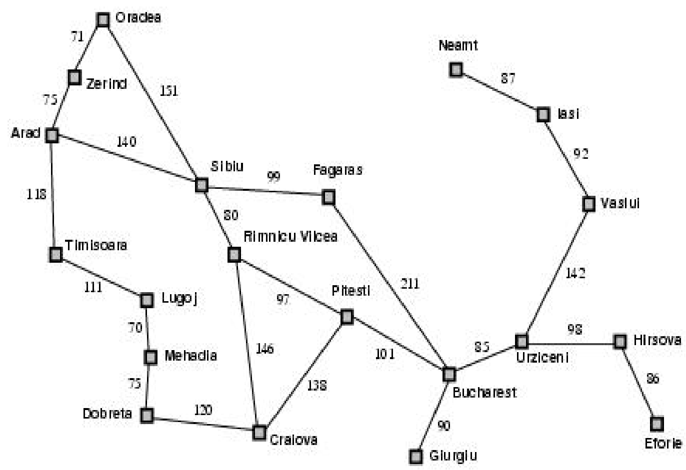
Problem Solving by Searching

Outline

- Example problems
- Problem formulation
  - Example problems
- Searching for solutions
  - Tree search
  - Graph search
- Basic search algorithms
  - Uninformed

2

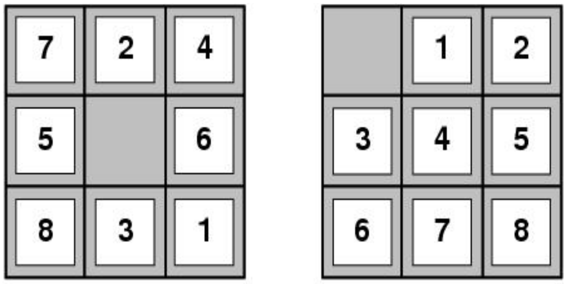
Example: Getting around in Romania



Example: Getting around in Romania

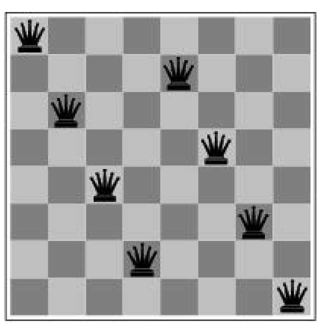
- On holiday in Romania; currently in Arad
  - Flight leaves tomorrow from Bucharest
- Formulate goal
  - Be in Bucharest
- Formulate problem
  - States: various cities
  - Actions: drive between cities
- Find solution
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, …

Example: 8-puzzle



Start State Goal State

Example: 8-queens problem



- Constraint satisfaction problems (CSP)

Example Problems

- Toy problems
  - 8-puzzle
  - 8-queens
  - Rubik's cube
- Real-world problems
  - Route-finding problem: air travel planning, military operations planning, routing in computer networks
  - Touring problem, traveling salesperson problem
  - VLSI layout
  - Internet searching
  - Class scheduling

7

Problem types

- Goal-based agents
- Assumption on environment:
  - Observable: the agent knows current state
  - Static
  - Discrete: finitely many actions to choose
  - Deterministic
    - Agent knows the effects of its actions
- solution is a sequence of actions

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
    - Design a representation that captures relevant aspects of the world
  - Search
    - Determine the possible sequence of actions that lead to the goal states and then choose the best sequence.
  - Execute
    - Given the solution perform the actions "eyes closed"

9

Problem formulation

- A search problem is defined by five components:
  - An initial state, e.g. "at *Arad*"
  - Actions: ACTIONS(*s*) applicable actions in *s*
  - Transition model (or Successor function, or Operators)

  RESULT(s,a): state that results from action a in state s

  initial state + actions + transition model define state space, a directed graph in which the nodes are states and the arcs between nodes are actions
  - Goal test, can be
    - Explicit set of possible goal states, e.g. *x='at bucharest'*
    - Implicit, e.g. *checkmate(x)*, *NoDirt(x)*
  - Path cost (additive): assigns a numeric cost to each path
    - e.g. sum of distances, number of actions executed, …
    - *c(s,a,s')* is the step cost, assumed to be >= 0

  A solution is a sequence of actions from initial to goal state.
  Optimal solution has the lowest path cost.

