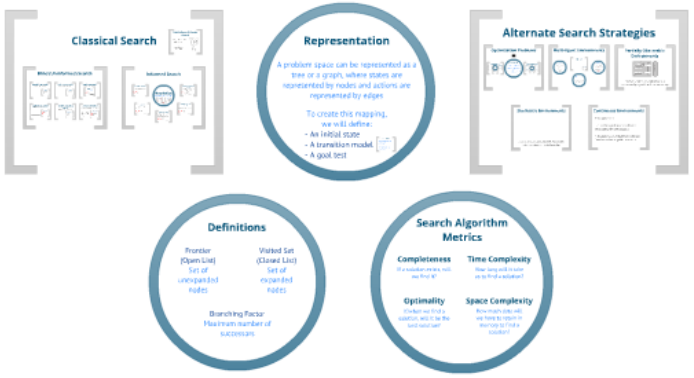
Problem Solving

Strategies for a Goal-Based Agent

Search



Constraint Satisfaction

Environment

Factored state representation.

Portions of state have related values (variables are dependent on each other).

Often used for solving scheduling problems.

Components

We represent a CSP with:
- Variables
- Domains
- Constraints

Example

1	2
3	4
5	6

Insert the letters A-F into spaces 1-6, with the following constraints:
- Only one letter per space
- A is not in row 1, column 2
- A and C are not in the same column
- E and F are in the same row
- C and F are not adjacent (horizontally or vertically)
- A and B are adjacent
- F and D are in the same column

Consistency

Consistency is used to propagate constraints.

Node consistency: One variable
Arc consistency: Two variables
Path consistency: Three variables
k-consistency: k variables

Heuristics

Minimum Remaining Values (MRV): Choose the variable with the smallest domain.

Least Restrictive Values (LRV): Choose the value that least restricts other variables' domains.

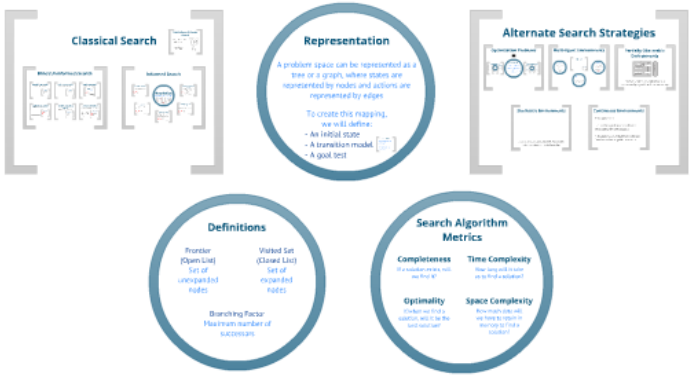
Algorithm

Alternate variable selection and constraint propagation, backtracking when reaching a dead end.

Problem Solving

Strategies for a Goal-Based Agent

Search



Constraint Satisfaction

Environment

Factored state representation.

Portions of state have related values (variables are dependent on each other).

Often used for solving scheduling problems.

Components

We represent a CSP with:
- Variables
- Domains
- Constraints

Example

1	2
3	4
5	6

Insert the letters A-F into spaces 1-6, with the following constraints:
- Only one letter per space
- A is not in row 1, column 2
- A and C are not in the same column
- E and F are in the same row
- C and F are not adjacent (horizontally or vertically)
- A and B are adjacent
- F and D are in the same column

Consistency

Consistency is used to propagate constraints.

Node consistency: One variable
Arc consistency: Two variables
Path consistency: Three variables
k-consistency: k variables

Heuristics

Minimum Remaining Values (MRV): Choose the variable with the smallest domain.

Least Restrictive Values (LRV): Choose the value that least restricts other variables' domains.

Algorithm

Alternate variable selection and constraint propagation, backtracking when reaching a dead end.

Search

Classical Search

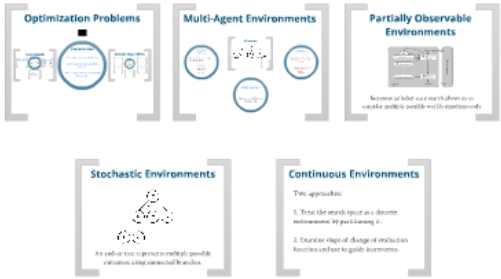
Representation

A problem space can be represented as a tree or a graph, where states are represented by nodes and actions are represented by edges

To create this mapping, we will define:

- An initial state
- A transition model
- A goal test

Alternate Search Strategies



Definitions

Frontier
(Open List)
Set of unexpanded nodes

Visited Set
(Closed List)
Set of expanded nodes

Branching Factor
Maximum number of successors

Search Algorithm Metrics

Completeness
If a solution exists, will we find it?

Time Complexity
How long will it take us to find a solution?

Optimality
If/when we find a solution, will it be the best solution?

Space Complexity
How much data will we have to retain in memory to find a solution?

Representation

A problem space can be represented as a tree or a graph, where states are represented by nodes and actions are represented by edges

To create this mapping,
we will define:

- An initial state
- A transition model
- A goal test

Transition Model

A transition model is a function of a state, that gives us:

* Available actions

* Cost of executing each action

* State resulting from each action (successors / children)

Representation

A problem space can be represented as a tree or a graph, where states are represented by nodes and actions are represented by edges

To create this mapping,
we will define:

- An initial state
- A transition model
- A goal test

Definitions

Frontier
(Open List)
Set of
unexpanded
nodes

Visited Set
(Closed List)
Set of
expanded
nodes

Branching Factor
Maximum number of
successors

Search Algorithm Metrics

Completeness

If a solution exists, will we find it?

Time Complexity

How long will it take us to find a solution?

Optimality

If/when we find a solution, will it be the best solution?

Space Complexity

How much data will we have to retain in memory to find a solution?

Completeness

If a solution exists, will we find it?

Optimality

If/when we find a solution, will it be the best solution?

trics

Time Complexity

How long will it take us to find a solution?

Space Complexity

How much data will we have to retain in memory to find a solution?

Classical Search

Limitations of Classic Search

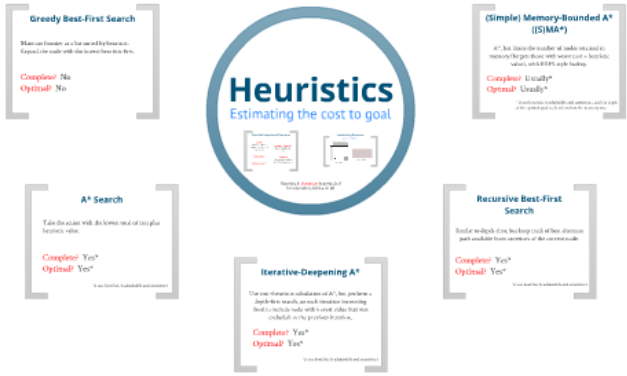
Goal state must be easily recognized

Search space is manageable

Environment is:
* Observable
* Deterministic
* Single-agent
* Discrete
* Static
* Known

Blind (Uninformed) Search

Informed Search



Limitations of Classic Search

Goal state must be easily recognized

Search space is manageable

Environment is:

* Observable
* Deterministic
* Single-agent
* Discrete
* Static
* Known

Searc

Goal state must be easily recognized

easily recognized

Search space is manageable

be

ed

s

Environment is:

