

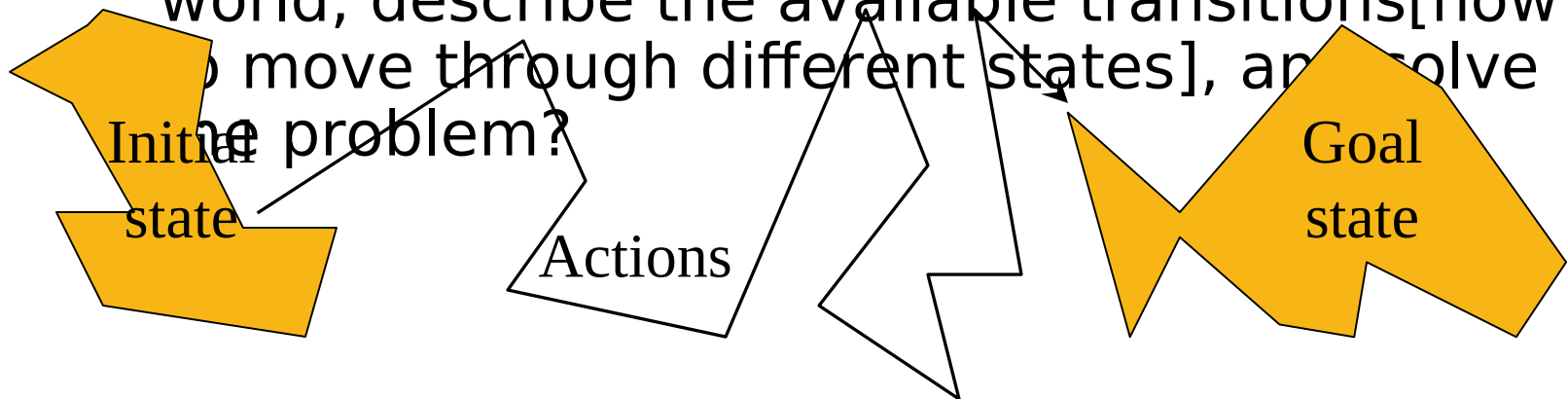
SEARCH STRATEGIES



Building goal-based agents

To build a goal-based agent we need to answer the following questions:

- What is the goal to be achieved?
- What are the actions available to the agent?
- What *relevant* information is necessary to encode in order to describe the state of the world, describe the available transitions[how to move through different states], and solve the problem?



What is the goal to be achieved?



- Could describe a situation we want to achieve, a set of properties that we want to hold, etc.
- Requires defining a “**goal test**” so that we know what it means to have achieved/satisfied our goal.
- Certainly psychologists and motivational speakers always stress the importance of people establishing clear goals for themselves as the first step towards solving a problem.

What are the actions?

- Characterize the **primitive actions** or events that are available for making changes in the world in order to achieve a goal.
- **Deterministic** world: no uncertainty in an action's effects. Given an action (a.k.a. **operator** or move) and a description of the **current world state**, the action completely specifies
 - whether that action *can* be applied to the current world (i.e., is it applicable and legal), and
 - what the exact state of the world will be after the action is performed in the current

Representing actions

- Note also that actions in this framework can all be considered as **discrete events** that occur at an **instant of time**.
 - For example, if “Mary is in class” and then performs the action “go home,” then in the next situation she is “at home.” There is no representation of a point in time where she is neither in class nor at home (i.e., in the state of “going home”).
- The number of actions / operators depends on the **representation** used in describing a state.
 - In the 8-puzzle, we could specify 4 possible moves for each of the 8 tiles, resulting in a total of **$4 \times 8 = 32$ operators**.
 - On the other hand, we could specify four moves for the “blank” square and we would only need **4 operators**.

Representing states

- What information is necessary to encode about the world to sufficiently describe all relevant aspects to solving the goal? That is, what knowledge needs to be represented in a state description to adequately describe the current state or situation of the world?
- The **size of a problem** is usually described in terms of the **number of states** that are possible.
 - Tic-Tac-Toe has about 3^9 states.
 - Checkers has about 10^{40} states.
 - Rubik's Cube has about 10^{19} states.
 - Chess has about 10^{120} states in a typical game.

Closed World Assumption

- We will generally use the **Closed World Assumption**.
- All necessary information about a problem domain is available in each percept[each point in time] so that each state is a complete description of the world.
- There is no incomplete information at any point in time – we have all relevant info at all points for each scenario

Some example problems

- Toy problems and micro-worlds
 - 8-Puzzle
 - Missionaries and Cannibals
 - Remove 5 Sticks
- Real-world problems

8-Puzzle

Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles.

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

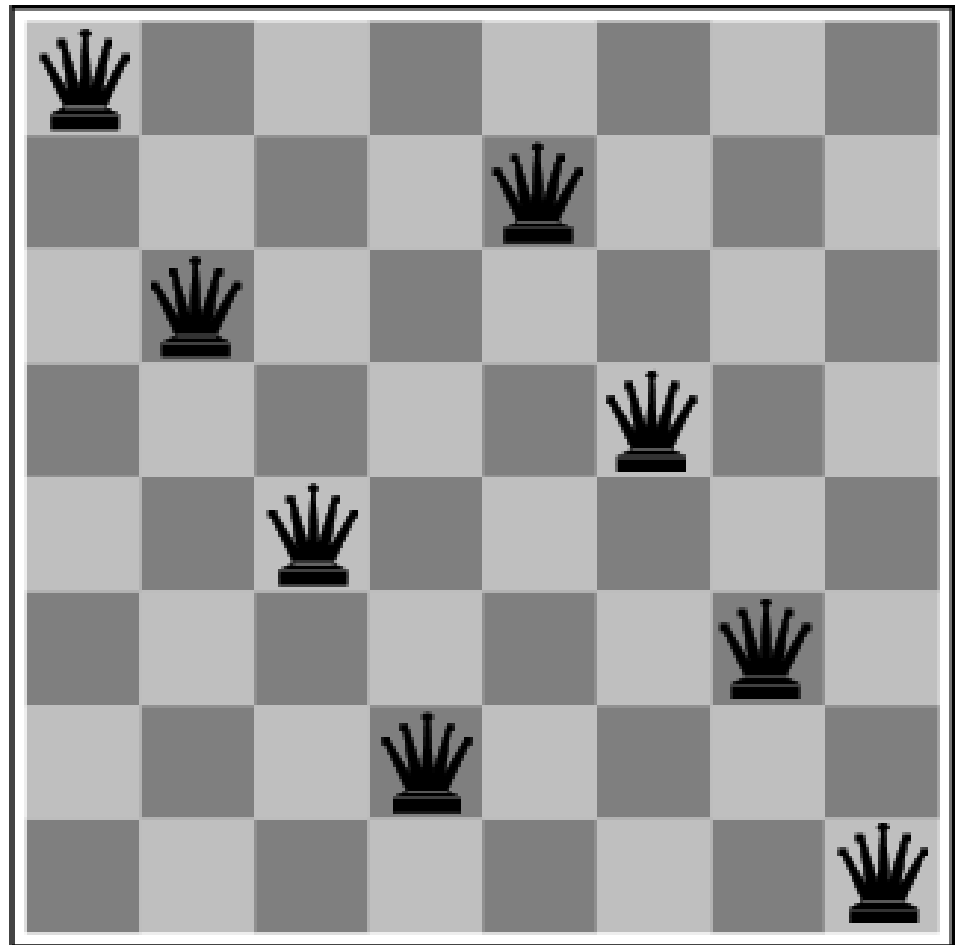
8 puzzle

- **State:** 3 x 3 array configuration of the tiles on the board.
- **Operators:** Move Blank Square Left, Right, Up or Down.
 - This is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used.
- **Initial State:** A particular configuration of the board.
- **Goal:** A particular configuration of the board.

The 8-Queens Problem



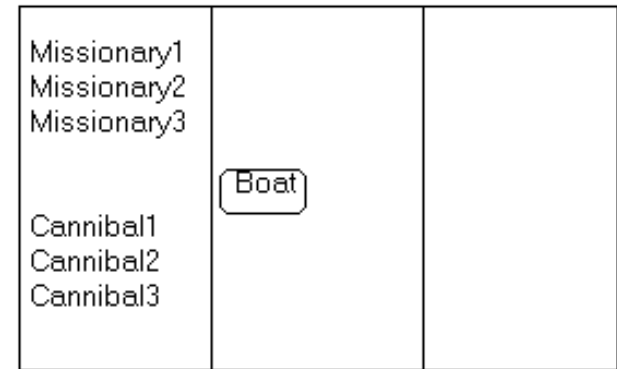
**Place eight
queens on a
chessboard
such that no
queen
attacks any
other!**



Missionaries and Cannibals - EXAMINABLE QN!!!

There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river.

- **Goal:** Move all the missionaries and cannibals across the river.
- **Constraint:** Missionaries can never be outnumbered by cannibals on either side of river, or else the missionaries are killed.
- **State:** configuration of missionaries and cannibals and boat on each side of river.
- **Operators:** Move boat



3 Missionaries and 3 Cannibals wish to cross the river. They have a boat that will carry two people. Everyone can navigate the boat. If at any time the Cannibals outnumber the missionaries on either bank of the river, they will eat the Missionaries. Find the smallest number of crossings that will allow everyone to cross the river safely.

The problem can be solved in 11 moves. But people rarely get the optimal solution, because the MC problem contains a 'tricky' state at the end, where two people move back across the river.