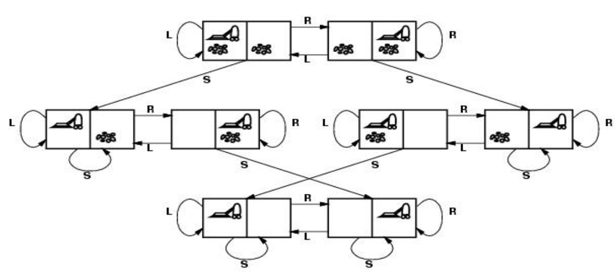
10

Selecting a state space

- Real world is absurdly complex.
  - State space must be *abstracted* for problem solving.
- Good representations
  - preserve the relevant aspects of the problem
  - expose the relevant problem structure
  - abstracts away unimportant details

11

Example: vacuum world

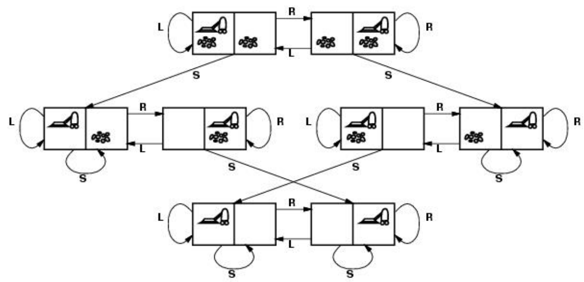


Clean both squares with minimum number of actions

- States??
- Initial state??
- Actions?? Transition model??
- Goal test??
- Path cost??

12

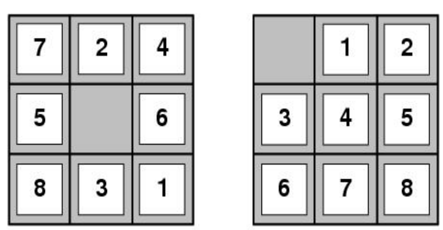
Example: vacuum world



- States?? two locations, with or without dirt: 2 x 2^2=8 states.
- Initial state?? Any state can be initial
- Actions?? {*Left*, *Right*, *Suck*}
- Goal test?? Check whether squares are clean.
- Path cost?? 1 per action

13

Example: 8-puzzle

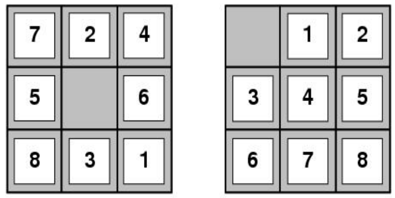


Start State Goal State

- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??

14

Example: 8-puzzle



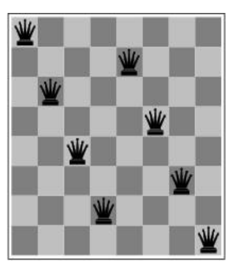
Start State Goal State

- States?? Integer location of each tile, 10^5 states
- Initial state?? Any state can be initial
- Actions?? {*Left*, *Right*, *Up*, *Down*}
- Goal test?? Check whether goal configuration is reached
- Path cost?? 1 per move

[Note: optimal solution of n-Puzzle family is NP-complete]

15

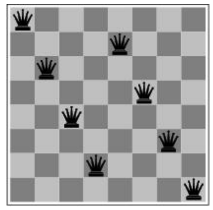
Example: 8-queens problem



- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??

16

Example: 8-queens problem

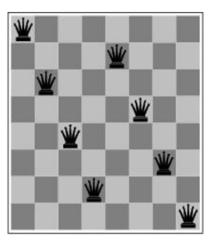


Incremental formulation

- States?? Any arrangement of 0 to 8 queens on the board
- Initial state?? No queens
- Actions?? Add queen in empty square
- Goal test?? 8 queens on board and none attacked
- Path cost?? None

  10^{14} possible sequences to investigate

Example: 8-queens problem

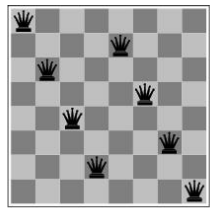


Incremental formulation (alternative)

- States?? *n* (0≤ *n*≤ 8) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
- Actions?? Add queen in leftmost empty column such that it is not attacking other queens