* Observable

* Deterministic

* Single-agent

* Discrete

* Static

* Known

Blind (Uninformed) Search

Breadth-First Search

Maintain a FIFO frontier queue

Nodes are goal-tested as they are generated

Attributes

Breadth-First Tree search

Complete? Yes
Optimal? Only if depth == cost
Time? $O(b^d)$
Space? $O(b^d)$

b = branching factor
d = depth of first solution

Uniform-Cost Search

Maintain an ordered frontier queue

Nodes are goal-tested as they are selected for expansion

Attributes

Uniform-Cost Tree Search

Complete? Yes
Optimal? Yes, assuming $\varepsilon > 0$
Time? $O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$
Space? $O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$

b = branching factor
C^* = cost of optimal solution
ε = minimum transition cost

Depth-First Search

Maintain a LIFO frontier queue

Attributes

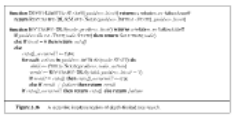
Depth-First Tree Search

Complete? No
Optimal? No
Time? $O(b^m)$
Space? $O(bm)$

b = branching factor
m = maximum tree depth

Depth-Limited Search

Depth-first with a maximum depth



Attributes

Depth-Limited Tree search

Complete? No
Optimal? No
Time? $O(b^l)$
Space? $O(bl)$

b = branching factor
l = depth limit

Iterative Deepening Search

Depth-limited with an increasing depth

Attributes

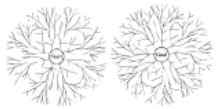
Iterative Deepening Tree Search

Complete? Yes
Optimal? Yes, if depth == cost
Time? $O(b^d)$
Space? $O(bd)$

b = branching factor
d = depth of first solution

Bidirectional Search

Simultaneous breadth-first search from goal and initial state



Attributes

Bi-directional Breadth-First Search

Complete? Yes
Optimal? Yes, if depth == cost
Time? $O(b^{d/2})$
Space? $O(b^{d/2})$

b = branching factor
d = depth of first solution

Breadth-First Search

Maintain a FIFO frontier queue

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Nodes are goal-tested as they are generated

Attributes

Breadth-First Tree search

Complete? Yes
Optimal? Only if depth == cost
Time? $O(b^d)$
Space? $O(b^d)$

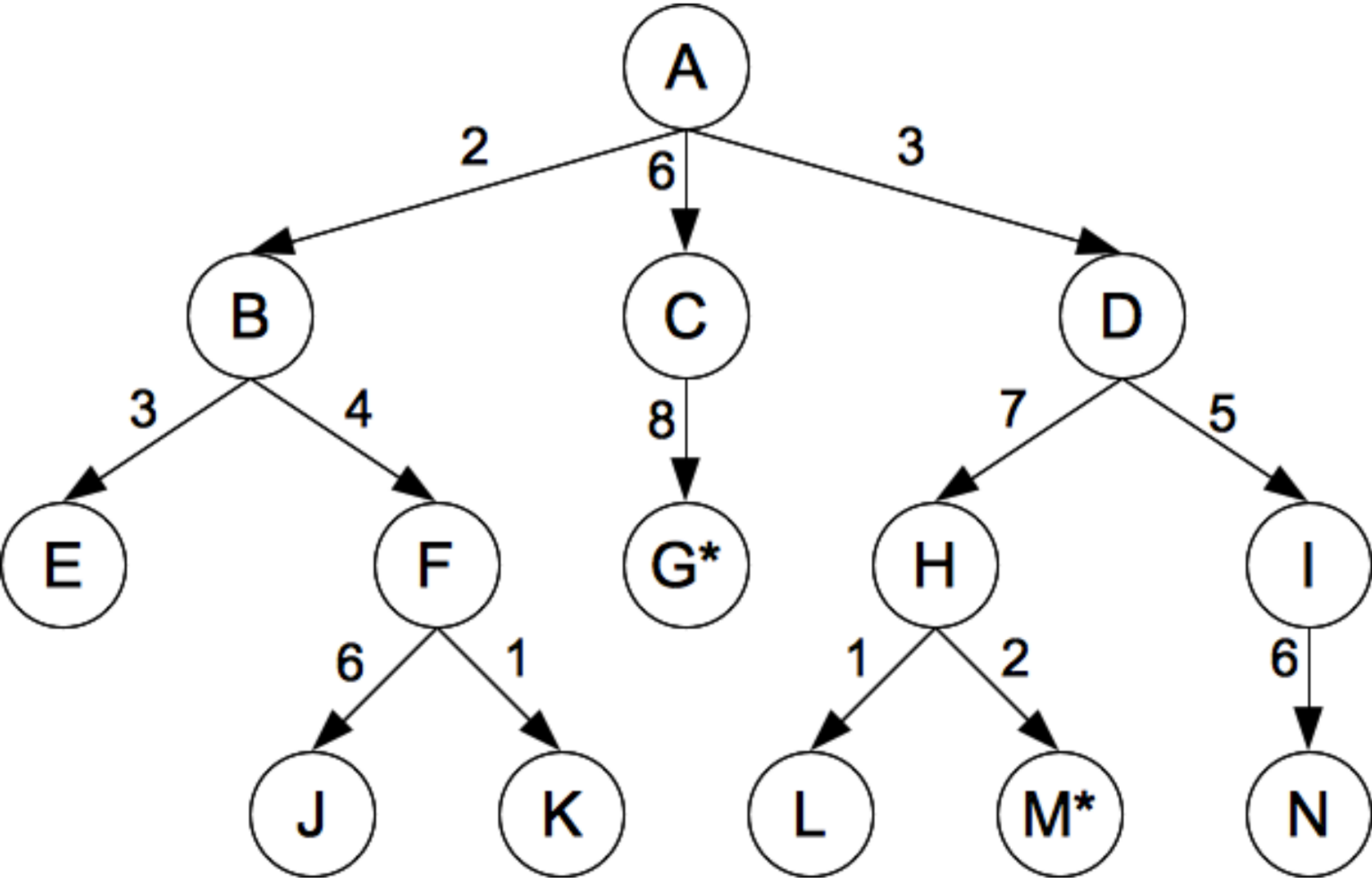
b = branching factor
d = depth of first solution

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Nodes are goal-tested as they are generated


A
2
6
3
B
C
D
3
4
8
7
5
E
F
G*
H
I
6
1
1
2
6
J
K
L
M*
N

Attributes

Breadth-First Tree search

Complete? Yes

Optimal? Only if depth == cost

Time? $O(b^d)$

Space? $O(b^d)$

b = branching factor

d = depth of first solution

Uniform-Cost Search

Maintain an ordered frontier queue

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure **??**, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Nodes are goal-tested as they are selected for expansion

Attributes

Uniform-Cost Tree Search

Complete? Yes

Optimal? Yes, assuming $\varepsilon > 0$

Time? $O(b^{1+\lceil \frac{C^*}{\varepsilon} \rceil})$

Space? $O(b^{1+\lceil \frac{C^*}{\varepsilon} \rceil})$

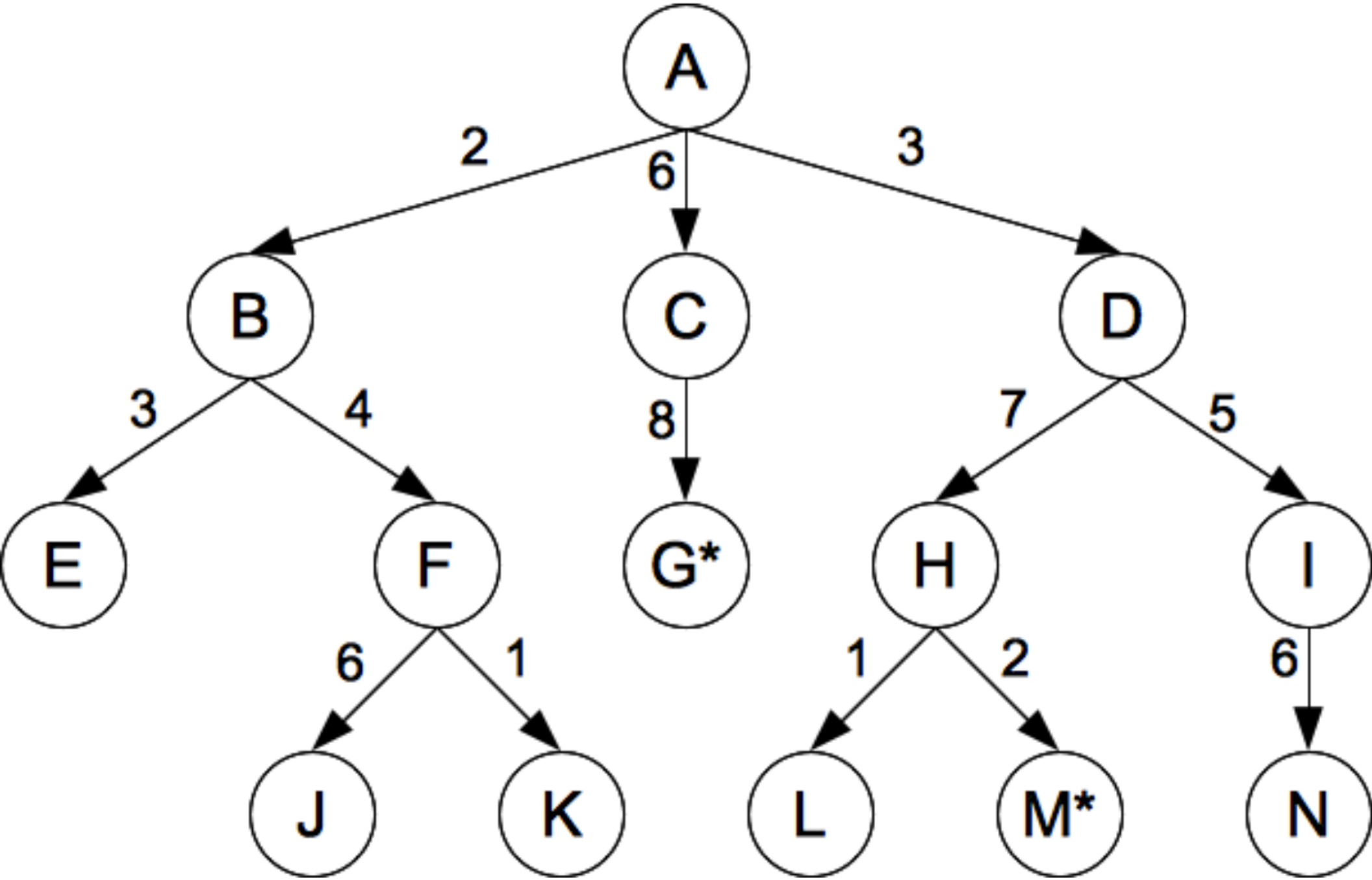
b = branching factor
C^* = cost of optimal solution
ε = minimum transition cost

