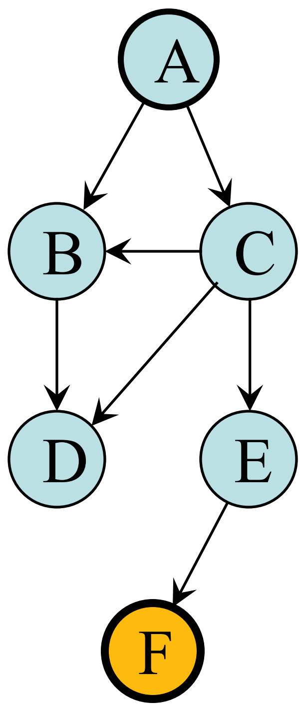
(E D)

(D F)

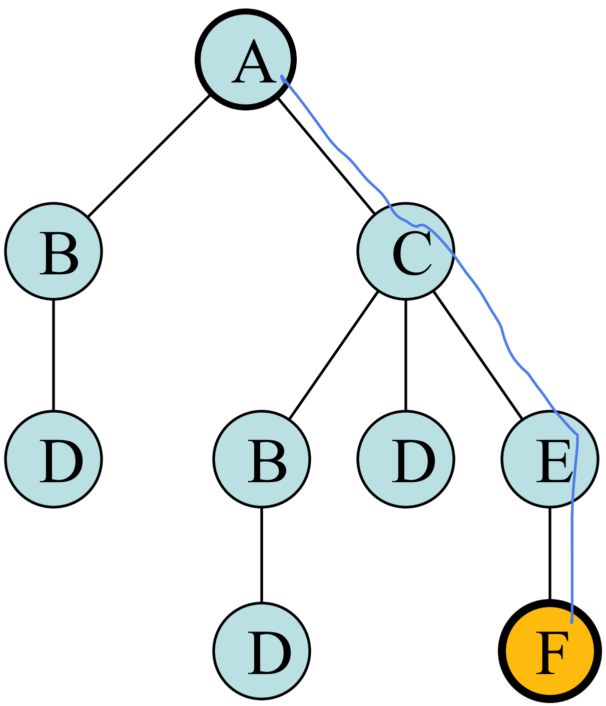
(F)

Example

State space graph



Search tree



Queue

(A)

(B C)

(C D)

(D B D E)

(B D E)

(D E D)

(E D)

(D F)

(F)

( )

Breadth-First Search

- Complete?

- Optimal?

- Time complexity?

- Space complexity?

b = branching factor (require finite b)
d = depth of shallowest solution

Breadth-First Search

- Complete? Yes
- Optimal?
- Time complexity?
- Space complexity?

b = branching factor (require finite b)
d = depth of shallowest solution

Breadth-First Search

- Complete? Yes
- Optimal? If shallowest goal is optimal
- Time complexity?
- Space complexity?

b = branching factor (require finite b)
d = depth of shallowest solution

Breadth-First Search

- Complete? Yes
- Optimal? If shallowest goal is optimal
- Time complexity? Exponential: **O(** b^{d+1} **)**
- Space complexity?

b = branching factor (require finite b)
d = depth of shallowest solution

Breadth-First Search

- Complete? Yes
- Optimal? If shallowest goal is optimal
- Time complexity? Exponential: **O(** b^{d+1} **)**
- Space complexity? Exponential: **O(** b^{d+1} **)**

b = branching factor (require finite b)
d = depth of shallowest solution

Breadth-First Search

- Complete? Yes
- Optimal? If shallowest goal is optimal
- Time complexity? Exponential: **O(** b^{d+1} **)**
- Space complexity? Exponential: **O(** b^{d+1} **)**

In practice, the memory requirements are typically worse than the time requirements

b = branching factor (require finite b)
d = depth of shallowest solution

Depth-First Search

Depth-First Search

- Always expands one of the nodes at the deepest level of the tree
  - Low memory requirements
  - Problem: depth could be infinite
- Uses a stack (LIFO)

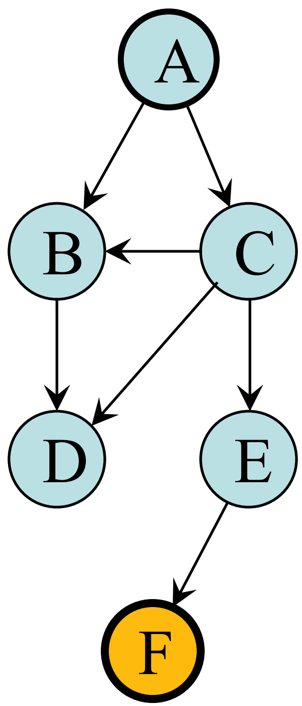
Depth-First Search

- Always expands one of the nodes at the deepest level of the tree
  - Low memory requirements
  - Problem: depth could be infinite
- Uses a stack (LIFO)