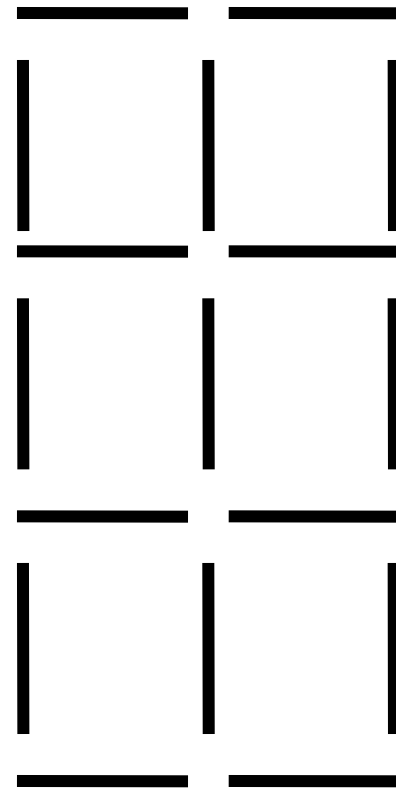
Missionaries and Cannibals Solution

	<u><i>Near side</i></u>		<u><i>Far side</i></u>	
0 Initial setup:	MMMCCC	B	-	
1 Two cannibals cross over:	MMMC		B	CC
2 One comes back:	MMMCC	B		C
3 Two cannibals go over again:	MMM		B	CCC
4 One comes back:	MMMC	B		CC
5 Two missionaries cross:	MC		B	MMCC
6 A missionary & cannibal return:	MMCC	B		MC
7 Two missionaries cross again:	CC		B	MMMC
8 A cannibal returns:	CCC	B		MMM
9 Two cannibals cross:	C		B	MMMCC
10 One returns:	CC	B		MMMC
11 And brings over the third:	-		B	MMMCCC

Remove 5 Sticks – FIND SOLUTION

B4 EXAM!!!

- Given the following configuration of sticks, remove exactly 5 sticks in such a way that the remaining configuration forms exactly 3 squares.



Some more real-world problems

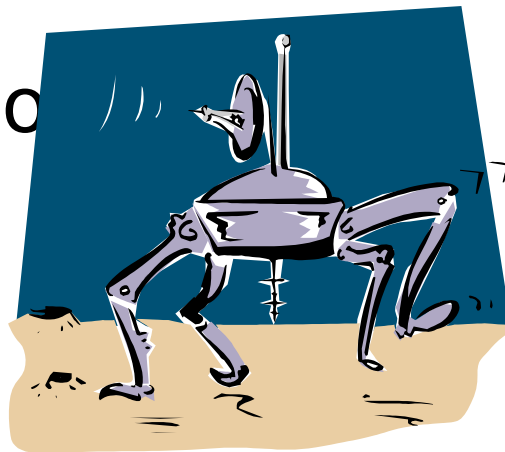
- Route finding

Touring (traveling salesman) – finding least expensive route to traverse a new town with numerous paths

- Logistics

- VLSI layout

- Robot navigation



Knowledge representation issues

- What's in a state ?
 - Is the color of the boat relevant to solving the Missionaries and Cannibals problem? Is sunspot activity relevant to predicting the stock market? What to represent is a very hard problem that is usually left to the system designer to specify.
- What **level of abstraction** or detail to describe the world.
 - Too fine-grained and we'll "miss the forest for the trees."
Too coarse-grained and we'll miss critical details for solving the problem.
- The number of states depends on the representation and level of abstraction chosen.
 - In the Remove-5-Sticks problem, if we represent the individual sticks, then there are 17-choose-5 possible ways of removing 5 sticks. On the other hand, if we represent the "squares" defined by 4 sticks, then there are 6 squares initially and we must remove 3 squares, so only 6-choose-3 ways of removing 3 squares.

Formalizing search in a state space

- A state space is a **graph** (V, E) where V is a set of **nodes** and E is a set of **arcs**, and each arc is directed from a node to another node.
- Each **node** is a data structure that contains a state description plus other information such as the parent of the node, the name of the operator that generated the node from that parent, and other bookkeeping data.
- Each **arc** corresponds to an instance of one of the operators. When the operator is applied to the state associated with the arc's source node, then the resulting state is the state associated with the arc's destination node.

Formalizing search II

- Each arc has a **fixed, positive cost** associated with it corresponding to the cost of the operator.
- Each node has a set of **successor nodes** corresponding to all of the legal operators that can be applied at the source node's state. (leaf node has no legal operators as it has no successor nodes)
 - The process of expanding a node means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as **start nodes**. (parent nodes with one or more successor nodes)

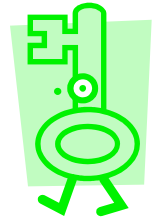
Formalizing search III

- A **solution** is a sequence of operators that is associated with a path in a state space from a start node to a goal node.
- The **cost of a solution** is the sum of the arc costs on the solution path.
 - If all arcs have the same (unit) cost, then the solution cost is just the length of the solution (number of steps / state transitions)

Formalizing search IV

- **State-space search** is the process of searching through a state space for a solution by **making explicit** a sufficient portion of an **implicit** state-space graph to find a goal node.
 - For large state spaces, it isn't practical to represent the whole space.
 - Initially $V = \{S\}$, where S is the start node; when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E . This process continues until a goal node is found.
- Each node implicitly or explicitly represents a partial solution path (and cost of the partial solution path) from the start node to the given node.
 - In general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix.

Key procedures to be defined



□ *EXPAND*

- Generate all successor nodes of a given node

□ *GOAL-TEST*

- Test if state satisfies all goal conditions

□ *QUEUING-FUNCTION*

- Used to maintain a ranked list of nodes that are candidates for expansion – if there are multiple successors from one specific node, must choose which successor node to expand first



Bookkeeping

- Typical node data structure includes:
 - State at this node
 - Parent node
 - Operator applied to get to this node
 - Depth of this node (number of operator applications since initial state)
 - Cost of the path (sum of each operator application so far) – cost associated with all the arcs traversed from the start node to get to a specific node

Evaluating Search Strategies (when there are multiple existing solutions)

□ **Completeness**

- Guarantees finding a solution whenever one exists

□ **Time complexity**

- How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded

□ **Space complexity**

- How much space is used by the algorithm? Usually measured in terms of the maximum size of the “nodes” list during the search

□ **Optimality/Admissibility**

- If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

UNINFORMED SEARCH

- Uninformed search?
- Uninformed search methods
 - Depth-first search
 - Breadth-first search
 - Iterative deepening search
 - Bi-directional search



Uninformed Search?

- Simply searches the state space.
- Can only distinguish between goal state and non-goal state.
- Sometimes called Blind search as it has no information or knowledge about its domain.

Uninformed Search

Characteristics

- Blind Searches have no preference as to which state (node) that is expanded next. - queueing function doesn't apply.
- The different types of blind searches are characterised by the order in which they expand the nodes.
- This can have a dramatic effect on how well the search performs when measured against the four criteria we defined earlier.

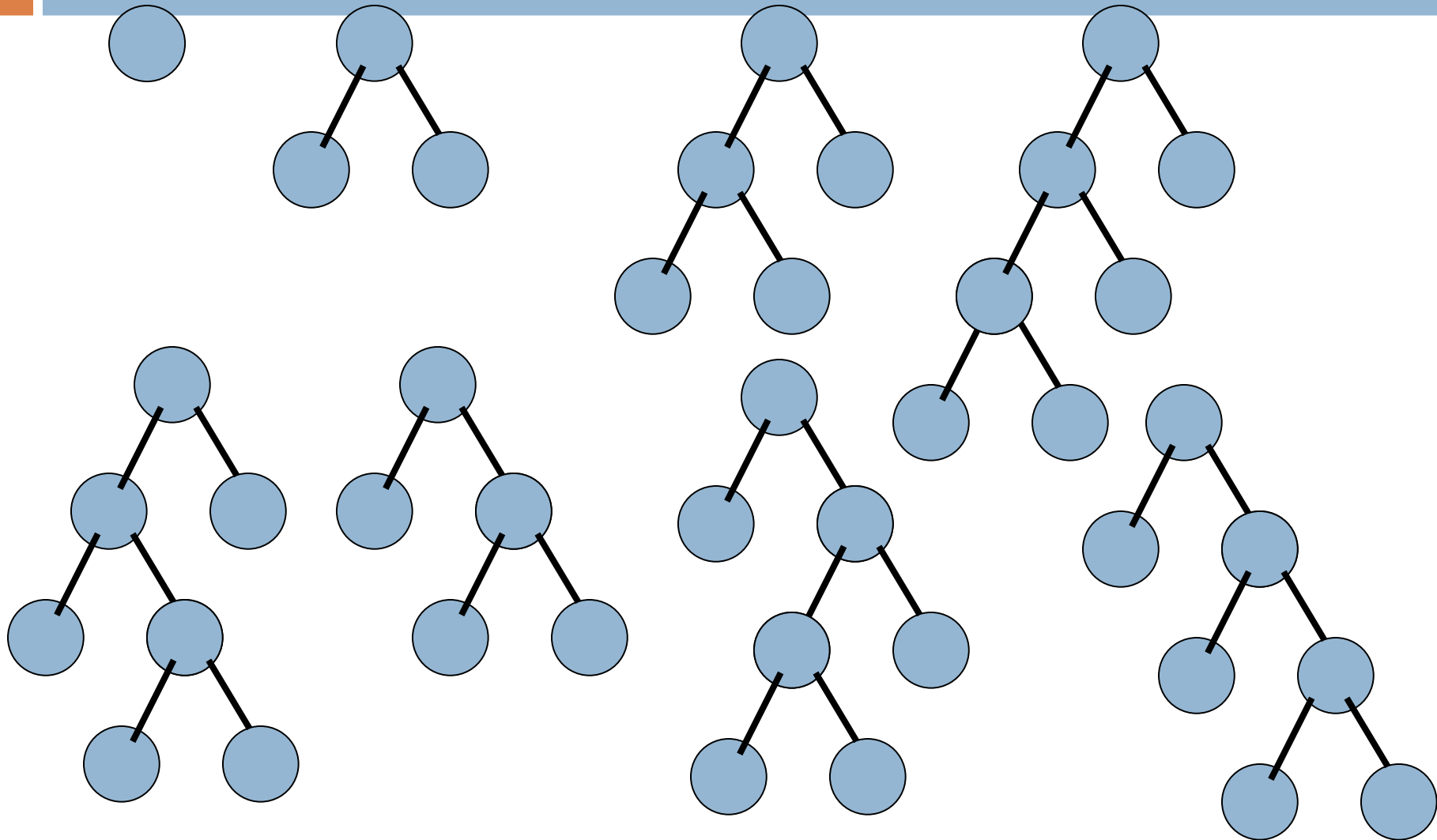
Uninformed (Blind) Search Methods

- Methods that do not use any specific knowledge about the problem.
- These are:
 - Depth-first search
 - Breadth-first search
 - Non deterministic search
 - Iterative deepening search
 - Bi-directional search