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure **??**, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Nodes are goal-tested as they are selected for expansion


A
2
6
3
B
C
D
3
4
8
7
5
E
F
G*
H
I
6
1
1
2
6
J
K
L
M*
N

Attributes

Uniform-Cost Tree Search

Complete? Yes

Optimal? Yes, assuming $\varepsilon > 0$

Time? $O(b^{1+\left\lceil \frac{C^*}{\varepsilon} \right\rceil})$

Space? $O(b^{1+\left\lceil \frac{C^*}{\varepsilon} \right\rceil})$

b = branching factor

C^* = cost of optimal solution

ε = minimum transition cost

Depth-First Search

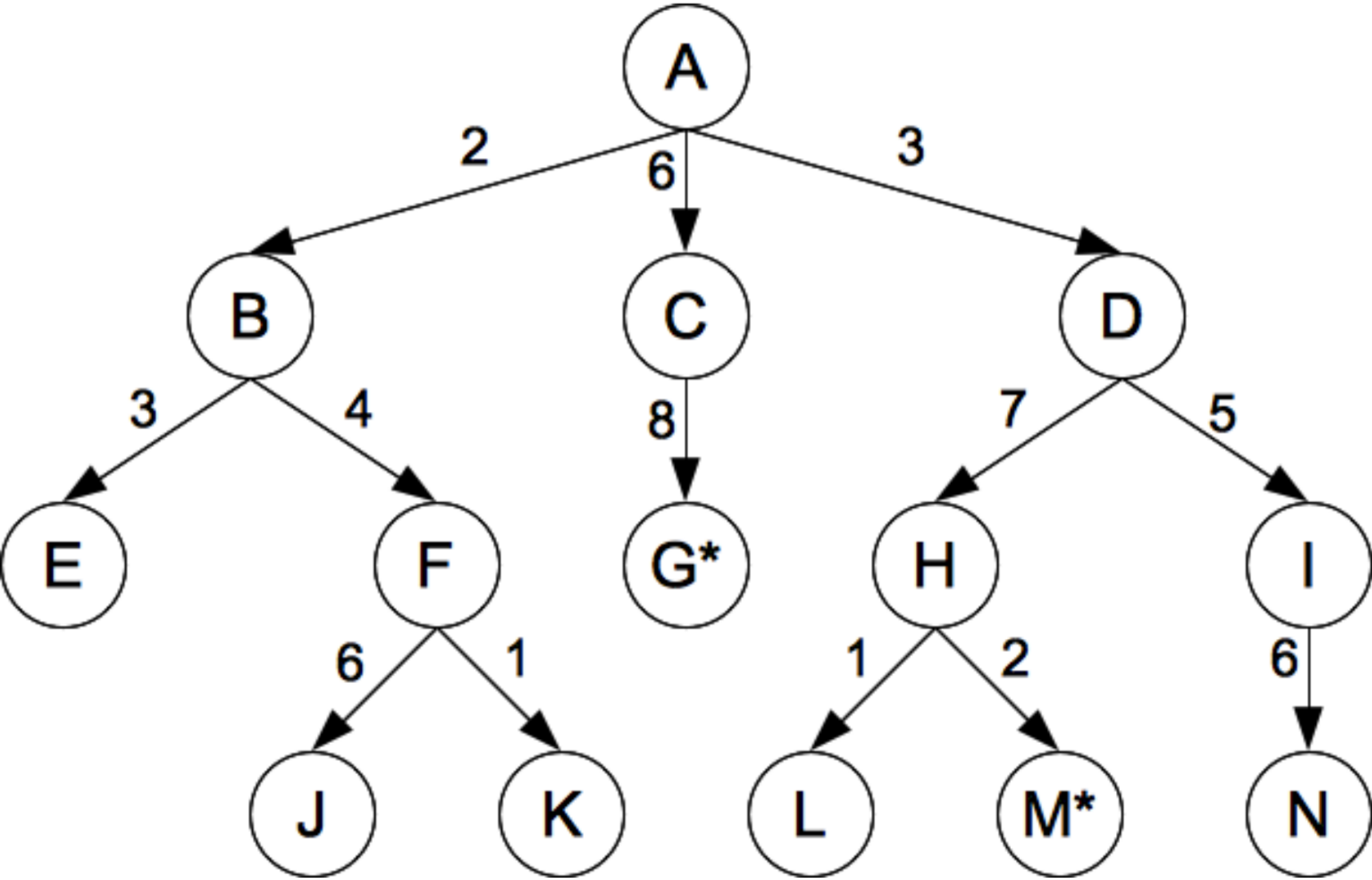
Maintain a LIFO frontier queue

Attributes

Depth-First Tree Search

Complete? No
Optimal? No
Time? $O(b^m)$
Space? $O(bm)$

b = branching factor
m = maximum tree depth


A
2
6
3
B
C
D
3
4
8
7
5
E
F
G*
H
I
6
1
1
2
6
J
K
L
M*
N

Attributes

Depth-First Tree Search

Complete? No

Optimal? No

Time? $O(b^m)$

Space? $O(bm)$

b = branching factor

m = maximum tree depth

Depth-Limited Search

Depth-first with a maximum depth

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

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.16 A recursive implementation of depth-limited tree search.