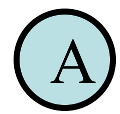
function DEPTH-FIRST-SEARCH(*problem*) **returns** a solution or failure
return **GENERAL-SEARCH**(*problem,* ENQUEUE-AT-FRONT)

Example

State space graph

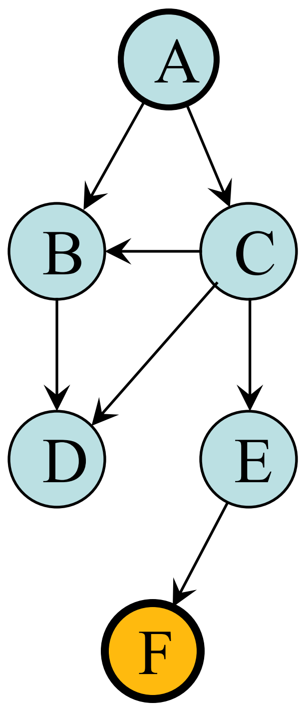


Search tree

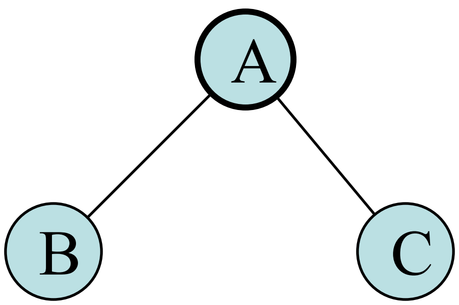


Example

State space graph

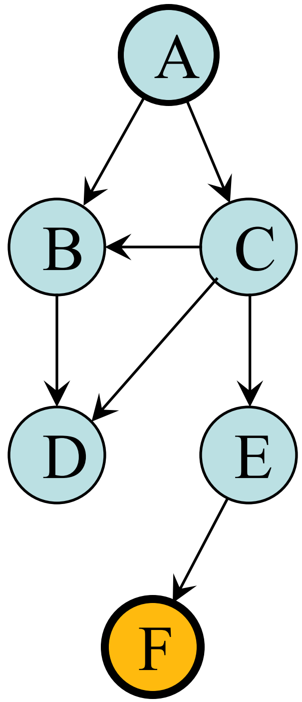


