Artificial Intelligence
Uninformed search

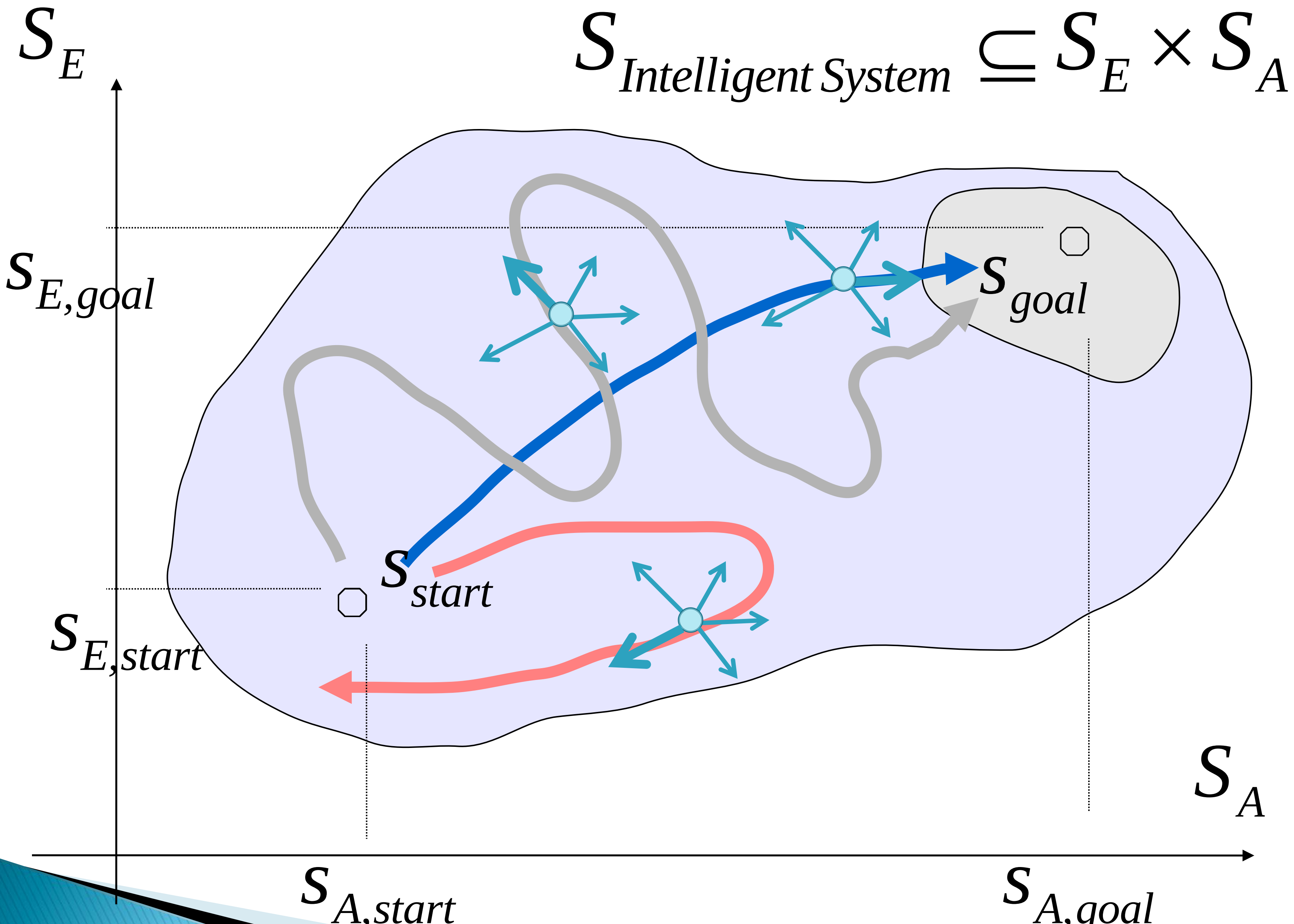
More about

Textbook, Chapter 3, Soving Problems by Searching

Outline

- Problem-solving (goal-oriented) agents
- Solving
  - Single state (fully observable)
  - Multiple state (search with partial information)

  Problem Types
- How to define a problem?
  - Example problems
- What algorithm can solve it actually?
  - Uninformed search algorithms


$S_{Intelligent\ System} \subseteq S_E \times S_A$
S_E
$S_{E,goal}$
$S_{E,start}$
S_{start}
S_{goal}
S_A
$S_{A,start}$
$S_{A,goal}$

AI as "symbol manipulation"
– expressing the goal and its direction

- The Logic Theorist, 1955 ➔ see lectures on logic
- The Dartmouth conference ("birth of AI", 1956)
- List processing (Information Processing Language, IPL)
- Means-ends analysis ("reasoning as search") ➔ see lecture on planning
- The **General Problem Solver**
- Heuristics to limit the search space ➔ see lecture on informed search
- The **Physical Symbol System Hypothesis**
  - intelligent behavior can be reduced to/emulated by symbol manipulation

    (A. Newel, H. Simon: Computer science as empirical inquiry: symbols and search, 1975)
- The unified theory of cognition (1990, cognitive architectures)

➔ expressing (human) problem solving symbolically

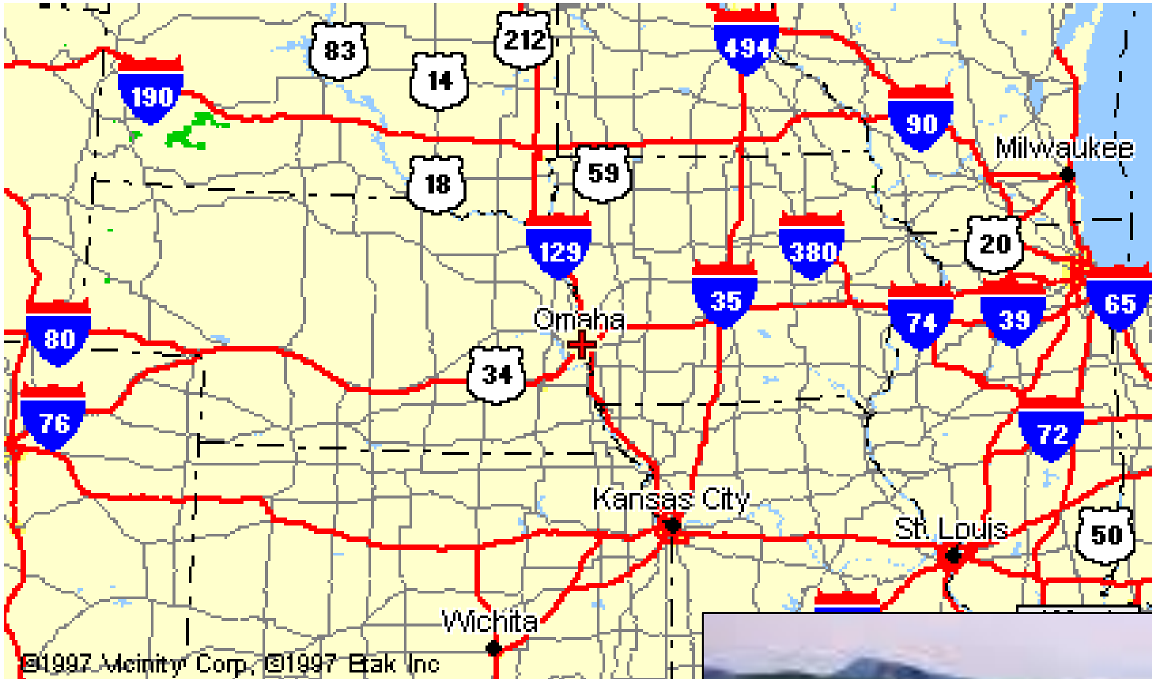
AI as “symbol manipulation” – required? enough?

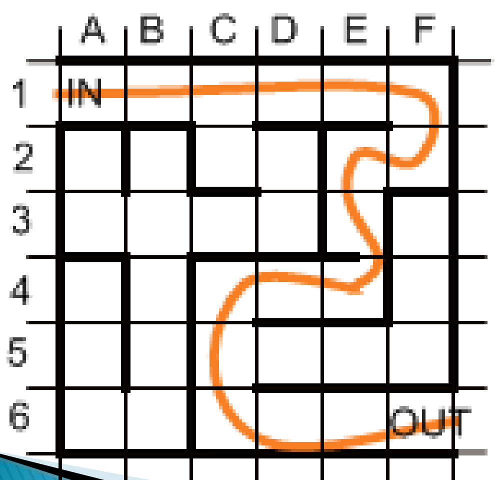
- The Box and Banana problem
  - Human, monkey, pigeon, crow, ..

Problem-solving agent

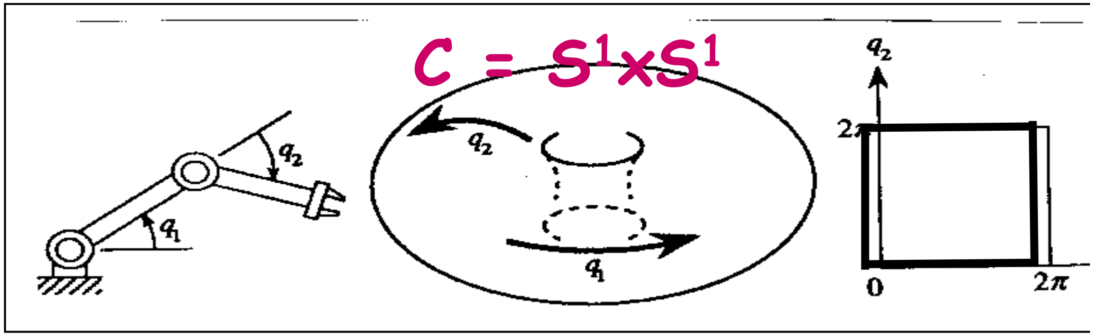
- Four general steps in problem solving:
  - (1) Goal formulation
    - What are the demanded, successful world states (**state-space of the problem**)
  - (2) Problem formulation
    - What actions and states are possible/legal to consider, given the goal
  - (3) Problem solving with search
    - Determine the possible sequence of actions that lead to the states of known values and then choose the best sequence.
  - (4) Executing the solution
    - Given the solution, perform its prescribed actions.