Attributes

Depth-Limited Tree search

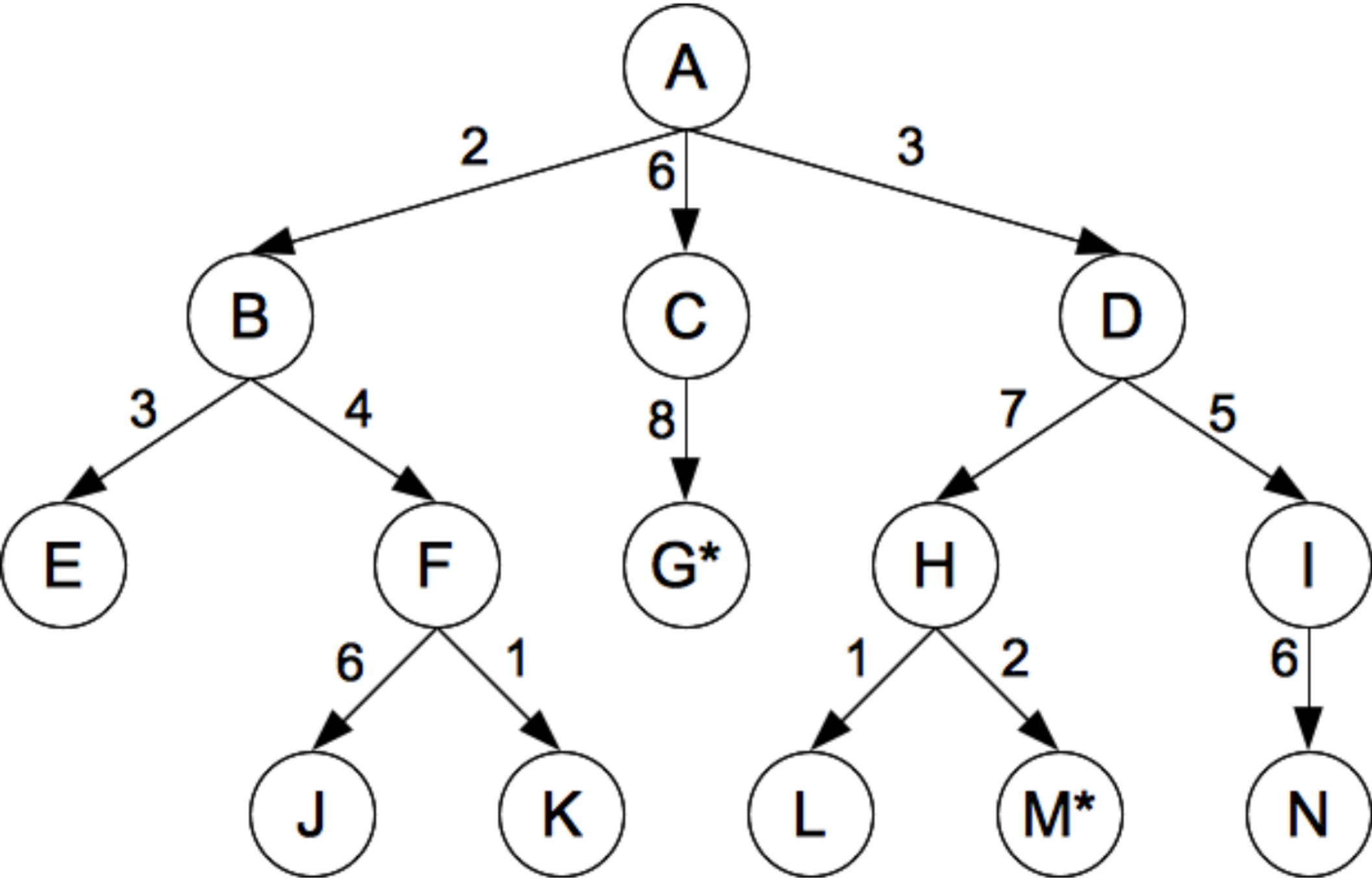
Complete? No
Optimal? No
Time? $O(b^l)$
Space? $O(bl)$

b = branching factor
l = depth limit

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

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.16 A recursive implementation of depth-limited tree search.


A
2
6
3
B
C
D
3
4
8
7
5
E
F
G*
H
I
6
1
1
2
6
J
K
L
M*
N

Attributes

Depth-Limited Tree search

Complete? No

Optimal? No

Time? $O(b^l)$

Space? $O(bl)$

b = branching factor

l = depth limit

Iterative Deepening Search

Depth-limited with an increasing depth

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.17 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Attributes

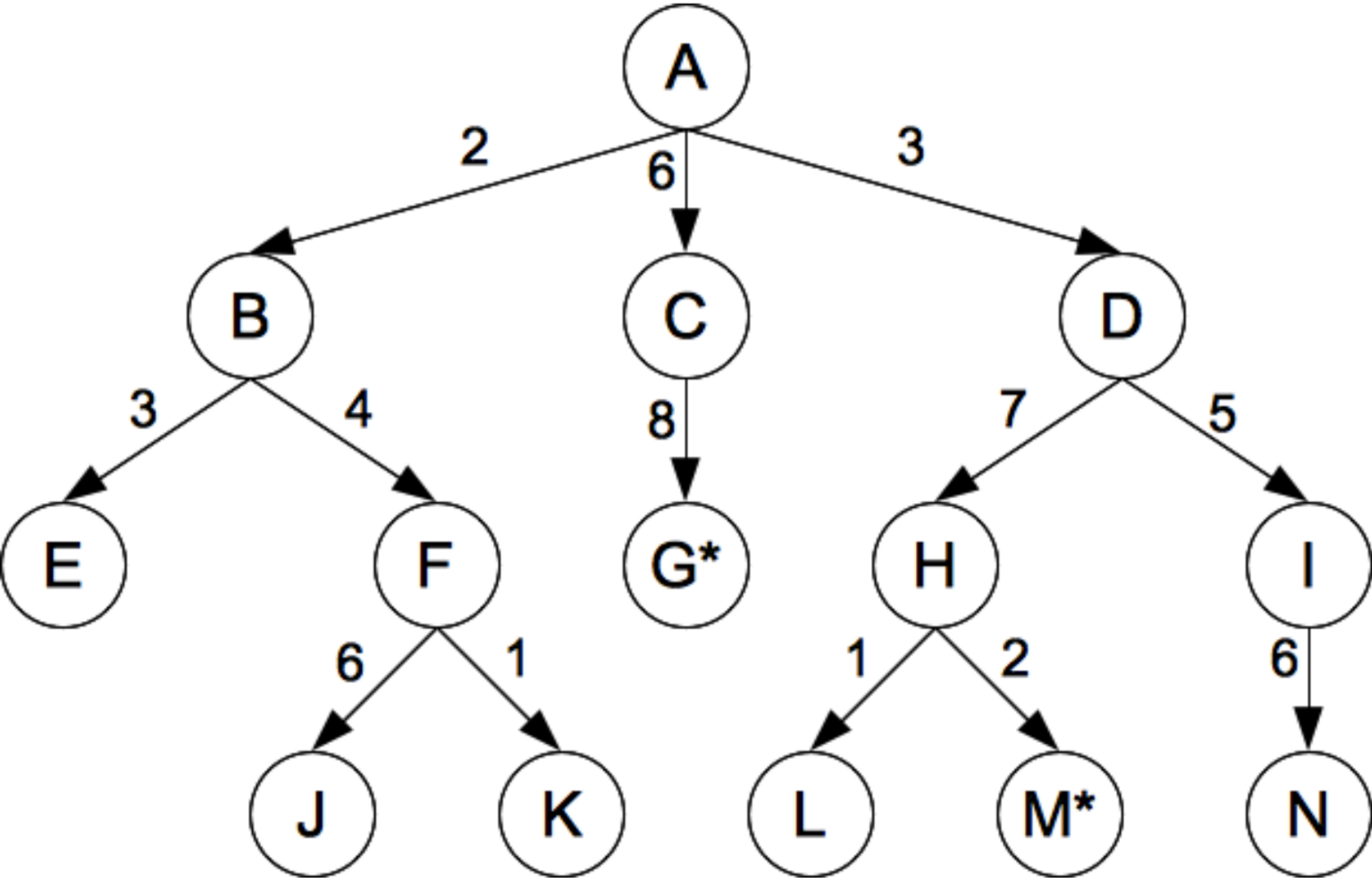
Iterative Deepening Tree Search

Complete? Yes
Optimal? Yes, if depth == cost
Time? $O(b^d)$
Space? $O(bd)$

b = branching factor
d = depth of first solution

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.17 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.