Search tree

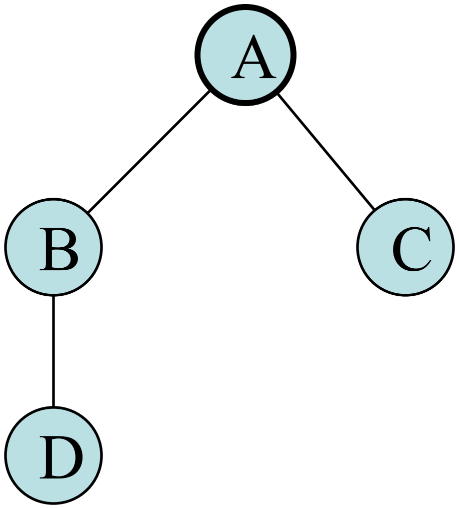


Example

State space graph

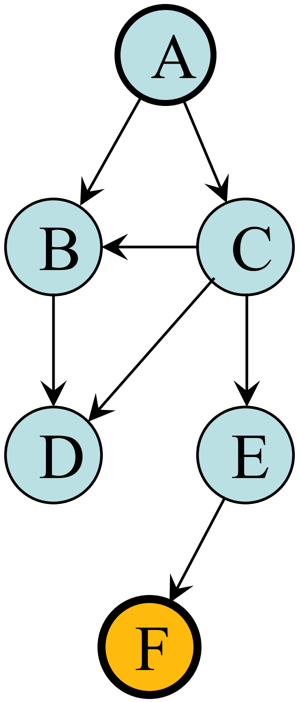


Search tree

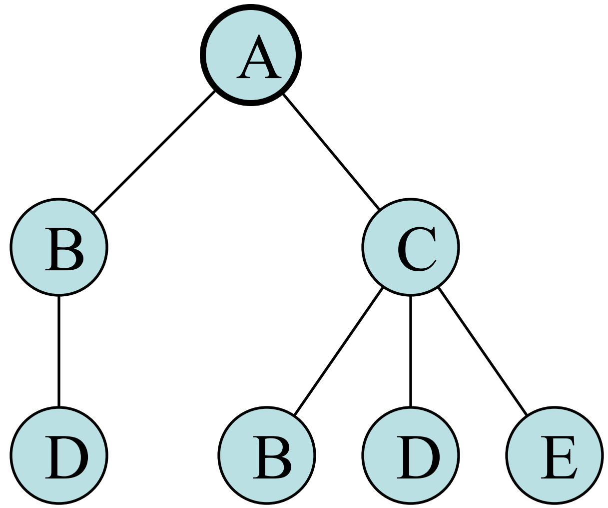


Example

State space graph

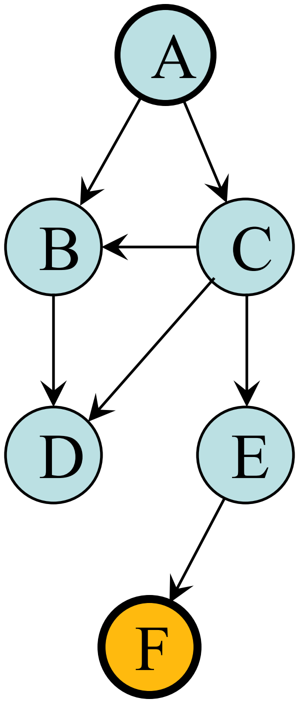


Search tree

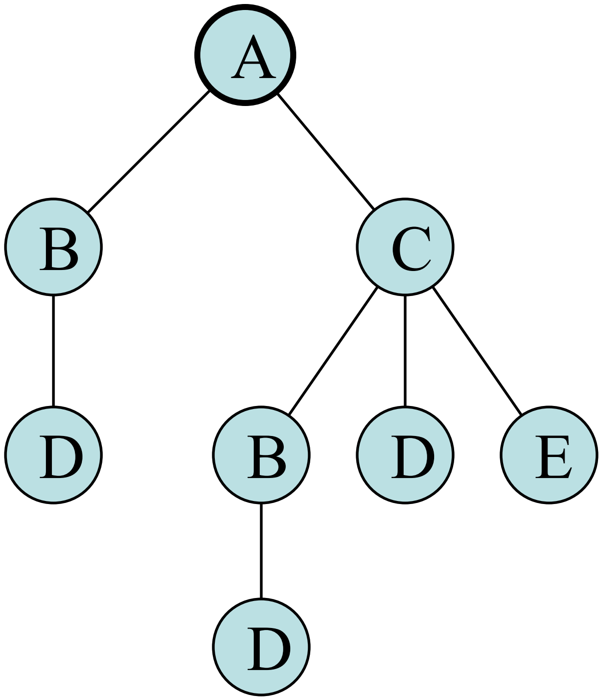


Example

State space graph