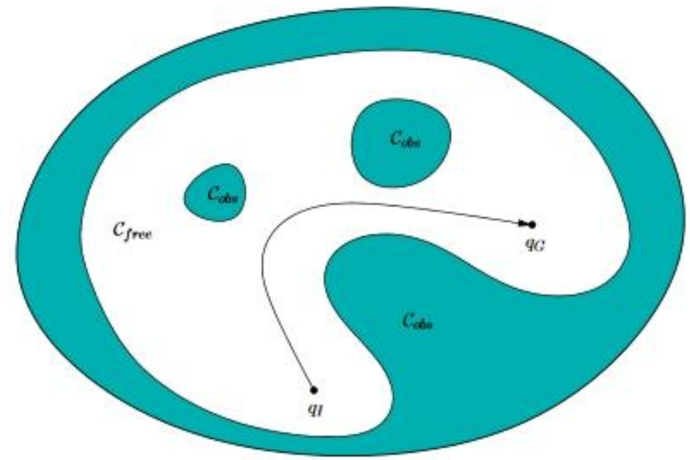
Physical State-Spaces

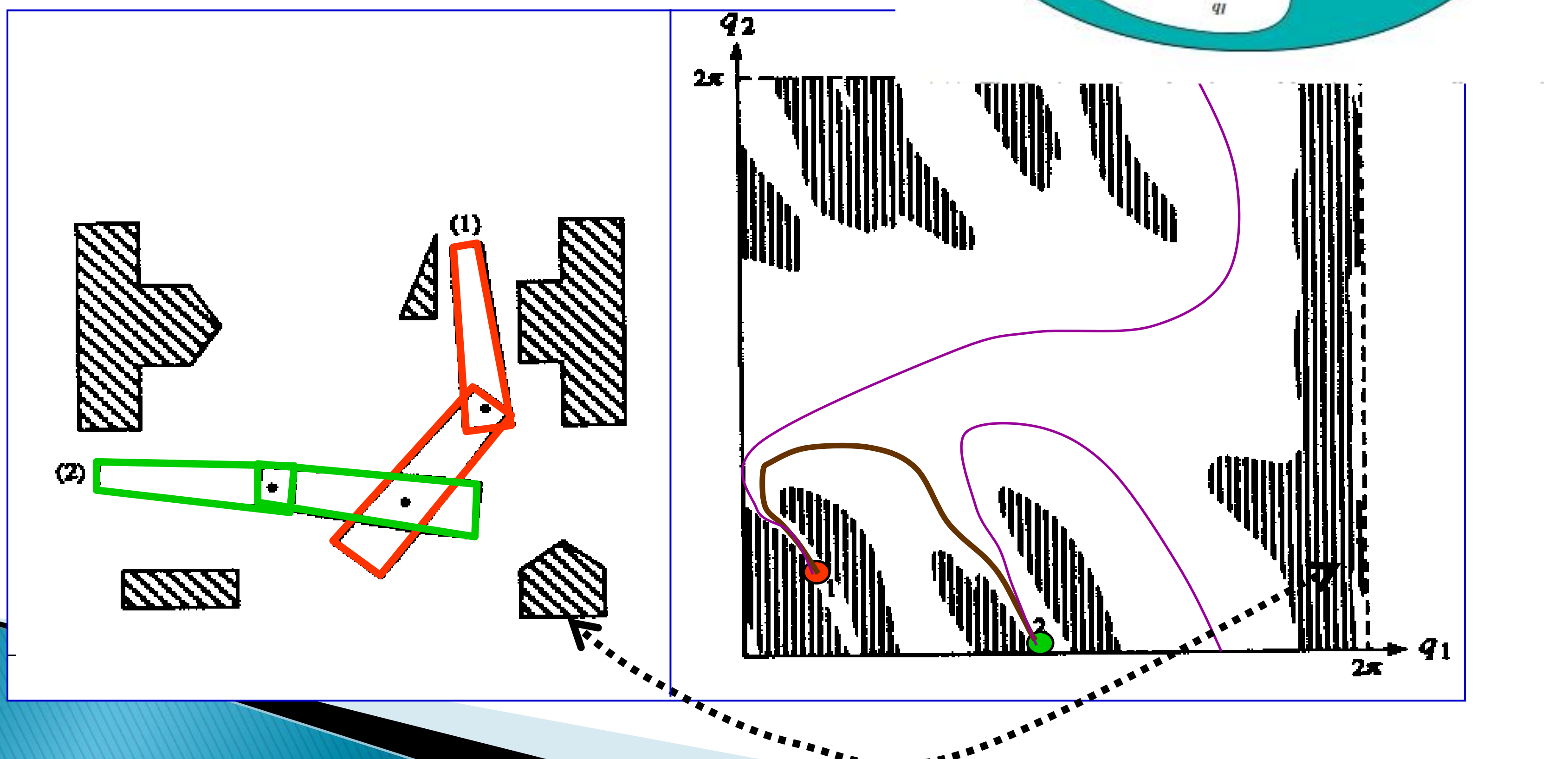




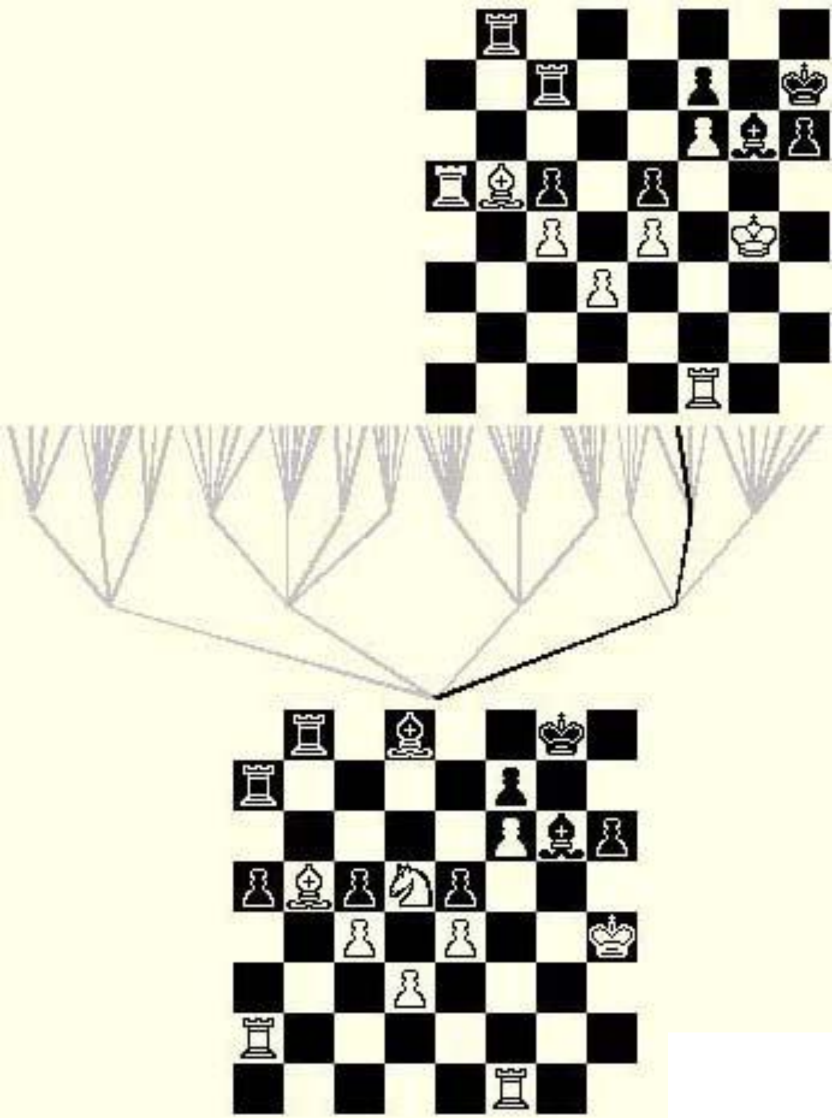
Reduce robot to a point → Configuration State-Spaces

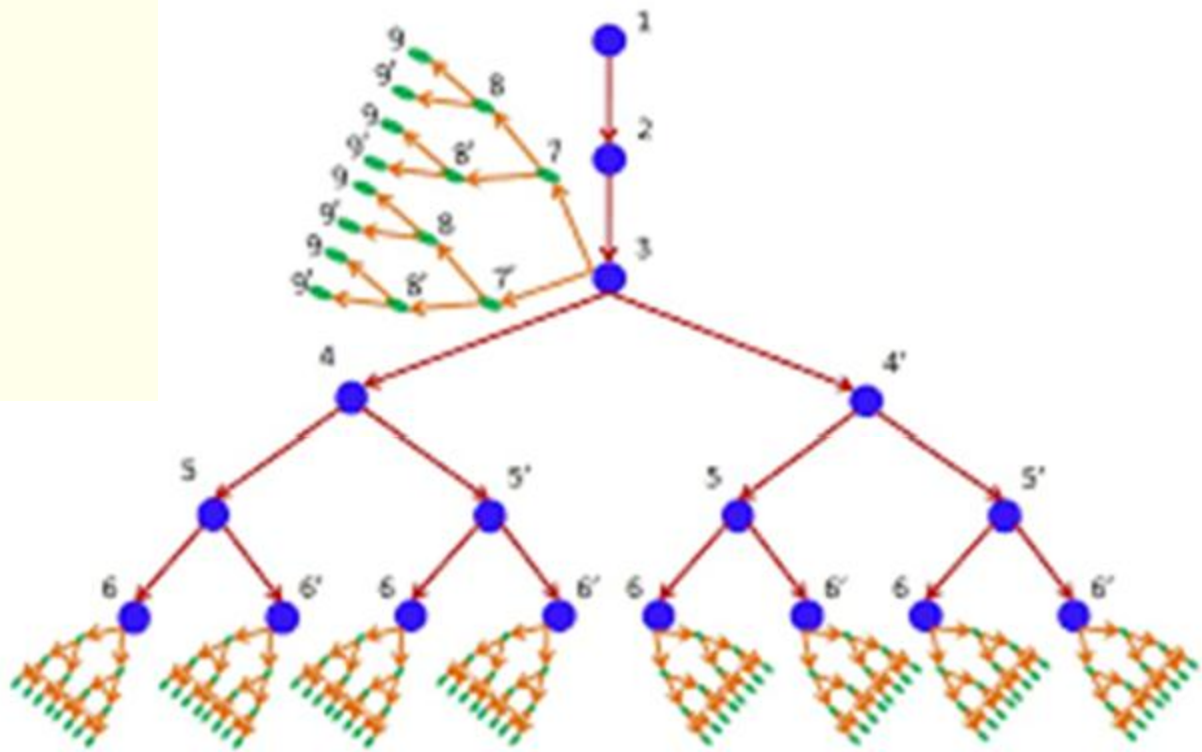






Abstract State-Spaces

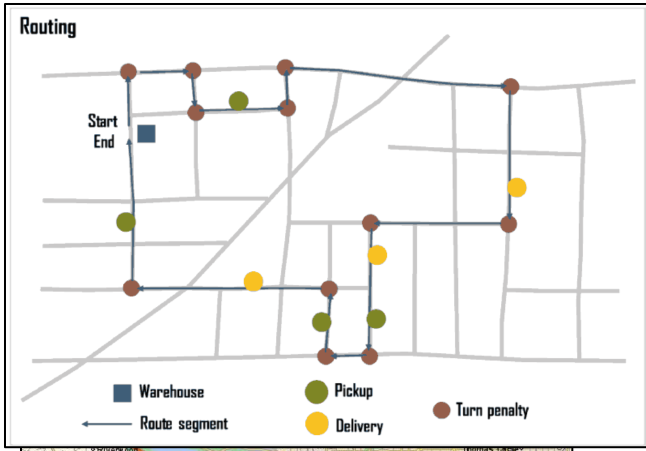




Example: Romania

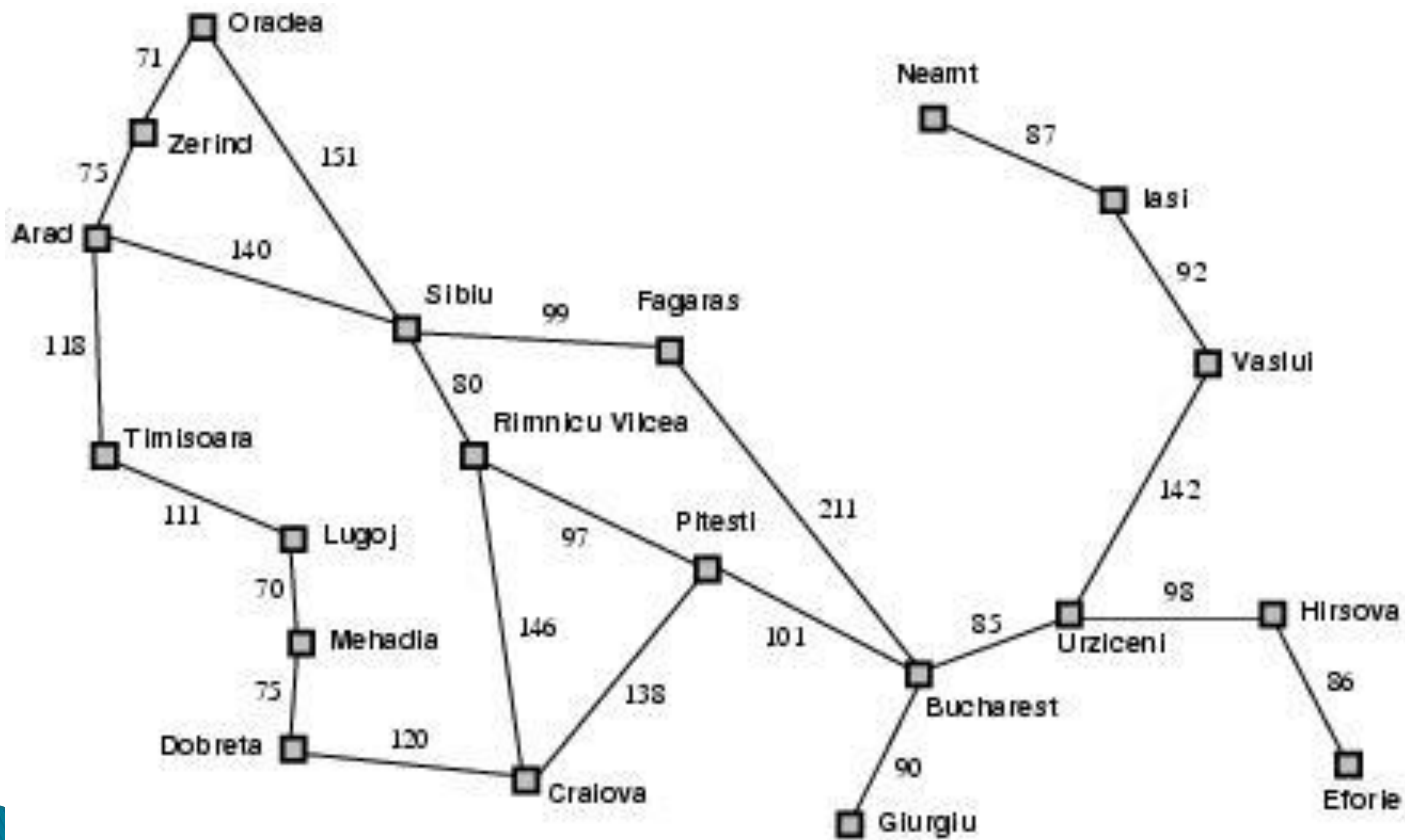
- On holidays in Romania; currently in town of Arad
  - Flight leaves home tomorrow from Bucharest
- Formulating the goal
  - Be in (time at the airport in) Bucharest
- Formulating the problem
  - States: various cities (closer or further from the goal!)
  - Actions: driving from a city to a city
- Finding solution
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, …, Bucharest (ending in the goal!)
- Executing the solution

Selecting a state space



- Real world is 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 action is a complex set of possible routes, detours, rest stops, etc.
  - The abstraction is valid if the path between two states is passable in the real world.
- (Abstract) solution = set of real paths that are real solutions in the real world.
- Abstract problem should be “easier” than the real problem.