A
2
6
3
B
C
D
3
4
8
7
5
E
F
G*
H
I
6
1
1
2
6
J
K
L
M*
N

Attributes

Iterative Deepening Tree Search

Complete? Yes

Optimal? Yes, if depth == cost

Time? $O(b^d)$

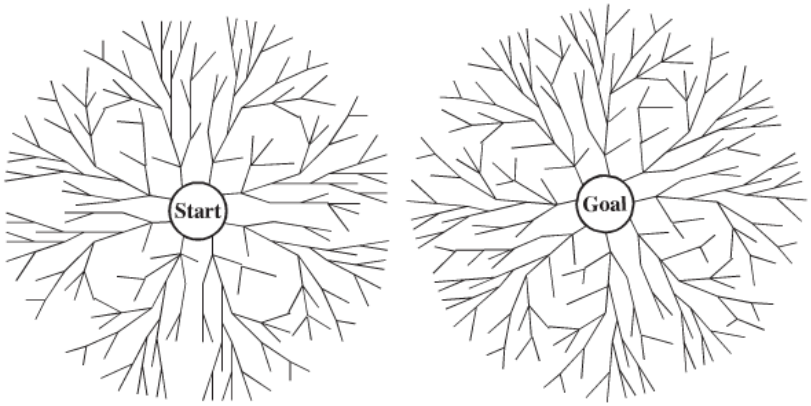
Space? $O(bd)$

b = branching factor

d = depth of first solution

Bidirectional Search

Simultaneous breadth-first search from goal and initial state

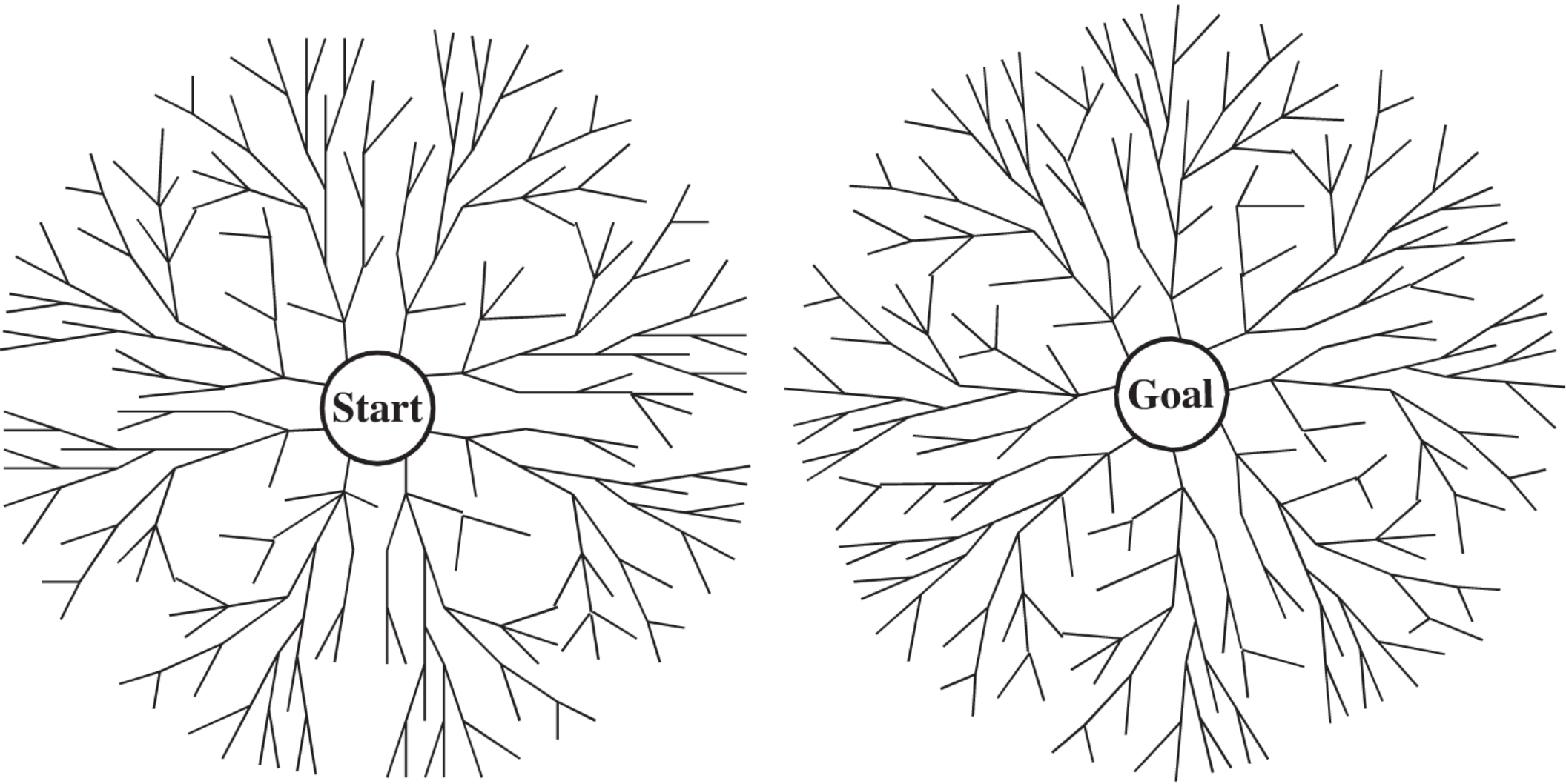


Attributes

Bi-directional Breadth-First Search

Complete? Yes
Optimal? Yes, if depth == cost
Time? $O(b^{d/2})$
Space? $O(b^{d/2})$

b = branching factor
d = depth of first solution


Start
Goal

Attributes

Bi-directional Breadth-First Search

Complete? Yes

Optimal? Yes, if depth == cost

Time? $O(b^{d/2})$

Space? $O(b^{d/2})$

b = branching factor

d = depth of first solution

Classical Search

Limitations of Classic Search

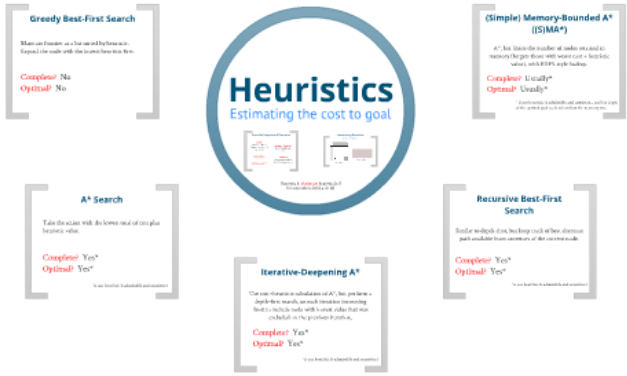
Goal state must be easily recognized

Search space is manageable

Environment is:
* Observable
* Deterministic
* Single-agent
* Discrete
* Static
* Known

Blind (Uninformed) Search

Informed Search



Informed Search

Greedy Best-First Search

Maintain frontier as a list sorted by heuristic.
Expand the node with the lowest heuristic first.

Complete? No
Optimal? No

Heuristics

Estimating the cost to goal

Heuristic h_1 *dominates* heuristic h_2 if for any node n, $h_1(n) \geq h_2(n)$

(Simple) Memory-Bounded A* ((S)MA*)

A*, but limits the number of nodes retained in memory (forgets those with worst cost + heuristic value), with RBFS-style backup.

Complete? Usually*
Optimal? Usually*

* if our heuristic is admissible and consistent, and the depth of the optimal goal node is less than the memory size.

A* Search

Take the action with the lowest total of cost plus heuristic value.

Complete? Yes*
Optimal? Yes*

*if our heuristic is admissible and consistent

Recursive Best-First Search

Similar to depth-first, but keep track of best alternate path available from ancestors of the current node.

Complete? Yes*
Optimal? Yes*

*if our heuristic is admissible and consistent

Iterative-Deepening A*

Use cost+heuristic calculation of A*, but perform a depth-first search, on each iteration increasing limit to include node with lowest value that was excluded on the previous iteration.