  2057 possible sequences to investigate; 10^52 for 100-queens

Example: 8-queens problem



complete-state formulation

- States?? 8 queens on the board, one per column
- Initial state?? any
- Actions?? Move a queen to a square in the same column
- Goal test?? 8 queens on board and none attacked
- Path cost?? None

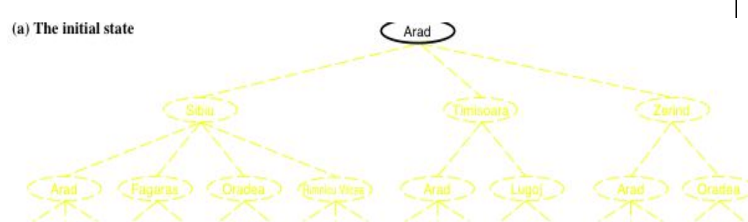
Tree search algorithms

- How do we find the solutions of previous problems?
  - Search the state space (exhaustive search)
  - Generate a **search tree**
    - ROOT= initial state.
    - Branches are actions; Nodes and leafs generated through transition model
    - **Expanding** the current state: applying each legal action to the current state **generating** a new set of states

Tree search algorithms

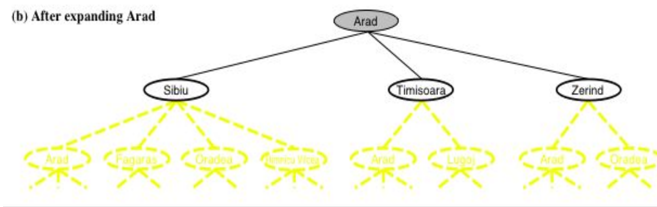
function TREE-SEARCH(*problem*, *strategy*) **return** a solution or failure
Initialize search tree to the *initial state* of the *problem*
loop do
if no candidates for expansion **then return** *failure*
choose leaf node for expansion according to *strategy*
if node contains goal state **then return**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

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

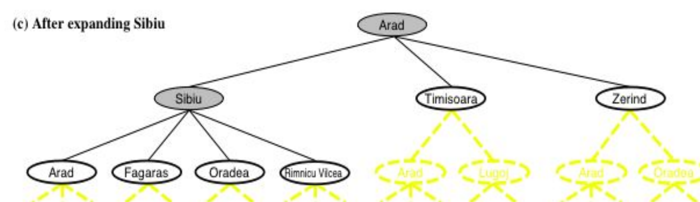
Simple tree search example



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

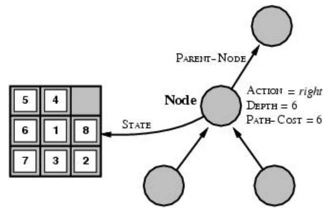
Simple tree search example



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* ←**Determines search process!!**
    if node contains goal state **then return** *solution*
    else expand the node and add resulting nodes to the search tree
  enddo

24

Implementation: states vs. nodes



- A *state* is a (representation of) a physical configuration
- A *node* is a bookkeeping data structure used to represent the search tree
  - *node= <state, parent-node, action, path-cost, …>*
- ***Fringe***= contains generated nodes which are not yet expanded=leaf nodes (aka, ***Frontier***, ***open list***)
  - Implemented as a queue

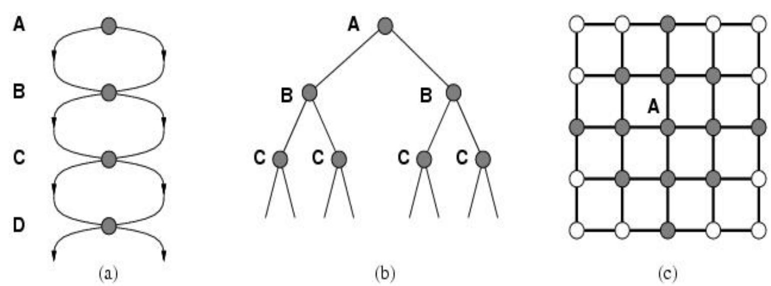
Tree search algorithm

function TREE-SEARCH(*problem*) **return** a solution or failure
    fringe ← INSERT(MAKE-NODE(problem.INITIAL-STATE))
    loop do
        if EMPTY?(*fringe*) **then return** failure
        node ← POP(*fringe*)
        if problem.GOAL-TEST(node.STATE)
            then return SOLUTION(*node*)
        for each action **in** problem.ACTIONS(node.STATE) **do**
            child ← Child-Node(problem, node, action)
            fringe ← INSERT(child, *fringe*)

Fringe implemented as a queue: FIFO, LIFO, priority queue

Repeated states

- Failure to detect repeated states can turn a solvable problems into unsolvable ones.