Example: (an abstract) Romania



Problem formulation

- A problem is defined by:
  - (1) An initial state, e.g. '*at Arad*'
  - (2) Successor function (an operator to move in state-space)

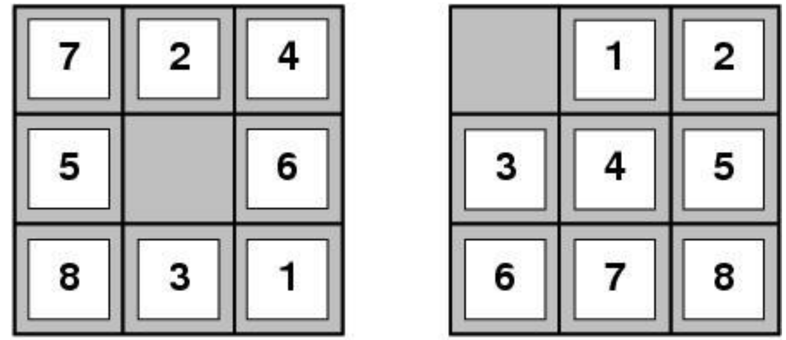
    S(X)= set of action-state pairs

    - e.g. $S(Arad)=\{<Arad \rightarrow Zerind, Zerind>,\ldots\}$

    intial state + successor function = state space as a graph
  - (3) Goal test, here can be:
    - *x='at Bucharest'*
  - (4) Path cost (additive)
    - e.g. sum of distances, number of actions executed, …
    - *c(x,a,y)* is the step cost (petrol?!), assumed to be >= 0

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

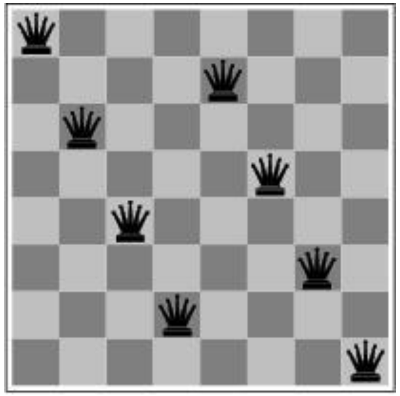
Examples: 8-puzzle



Start State Goal State

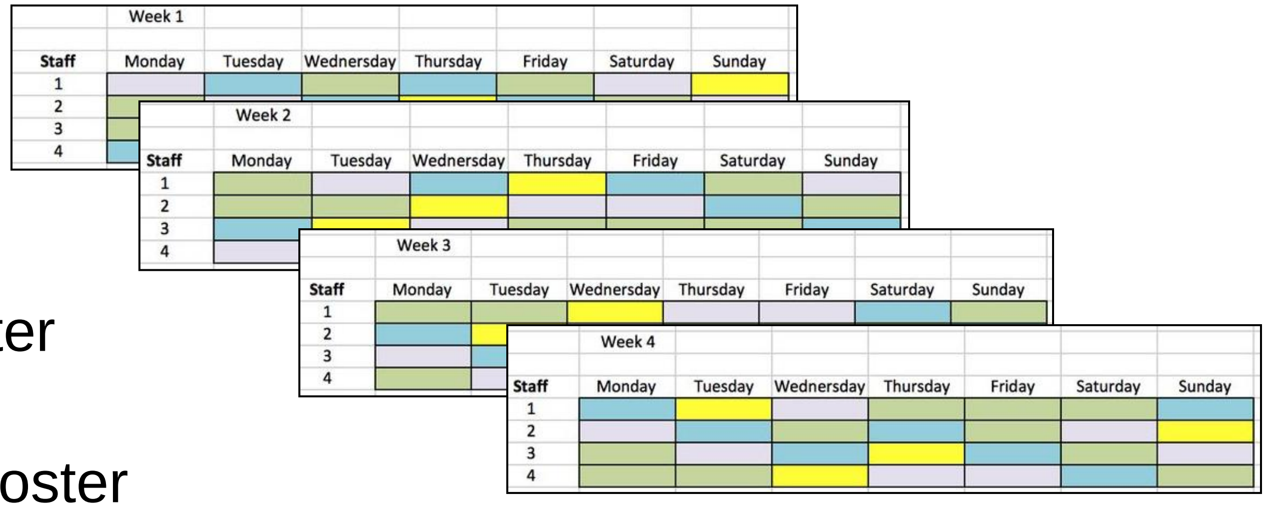
- States
  - The location of the eight tiles, and the blank.
- Initial state
  - {(7.0),(2,1),(4,2), …, (8,6),(3,7),(1,8)}
- Actions (operators)
  - 4 actions (blank moves Left, Right, Up, Down)
- Goal test
  - Is a given state a goal state = {(_,0),(1,1),…,(7,7),(8,8)}?
- Path cost
  - Each step costs 1

Examples: 8-queens problem



- States
  - Complete-state: Any arrangement of all 8 queens
  - Incr/1-state: Any arrangement of 0 to 8 queens
  - Incr/2-state: *n* ($0 \leq n \leq 8$) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
- Initial state
  - Complete-state: Any arrangement of all 8 queens on the board
  - Incr/1, Incr/2: no queens on board
- Actions
  - Complete-state: move a queen to an empty square
  - Incr/1: Add a queen to an empty square
  - Incr/2: Add a queen in the leftmost empty column, not attacking others
  - From 3×10^{14} to 2057 possible sequences to investigate, depending on problem formulation.
- Goal test
  - All 8 queens on board and none attacked
- Path cost
  - None

Examples: Police precinct shift roster



- States
  - Weekly roster
- Initial state
  - Last week roster
- Action (operator)
  - Moving shifts, moving weekdays, scheduled holidays, participating in special events
- Goal test
  - Captain input
- Path cost
  - Complicated: paid overtime, burden of late shifts, personal incompatibility, paid skipped holidays, …

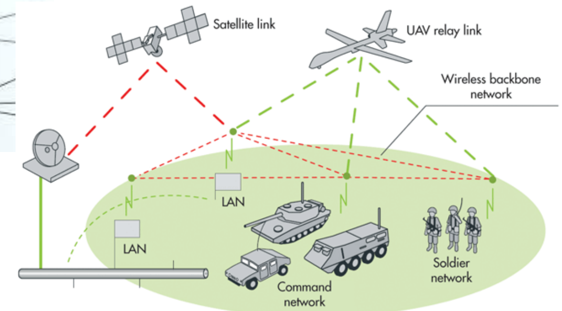
Examples: Route Finding Problems

Car Navigation

Air Travel Planning

Routing in Computer Networks

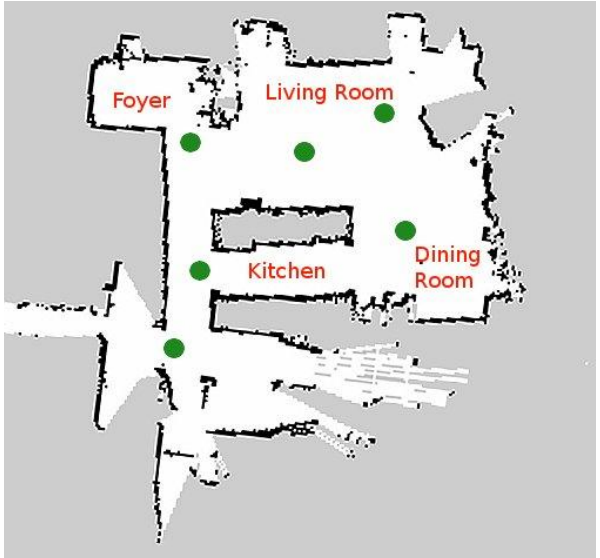
Military Operation Planning



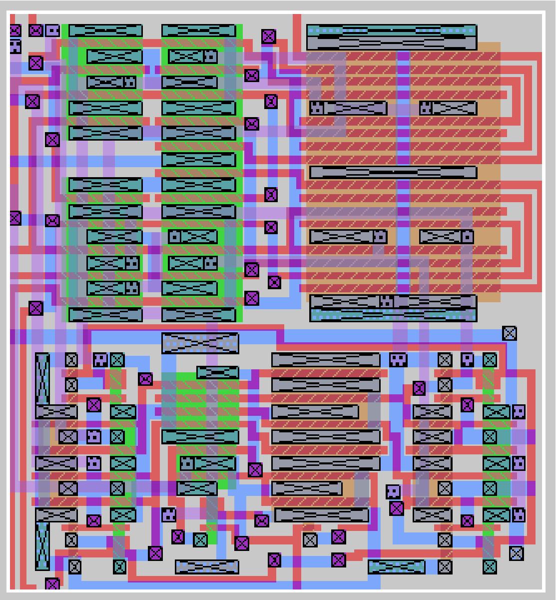
- States
  - Locations
- Initial state
  - Starting point
- Action (operator)
  - Move from one location to another
- Goal test
  - Arriving at a certain location
- Path cost
  - May be quite complex, money, time, travel comfort, scenery, …

Examples: Robot Navigation

- States
  - Locations, Position of actuators
- Initial state
  - Starting poisition (task?)
- Action (operator)
  - Movement, actions of actuators
- Goal test
  - Task-dependent
- Path cost
  - May be quite complex distance, energy consumption, …



Examples: VLSI layout problem



- States
  - Positions of components, wires on chip
- Initial state
  - Incremental: no components placed
  - Complete-state: all components placed (e.g. randomly, manually)
- Action (operator)
  - Incremental: place components, route wire
  - Complete-state: move component, move wire
- Goal test
  - All coponents placed. Components connected as specified
- Path cost
  - May be quite complex, distance, capacity, number of connections per components, …

Basic search algorithms

How do we find the solutions of previous problems?

- Traversal of the search space (a tree or a graph)
- From the initial state to a goal state
- Legal sequence of actions as defined by successor function

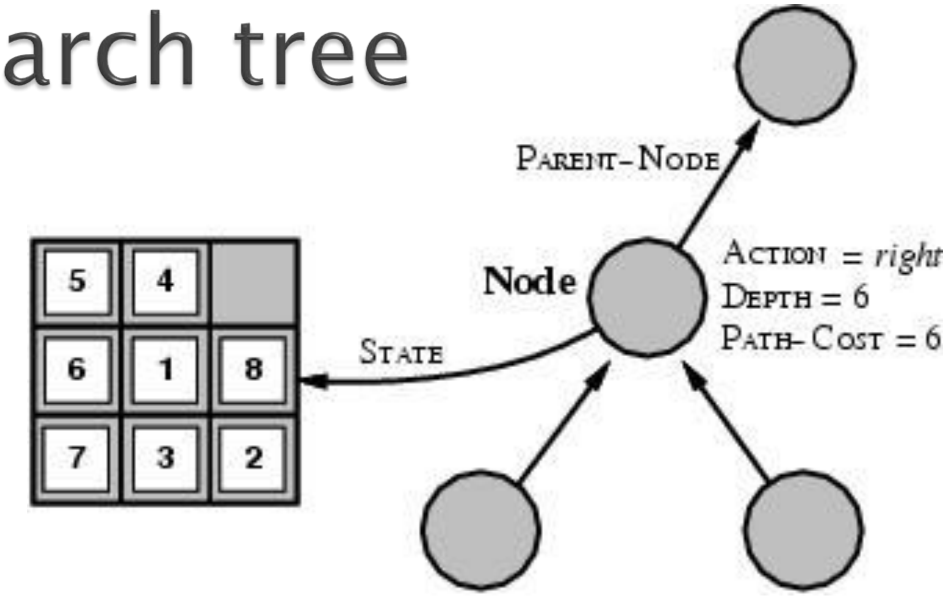
General procedure

- Check for goal state
- Expand the current state
- Determine the set of reachable states
- Return „failure” if the set is empty
- Select one from the set of reachable states
- Move to the selected state

A search tree is generated

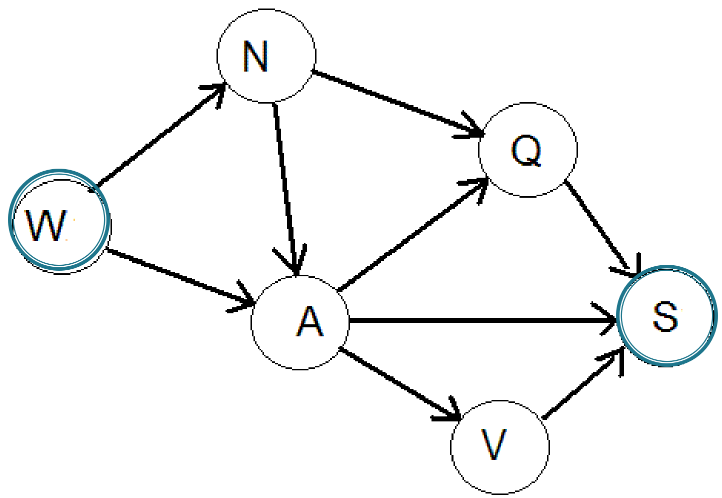
- Nodes are added as more states are visited
- The tree specifies possible paths through the search space

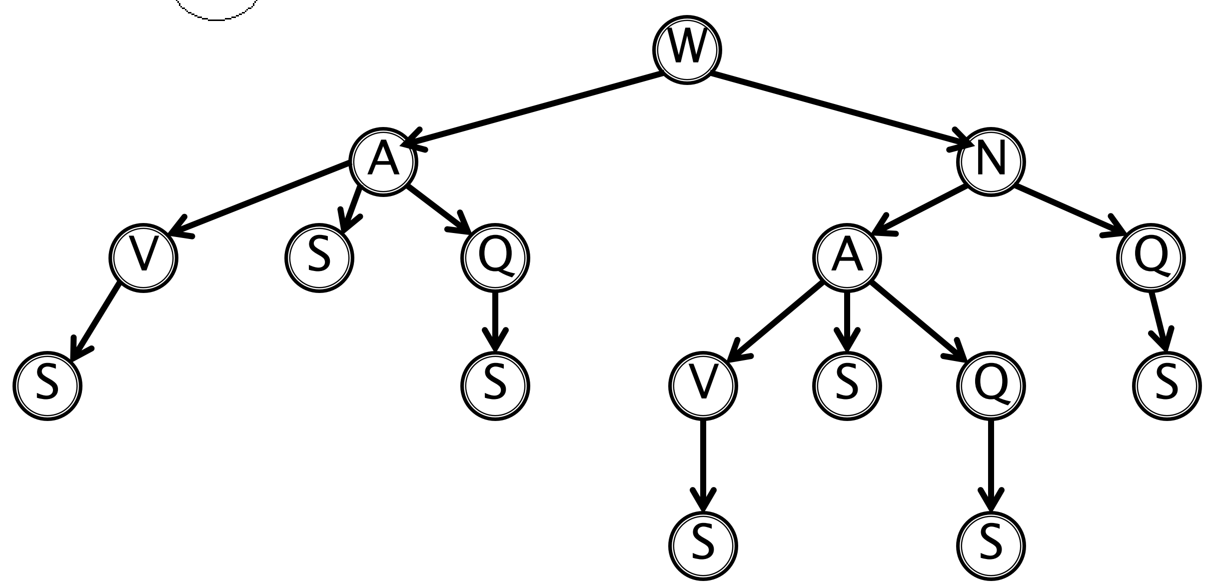
State space vs. search tree



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

State space vs. search tree





Tree search algorithm (1)

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

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
  loop do
    if EMPTY?(*fringe*) **then return** failure
    node ← REMOVE-FIRST(*fringe*)
    if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
      then return SOLUTION(*node*)
    fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

Tree search algorithm (2)

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

successors ← the empty set

for each <*action*, *result***> in** SUCCESSOR**[***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*

Graph-search – handling repeated states and loops, later …

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?*
  - Space Complexity: *number of nodes stored in memory during search?*
  - Time Complexity: *number of nodes generated/expanded?*
- Time and space complexity measure problem difficulty and are 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 ∞)*

Uninformed search strategies

- (a.k.a. blind search) = use only information available in problem definition.
  - When strategies can determine whether one non-goal state is better than another $\rightarrow$ informed search.