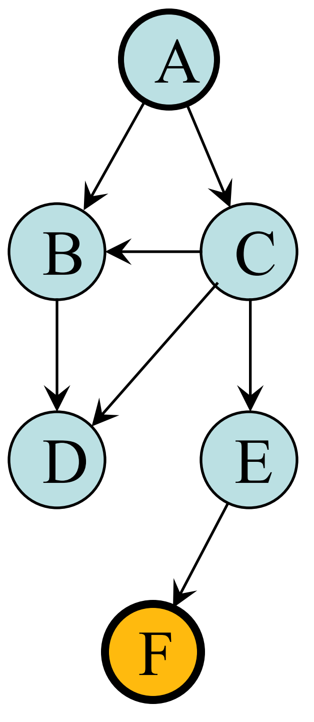


Search tree

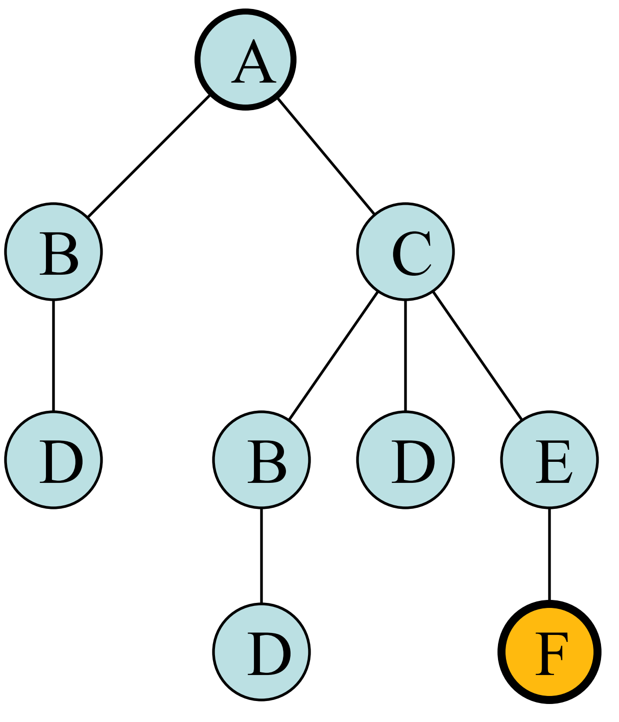


Example

State space graph

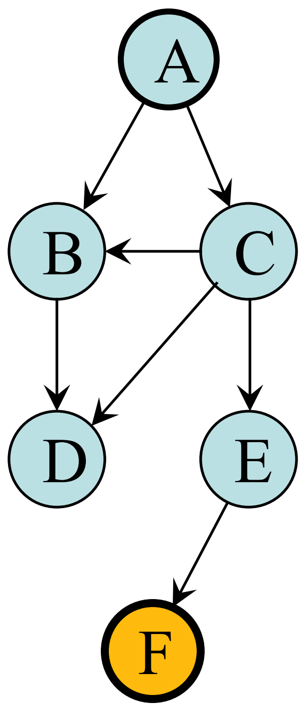


Search tree

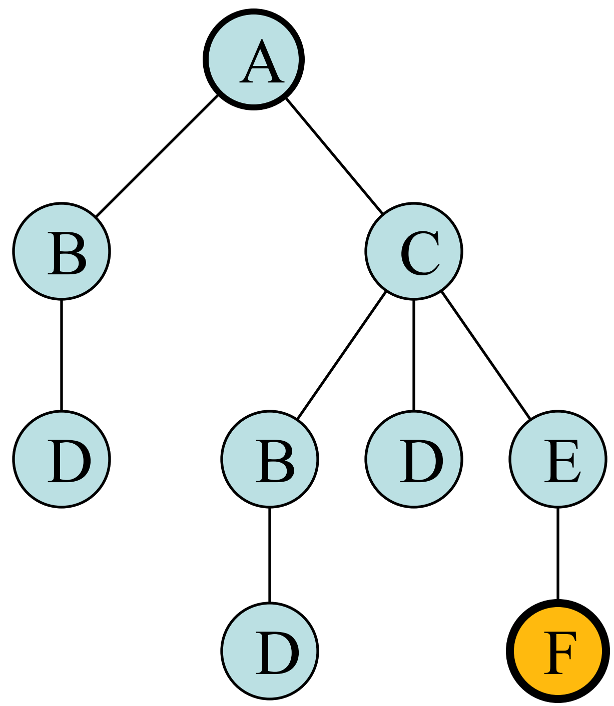


Example

State space graph



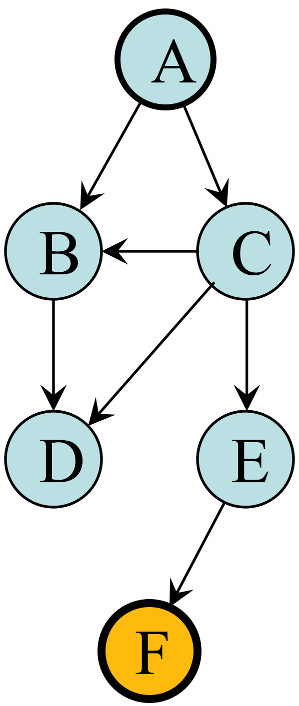
Search tree



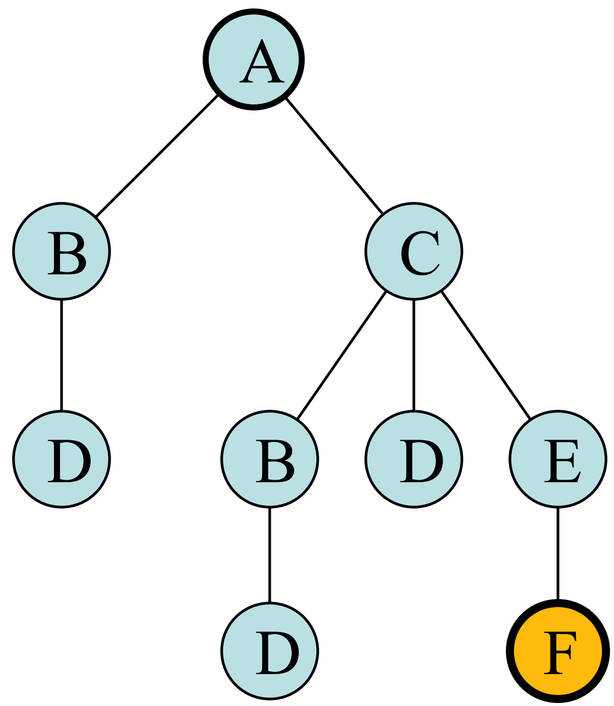
Queue

Example

State space graph