27

Graph search algorithm

- **Closed list** (or **explored set**) stores all expanded nodes

function GRAPH-SEARCH(*problem*) **return** a solution or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(problem.INITIAL-STATE))
  loop do
    if EMPTY?(*fringe*) **then return** failure
    node ← POP(*fringe*)
    if problem.GOAL-TEST(node.STATE)
      then return SOLUTION(*node*)
    add node.STATE to *closed*
    for each action **in** problem.ACTIONS(node.STATE) **do**
      child ← Child-Node(problem, node, action)
      if child.STATE is not in closed or fringe **then**
        fringe ← INSERT(child, *fringe*)

28

Uninformed search strategies

- (a.k.a. blind search) = use only information available in problem definition.
  - When strategies know whether one non-goal state is better than another → informed search.
- A strategy is defined by picking the **order of node expansion**:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search.
  - Bidirectional search

29

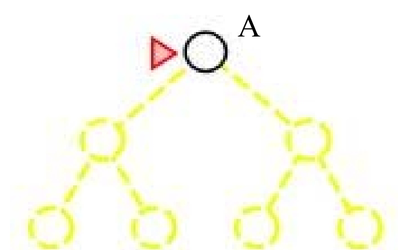
Performance Measure

- Algorithm 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?*
  - **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 shallowest solution*
  - *m - maximum length of the state space (may be* ∞*)*

30

BF-search, an example

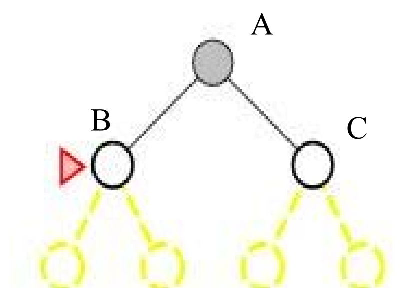
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



31

BF-search, an example

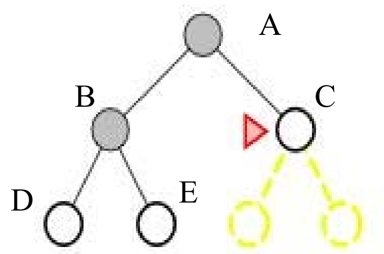
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



32

BF-search, an example

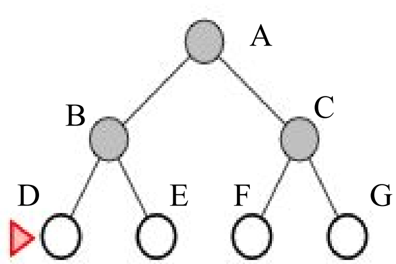
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



33

BF-search, an example

- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



34

BF Graph search algorithm

```
function BF-SEARCH(problem) return a solution or failure
   closed ← an empty set
   fringe ← a FIFO queue with MAKE-NODE(problem.INITIAL-STATE)
   loop do
       if EMPTY?(fringe) then return failure
       node ← POP(fringe)
       add node.STATE to closed
       for each action in problem.ACTIONS(node.STATE) do
              child ←  Child-Node(problem, node, action)
              if child.STATE is not in closed or fringe then
                     if problem.GOAL-TEST(child.STATE)
                            then return SOLUTION(child)
                     fringe ← INSERT(child, fringe)
```

35

BF-search; evaluation

- Completeness:
  - *Does it always find a solution if one exists?*

36

BF-search; evaluation

- Completeness:
  - *Does it always find a solution if one exists?*
  - YES
    - Condition: If *b* is finite
      - (maximum number Of successor nodes is finite)

37

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Assume a state space where every state has *b* successors.
  - Assume solution is at depth *d*

38

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity (worst case tree search):
  - Assume a state space where every state has *b* successors.
    - root has *b* successors, each node at the next level has again b successors (total b^2), …
  - Assume solution is at depth *d*
    - Total number of nodes generated in the worst case:

$$b + b^2 + b^3 + \ldots + b^d = O(b^d)$$

39

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated:

$$b + b^2 + b^3 + \ldots + b^d = O(b^d)$$

  - Graph search: more efficient, worst case proportional to the size of the state space (may be much smaller than $O(b^d)$).