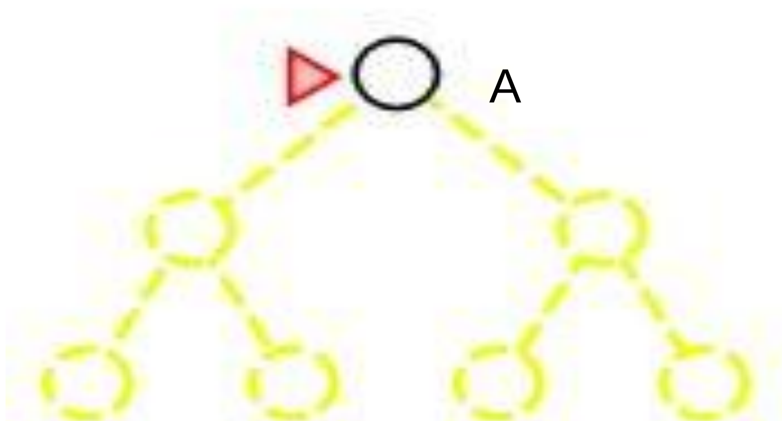
    We do not have this information here!

- Search methods defined by node expansion algorithm:
  - Breadth-first (BF) search
  - Uniform-cost search
  - Depth-first (DF) search
  - Depth-limited search
  - Iterative deepening (ID) search
  - Bidirectional search

BF-search, an example

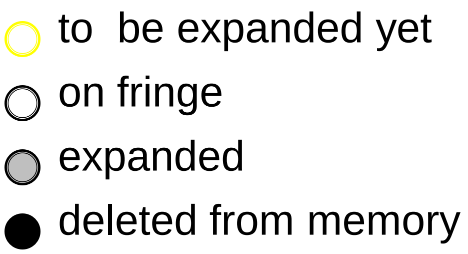
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue

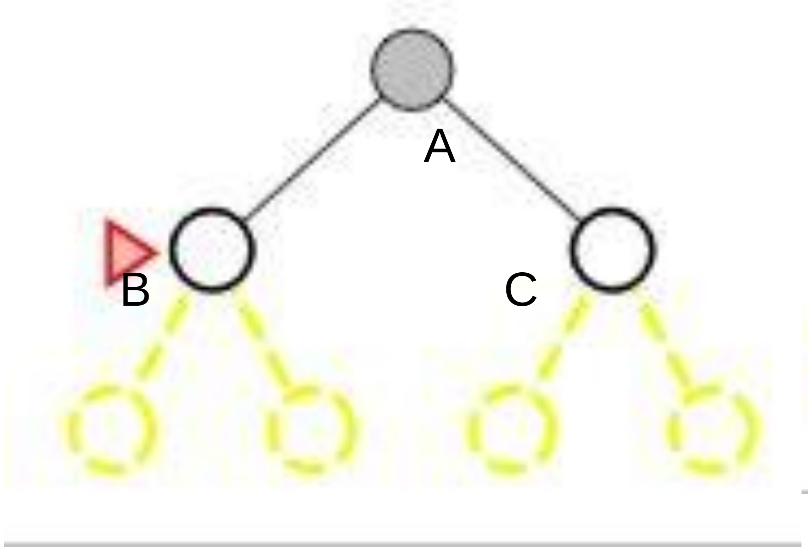
to be expanded yet
on fringe
expanded
deleted from memory



BF-search, an example

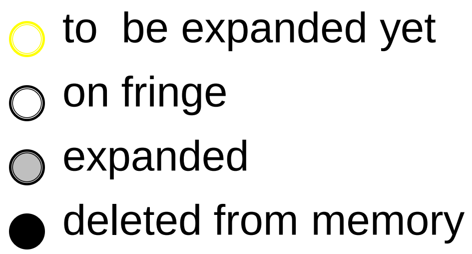
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue

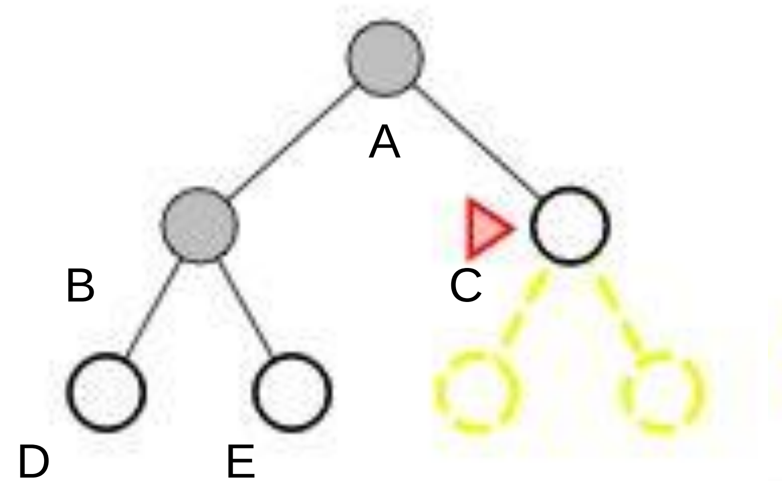




BF-search, an example

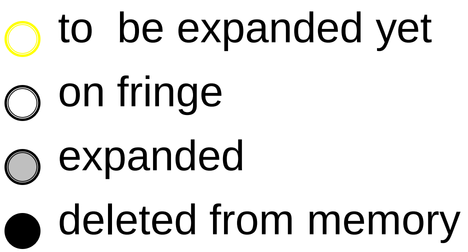
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue

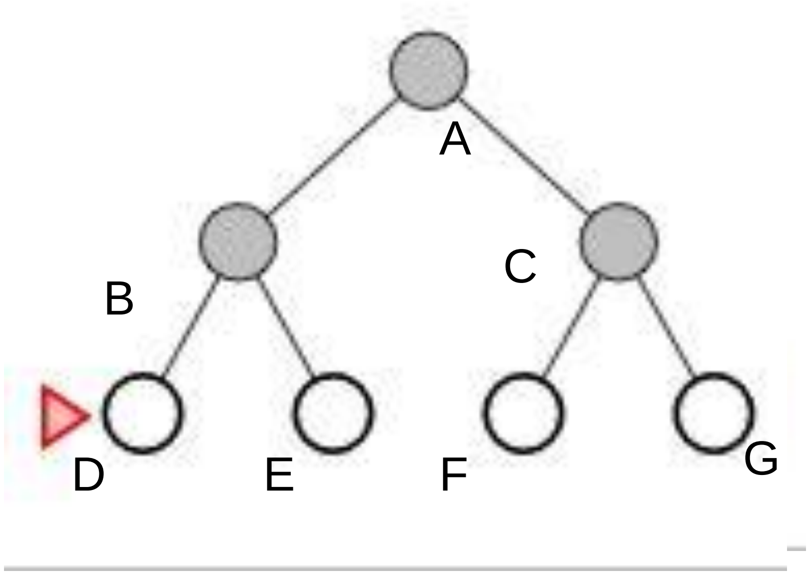




BF-search, an example

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





BF-search; evaluation

- Completeness:
  - *Does it always find a solution if one exists?*
  - YES
    - If the shallowest goal node is at some finite depth d
    - Condition: If *b* is finite
      - (maximum num. of succ. nodes is finite)

BF-search; evaluation

- Completeness:
  - YES (if b is finite)
- Time complexity:
  - Assume a state space where every state has b successors.
    - the root has b successors, each node at the next level has again b successors (total b^2), …
    - Assume solution is at depth d
    - Worst case; expand all but the last node at depth d
    - Total number of nodes generated:

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total number of nodes generated:
- Space complexity:
  - Idem, because each node is retained in the memory

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

BF-search; evaluation

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

$$1+b+b^2+b^3+...+b^d+(b^{d+1}-b)=O(b^{d+1})$$

- Space complexity:
  - Idem, because each node is retained in the memory
- Optimality:
  - *Does it always find the least-cost solution?*
  - In general YES
    - unless actions have different cost.

BF-search; evaluation

- Two lessons:
  - Maintaining large memory is a bigger problem than the execution time.
  - Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

BF-search; evaluation

b = 10 **branching factor** (Chess app. 35 !)

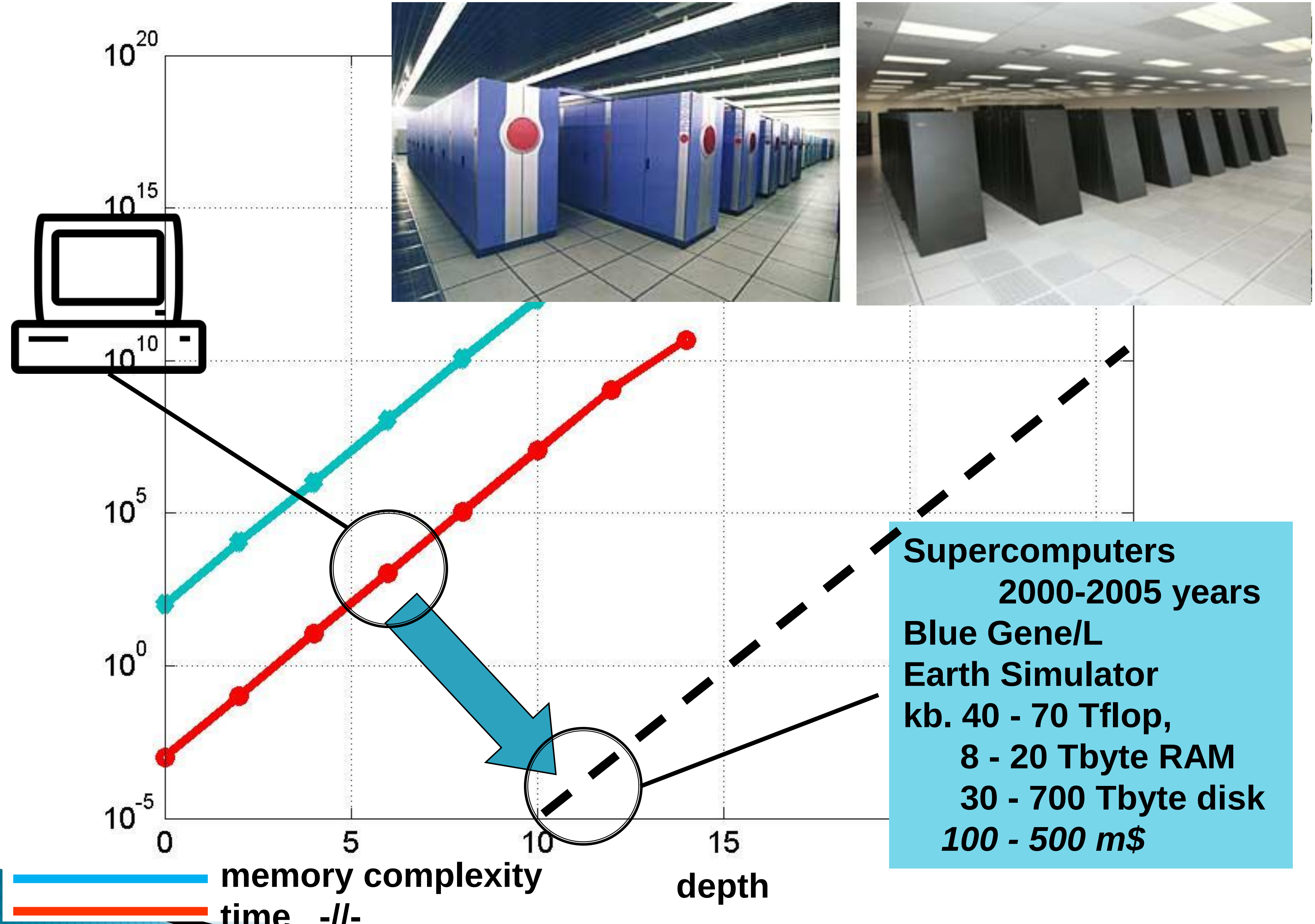
1000 decision / sec ≅ 1kflop
1 decision information: 100 byte

1 byte ≅ 1 letter

Depth	Decisions	Time Demand	Memory Demand
0	1	0.001 sec	100 byte
2	111	0.1 sec	11 kbyte
4	11111	11 sec	1 Mbyte
6	10^6	18 minutes	111 Mbyte
8	10^8	31 hours	11 Gbyte (PC)
10	10^{10}	128 days	1 Tbyte
12	10^{12}	35 years	111 Tbyte
14	10^{14}	1500 years	11111 Tbyte

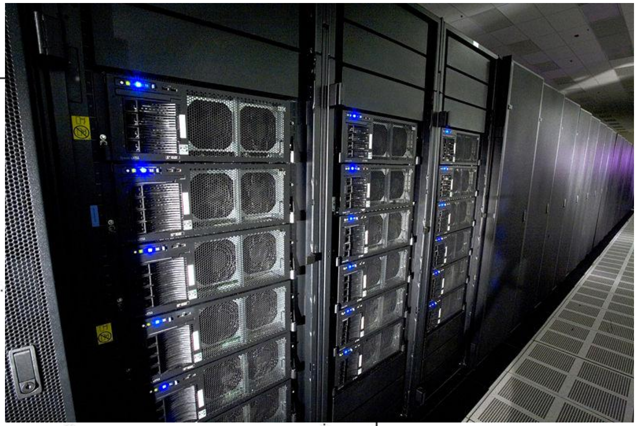
k = 10^3, M = 10^6, G = 10^9, T = 10^{12}, P = 10^{15}