Search tree

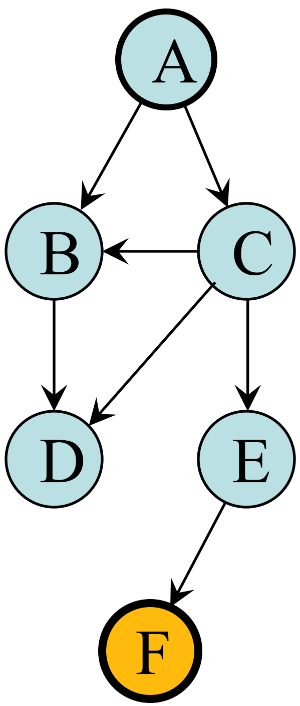


Queue

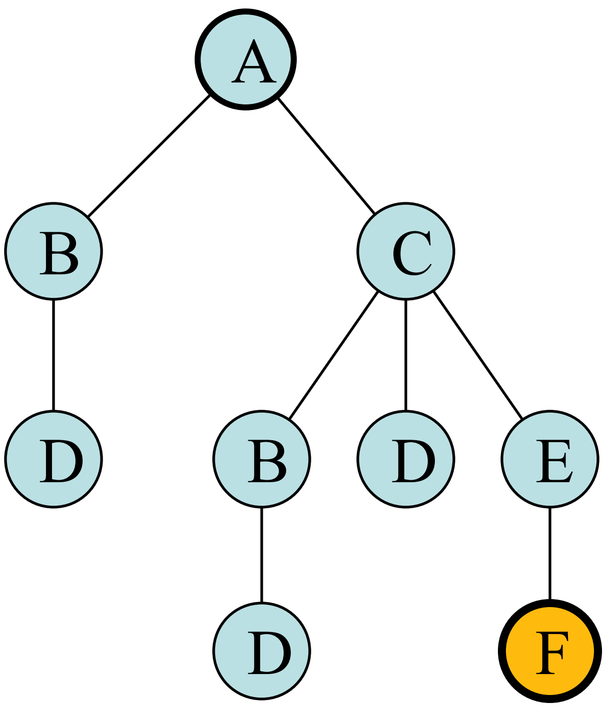
(A)

Example

State space graph



Search tree



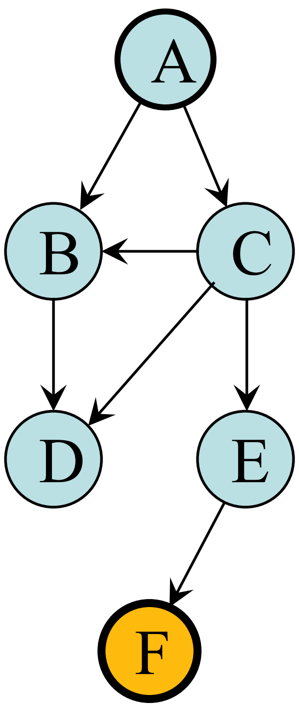
Queue

(A)

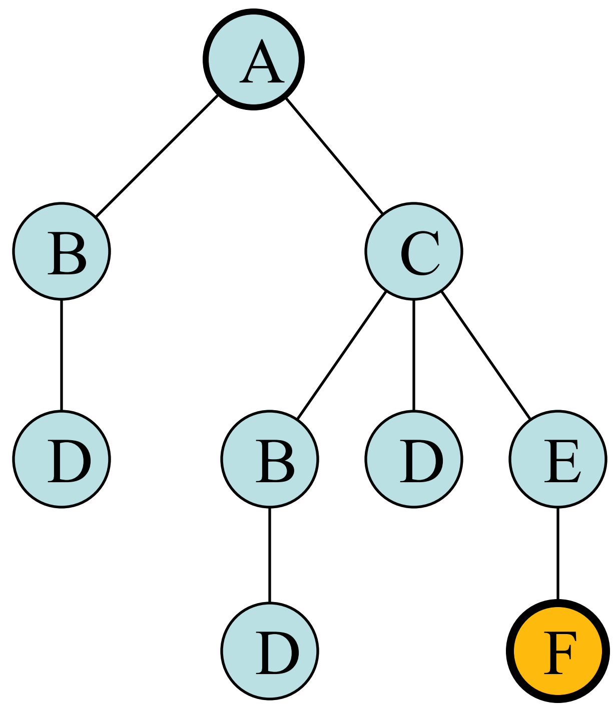
(B C)

Example

State space graph



Search tree



Queue

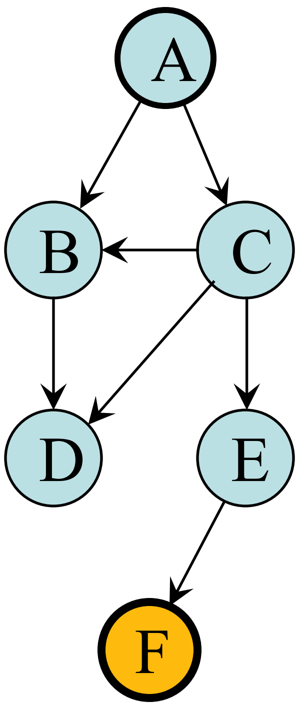
(A)

