SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30019: Introduction to Artificial Intelligence

Problem solving and Search

1

Previously on COS30019 …

- 4 paradigms of AI:
  - Systems that think/act like a human/rationally
- Intelligent agents are systems that act rationally
  - chooses whichever action that maximizes the expected value of the performance measure given the percept sequence to date and prior environment knowledge
- 4 basic types of agent & 4 (basic type + **learning**) agents
  - Simple reflex
  - State-based reflex
  - Goal-based agent
  - Utility-based agent

2

Outline

- Problem-solving agents
  - A kind of goal-based agent
- Problem types
  - Single state (fully observable)
  - Search with partial information
- Problem formulation
  - Example problems
- Basic search algorithms
  - Uninformed

3

Example: 8-puzzle

7	2	4
5		6
8	3	1

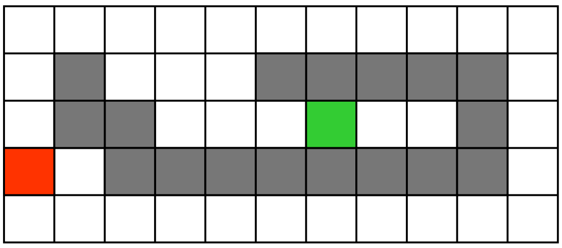
Start State

	1	2
3	4	5
6	7	8

Goal State

4

Example: Robot Navigation

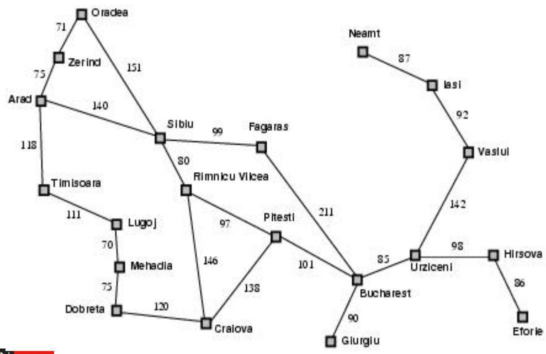


5

Example: Route finding

6

Example: Romania



7

Example: Romania

- On holiday in Romania; currently in Arad
  - Flight leaves tomorrow from Bucharest
- Formulate goal
  - Be in Bucharest
- Formulate problem
  - States: various cities
  - Actions: drive between cities
- Find solution
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, …

8

Problem-solving agent

- Four general steps in problem solving:
  - Goal formulation
    - What are the successful world states
  - Problem formulation
    - What actions and states to consider given the goal
  - Search
    - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
  - Execute
    - Given the solution, perform the actions.

9

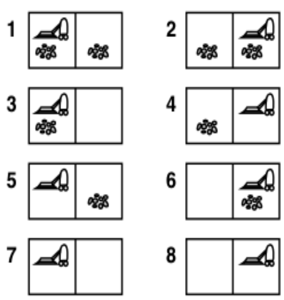
Problem types

- Deterministic, fully observable ⇒ *single state problem*
  - Agent knows exactly which state it will be in; solution is a sequence.
- Partial knowledge of states and actions:
  - Non-observable ⇒ *sensorless or conformant problem*
    - Agent may have no idea where it is; solution (if any) is a sequence.
  - Nondeterministic and/or partially observable ⇒ *contingency problem*
    - Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
  - Unknown state space ⇒ *exploration problem* ("online")
    - When states and actions of the environment are unknown.

10

Example: vacuum world

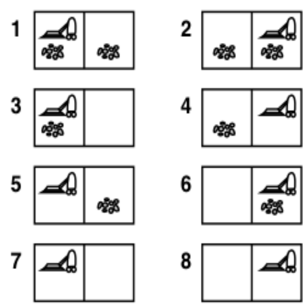
- Single state, start in #5.
  Solution??



11

Example: vacuum world

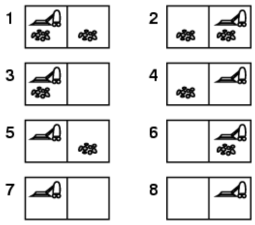
- Single state, start in #5.
  Solution??
  - *[Right, Suck]*



12

Example: vacuum world

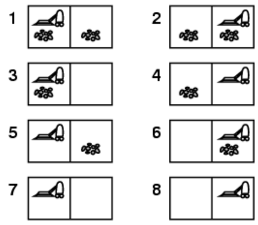
- Single-state, start in #5.
  Solution? *[Right, Suck]*
-
- Sensorless, start in
  {1,2,3,4,5,6,7,8} e.g.,
  Right goes to *{2,4,6,8}*
  Solution?
-



13

Example: vacuum world

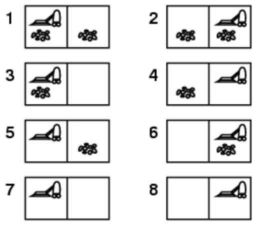
- Sensorless, start in
  {1,2,3,4,5,6,7,8} e.g.,
  Right goes to *{2,4,6,8}*
  Solution?
  [Right,Suck,Left,Suck]
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[A, Clean]*, i.e., start in #5 or #7
    Solution?