- Space complexity?

40

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated:
    $$b + b^2 + b^3 + \ldots + b^d = O(b^d)$$
  - Graph search: proportional to the size of the state space (may be much smaller than $O(b^d)$).
- Space complexity: $O(b^d)$
  - Same as Time as each node is retained in memory
  - Tree search not save much space, use more time

41

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated: $O(b^d)$
  - Graph search: proportional to the size of the state space (may be much smaller than $O(b^d)$).
- Space complexity: $O(b^d)$
  - Same as Time as each node is retained in memory
- Optimality?
  - *Does it always find the least-cost solution?*

42

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated: $O(b^d)$
  - Graph search: more efficient, proportional to the size of the state space
- Space complexity: $O(b^d)$
  - Same as Time as each node is retained in memory
- Optimality?
  - *Does it always find the least-cost solution?*
  - Yes if unit cost per step.
  - No in general: actions have different cost.

43

BF-search; evaluation

- Two lessons:
  - Memory requirements are a bigger problem than its execution time.
  - Time: Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH	NODES	TIME	MEMORY
2	110	0.11 milliseconds	107 kilobyte
4	11110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabytes
8	10^8	2 minutes	103 gigabyte
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabytes
14	10^{14}	3.5 years	99 petabytes

b=10; 1 million nodes/second, 1000 bytes/node

44

Uniform-cost search

- Extension of BF-search:
  - Expand node with *lowest path cost g(n)*
- Implementation: *fringe* = priority queue ordered by path cost.
- UC-search is similar to BF-search when all step-costs are equal, but more costly

45

Uniform-cost Graph search

```
function Uniform-Cost-SEARCH(problem) return a solution or failure
   closed ← an empty set
   fringe ← a priority queue with MAKE-NODE(problem.INITIAL-STATE)
   loop do
        if EMPTY?(fringe) then return failure
        node ← POP(fringe)
        if problem.GOAL-TEST(node.STATE)
                then return SOLUTION(node)
        add node.STATE to closed
        for each action in problem.ACTIONS(node.STATE) do
                child ←  Child-Node(problem, node, action)
                if child.STATE is not in closed or fringe then
                        fringe ← INSERT(child, fringe)
                else if child.STATE is in fringe with higher path cost then
                        replace that fringe node with child
```

46

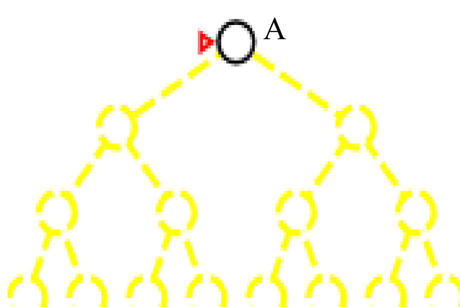
Uniform-cost search

- Completeness:
  - YES, if step-cost ≥ ε (small positive constant)
- Time complexity:
  - Expand nodes with cost less than C*, the cost of the optimal solution.
  - Worst-case tree-search: $O(b^{1+C^*/\varepsilon})$
- Space complexity:
  - Same as time complexity
- Optimality:
  - nodes expanded in order of increasing path cost.
  - YES, if complete.

47

DF-search, an example

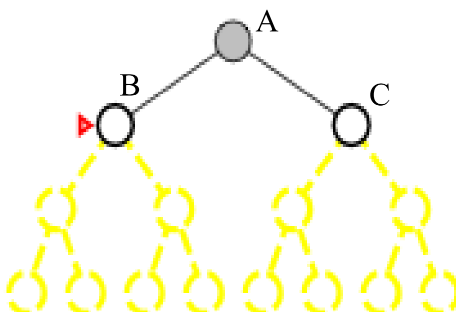
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