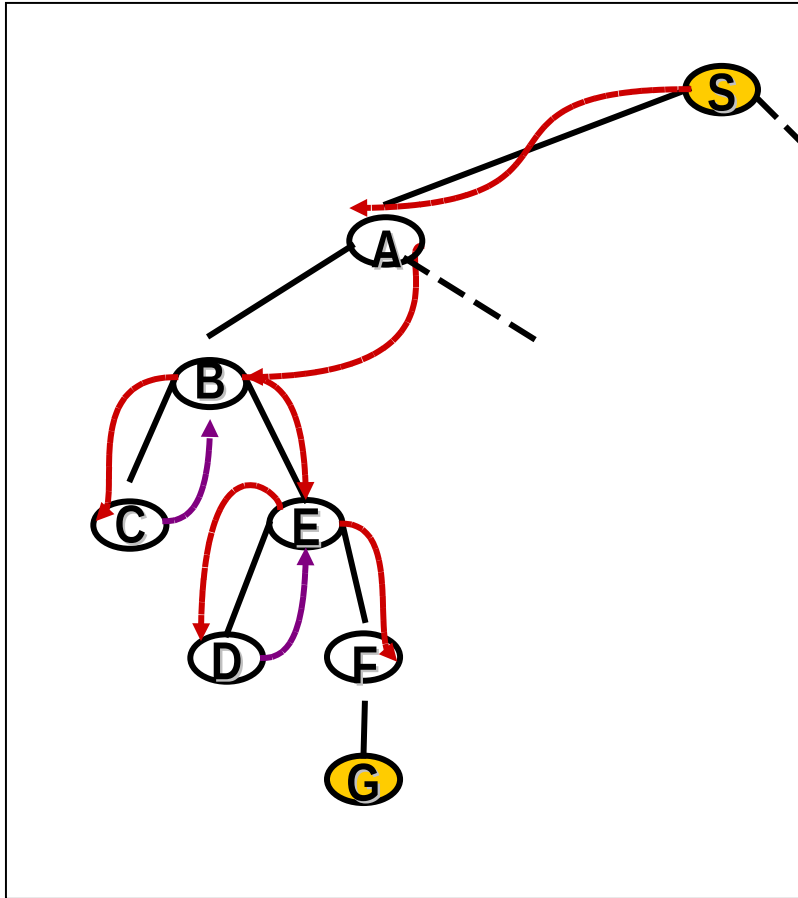
1. Depth-first Search

- Expand the tree as deep as possible, returning to upper levels when needed.
- One can return to upper levels if they encounter a leaf node before reaching the goal state.

Depth-First Search – expanding childs (left to right) in search of goal state



Depth-first search = Chronological backtracking

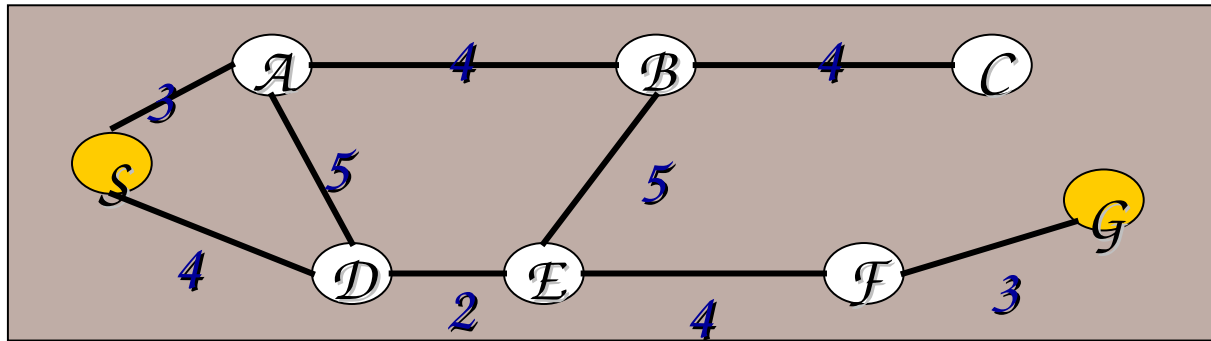


- Select a child
 - convention: left-to-right
- Repeatedly go to next child, as long as possible.
- Return to left-over alternatives (higher-up i.e. from the beginning) only when needed.

Depth-first algorithm:

1. QUEUE \leftarrow path only containing the root;
2. WHILE { QUEUE is not empty
 AND goal is not reached

 DO remove the first path from the QUEUE [s];
 create new paths (to all children) [successor nodes of s];
 reject the new paths with loops;
 add the new paths to front of QUEUE;
3. IF goal reached
 THEN success;
 ELSE failure;



1. **QUEUE** <-- path only containing the root;
2. **WHILE** { **QUEUE** is not empty
AND goal is not reached

DO remove the first path from the **QUEUE**;
 create new paths (to all children);
 reject the new paths with loops;
 add the new paths to front of **QUEUE**;
3. **IF** goal reached
THEN success;
ELSE failure;

Trace of depth-first for running example:

- (S) S removed, (SA,SD) computed and added
- (SA, SD) SA removed, (SAB,SAD,SAS) computed, (SAB,SAD) added
- (SAB,SAD,SD) SAB removed, (SABA,SABC,SABE) computed, (SABC,SABE) added
- (SABC,SABE,SAD,SD) SABC removed, (SABCB) computed, nothing added
- (SABE,SAD,SD) SABE removed, (SABEB,SABED,SABEF) computed, (SABED,SABEF) added
- (SABED,SABEF,SAD,SD) SABED removed, (SABEDS,SABEDA,SABEDE) computed,

Evaluation criteria:

- **Completeness**
 - Does the algorithm always find a path?
 - (for every state space such that a path exists) if a solution exists, it will be found.
- **Speed** (worst time complexity) :
 - What is the highest number of nodes that may need to be created?
- **Memory** (worst space complexity) :
 - What is the largest amount of nodes that may need to be stored?
- Expressed in terms of:
 - d = depth of the tree
 - b = (average) branching shallowest factor of the tree
 - m = depth of the solution

Repeated States problem

In the above discussion we have ignored an important complication that often arises in search processes – the possibility that we will waste time by expanding states that have already been expanded before somewhere else on the search tree.

Avoiding Repeated States

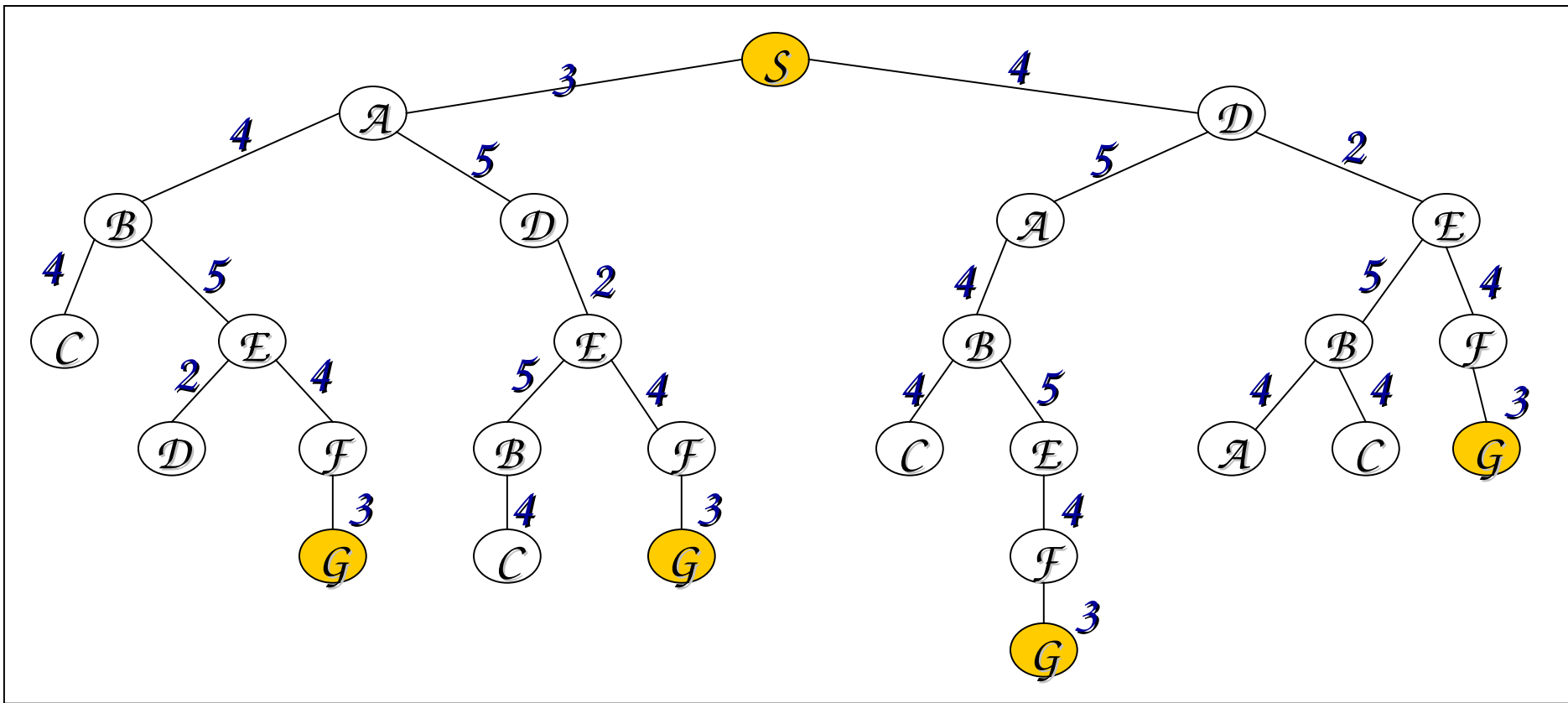
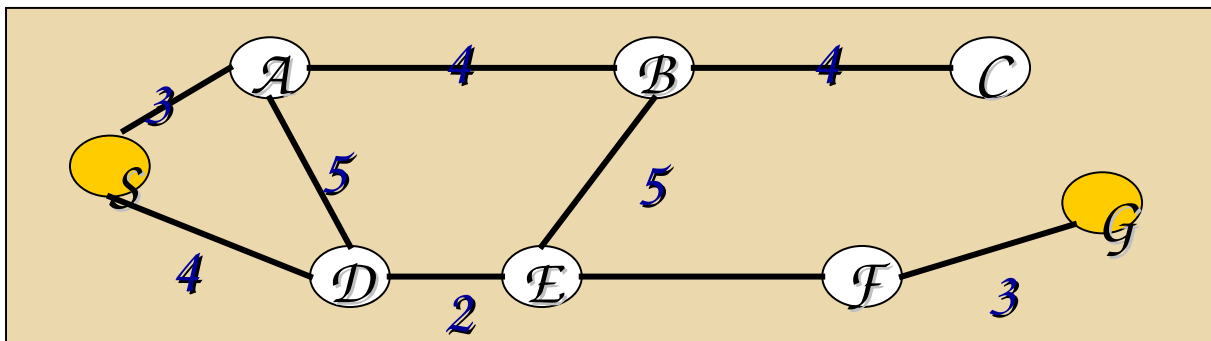
- 1. Never return to the state you have just come from - NO BACKTRACKING unless you've reached leaf nodes**
- 2. Never create search paths with cycles in them**
- 3. Never generate states that have already been generated before - store all generated states in memory.**

Note: approximations !!