(B C)

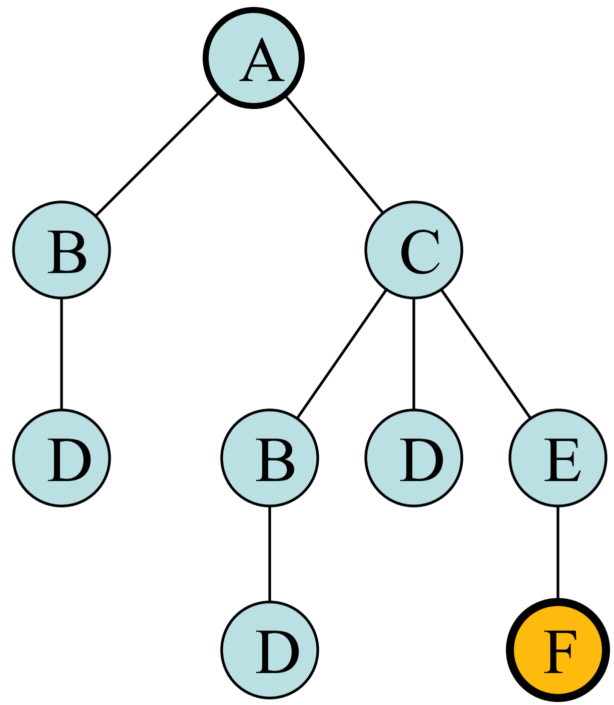
(D C)

Example

State space graph



Search tree



Queue

(A)

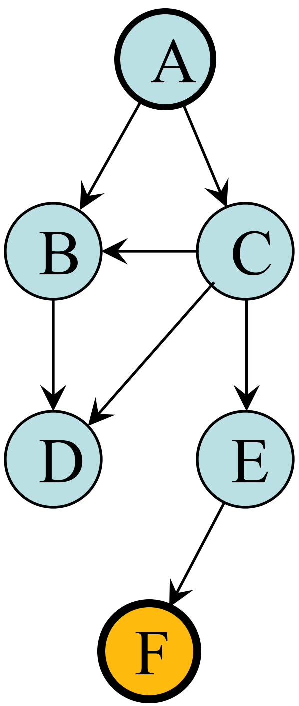
(B C)

(D C)

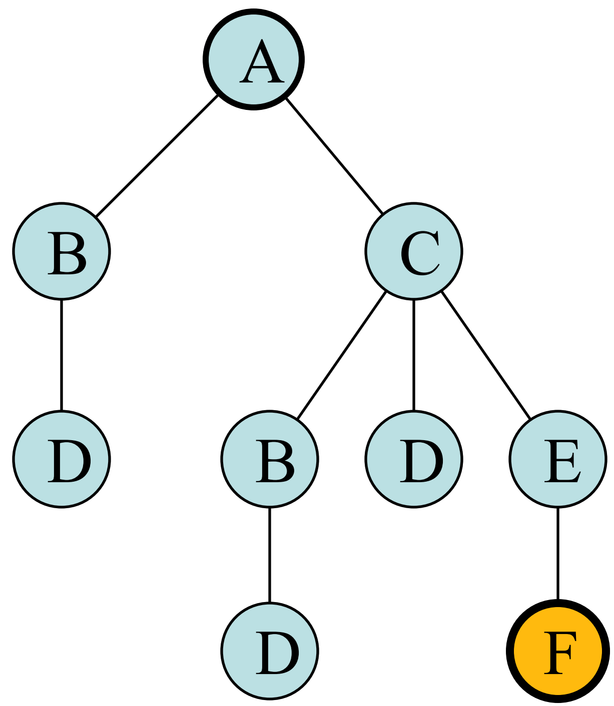
(C)

Example

State space graph



Search tree



Queue

(A)

(B C)

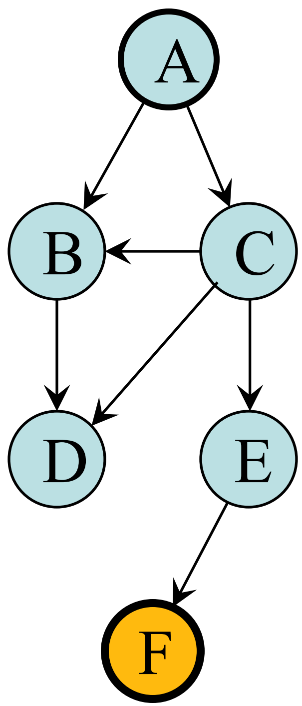
(D C)

(C)