48

DF-search, an example

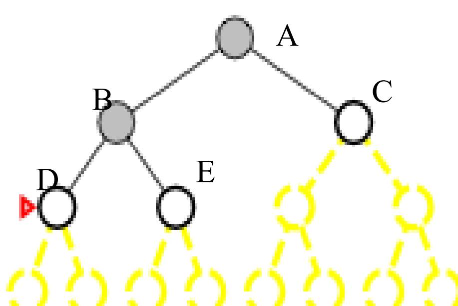
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



49

DF-search, an example

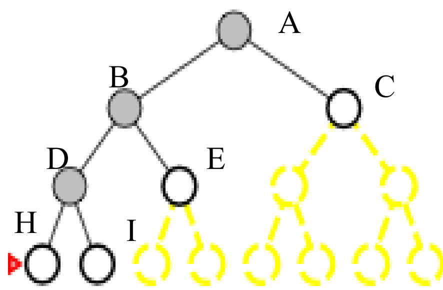
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



50

DF-search, an example

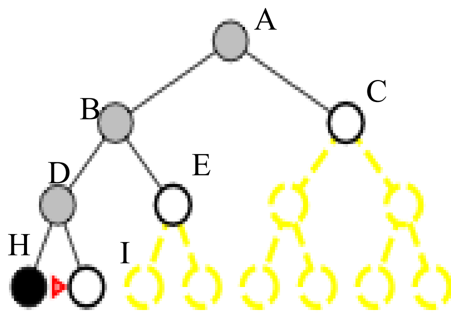
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



51

DF-search, an example

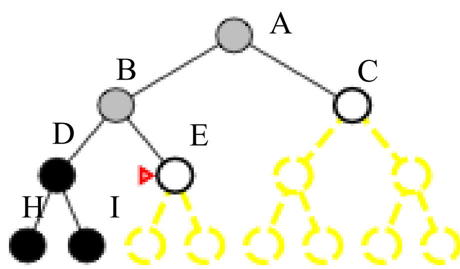
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



52

DF-search, an example

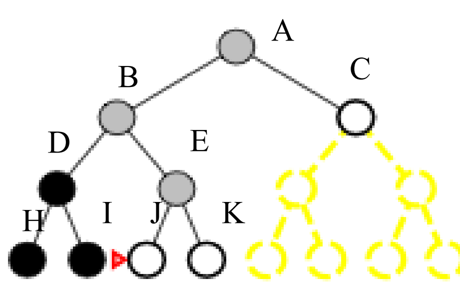
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



53

DF-search, an example

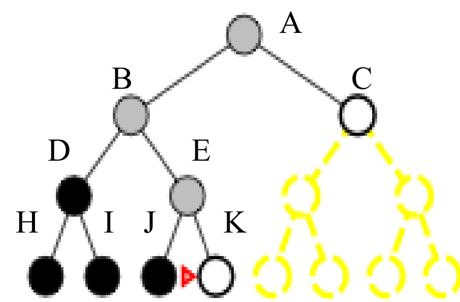
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



54

DF-search, an example

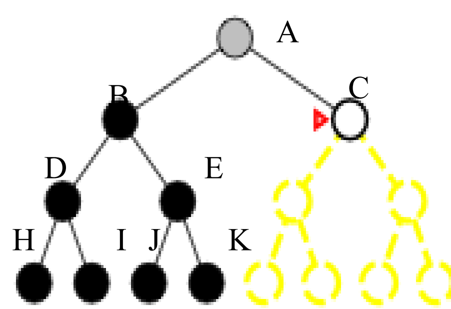
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



55

DF-search, an example

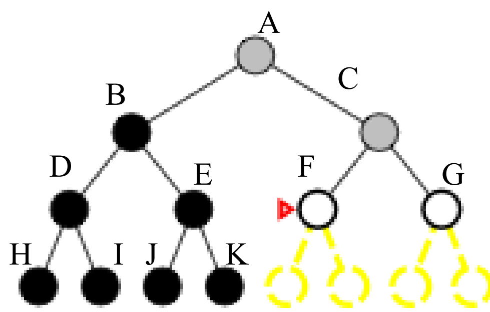
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