- In our complexity analysis, we do not take the built-in **loop-detection** into account [time complexity].
- The results only ‘formally’ apply to the variants of our algorithms **WITHOUT** loop-checks.
- Studying the effect of the loop-checking on the complexity is hard:
 - overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree.
- Also: our analysis **DOES NOT** take the length (space) of representing paths into account !!

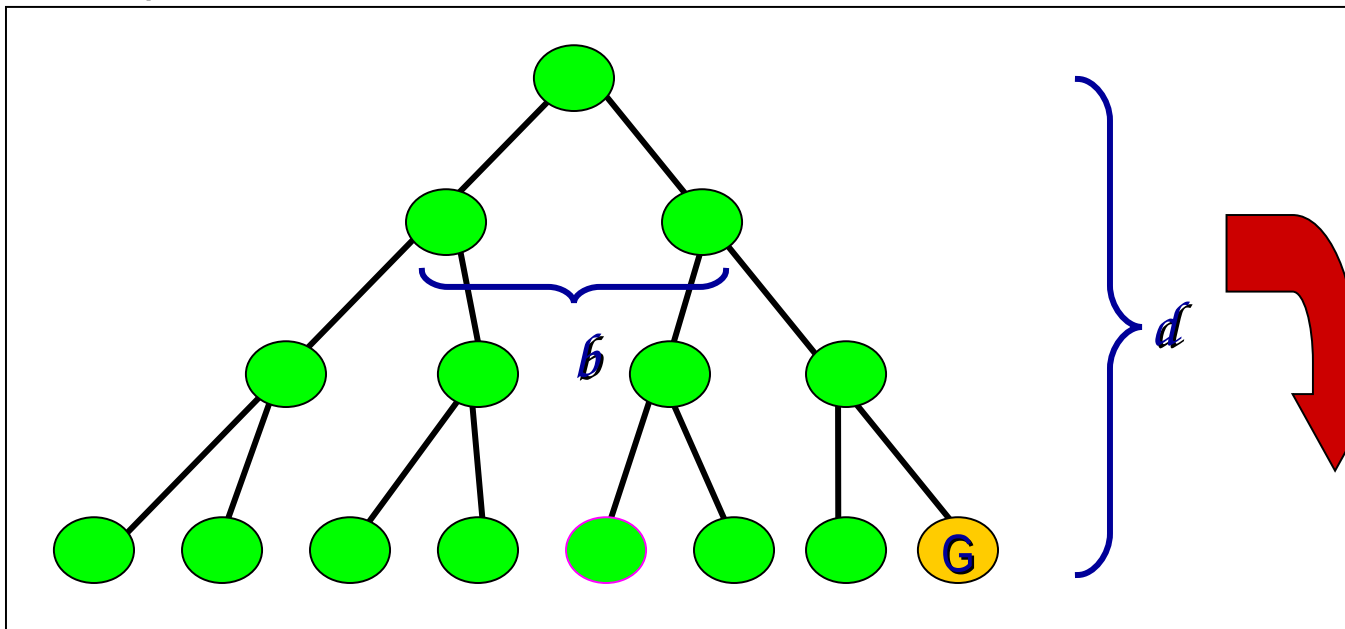
Completeness (depth-first)

- Complete for FINITE (implicit) NETS.
 - (= State space with finitely many nodes)
- **IMPORTANT:**
 - This is due to integration of LOOP-checking in this version of Depth-First (and in all other algorithms that will follow) !
 - IF we do not remove paths with loops, then Depth-First is not complete (may get trapped in loops of a finite State space)
- **Note:** does NOT find the shortest path.



Speed (depth-first)

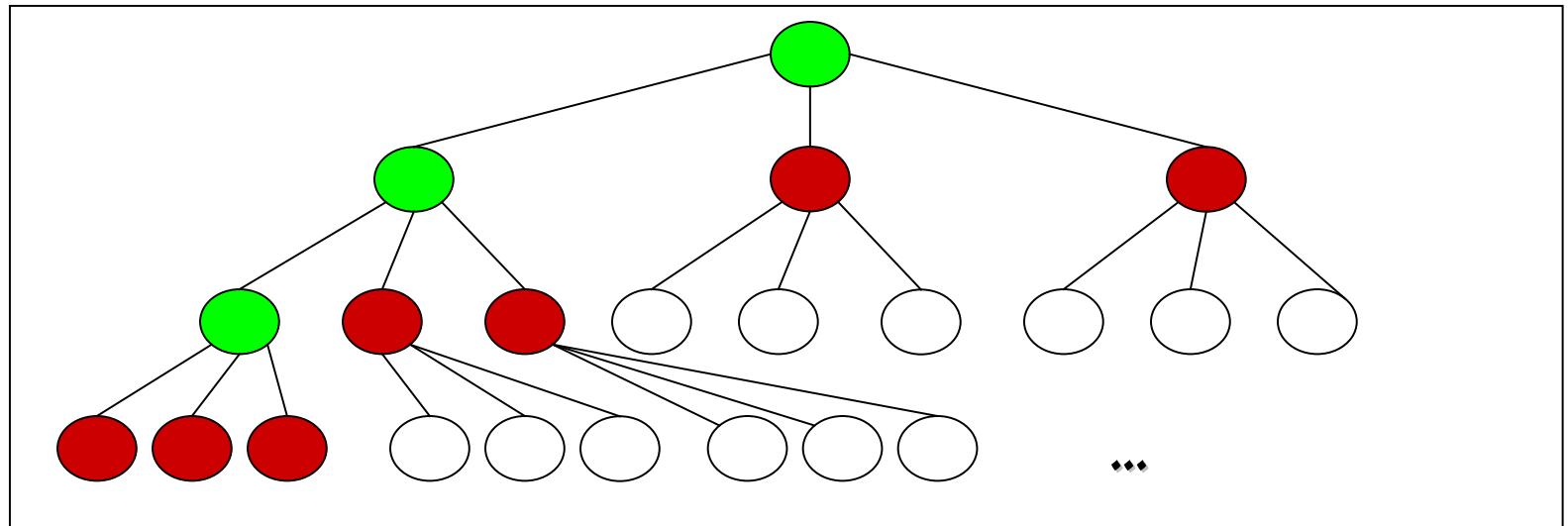
- In the worst case scenario:
 - the (only) goal node may be on the right-most branch,




- Time complexity $= b^d + b^{d-1} + \dots + 1 = \frac{b^{d+1} - 1}{b - 1}$
- Thus: $O(b^d) - O(\text{breadth}[\text{initial successor nodes}] \text{ raised to depth})$

Memory (depth-first)

- Largest number of nodes in QUEUE is reached in bottom left-most node.



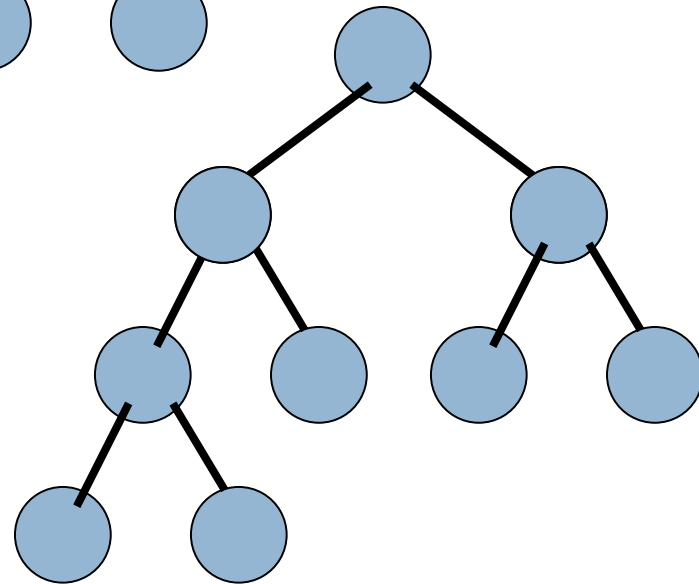
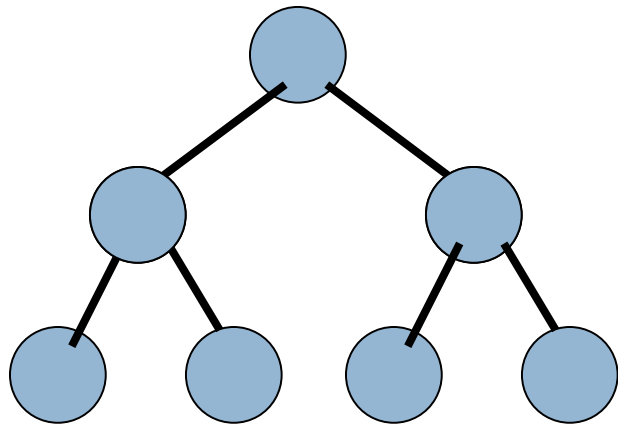
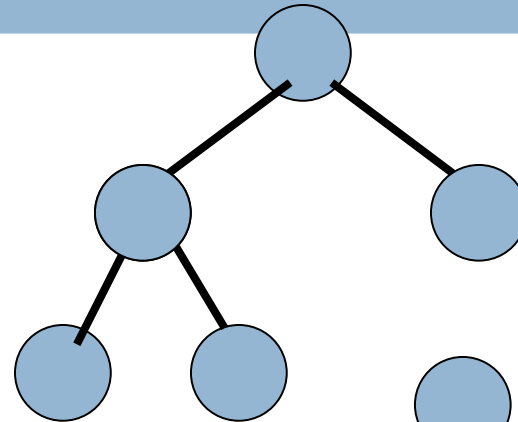
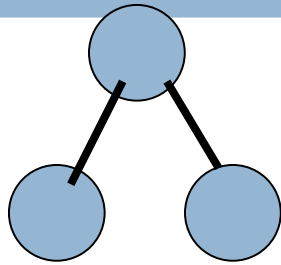
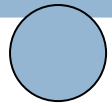
- QUEUE contains all  nodes. Thus: 7.
- In General: $((b-1) * d) + 1$