Complete? Yes*
Optimal? Yes*

*if our heuristic is admissible and consistent

Heuristics

Estimating the cost to goal

Desirable Properties of Heuristics

Accurate
Strong correlation between heuristic and actual result

Admissible (Optimistic)
Never overestimates costs

Efficient (time)

Consistent
Differences in heuristics should also be optimistic

Efficient (memory)

Constructing Heuristics

Constraint relaxation

Maze Solver

Tower of Hanoi

Heuristic h_1 *dominates* heuristic h_2 if
for any node n, $h_1(n) \geq h_2(n)$

Desirable Properties of Heuristics

Accurate

Strong correlation between heuristic and actual result

Admissible (Optimistic)

Never overestimates costs

Efficient (time)

Consistent

Differences in heuristics should also be optimistic

Efficient (memory)

Accurate

Strong correlation between heuristic and actual result

Efficient (time)

Efficient (memory)

Admissible (Optimistic)

Never overestimates costs

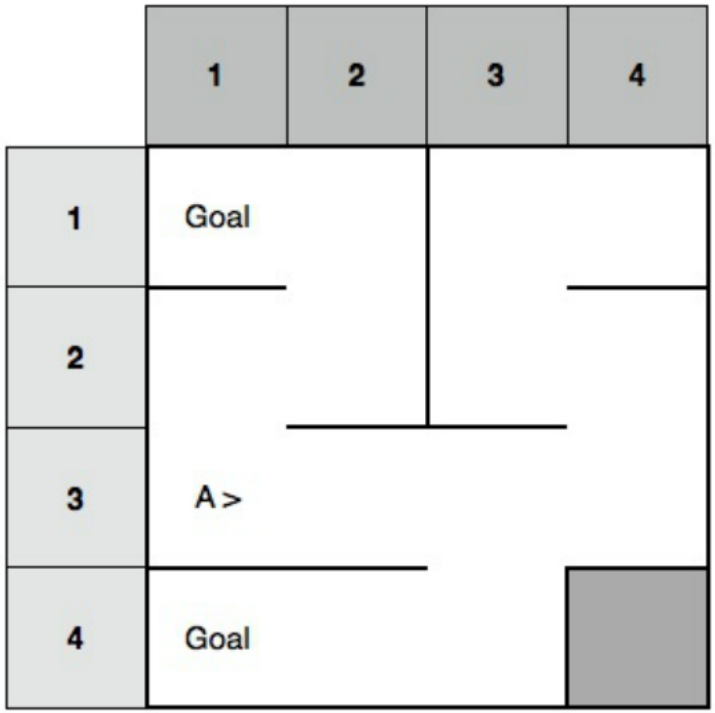
costs

Consistent

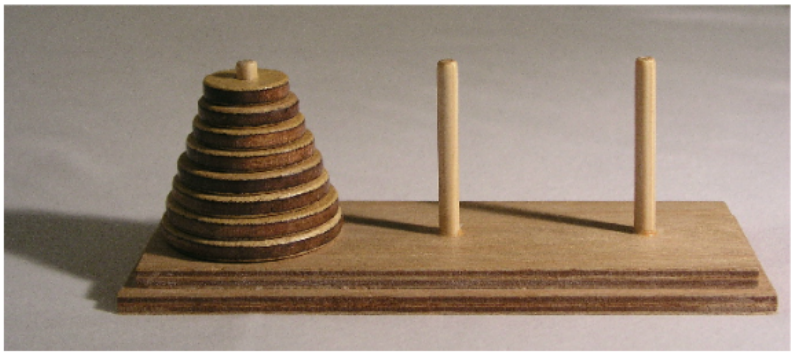
Differences in heuristics should also be optimistic

Constructing Heuristics

Constraint relaxation



Maze Solver



Tower of Hanoi

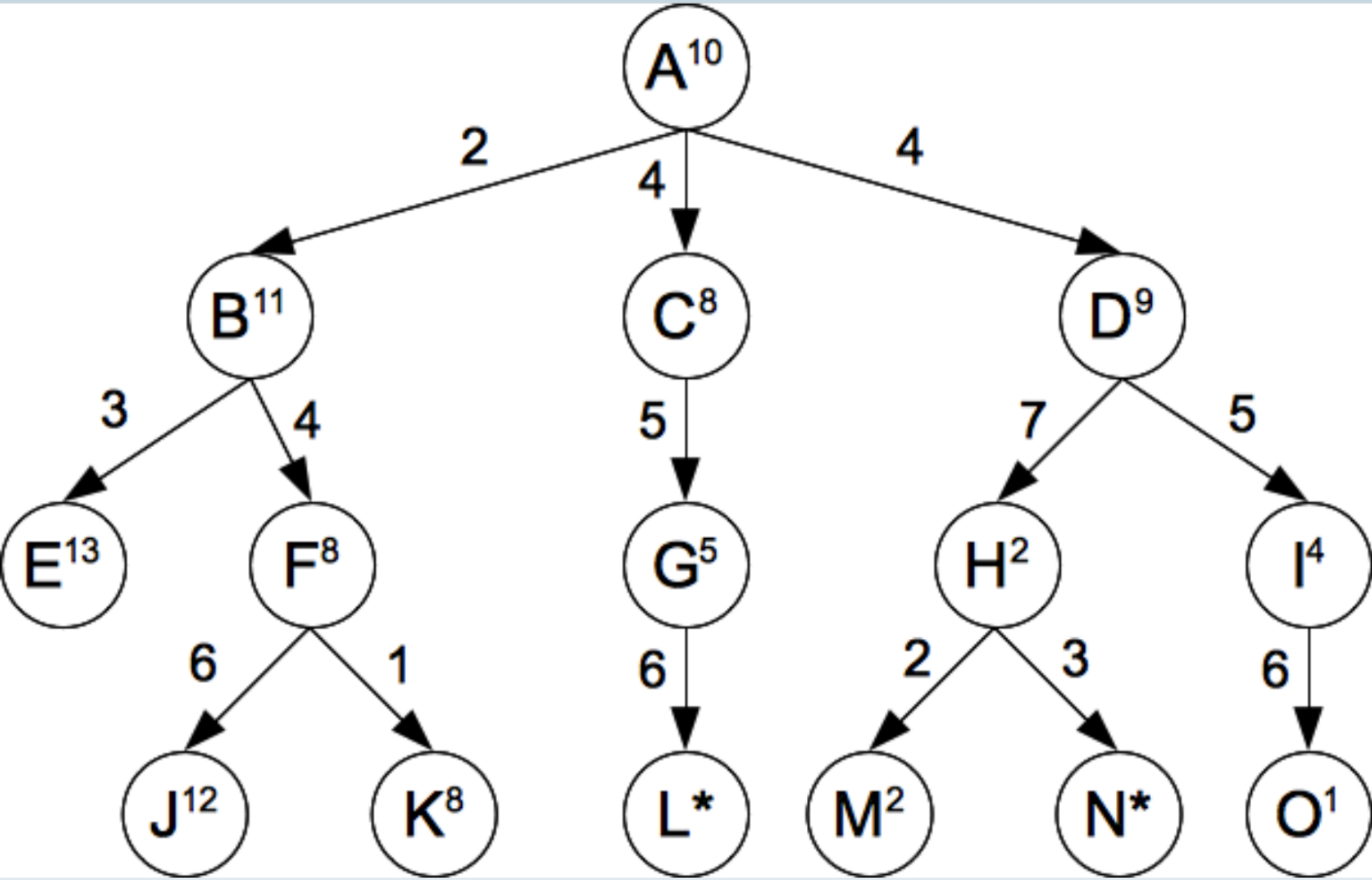
Heuristic h_1 *dominates* heuristic h_2 if for any node n, $h_1(n) \geq h_2(n)$

Greedy Best-First Search

Maintain frontier as a list sorted by heuristic.
Expand the node with the lowest heuristic first.