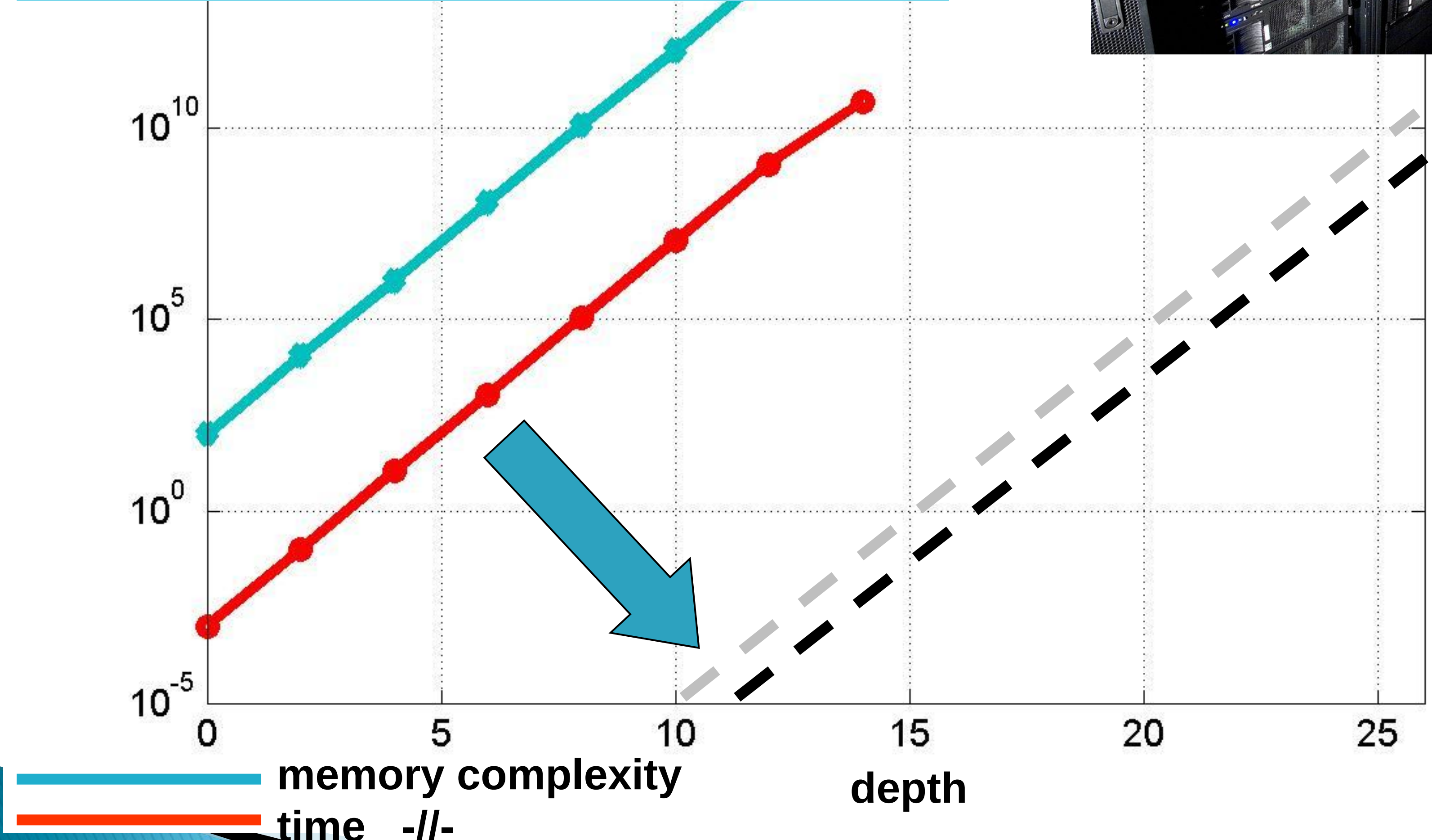
11111 Tbyte ≅ 3 milliard human libraries



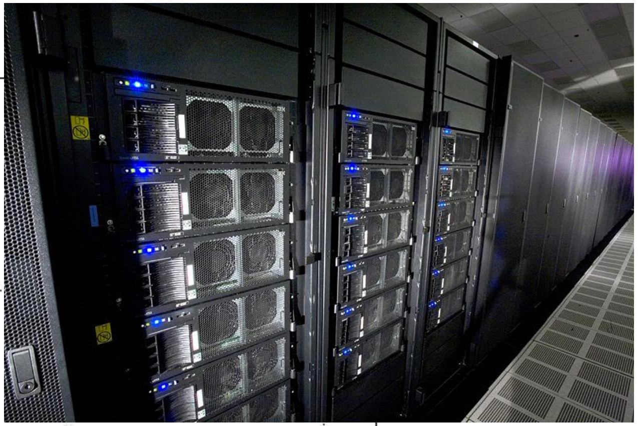
10^{20}
10^{15}
10^{10}
10^{5}
10^{0}
10^{-5}
0
5
10
15
depth
memory complexity
time -//-
Supercomputers
2000-2005 years
Blue Gene/L
Earth Simulator
kb. 40 - 70 Tflop,
8 - 20 Tbyte RAM
30 - 700 Tbyte disk
100 - 500 m$

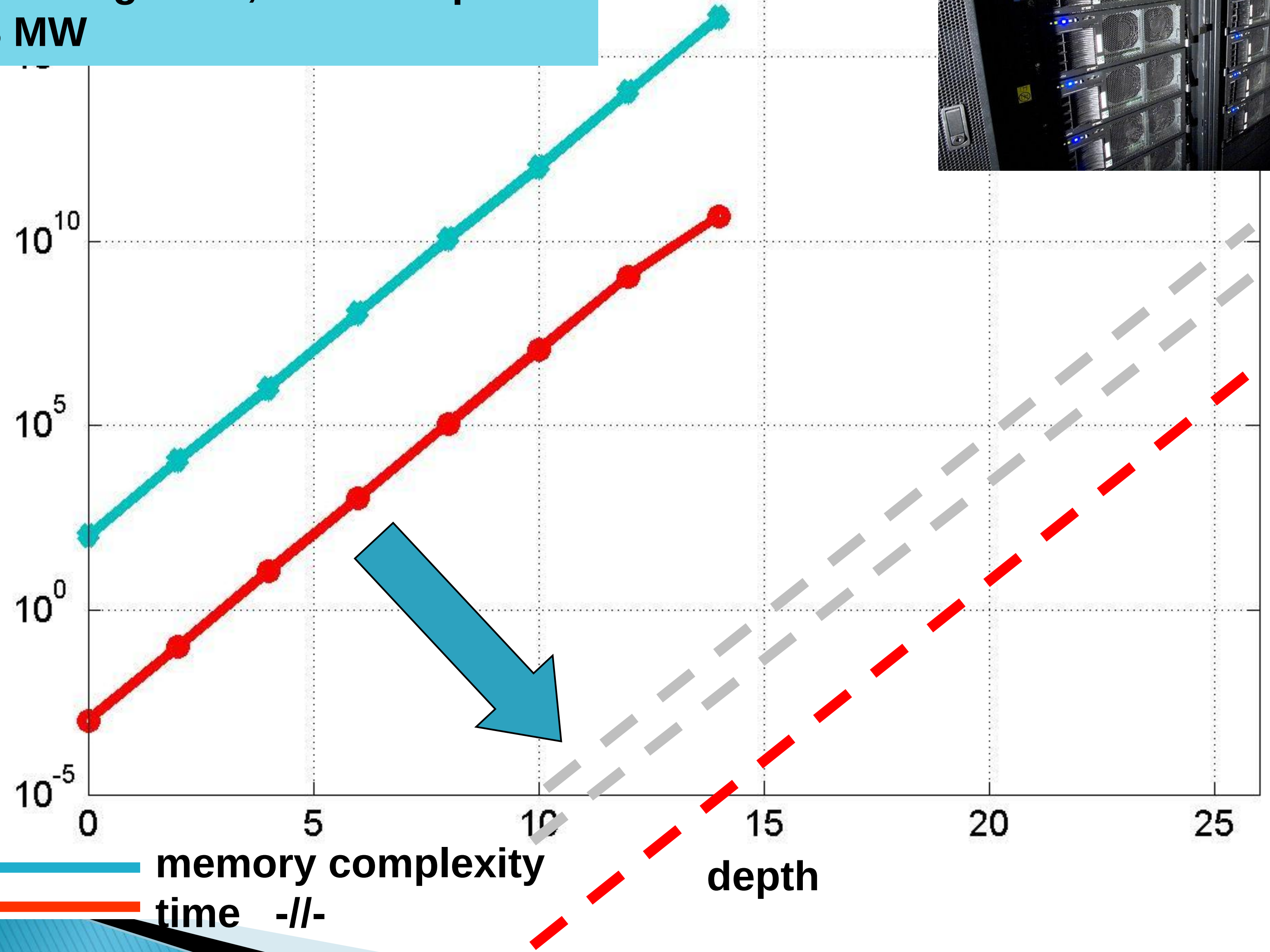
Supercomputers 2007-2008 years
IBM Blue Gene/L, upgrade
Lawrence Livermore Nat Lab, 478 Tflop
IBM Roadrunner, Los Alamos Nat Lab
1.026 Pflop



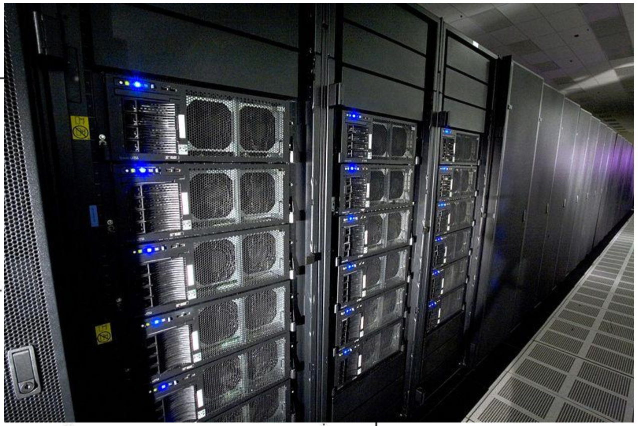


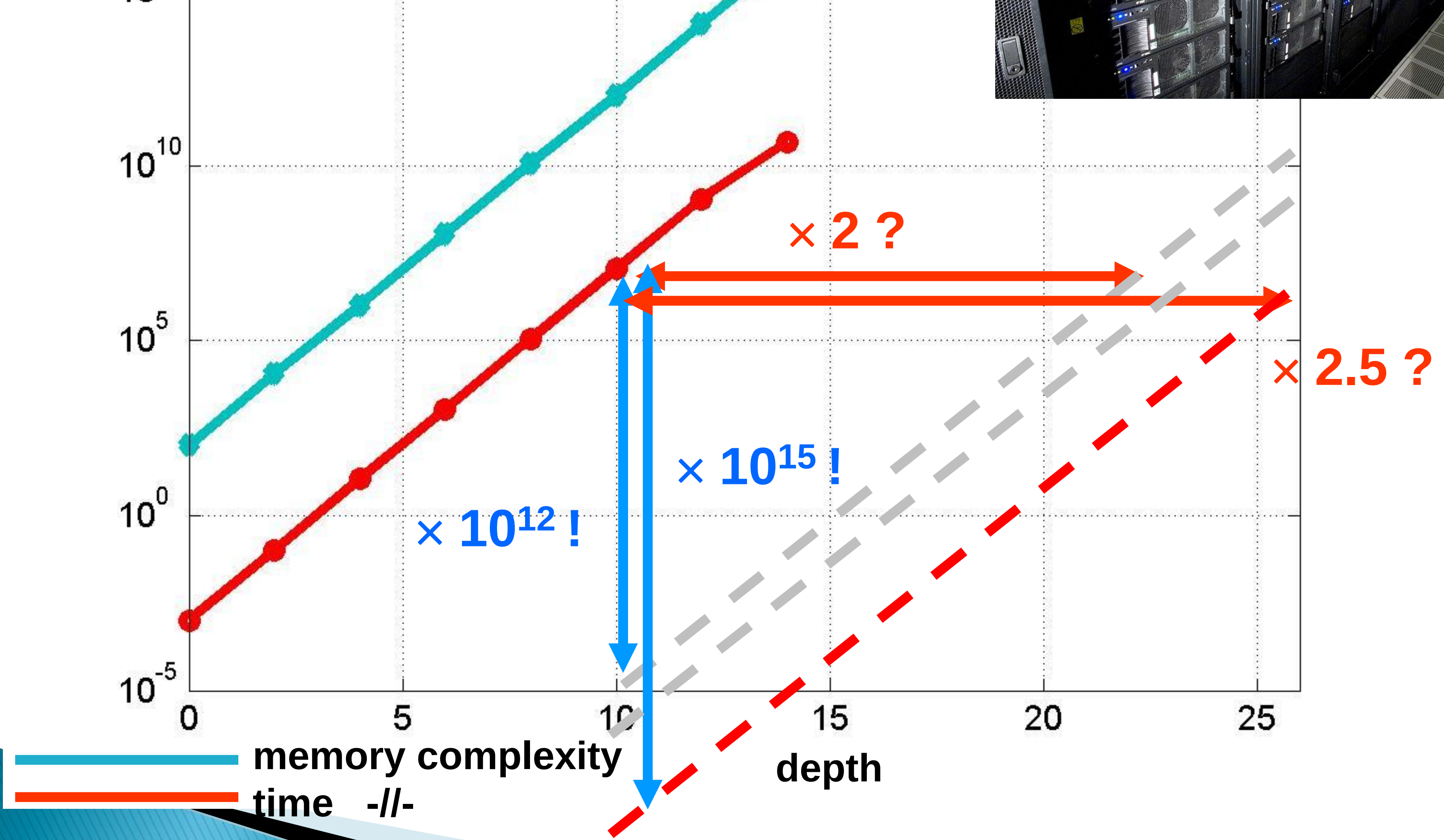
Supercomputers 2018 year
IBM Summit
Oak Ridge Lab, 1223 Pflop
13 MW





Supercomputers 2018 year
IBM Summit
Oak Ridge Lab, 1223 Pflop
13 MW





Uniform-cost search

- Extension of BF-search:
  - Expand node with *lowest path cost*
- Implementation: *fringe* = queue ordered by path cost.

- UC-search is the same as BF-search when all step-costs are equal.