Order: $O(d*b)$ – O (breadth[initial successor nodes] multiplied by depth[number of node levels])

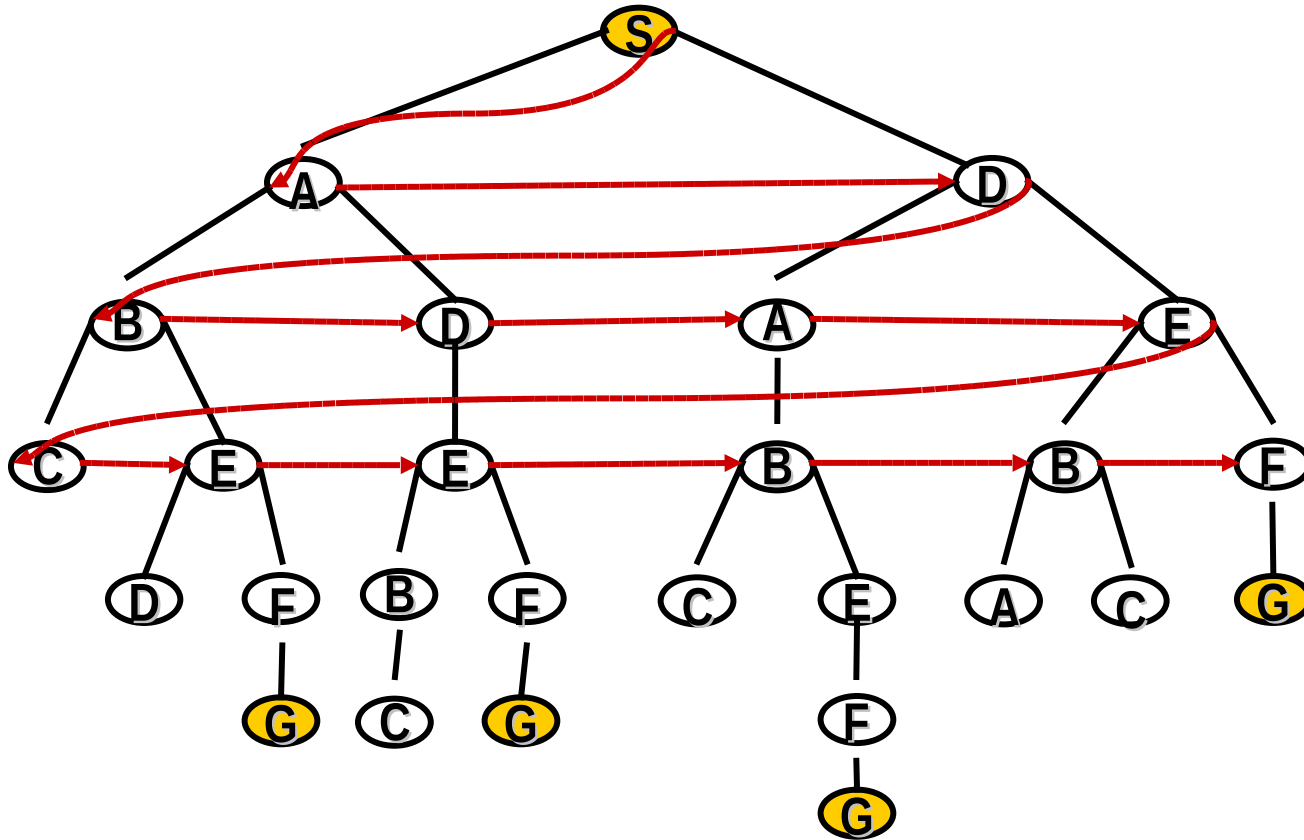
2. Breadth-first Search

- Expand the tree layer by layer, progressing in depth – deals with one node level at a time.
- Generate children without expanding them first.
- In other words,
 - Expand root node first
 - Expand all nodes at level 1 before expanding level 2OR
 - Expand all nodes at level d before expanding

Breadth-First Search



Breadth-first search:



- Move downward, level by level [layer by layer], until goal is reached.

Breadth-first algorithm: (same as depth-first search algorithm except highlighted difference).

1. QUEUE \leftarrow path only containing the root;

2. WHILE QUEUE is not empty
AND goal is not reached

{ DO remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths to back of QUEUE;

3. IF goal reached
THEN success;
ELSE failure;



**ONLY
DIFFERENCE !**

Trace of breadth-first for running example:

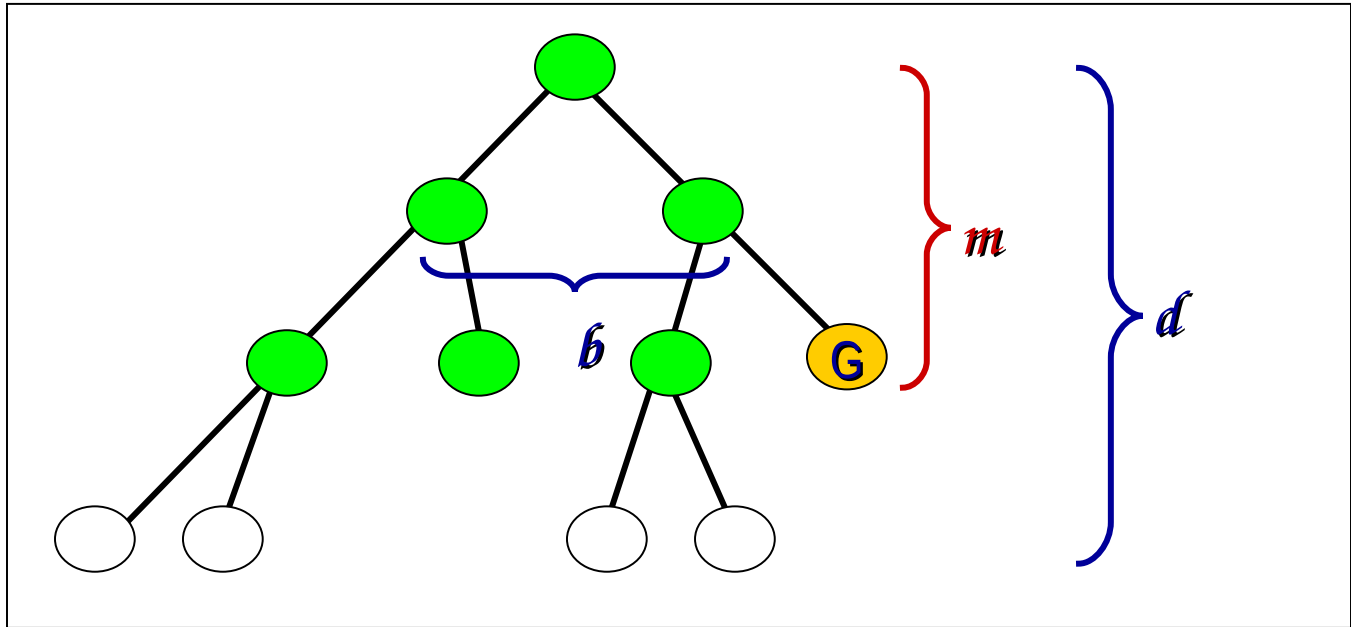
- (S) S removed, (SA,SD) computed and added
- (SA, SD) SA removed, (SAB,SAD,SAS) computed, (SAB,SAD) added
- (SD,SAB,SAD) SD removed, (SDA,SDE,SDS) computed, (SDA,SDE) added
- (SAB,SAD,SDA,SDE) SAB removed, (SABA,SABE,SABC) computed, (SABE,SABC) added
- (SAD,SDA,SDE,SABE,SABC) SAD removed, (SADS,SADA,SADE) computed, (SADE) added
- etc, until QUEUE contains:

Completeness (breadth-first)

- **Complete**
 - even for infinite implicit NETS !
- Would even remain complete without our loop-checking.
- **Note:** ALWAYS finds the shortest path.

Speed (breadth-first)

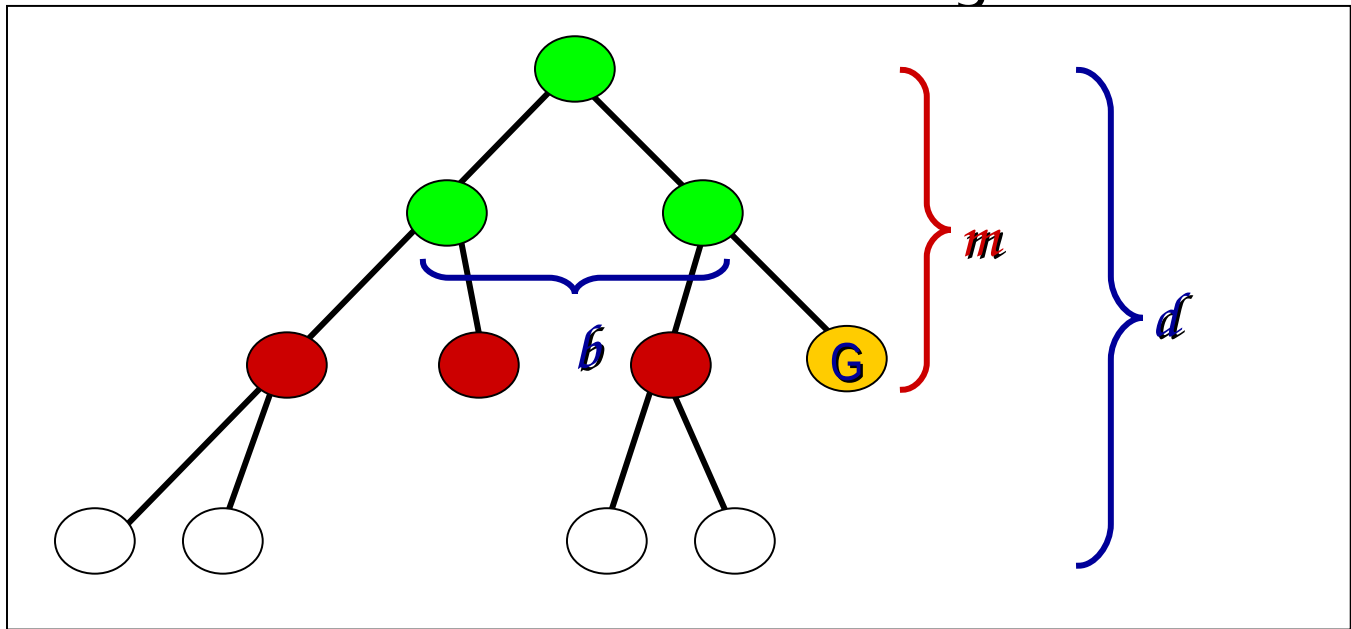
- If a goal node is found on depth **m** of the tree, all nodes up till that depth are created.