56

DF-search, an example

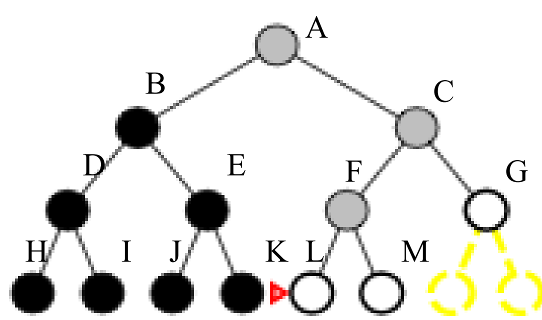
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



57

DF-search, an example

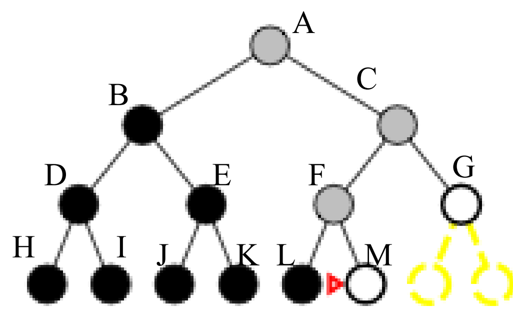
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



58

DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



59

DF-search; evaluation

- Completeness
  - *Does it always find a solution if one exists?*

60

DF-search; evaluation

- Completeness
  - *Does it always find a solution if one exists?*

  - NO: fails in infinite state spaces
    - *G*raph search complete in finite state spaces
    - Tree search complete in finite depth trees (can be modified to avoid infinite loops)

61

DF-search; evaluation

- Completeness;
  - NO unless finite state space / depth
- Time complexity?
  - Assume maximum depth *m*

62

DF-search; evaluation

- Completeness;
  - NO unless finite state space / depth.
- Time complexity;
  - Tree search: worst case $O(b^m)$
  - Terrible if *m* is much larger than *d* (depth of optimal solution)
  - But if many solutions, may be much faster than BF-search

63

DF-search; evaluation

- Completeness;
  - NO unless finite state space / depth.
- Time complexity; $O(b^m)$
- Space complexity?

64

DF-search; evaluation

- Completeness;
  - NO unless finite state space / depth.
- Time complexity; $O(b^m)$
- Space complexity;
  - Tree search: linear space $O(bm)$; nodes expanded with no descendants in fringe can be removed from memory
  - Graph search: as Time, size of state space

65

DF-search; evaluation

- Completeness;
  - NO unless finite state space / depth.
- Time complexity; $O(b^m)$
- Space complexity;
  - Tree search: linear space $O(bm)$
- Optimality?

66

DF-search; evaluation

- Completeness;
  - NO unless finite state space / depth.
- Time complexity; $O(b^m)$
- Space complexity;
  - Tree search: linear space $O(bm)$
- Optimality; No

67

Depth-limited search

- DF-search with predetermined depth limit *l*.
  - i.e. nodes at depth *l* have no successors.
  - Problem knowledge can be used
- Solves the infinite-path problem.
- If $l < d$ then incompleteness results.
- not optimal.
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

68

Depth-limited algorithm

function DEPTH-LIMITED-SEARCH(*problem*,*limit*) **return** a solution or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), *problem*,*limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **return** a solution or failure/cutoff
cutoff_occurred? ← false
if problem.GOAL-TEST(node.STATE) **then return** SOLUTION(*node*)
else if DEPTH[*node*] == *limit* then return *cutoff*
else for each *action* **in** problem.ACTIONS(*node*.STATE) **do**
child ← Child-Node(problem, node, action)
result ← RECURSIVE-DLS(*child*, *problem*, *limit*)
if *result* == *cutoff* **then** *cutoff_occurred?* ← true
else if *result* ≠ *failure* **then return** *result*
if *cutoff_occurred?* **then return** *cutoff* **else return** failure

Recursive implementation

Iterative deepening DF search

- Iterative deepening depth-first tree search
- A general strategy to find best depth limit *l*.
  - Gradually increasing the depth limit until a goal is found at depth *d*, the depth of the shallowest goal-node.
- Combines benefits of DF and BF search

Iterative deepening search

function ITERATIVE_DEEPENING_SEARCH(*problem*)
  return a solution or failure

  for *depth* ← 0 to ∞ **do**
    result ← DEPTH-LIMITED_SEARCH(*problem*, *depth*)
    if *result* ≠ *cuttoff* **then return** *result*