14

Example: vacuum world

- Sensorless, start in
  {1,2,3,4,5,6,7,8} e.g.,
  Right goes to *{2,4,6,8}*
  Solution?
  [Right,Suck,Left,Suck]
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[A, Clean]*, i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*



15

Single-state problem formulation

A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = *{<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

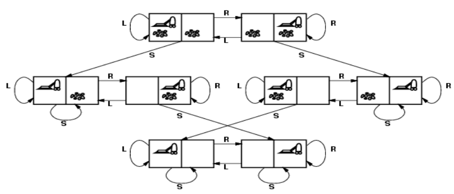
16

Selecting a state space

- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

17

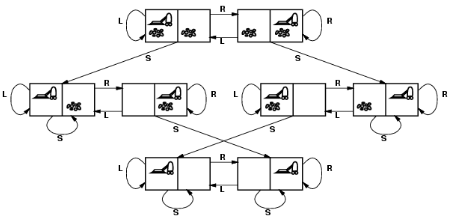
Vacuum world state space graph



- states?
- actions?
- goal test?
- path cost?
-

18

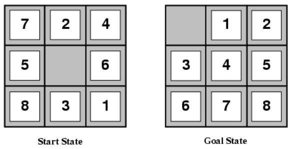
Vacuum world state space graph



- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

19

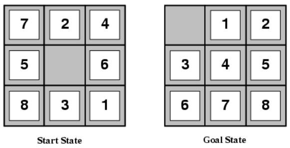
Example: The 8-puzzle



- states?
- actions?
- goal test?
- path cost?

20

Example: The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

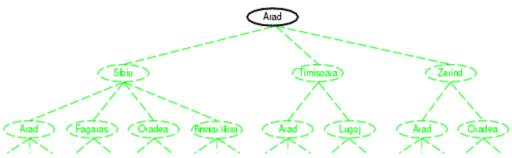
21

Basic search algorithms

- How do we find the solutions of previous problems?
  - Search the state space (remember complexity of space depends on state representation)
  - Here: search through *explicit tree generation*
    - ROOT= initial state.
    - Nodes and leafs generated through successor function.
  - In general search generates a graph (same state through multiple paths)

22

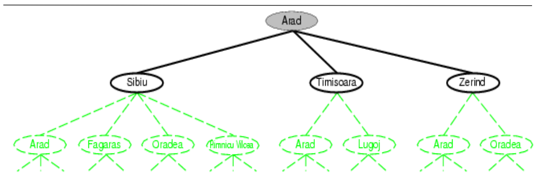
Simple tree search example



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure
Initialize search tree to the *initial state* of the *problem*
do
if no candidates for expansion **then return** *failure*
choose leaf node for expansion according to *strategy*
if node contains goal state **then return** *solution*
else expand the node and add resulting nodes to the search tree
enddo

23

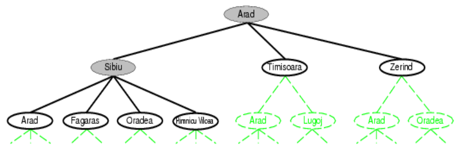
Simple tree search example



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure
Initialize search tree to the *initial state* of the *problem*
do
if no candidates for expansion **then return** *failure*
choose leaf node for expansion according to *strategy*
if node contains goal state **then return** *solution*
else expand the node and add resulting nodes to the search tree
enddo

24

Simple tree search example



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

choose leaf node for expansion according to *strategy* ←**Determines search process!!**

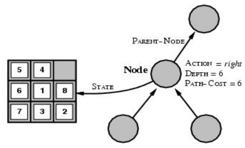
if node contains goal state **then return** *solution*

else expand the node and add resulting nodes to the search tree

enddo

25

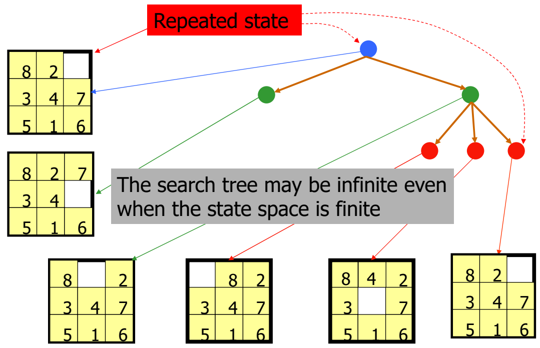
State space vs. search tree



- A *state* is a (representation of) a physical configuration
- A *node* is a data structure belong to a search tree
  - A node has a parent, children, … and includes path cost, depth, …
  - Here *node*= *<state, parent-node, action, path-cost, depth>*
  - *FRONTIER*= contains generated nodes which are not yet expanded.
    - White nodes with black outline