- Thus: $O(b^m)$ [m reps a state where there are no ‘dangling’ nodes]
- note: depth-first would also visit deeper nodes.

Memory (breadth-first)

- Largest number of nodes in QUEUE is reached on the level m of the goal node.



- QUEUE contains all ● and G nodes. (Thus: 4) .

In General: b^m [m reps a state where there are no 'dangling' nodes]

- This usually is MUCH worse than depth-first !!

Exponential Growth (breadth-first)

Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	kbytes
2	111	0.1	second	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

Time and memory requirements for breadth-first search, assuming a branching factor of 10, 100 bytes per node and searching 1000 nodes/second

Exponential Growth - Observations

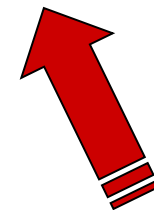
- Space is more of a factor to breadth first search than time
- Time is still an issue. Who has 35 years to wait for an answer to a level 12 problem (or even 128 days to a level 10 problem)
- It could be argued that as technology gets faster then exponential growth will not be a problem. But even if technology is 100 times faster we would still have to wait 35 years for a level 14 problem and what if we hit a level 15 problem!

Practical evaluation:

- **1.Depth-first search:**
 - IF the search space contains very deep branches without solution, THEN Depth-first may waste much time in them.
- **2. Breadth-first search:**
 - Is VERY demanding on memory !
- **Solutions ??**
 - Non-deterministic search
 - Iterative deepening

Non-deterministic search: (same as depth-first search algorithm except highlighted difference).

1. QUEUE \leftarrow path only containing the root;
2. WHILE $\left\{ \begin{array}{l} \text{QUEUE is not empty} \\ \text{AND goal is not reached} \end{array} \right.$
 - DO remove the first path from the QUEUE;
 - create new paths (to all children);
 - reject the new paths with loops;
 - add the new paths in random places in QUEUE;
3. IF goal reached
 - THEN success;
 - ELSE failure;



3. Iterative deepening Search

- Restrict a depth-first search to a fixed depth.
- If no path was found, increase the depth and restart the search from parent node.

Depth-limited search:

1. **DEPTH** <-- <some natural number>
QUEUE <-- path only containing the root;
2. **WHILE** **QUEUE** is not empty
 AND goal is not reached

 DO { remove the first path from the **QUEUE**;
 IF path has length smaller than **DEPTH**
 create new paths (to all children);
 reject the new paths with loops;
 add the new paths to front of **QUEUE**;
3. **IF** goal reached
 THEN success;
 ELSE failure;

Iterative deepening algorithm:

1. ***DEPTH*** \leftarrow 1

2. ***WHILE*** goal is not reached

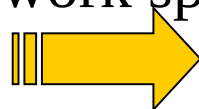
DO perform Depth-limited search;
 increase ***DEPTH*** by 1;

Iterative deepening: the best 'blind' search.

- Complete: yes - even finds the shortest path (like breadth first) .
- Memory: b^*m (combines advantages of depth- and breadth-first)
- Speed:
 - If the path is found for **Depth** = m , then how much time was wasted constructing the smaller trees??

$$\blacksquare \quad b^{m-1} + b^{m-2} + \dots + 1 = \frac{b^m - 1}{b - 1} = O(b^{m-1})$$

- While the work spent at **DEPTH** = m itself is $O(b^m)$



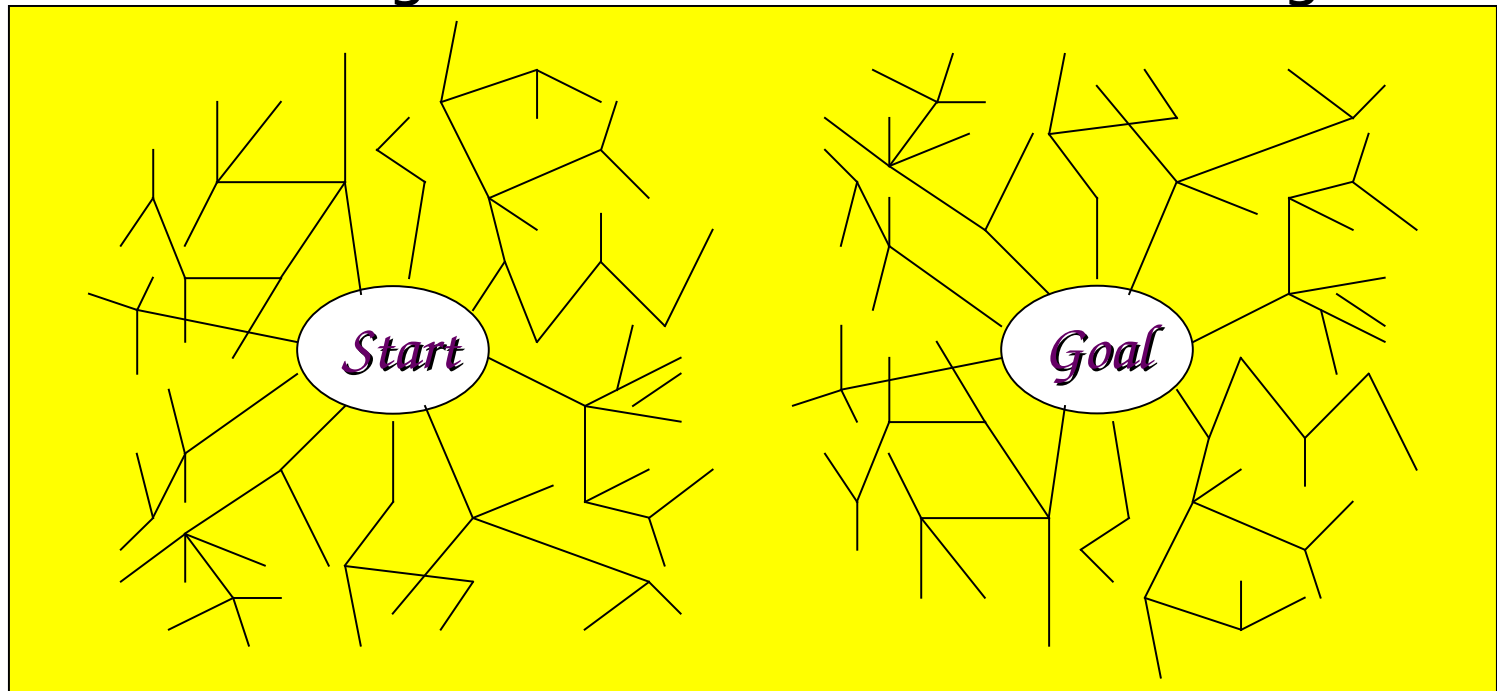
In general: VERY good trade-off

4. Bi-directional Search

- Compute the tree from the start node and from a goal node, until these two meet.

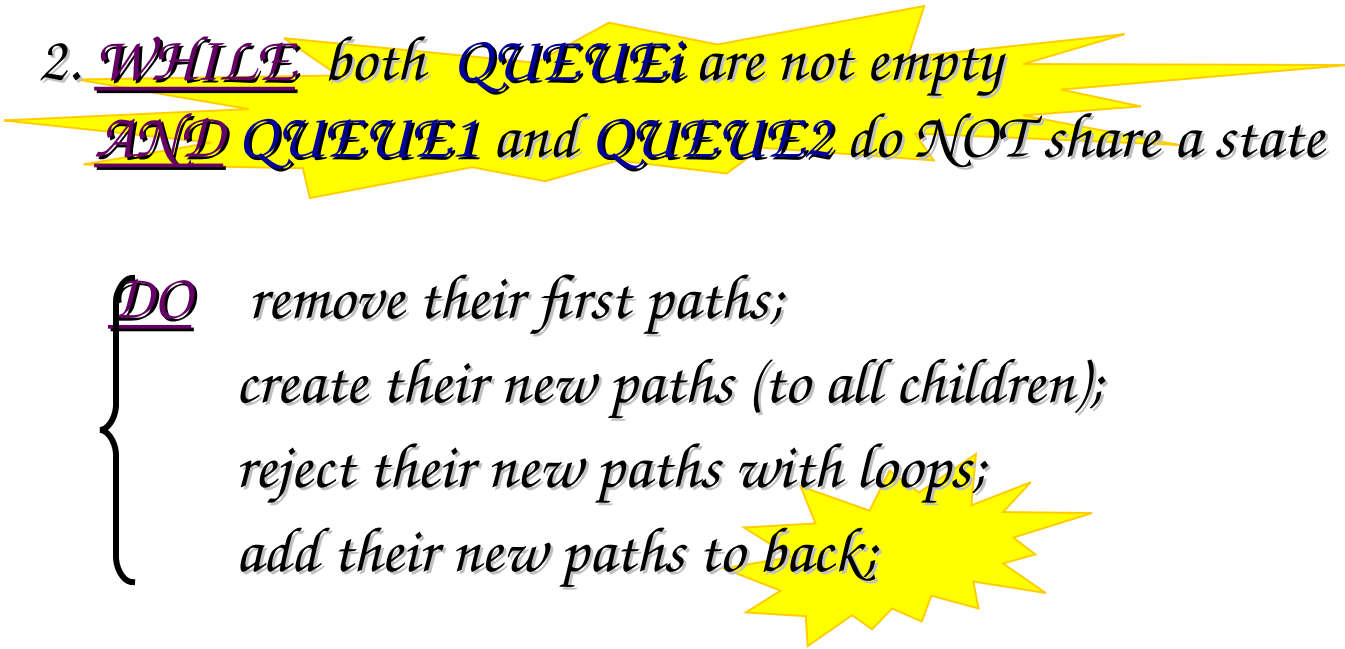
Bi-directional search

- IF you are able to EXPLICITLY describe the GOAL state, AND
- you have BOTH rules for FORWARD reasoning AND BACKWARD reasoning:



Bi-directional algorithm:

1. *QUEUE1* \leftarrow path only containing the root;
QUEUE2 \leftarrow path only containing the goal;
2. WHILE both *QUEUEi* are not empty
AND *QUEUE1* and *QUEUE2* do NOT share a state

DO  remove their first paths;
create their new paths (to all children);
reject their new paths with loops;
add their new paths to back;
3. IF *QUEUE1* and *QUEUE2* share a state
THEN success;
ELSE failure;

Properties (Bi-directional):

- Complete: Yes.
- Speed: If the test on common state can be done in constant time (hashing):
 - $2 * O(b^{m/2}) = O(b^{m/2})$
- Memory: similarly: $O(b^{m/2})$

Disadvantage of BDS

- We don't really know the solution, only a description of it.
- There may be many solutions, and we have to choose some to work backwards from.
- We cannot reverse our operators to work backwards from the solution.
- We have to record all the paths from both sides to see if any two meet at the same point - this may take up a lot of memory, and checking. Requires a lot of computational power.