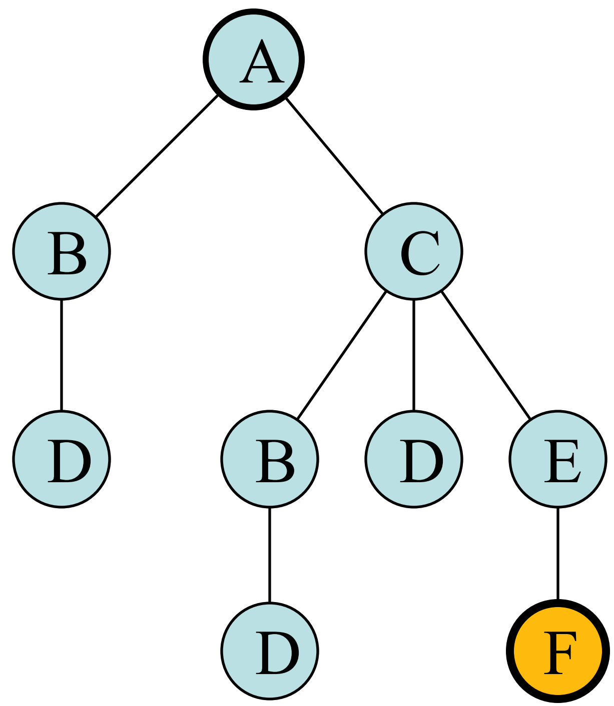
(B D E)

Example

State space graph



Search tree



Queue

(A)

(B C)

(D C)

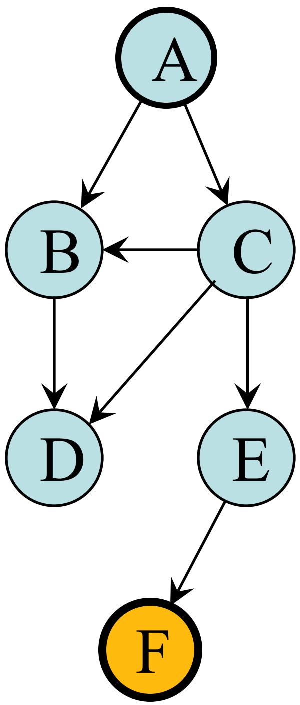
(C)

(B D E)

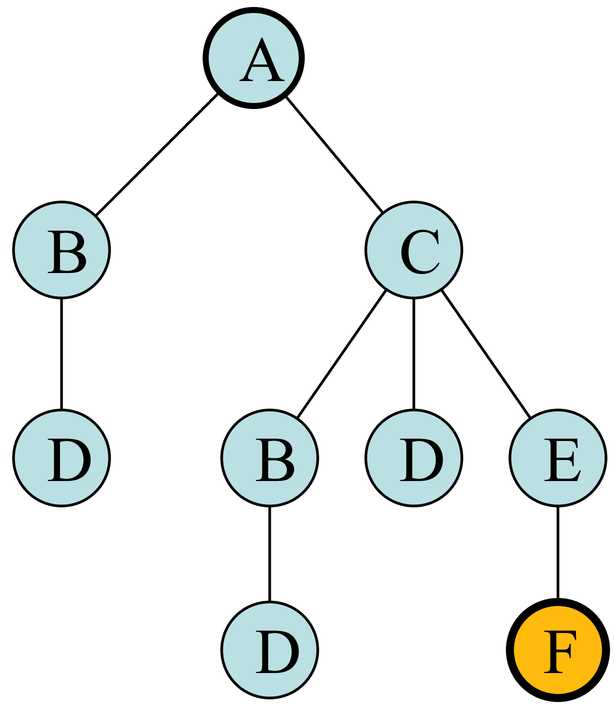
(D D E)

Example

State space graph



Search tree



Queue

(A)

(B C)

(D C)

(C)

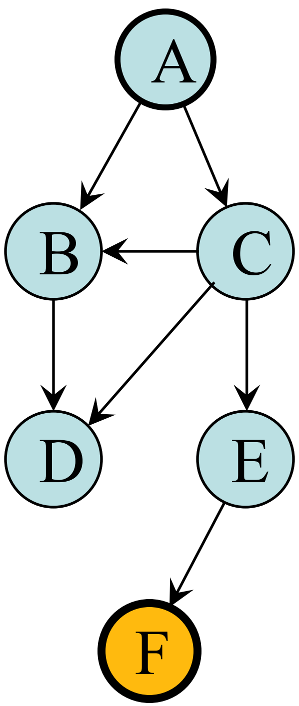
(B D E)

(D D E)