26

Search Nodes ≠ States



27

Tree search algorithm

function TREE-SEARCH(*problem, frontier*) **return** a solution or failure

frontier ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *frontier*)

loop do

if EMPTY?(*frontier*) **then return** failure

node ← REMOVE-FIRST(*frontier*)

if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds

then return SOLUTION(*node*)

frontier ← INSERT-ALL(EXPAND(*node*, *problem*), *frontier*)

28

Tree search algorithm (2)

function EXPAND(*node,problem*) **return** a set of nodes

successors ← the empty set

for each <*action*, *result*> **in** SUCCESSOR-FN**[***problem***]**(STATE[*node*]) **do**

s ← a new NODE

STATE[*s*] ← *result*

PARENT-NODE[*s*] ← *node*

ACTION[*s*] ← *action*

PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node*, *action*,*s*)

DEPTH[*s*] ← DEPTH[*node*]+1

add *s* to *successors*

return *successors*

29

Search strategies

- A strategy is defined by picking the order of node expansion.
- Problem-solving performance is measured in four ways:
  - Completeness; *Does it always find a solution if one exists?*
  - Optimality; *Does it always find the least-cost solution?*
  - Time Complexity; *Number of nodes generated/expanded?*
  - Space Complexity; *Number of nodes stored in memory during search?*
- Time and space complexity are measured in terms of problem difficulty defined by:
  - *b - maximum branching factor of the search tree*
  - *d - depth of the least-cost solution*
  - *m - maximum depth of the state space (may be ∞)*

30

Blind vs. Heuristic Strategies

- Blind (or uninformed) strategies do not exploit any of the information contained in a state
- Heuristic (or informed) strategies exploits such information to assess that one node is "more promising" than another

31

Uninformed search strategies

- Categories defined by expansion algorithm:
  - ☑ Breadth-first search
  - ☑ Uniform-cost search
  - ☑ Depth-first search
  - ☐ Depth-limited search
  - ☐ Iterative deepening search.
  - ☐ Bidirectional search

32

Uninformed search strategies

- Breadth-first
  - ☐ Bidirectional
- Depth-first
  - ☐ Depth-limited
  - ☐ Iterative deepening
- Uniform-Cost

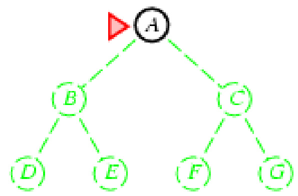
Step cost = 1

Step cost = c(action) $\geq \varepsilon > 0$

33

Breadth-first search

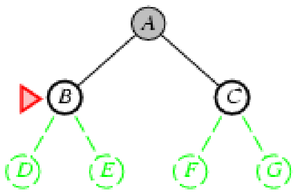
- Expand shallowest unexpanded node
- Implementation:
  - ☐ *frontier* is a FIFO queue, i.e., new successors go at end
  - ☐



34

Breadth-first search

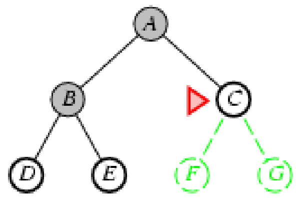
- Expand shallowest unexpanded node
- Implementation:
  - ☐ *frontier* is a FIFO queue, i.e., new successors go at end
  - ☐



35

Breadth-first search

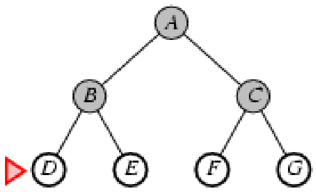
- Expand shallowest unexpanded node
- Implementation:
  - ☐ *frontier* is a FIFO queue, i.e., new successors go at end
  - ☐



36

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end
  -



37

Properties of breadth-first search

- Complete? Yes (if *b* is finite)
- Optimal? Yes (if cost = 1 per step)
-
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Space is the bigger problem (more than time)
-

38

Time and Memory Requirements

d	#Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim10^6$	1 sec	100 Mb
8	$\sim10^8$	100 sec	10 Gbytes
10	$\sim10^{10}$	2.8 hours	1 Tbyte
12	$\sim10^{12}$	11.6 days	100 Tbytes
14	$\sim10^{14}$	3.2 years	10,000 Tb