INFORMED SEARCH

- Informed search?
- Informed search methods
 - Best-first search
 - Greedy search
 - A*Search
 - A* Variants



INFORMED SEARCHES

Informed search uses some kind of evaluation function to tell us how far each expanded state is from a goal state, and/or some kind of heuristic function to help us decide which state is likely to be the best one to expand next.

Goal:

- To see how information about the state space can prevent algorithms from stumbling in the dark

Heuristic Functions

- A heuristic search may use a **heuristic function**, which is a calculation to *estimate* how costly (in terms of the path cost) a path from a state will be to a goal state.
- Heuristic functions can be derived in a number of ways;

Cont.

- Mathematically
- Think them up as good ideas
- Identify common elements in various search solutions, and convert these to useful heuristic functions.
- Use of computer programs to derive heuristic functions. E.g. ABSOLVER program.

Heuristic Search Strategies

□ Uniform Path Cost Search

- Similar to a BFS
- A uniform path cost search chooses which node to expand by looking at the path cost for each node: the node which has least cost to get to is expanded first.
- It is an optimal search strategy. It is guaranteed to find the least expensive goal.

Best-first search

- = An instance of TREE/GRAPH-SEARCH algorithm
- Expand the most desirable node:
- Expand node n based on an **evaluation function**, $f(n)$
- $f(n)$ measures distance to the goal
 - Select the node with the lowest $f(n)$
- Fringe implementation:
 - Priority queue
 - = data structure that will maintain the fringe in ascending order of f -values
- Expand what “appears” to be best (according to f), not really the best node
 - Unless f is indeed accurate, then it truly is “best”

Best-First Search Algorithms

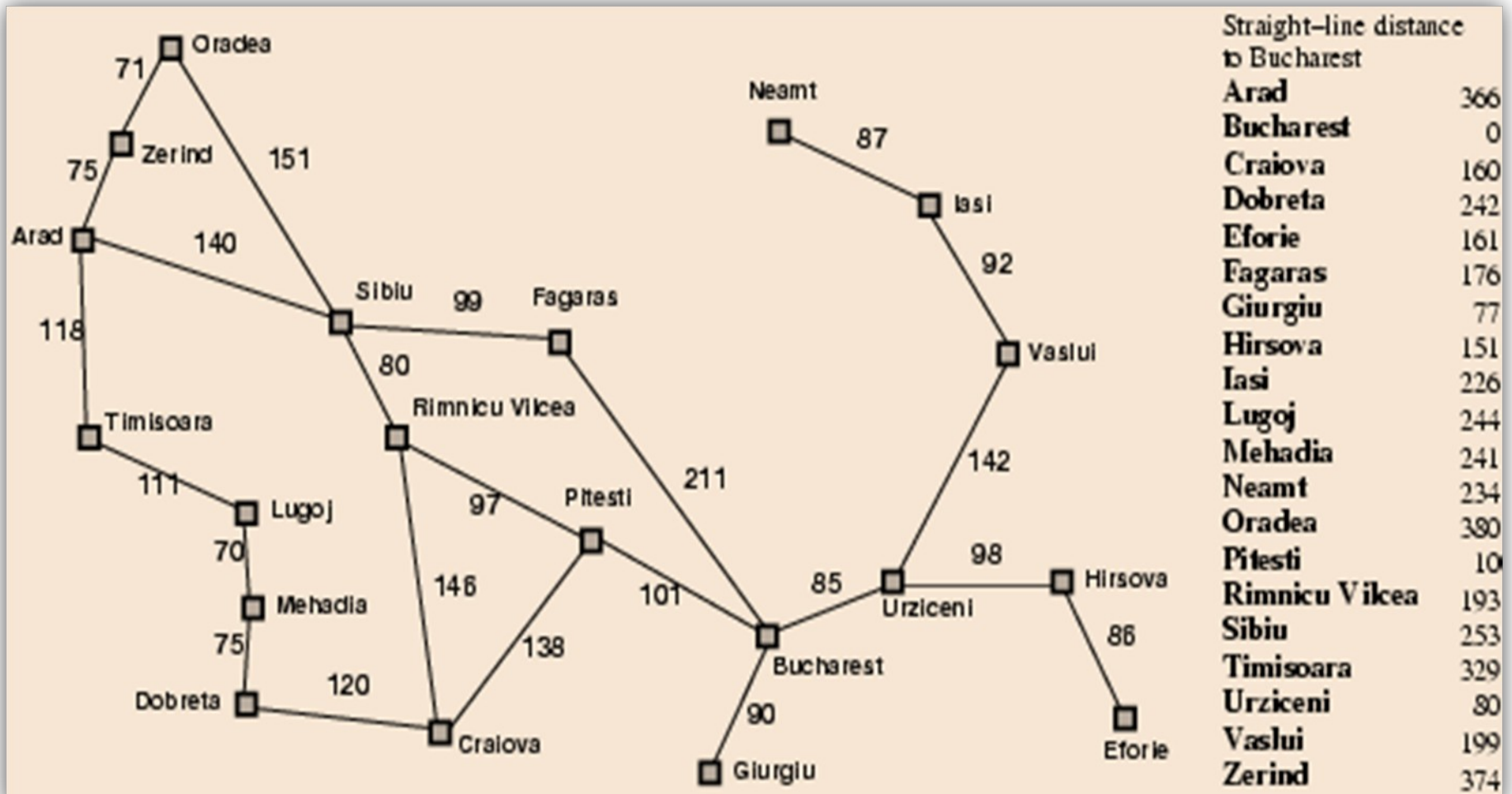
Different evaluation functions

- Heuristic function: $h(n)$
 - Estimated cost of the cheapest path from node n to a goal node.
- E.g. Holiday in Romania
 - $h = \text{the straight distance Arad-Bucharest}$
- Assumption:
 - If $n = \text{goal node}$, then $h(n) = 0$
- Two Best-first search algorithms:
 - Greedy best-first search
 - A* search

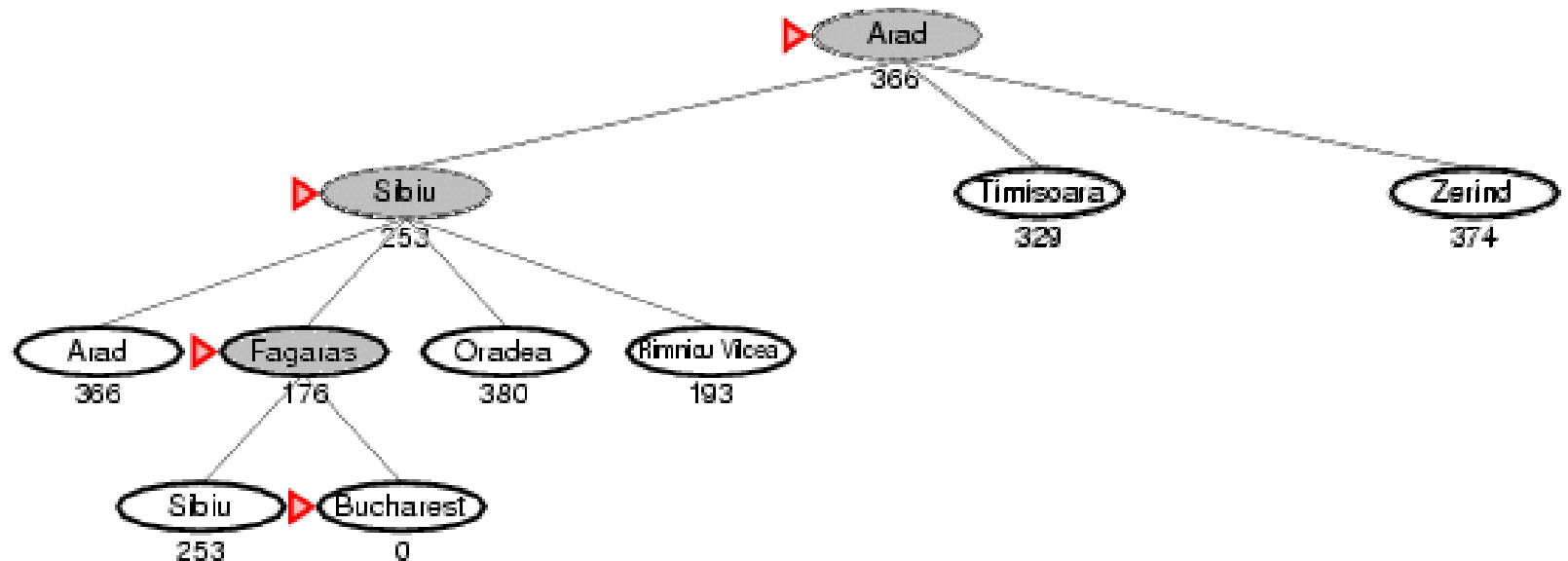
Greedy Best-First Search

- A.k.a. “greedy search” or “best-first search”
- Method:
 - Expand the node that **appears** to be closest to the goal
 - i.e. it is likely to lead to a solution quickly
 - Evaluate nodes using just the heuristic function:
 - $f(n) = h(n)$
- Example: Holiday in Romania
 - Heuristic (h): straight-line distance
 - $h(n)$ = straight line distance from n to

Romania with step costs in km



Visualization of Greedy Search



Issues with Greedy Search

- False starts
 - Because of minimizing $h(n)$
 - E.g. Consider Iași → Făgărași
 - Neamț = to be expanded first; → dead end
- Repeated states
 - If they are not avoided → the solution will never be found (infinite loops)
 - E.g. same as above:
 - Iași → Neamț → Iași → Neamț → ...
- Resembles DFS (follow one path to the goal)
- Same defects as DFS:
 - Not optimal, incomplete
 - Complete **if** unique states and finite search space

Cont.