Uniform-cost search

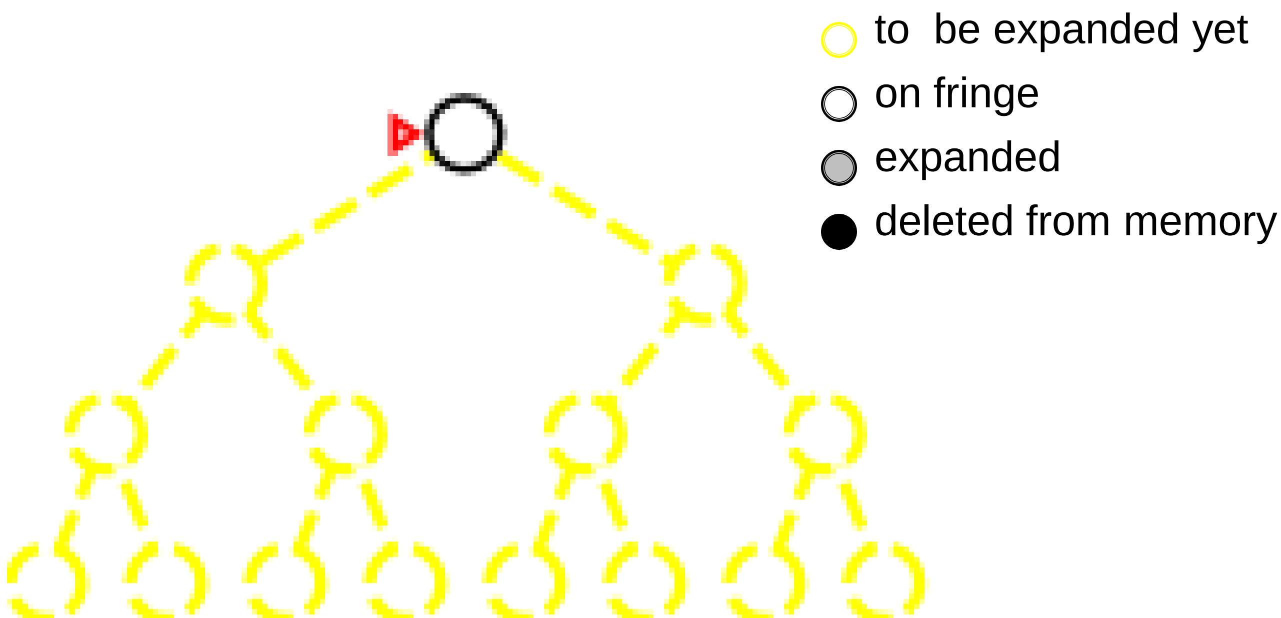
- Completeness:
  - YES, if step-cost > ε (smal positive constant)
- Time complexity:
  - Assume C* the cost of the optimal solution.
  - Assume that every action costs at least ε
  - Worst-case:

$$O(b^{C^*/\varepsilon})$$

- Space complexity:
  - Idem to time complexity
- Optimality:
  - nodes expanded in order of increasing path cost.
  - YES, if complete.

DF-search, an example

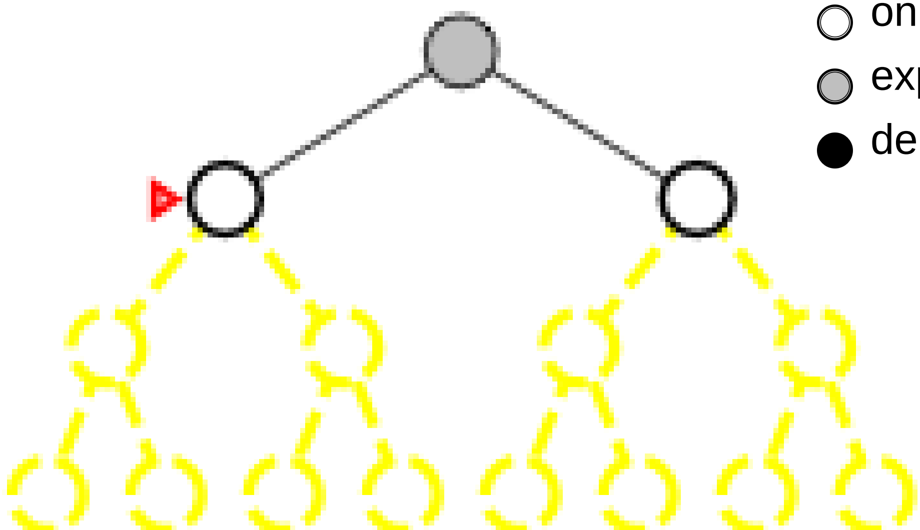
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



DF-search, an example

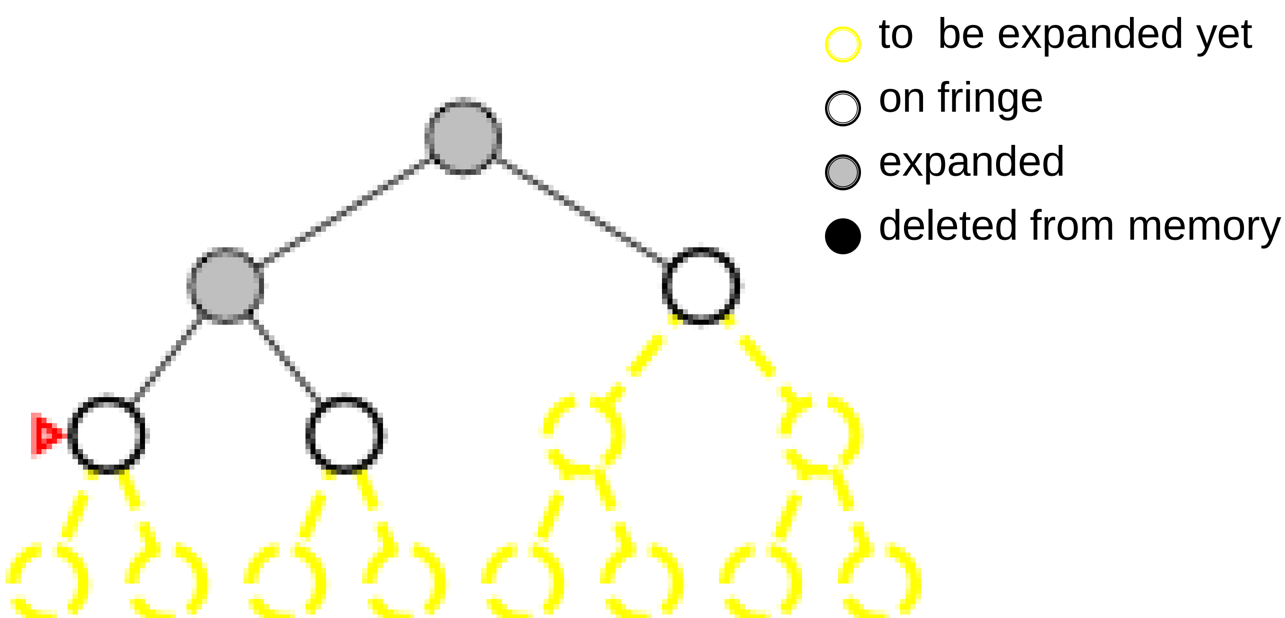
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)

to be expanded yet
on fringe
expanded
deleted from memory



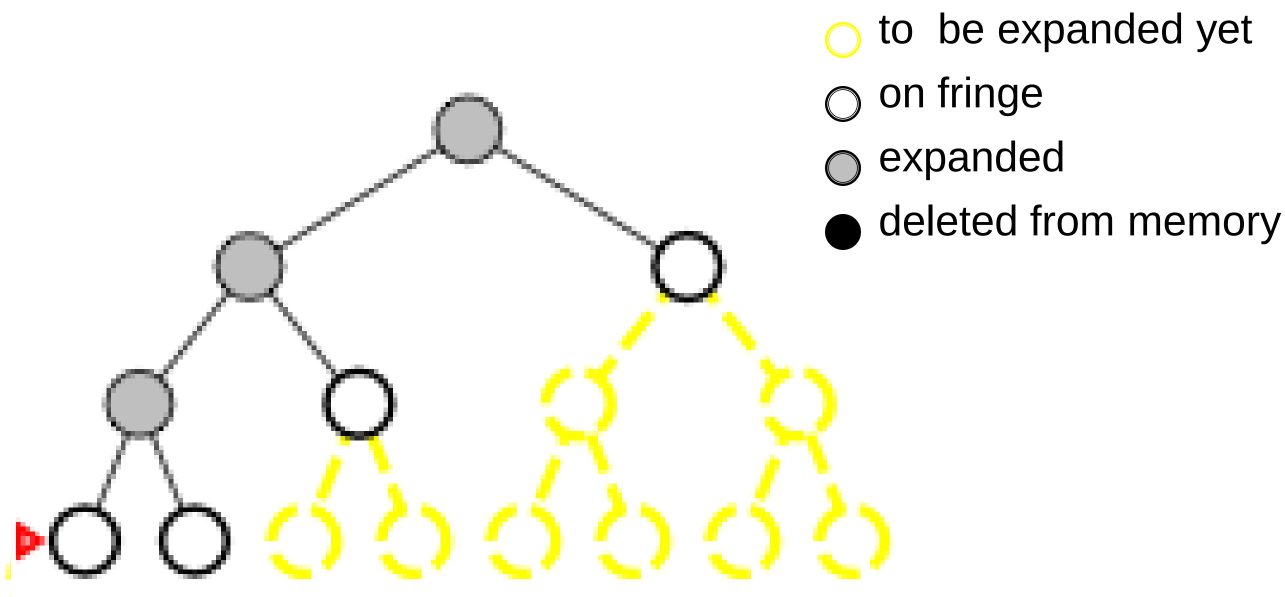
DF-search, an example

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



DF-search, an example

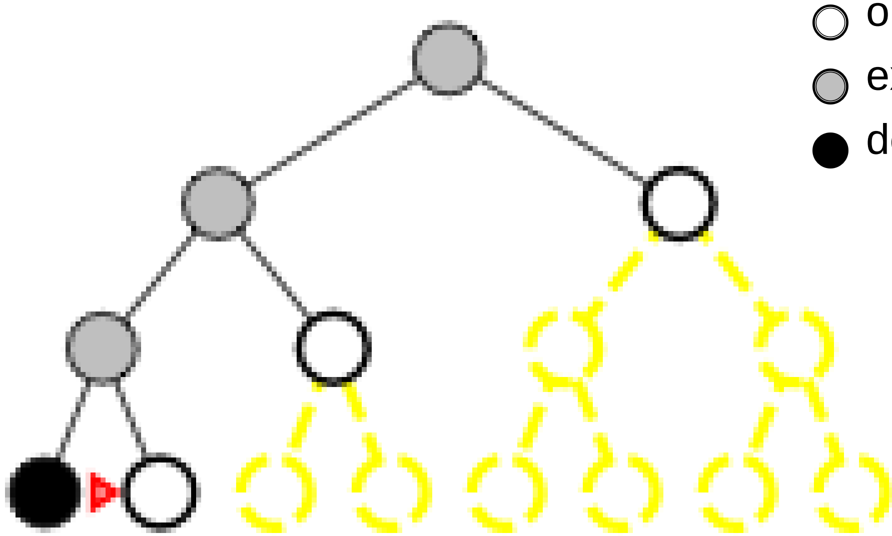
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



DF-search, an example

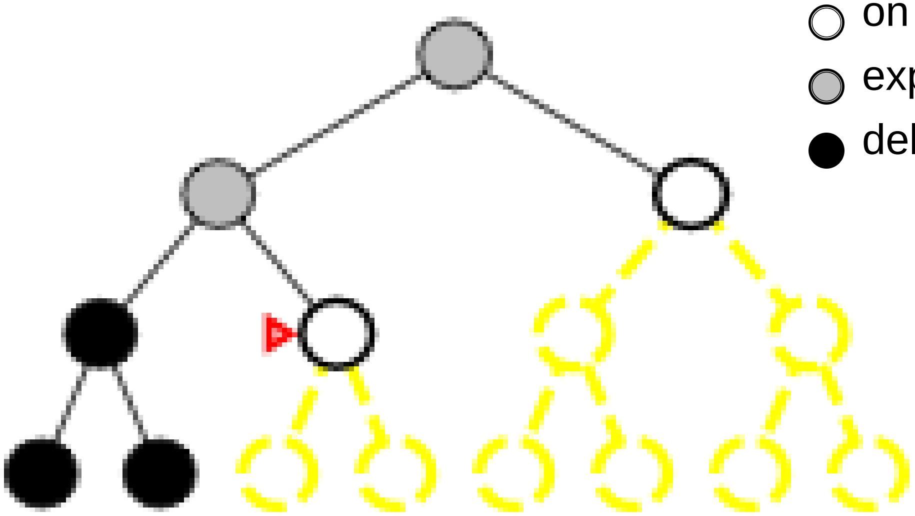
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)

to be expanded yet
on fringe
expanded
deleted from memory



DF-search, an example

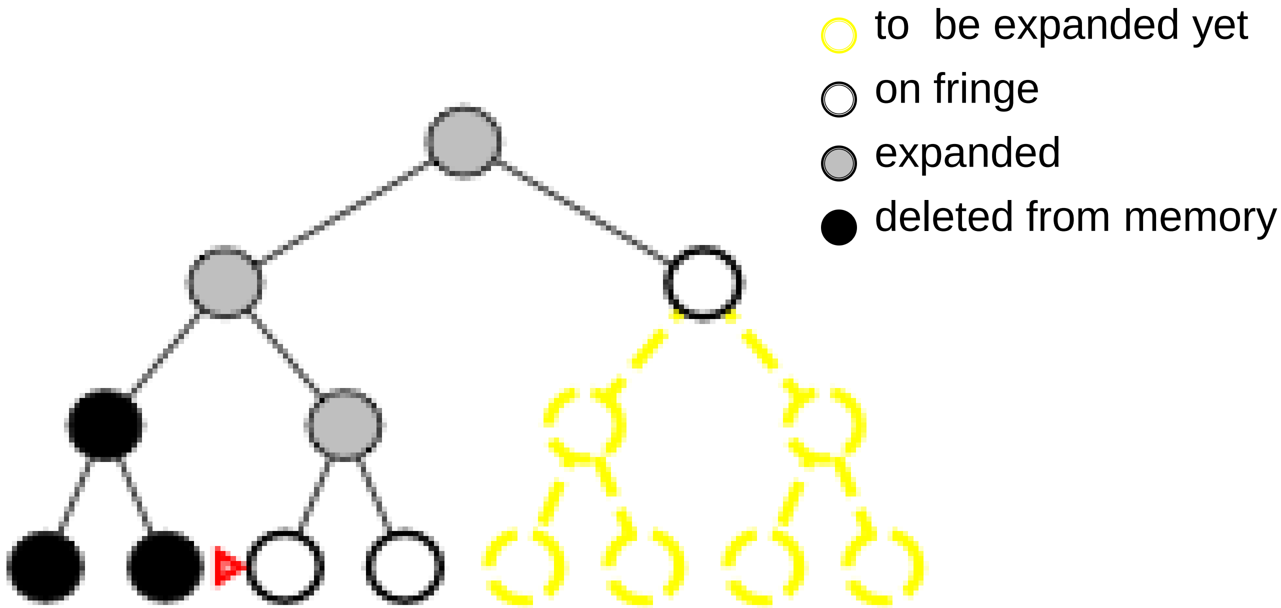
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