71

ID-search, example

- Limit=0

72

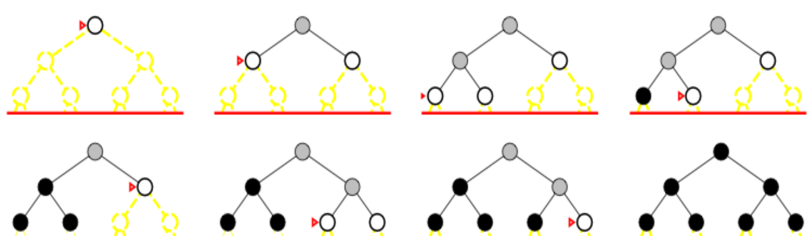
ID-search, example

- Limit=1

73

ID-search, example

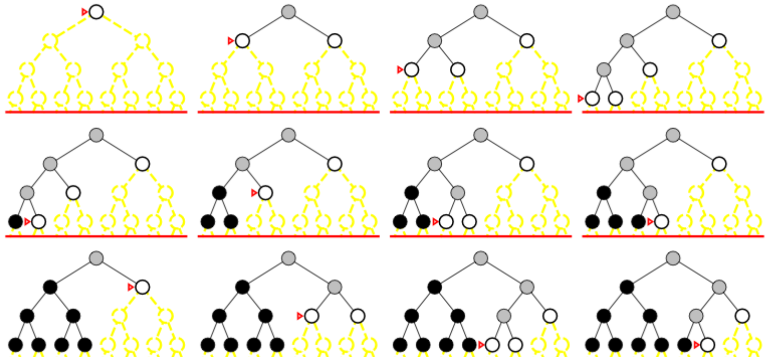
- Limit=2



74

ID-search, example

- Limit=3



75

ID search, evaluation

- Complete?

76

ID search, evaluation

- Completeness:
  - YES
- Time complexity?

ID search, evaluation

- Completeness:
  - YES
- Time complexity: $O(b^d)$
  - Algorithm seems costly due to repeated generation of certain states.
  - Node generation:
    - level d: once
    - level d-1: 2
    - level d-2: 3
    - …
    - level 2: d-1
    - level 1: d

$$N(IDS) = (d)b + (d-1)b^2 + ... + (1)b^d$$

$$N(BFS) = b + b^2 + ... + b^d$$

Num. Comparison for b=10 and d=5 solution at far right

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123{,}450$$

$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 = 111{,}110$$

ID search, evaluation

- Completeness:
  - YES
- Time complexity: $O(b^d)$
- Space complexity?

79

ID search, evaluation

- Completeness:
  - YES
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
  - depth-first tree search
- Optimal?

80

ID search, evaluation

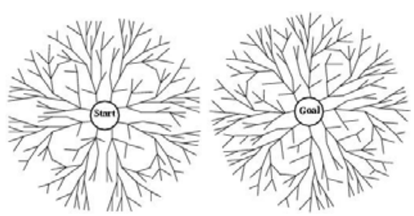
- Completeness:
  - YES
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$

- Optimality:
  - YES if step cost is 1.
  - Iterative analog to uniform-cost search?

- *ID is the preferred uninformed search method when there is a large search space and the depth of the solution is not known*

81

Bidirectional search

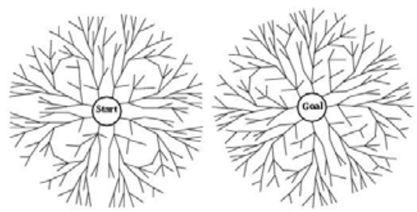


- Two simultaneous searches from start and goal.
  - Motivation: $b^{d/2} + b^{d/2} << b^d$

- Check whether the node belongs to the other fringe before expansion.
- Complete and optimal (for uniform step costs) if both searches are BF.
- Space complexity $O(b^{d/2})$ is the most significant weakness.

82

How to search backwards?



- Goal implicitly defined?
- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.

83

Summary of tree-search algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional BF search
Complete?	YES	YES	NO	NO	YES	YES
Time	b^d	$b^{C^*/e+1}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	$b^{C^*/e+1}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES	NO	NO	YES*	YES*

* Optimal if identical step costs

84