- One way to counteract this false start (blind-alley) effect is to turn off the heuristics until a proportion of the search space has been covered, so that the truly high scoring states can be identified.

A*Search_

- A* search combines the best parts of **uniform cost search**, namely the fact that it's optimal and complete, and the best parts of **greedy search**, namely its speed.

A* Search

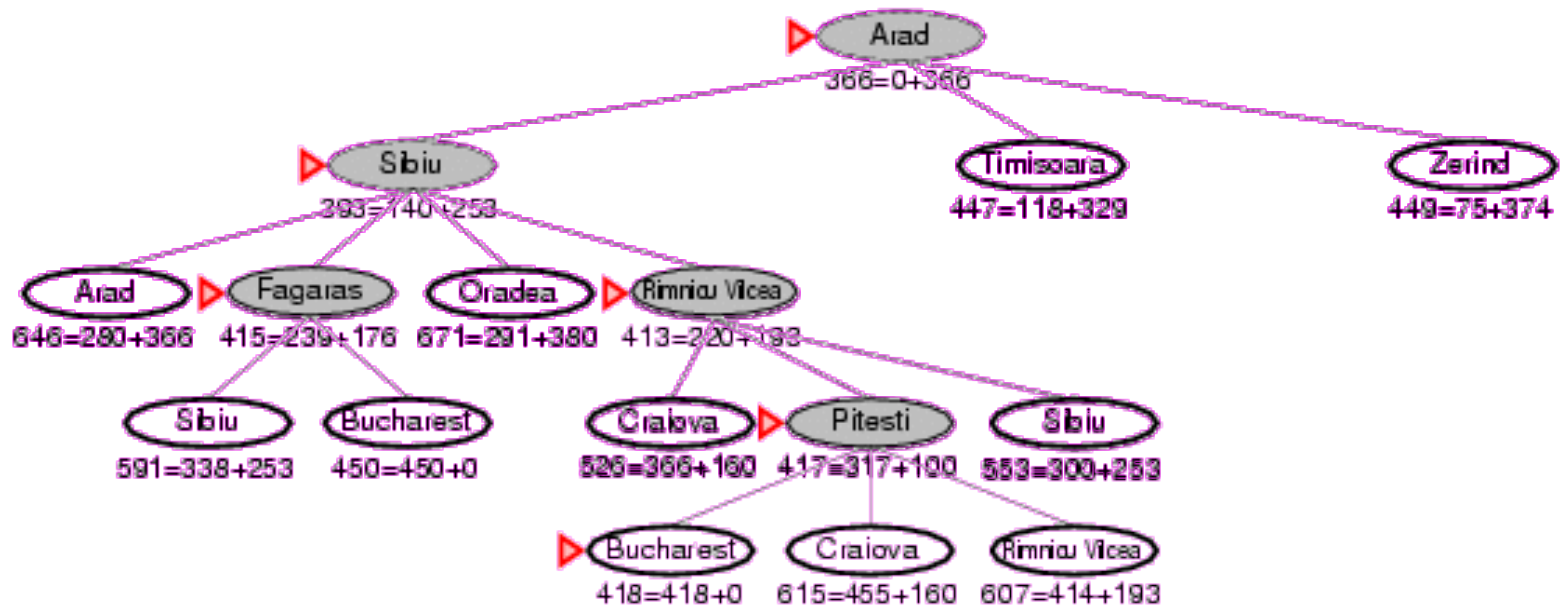
- Goal:
 - Minimize the **total** estimated solution cost
- Evaluation function:
 - $f(n) = g(n) + h(n)$
 - $g(n)$ = the cost to reach ***n***
 - $h(n)$ = the estimated cost from ***n*** to the goal node
 - $f(n)$ = estimated cost of the cheapest solution **through *n***
- Minimize $g(n) + h(n)$
- A* = optimal & complete

Admissible Heuristic

- A heuristic $h(n)$ is admissible if:
 - $h(n)$ **never overestimates** the cost to reach the goal, i.e. it is **optimistic**
- A heuristic $h(n)$ is **admissible** if for every node n ,
 $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n .
- Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G
 - $h_{SLD}(n)$ never overestimates the actual road

Visualization of A* Search

Slide 78



Recap....

- *In which we see how an agent can find a sequence of actions that achieves its goals, when no single action will do.*
 - *Stuart Russell and Peter Norvig*
- *Focus is on problem solving agents – decide what to do by finding sequences of actions that lead to desirable states.*
- *The process of looking for such a sequence is called Search.*

Overview of AI Work

- Most AI work can be explained as follows;
 - Specifying a problem
 - Defining a search space with the solution
 - Selecting a search strategy
 - Choosing an agent to implement the strategy.

Definition of a Problem in AI

- A problem is a specific task where the agent starts with the environment in a given **state** and acts upon the environment until the altered state has some pre-determined quality
- Class definition?

Types of Problems

- There are essentially 4 types of problems;
 - **Single state problems** – the action sequence is a 1 step transition.

→
[a1(initial state) to a2(goal state)]

Cont. of problem types

□ **Multiple State Problems**

From the initial state, there are a number of action sequences.

Consists of many intermediate states before the goal state.

Cont.

□ Contingency Problems

Many of the problems in the real world are contingency problems because making an exact prediction is impossible. **The environment is partially observable or the actions are uncertain.**

Example;

- Keep your eyes open while walking
- Beware! Have your senses awake

Cont.

□ Exploration Problem

The agent has no information about the effects of its actions e.g. ***an intelligent agent*** in a strange city must gradually discover what its actions do and what states exist.

Important Terms

- States
 - “Places” where the search *can* visit
- Search space
 - The set of possible states
- Search path
 - The states which the search agent *actually* visits
- Solution
 - A state with a particular property that solves the problem (achieves the task) at hand
 - There may exist more than one solution to a problem
- Strategy
 - How to choose the next state in the path at any given state

Search Problem considerations

- Consider the following when specifying a search problem;
 1. Initial state
 - Inform the agent where the search begins e.g. a city
 2. Operator set
 - Description of the possible actions available to the agent e.g. functions that change one state to another
 - Specify how the agent can move around search space
 - Final search strategy boils down to choosing states & operators

Cont.

3. Goal test

- Describes how the agent knows if the solution (goal) has been achieved.

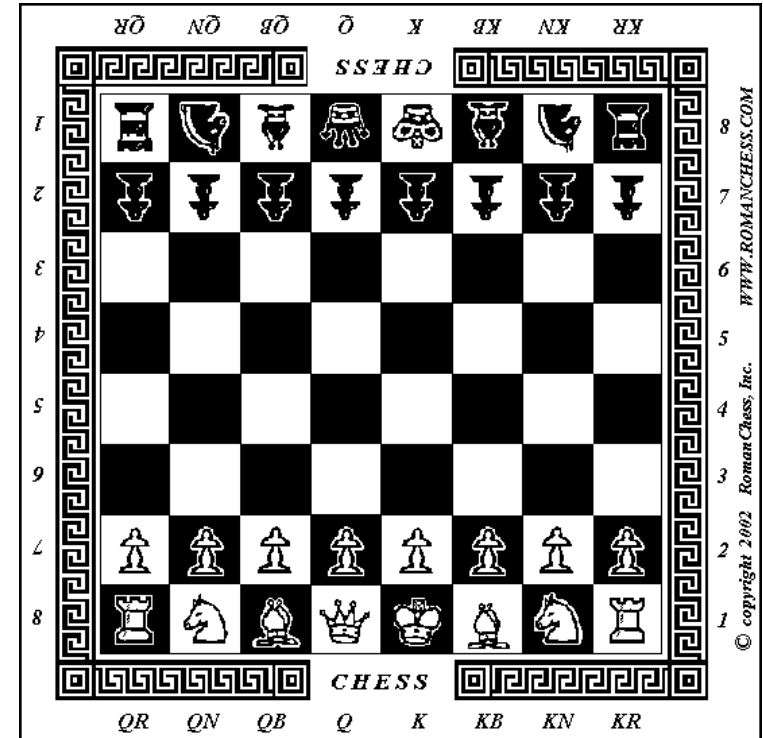
4. Path Cost

- This is a function that assigns a numeric cost to each path. The problem solving agent chooses a cost function that reflects its own performance measure i.e. time, distance etc.

Formulating Problems:

Example 1 - Chess

- Initial state
 - As in picture
- Operators
 - Moving pieces
- Goal test
 - Checkmate
 - king cannot move without being taken



Example 2 – Route Planning

- Initial state
 - City the journey starts in
- Operators
 - Driving from city to city
- Goal test
 - Destination city

Properties of Search Algorithms

- Consider the following when selecting an algorithm;
 - **Completeness** - Is a solution guaranteed to be found if at least one solution exists?
 - **Optimality** - Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?

Cont. of Search Properties

- **Time Complexity** - The upper bound on the time required to find a solution, as a function of the complexity of the problem.
 - How long does it take to find the solution?
- **Space Complexity** - The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.
 - what are the memory requirements?

TREES

- These are state graphs that have no cycles in them.
- Many AI searches can be represented as Trees.
- The **root** of the tree represents the search node corresponding to the initial state.
- The **Leaf nodes** correspond to all the other possible states in the problem.

Reading Assignment



- Read on the IDA* search
- MA* search (memory bounded A*)
- SMA* (Simplified Memory-Bounded A*) search
- **Source:**

Artificial Intelligence: A Modern Approach,
Russell SJ & Norvig P, Prentice-Hall Inc.
1995.

See you next
session!