- to be expanded yet
- on fringe
- expanded
- deleted from memory

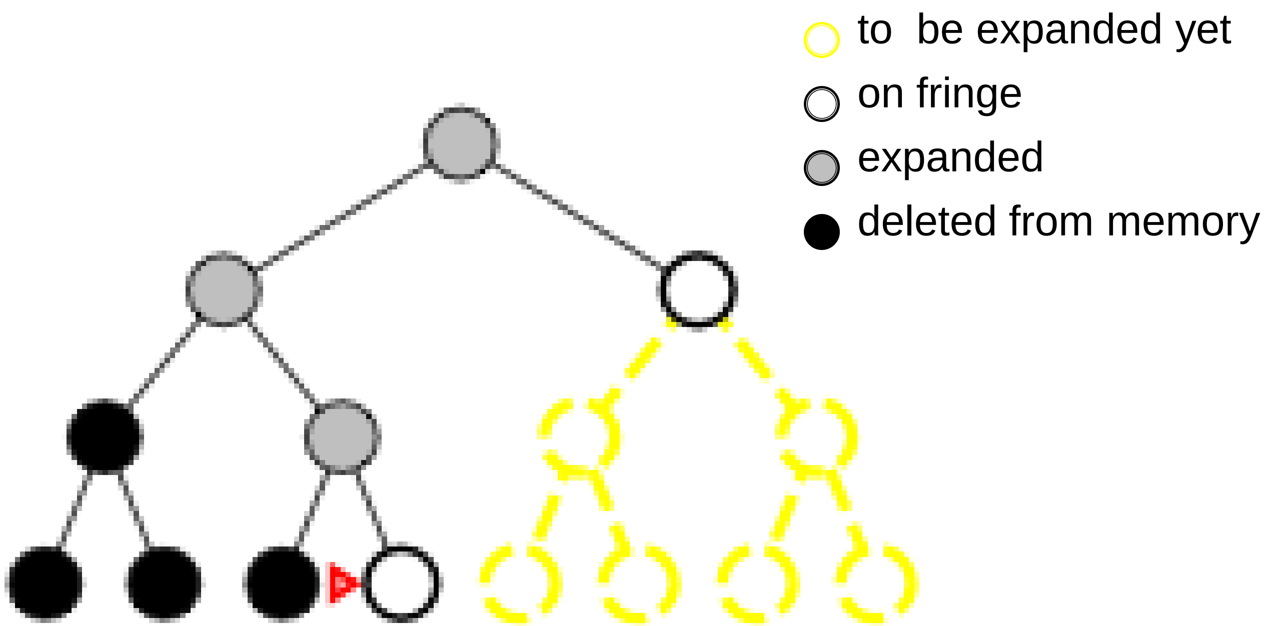
DF-search, an example

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



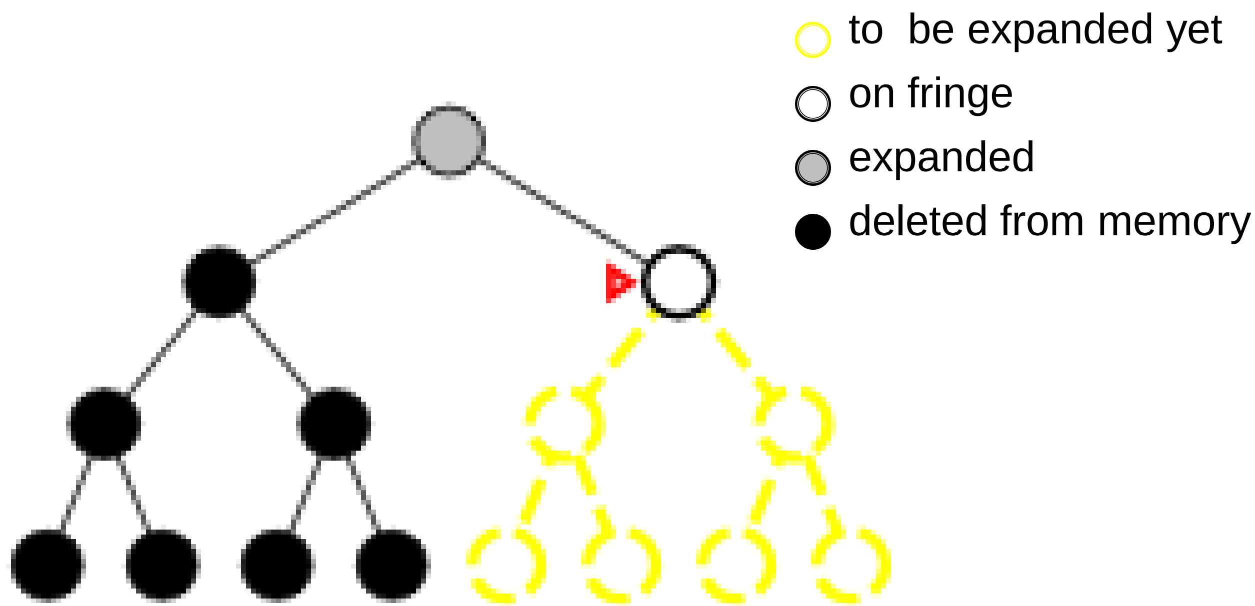
DF-search, an example

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



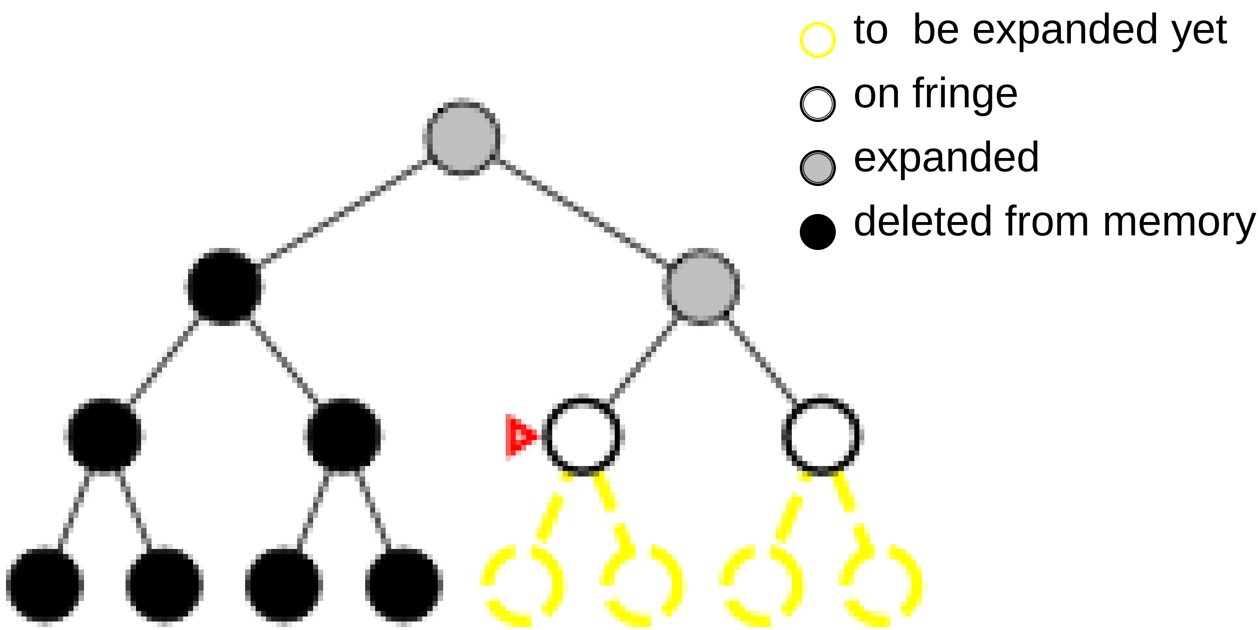
DF-search, an example

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



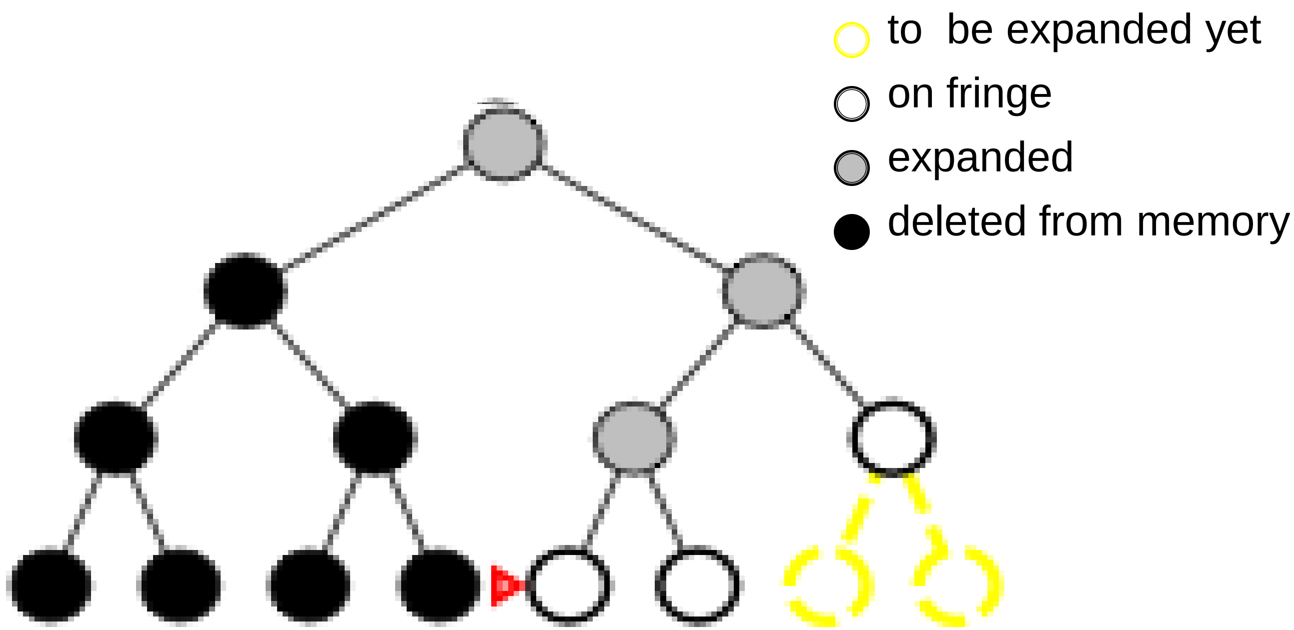
DF-search, an example

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



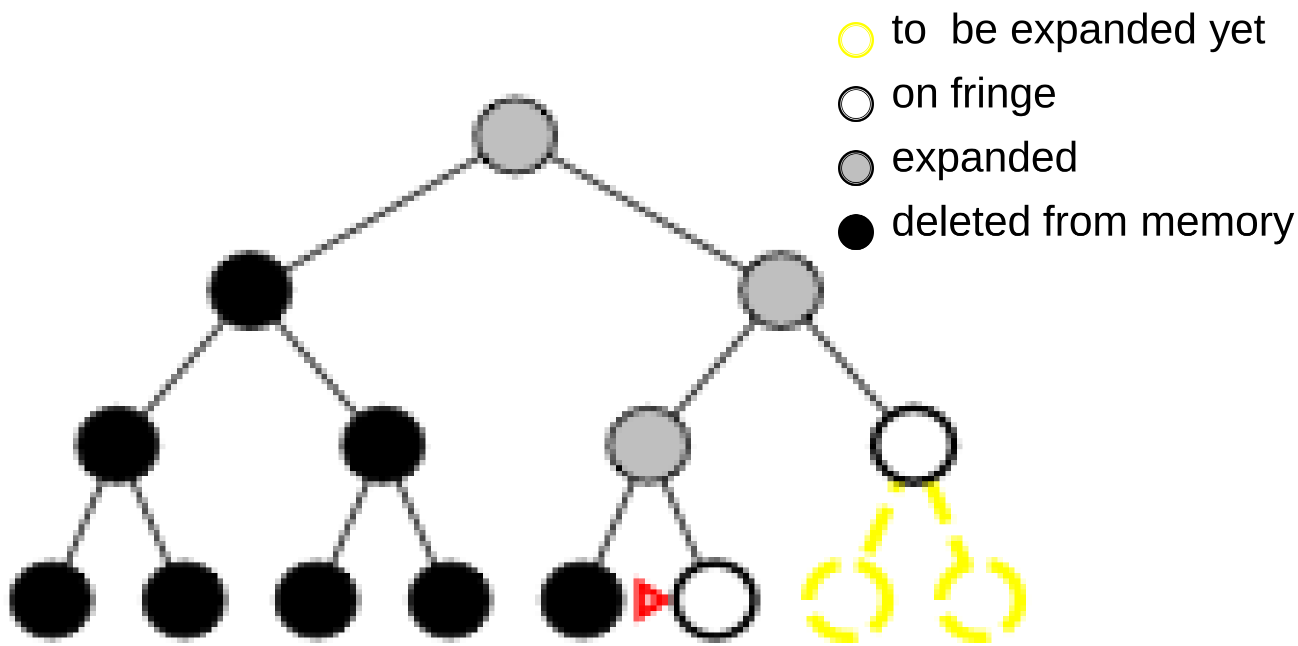
DF-search, an example

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



DF-search, an example

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



DF-search; evaluation

- Completeness;
  - *Does it always find a solution if one exists?*
  - NO
    - *unless* search space is finite and no loops are possible.