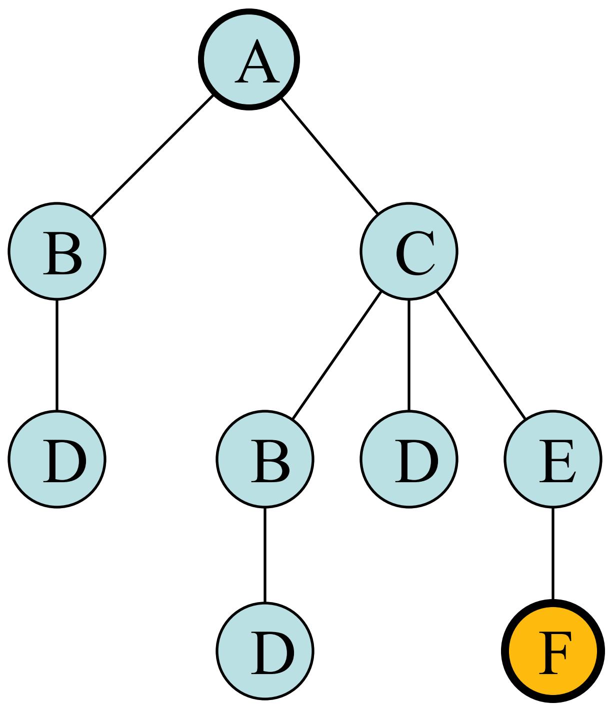
(D E)

Example

State space graph



Search tree



Queue

(A)

(B C)

(D C)

(C)

(B D E)

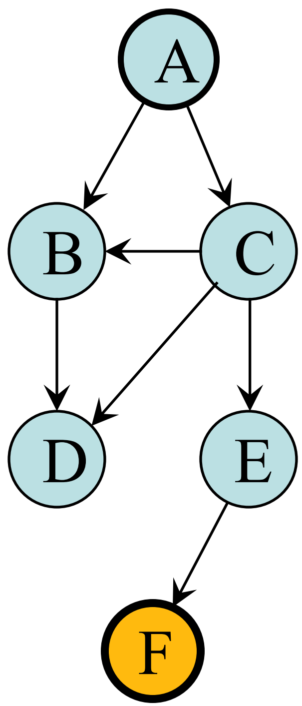
(D D E)

(D E)

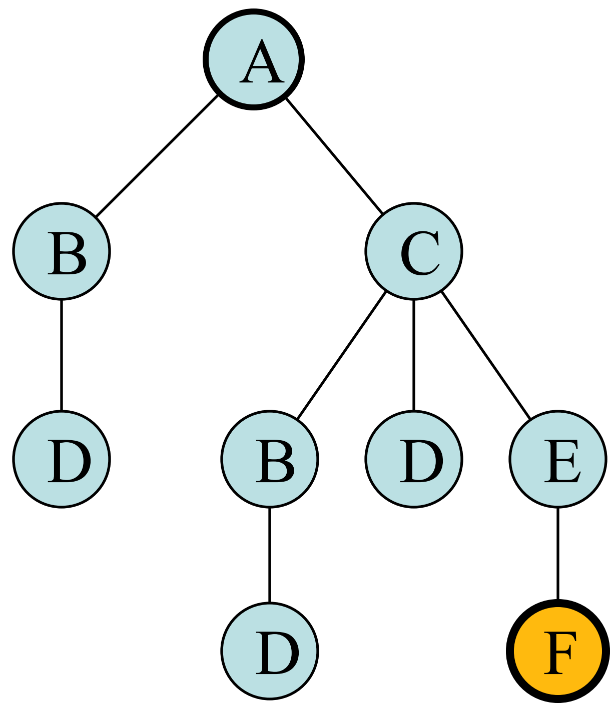
(E)

Example

State space graph



Search tree



Queue

(A)

(B C)

(D C)

(C)

(B D E)

(D D E)

(D E)

(E)

(F)

Depth-First Search

- Complete?

- Optimal?

- Time complexity?

- Space complexity?

m = maximum depth of the search tree
(may be infinite)

Depth-First Search

- Complete? No
- Optimal?
- Time complexity?
- Space complexity?

m = maximum depth of the search tree
(may be infinite)

Depth-First Search

- Complete? No
- Optimal? No
- Time complexity?
- Space complexity?

m = maximum depth of the search tree
(may be infinite)

Depth-First Search

- Complete? No
- Optimal? No
- Time complexity? Exponential: **O(b^m)**
- Space complexity?

m = maximum depth of the search tree
(may be infinite)

Depth-First Search

- Complete? No
- Optimal? No
- Time complexity? Exponential: **O(** b^m **)**
- Space complexity? Polynomial: **O(** bm **)**

m = maximum depth of the search tree
(may be infinite)

What is the difference between the BFS / DFS that you learned from the algorithm / data structure course?

- Nothing, except:
  - Now you are applying them to solve an AI problem
  - The graph can be infinitely large
  - The graph does not need to be known ahead of time (you only need local information: Goal-state checker, Successor function)

Next lecture

- Informed search
- Start game solving / minimax search
- You should:
  - Read Chapter 3 of AIMA textbook