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

39

Time and Memory Requirements

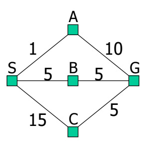
d	#Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim10^6$	1 sec	100 Mb
8	$\sim10^8$	100 sec	10 Gbytes
10	$\sim10^{10}$	2.8 hours	1 Tbyte
12	$\sim10^{12}$	11.6 days	100 Tbytes
14	$\sim10^{14}$	3.2 years	10,000 Tb

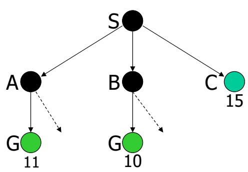
Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

40

Uniform-Cost Strategy

- Each step has some cost $\geq \varepsilon > 0$.
- The cost of the path to each frontier node N is
  g(N) = Σ costs of all steps.
- The goal is to generate a solution path of minimal cost.
- The queue FRONTIER is sorted in increasing cost.

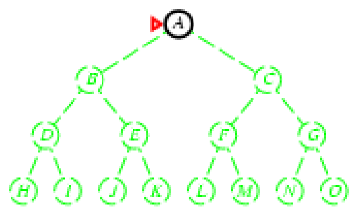




41

Depth-first search

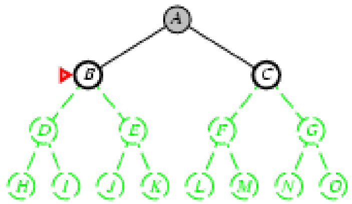
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



42

Depth-first search

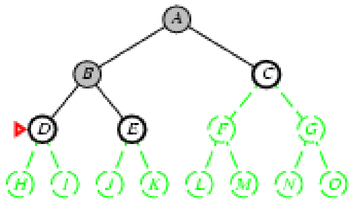
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



43

Depth-first search

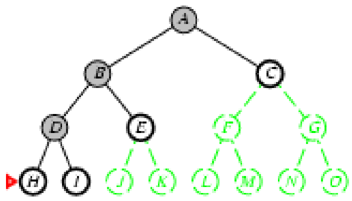
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



44

Depth-first search

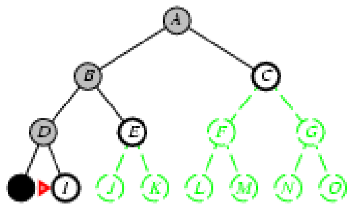
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



45

Depth-first search

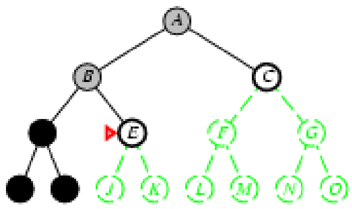
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



46

Depth-first search

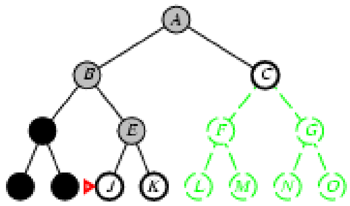
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



47

Depth-first search

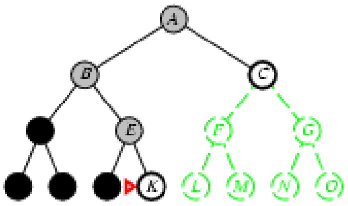
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



48

Depth-first search

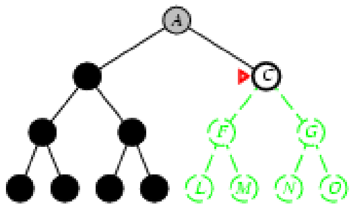
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



49

Depth-first search

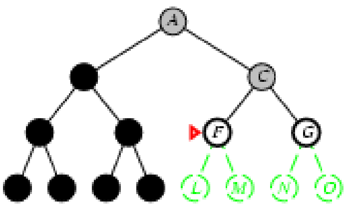
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



50

Depth-first search

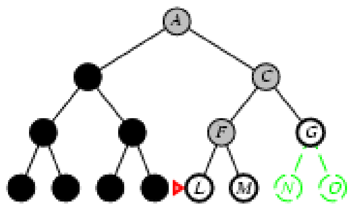
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



51

Depth-first search

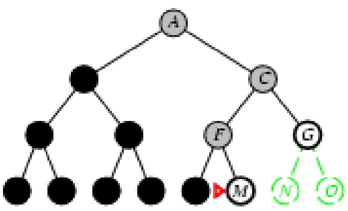
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



52

Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front
  -



53

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → complete in finite spaces
- Optimal? No
-
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!

54

Summary

- Search tree ≠ state space
- Search strategies: breadth-first, depth-first, and variants
- Evaluation of strategies: completeness, optimality, time and space complexity
- Avoiding repeated states
- Optimal search with variable step costs

55