DF-search; evaluation

- Completeness;
  - NO unless search space is finite. $O(b^m)$
- Time complexity;
  - May be terrible if *m* is much larger than *d* (depth of optimal solution)
  - But if many solutions, then faster than BF-search

DF-search; evaluation

- Completeness;
  - NO unless search space is finite.
- Time complexity; $O(b^m)$
- Space complexity; $O(bm+1)$
  - Backtracking search uses even less memory
    - One successor instead of all *b.*

DF-search; evaluation

- Completeness;
  - NO unless search space is finite.
- Time complexity; $O(b^m)$
- Space complexity; $O(bm+1)$
- Optimality; No
  - Same issues as completeness

Depth-limited search

- DF-search with a depth limit *l*.
  - i.e. nodes at depth *l* are not expaneded for successors.
  - Problem knowledge can be used
- Solves the infinite-path problem, but
- If $l < d$ then incompleteness results.
- If $l > d$ then not optimal.
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$

Depth-limited algorithm

function DEPTH-LIMITED-SEARCH(*problem, limit*) **return** a solution or failure/cutoff

return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[*problem*]), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem, limit*) **return** a solution or failure/cutoff

cutoff_occurred? ← false

if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)

else if DEPTH[*node*] == *limit* **then** return *cutoff*

else for each *successor* **in** EXPAND(*node*, *problem*) **do**

    result ← RECURSIVE-DLS(*successor*, *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

Iterative deepening search

- What it is?
  - A general strategy to find best depth limit *l*.
    - Goal is found at depth *d*, the depth of the shallowest goal-node.
  - Then use Depth-limited search
- Combines benefits of DF- en BF-search

Iterative deepening search

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

inputs: *problem*

for *depth* $\leftarrow$ 0 to ∞ **do**
  result $\leftarrow$ DEPTH-LIMITED_SEARCH(*problem*, *depth*)
  if *result* $\neq$ *cuttoff*
  then return *result*

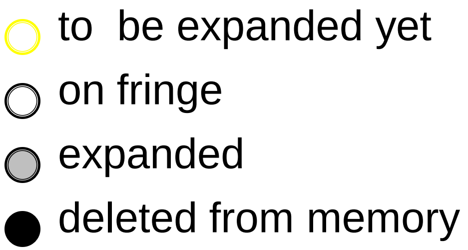
ID-search, example

- to be expanded yet
- on fringe
- expanded
- deleted from memory

- Limit=0

ID-search, example

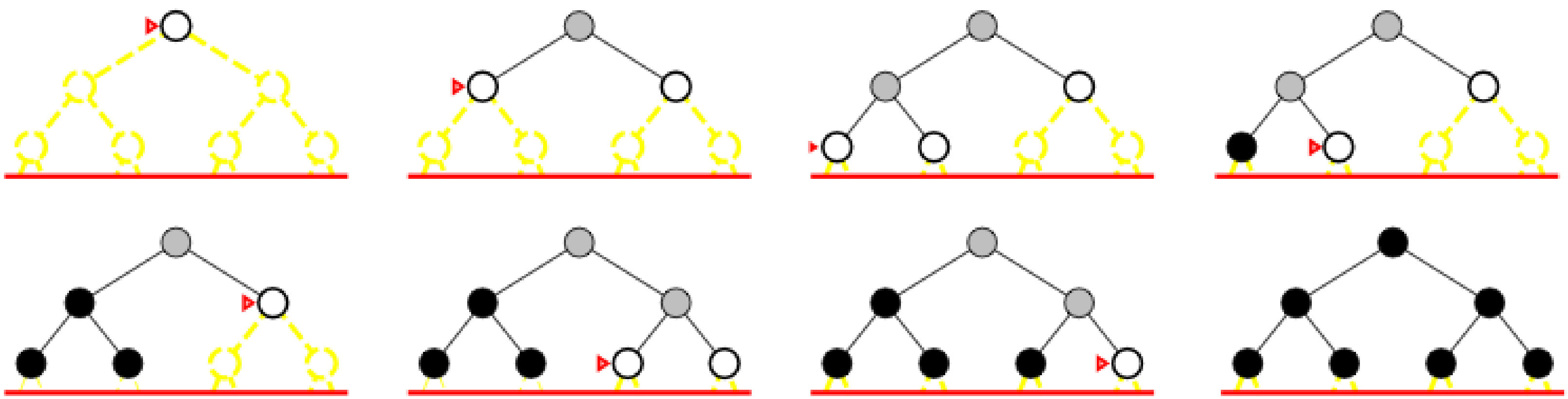
- Limit=1



ID-search, example

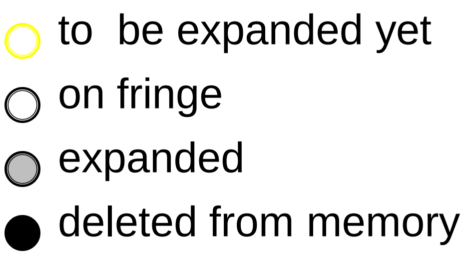
- to be expanded yet
- on fringe
- expanded
- deleted from memory

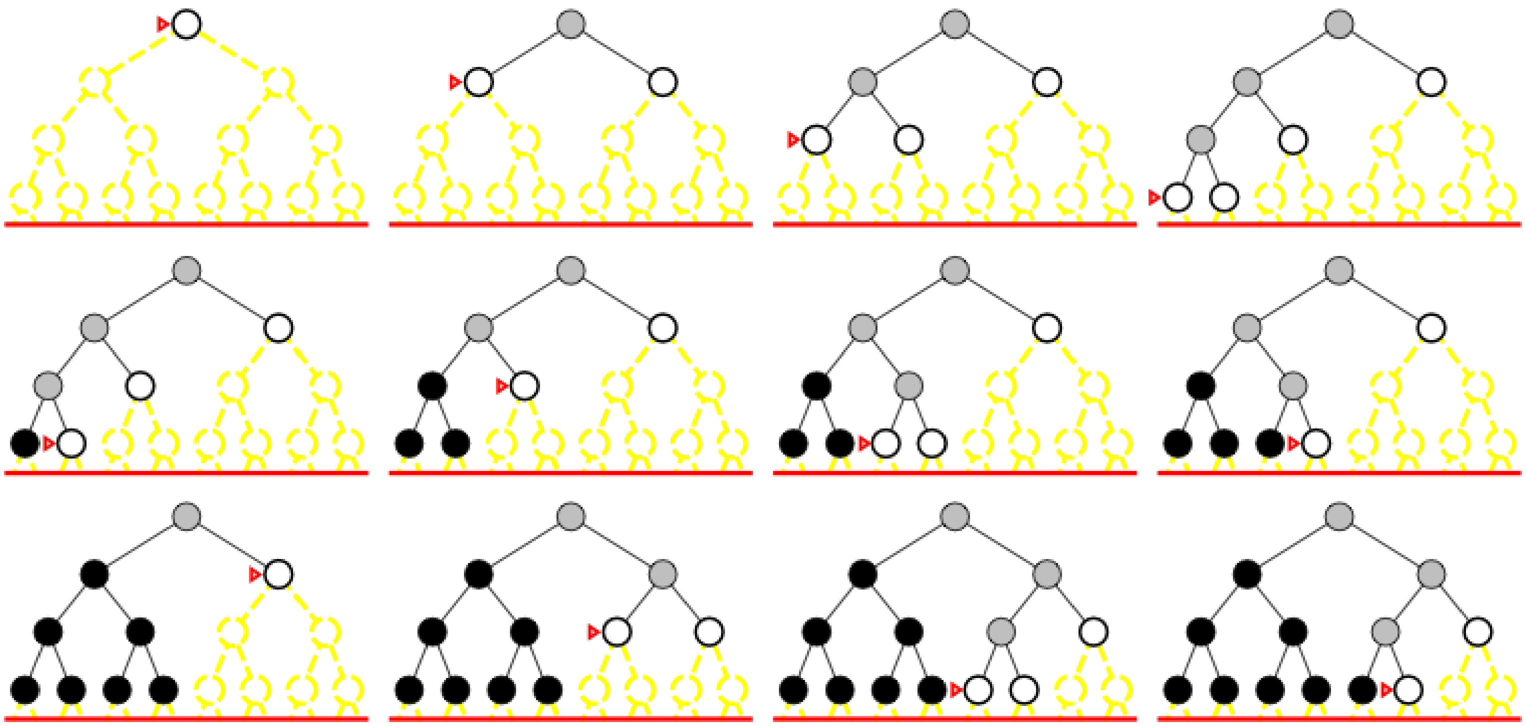
- Limit=2



ID-search, example

- Limit=3





ID search, evaluation

- Completeness:
  - YES (no infinite paths)

ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity:
  - Algorithm seems costly due to repeated generation of certain states.
  - Node generation: $O(b^d)$
    - level d: once
    - level d-1: x 2
    - level d-2: x 3
    - …
    - level 2: x (d-1)
    - level 1: x d

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

$$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

Compare for b=10 and d=5 (solution at far right)

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990$$
$$= 1111100$$

ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity:
  - Cfr. depth-first search $O(bd)$

ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality:
  - YES if step cost is 1.
  - Can be extended to iterative lengthening search
    - Same idea as uniform-cost search
    - Increases overhead.

Bidirectional search

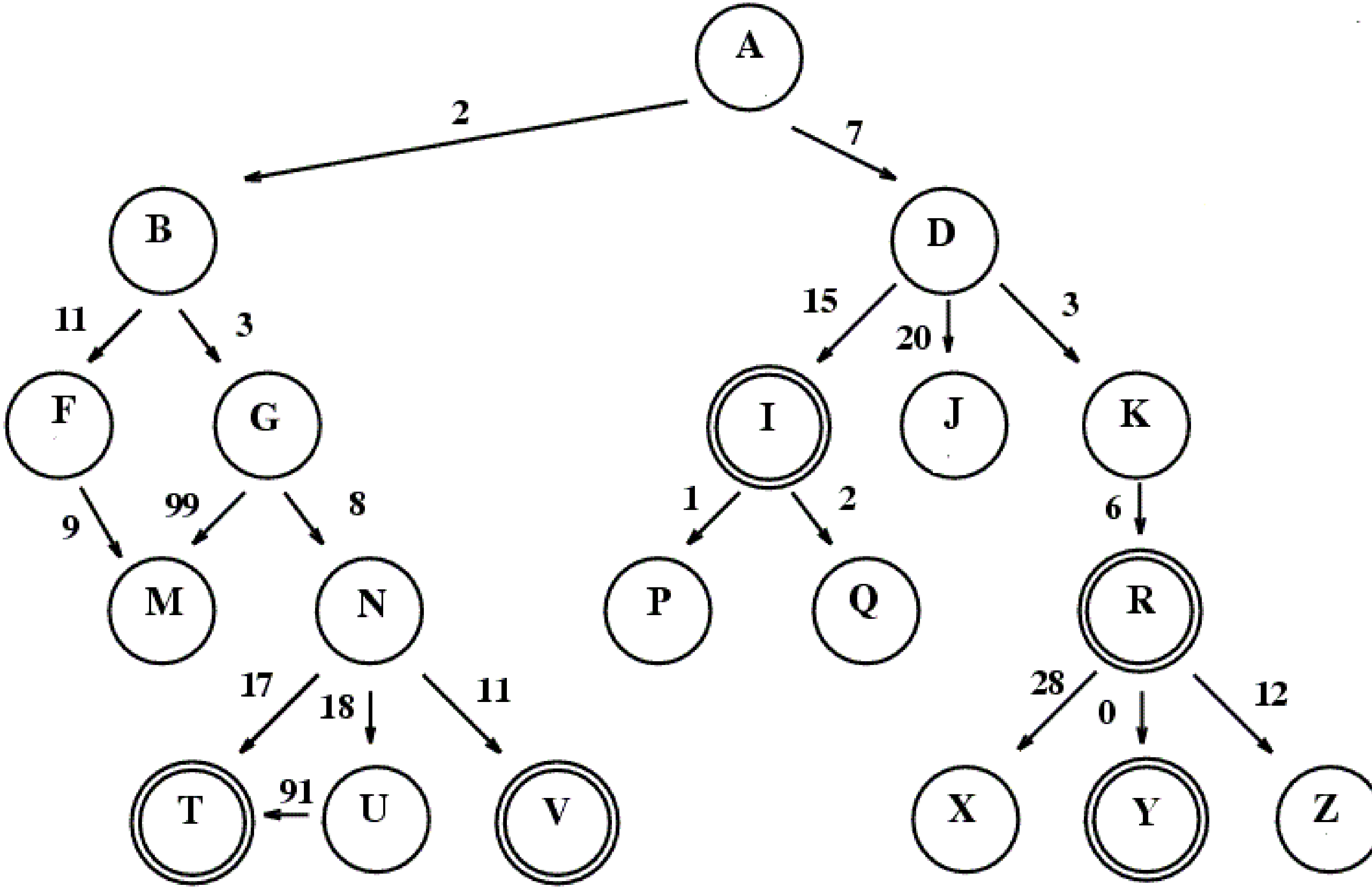
- Completeness:
  - YES, if at least one direction BF-like
- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$
- Optimality:
  - IF …
  - Complexity of checking for a node in the other search tree
  - Doing search „backwards” from the goal
  - …

Summary of algorithms

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

Summary

- The symbols & search paradigm in AI
- Uninformed search
  - Space complexity: OK!
  - Time complexity: exp. ➔ the knowledge paradigm in AI
- Suggested reading
  - Newel & Simon: Computer science as empirical inquiry: symbols and search, 1975
  - Cognitive architectures: ACT-R
    - http://act-r.psy.cmu.edu/
    - http://act-r.psy.cmu.edu/about/
    - Allen Newell describes cognitive architectures as the way to answer one of the ultimate scientific questions: "How can the human mind occur in the physical universe?
      - http://act-r.psy.cmu.edu/misc/newellclip.mpg



A
2
7
B
D
11
3
15
20
3
F
G
I
J
K
9
99
8
1
2
6
M
N
P
Q
R
17
18
11
28
0
12
T
91
U
V
X
Y
Z