Complete? No

Optimal? No


A10
2
4
4
B11
C8
D9
3
4
5
7
5
E13
F8
G5
H2
I4
6
1
6
2
3
6
J12
K8
L*
M2
N*
O1

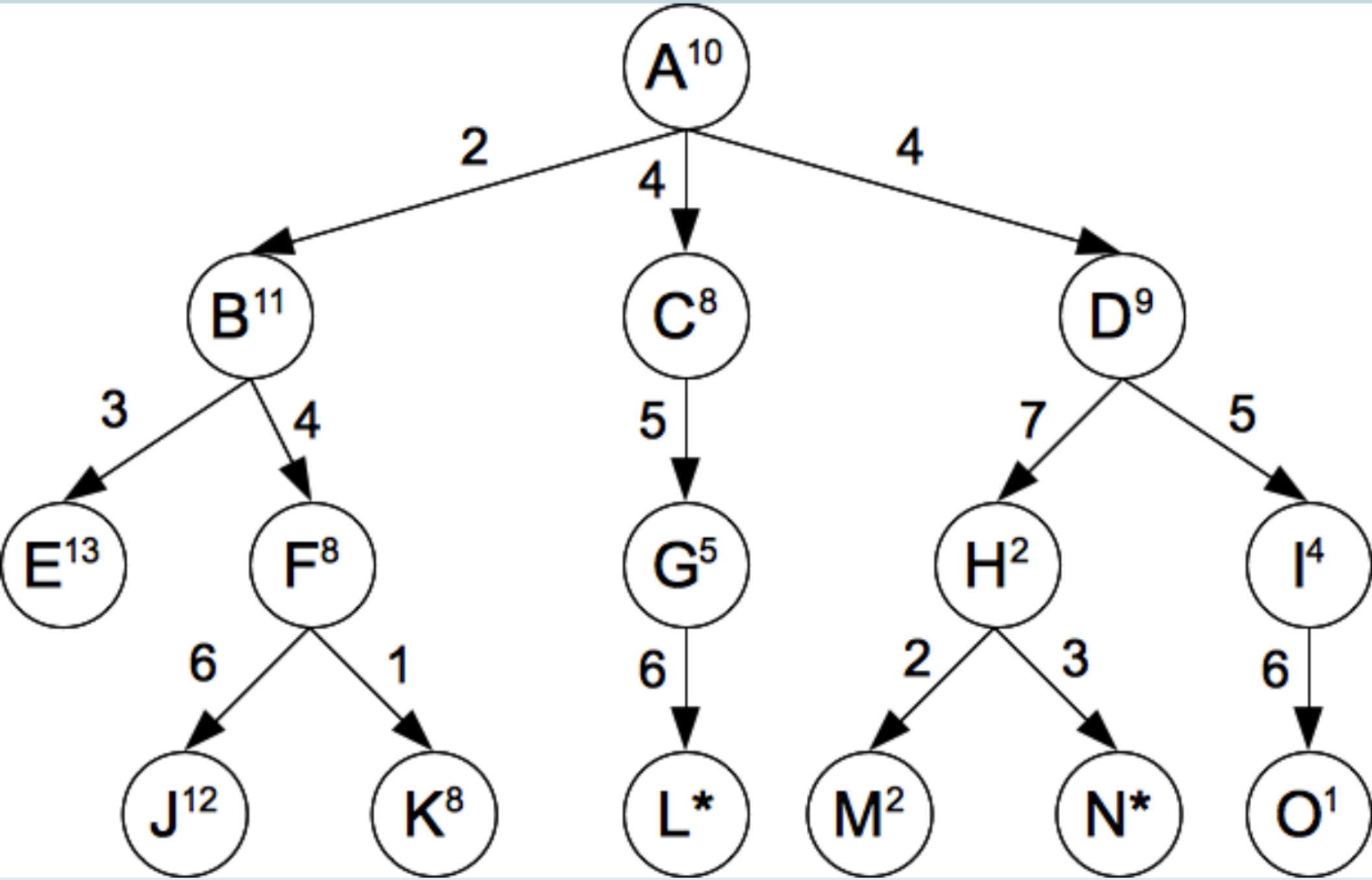
A* Search

Take the action with the lowest total of cost plus heuristic value.

Complete? Yes*

Optimal? Yes*

*if our heuristic is admissible and consistent


A10
2
4
4
B11
C8
D9
3
4
5
7
5
E13
F8
G5
H2
I4
6
1
6
2
3
6
J12
K8
L*
M2
N*
O1

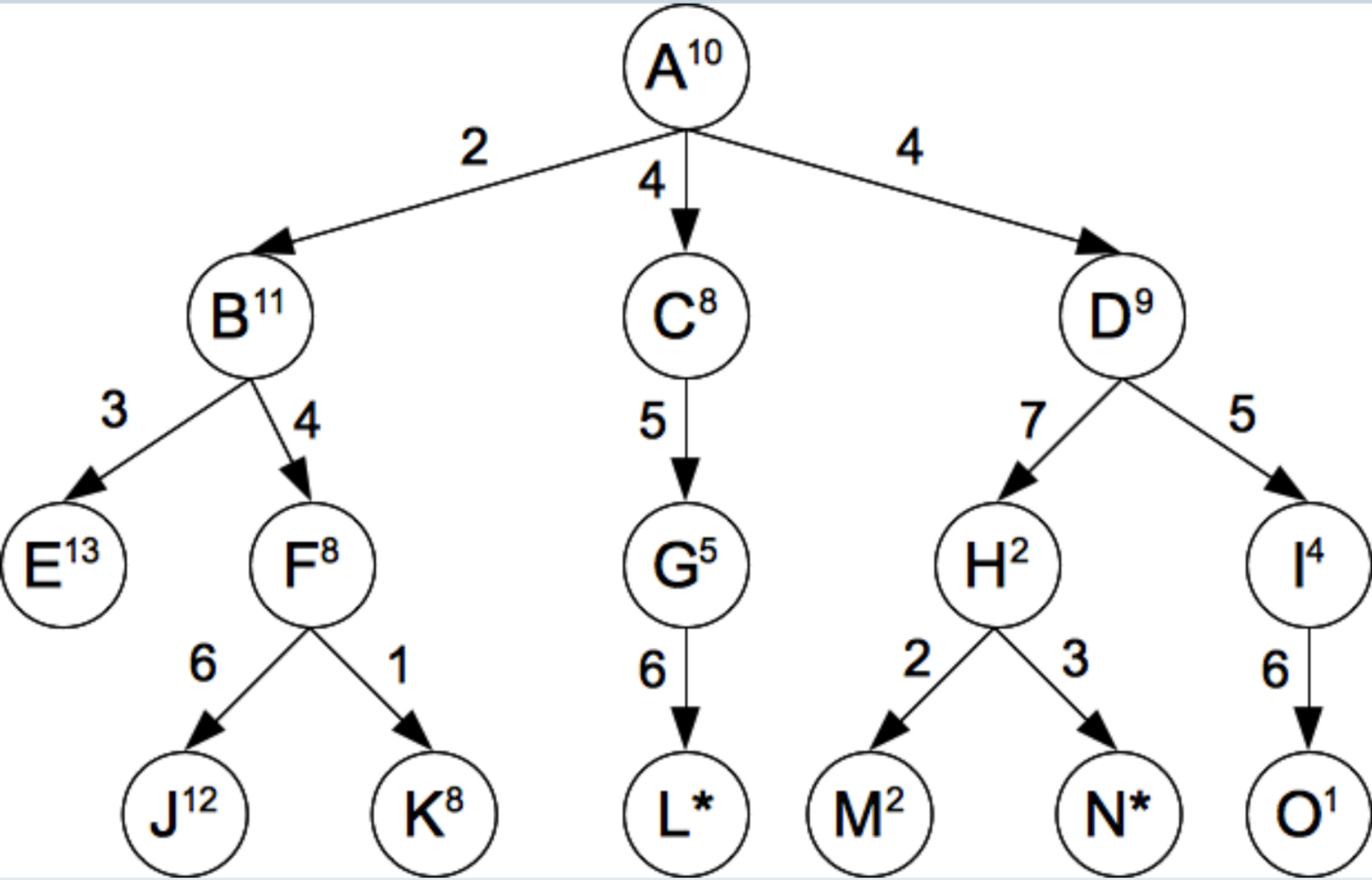
Iterative-Deepening A*

Use cost+heuristic calculation of A*, but perform a depth-first search, on each iteration increasing limit to include node with lowest value that was excluded on the previous iteration.

Complete? Yes*

Optimal? Yes*

*if our heuristic is admissible and consistent


A10
2
4
4
B11
C8
D9
3
4
5
7
5
E13
F8
G5
H2
I4
6
1
6
2
3
6
J12
K8
L*
M2
N*
O1

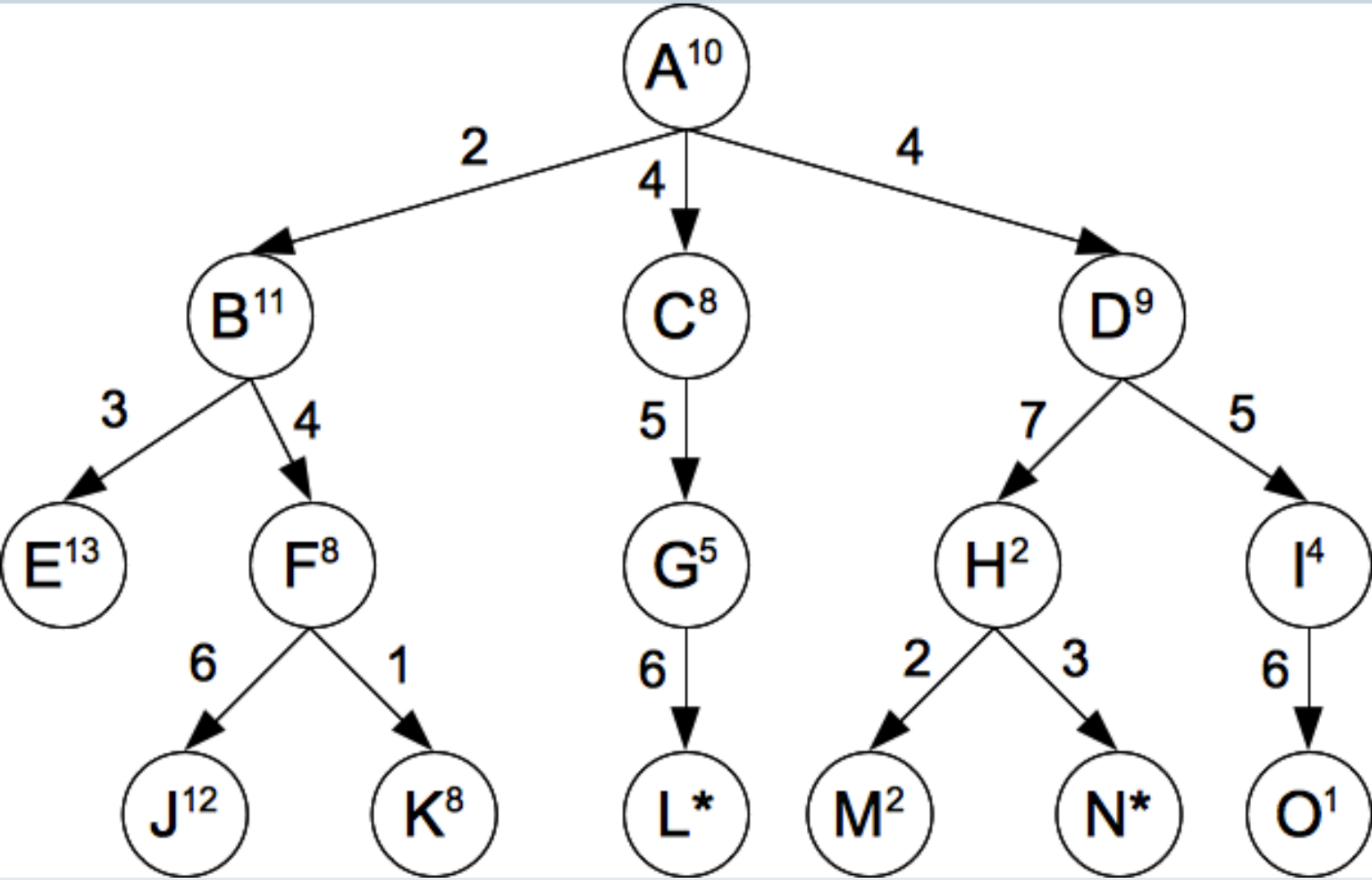
Recursive Best-First Search

Similar to depth-first, but keep track of best alternate path available from ancestors of the current node.

Complete? Yes*

Optimal? Yes*

*if our heuristic is admissible and consistent


A10
2
4
4
B11
C8
D9
3
4
5
7
5
E13
F8
G5
H2
I4
6
1
6
2
3
6
J12
K8
L*
M2
N*
O1

(Simple) Memory-Bounded A* ((S)MA*)

A*, but limits the number of nodes retained in memory (forgets those with worst cost + heuristic value), with RBFS-style backup.

Complete? Usually*

Optimal? Usually*

* if our heuristic is admissible and consistent, and the depth of the optimal goal node is less than the memory size.

Limitations of Classic Search

Goal state must be easily recognized

Search space is manageable

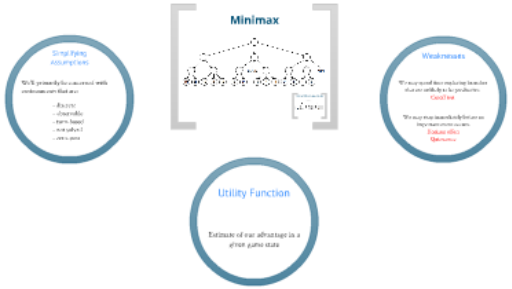
Environment is:

* Observable
* Deterministic
* Single-agent
* Discrete
* Static
* Known

Alternate Search Strategies

Optimization Problems

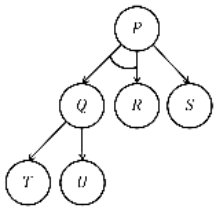
Multi-Agent Environments



Partially Observable Environments

Incremental belief state search allows us to consider multiple possible worlds simultaneously

Stochastic Environments



An and-or tree represents multiple possible outcomes using connected branches.

Continuous Environments

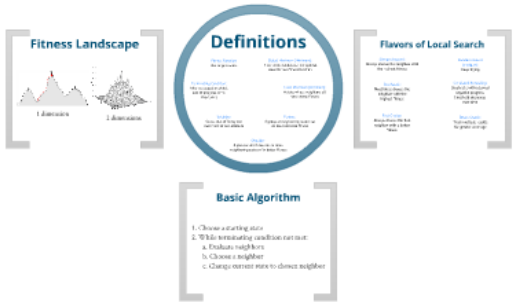
Two approaches:

1. Treat the search space as a discrete environment by partitioning it.

2. Examine slope of change of evaluation function and use to guide increments.

Optimization Problems

Local Search



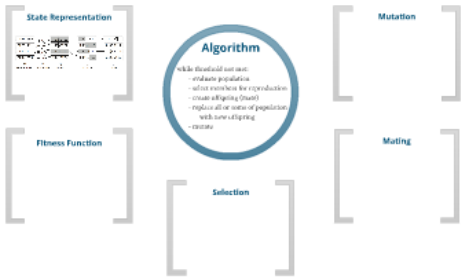
Characteristics

- Goal state is not clearly defined

- Search space is vast, potentially infinite

- Path to reach solution is generally unimportant

Genetic Algorithms




Friends
YouTube

Characteristics

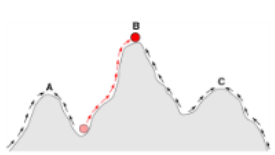
- Goal state is not clearly defined

- Search space is vast, potentially infinite

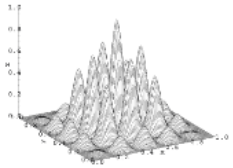
- Path to reach solution is generally unimportant

Local Search

Fitness Landscape



1 dimension



2 dimensions

Definitions

Fitness Function:
Our target metric

Global Maximum (Minimum):
In our entire landscape, the optimal value for our fitness function

Terminating Condition:
When to stop (threshold, lack of progress, time, steps, etc.)

Local Maximum (Minimum):
A state whose neighbors all have worse fitness

Neighbor:
States that differ by one increment of one attribute

Plateau:
A group of neighboring state that all have identical fitness

Shoulder:
A plateau which has one or more neighboring states with better fitness

Flavors of Local Search

Steepest-ascent:
Always choose the neighbor with the highest fitness

Random restart (shotgun):
Keep trying

Stochastic:
Most likely choose the neighbor with the highest fitness

Simulated Annealing:
Stochastic with allowed negative progress, threshold decreases over time

First Choice:
Always choose the first neighbor with a better fitness

Beam Search:
Track multiple results for greater coverage

Basic Algorithm

1. Choose a starting state
2. While terminating condition not met:
   a. Evaluate neighbors
   b. Choose a neighbor
   c. Change current state to chosen neighbor

Definitions

Fitness Function:
Our target metric

Global Maximum (Minimum):
In our entire landscape, the optimal value for our fitness function

Terminating Condition:
When to stop (threshold, lack of progress, time, steps, etc.)

Local Maximum (Minimum):
A state whose neighbors all have worse fitness

Neighbor:
States that differ by one increment of one attribute

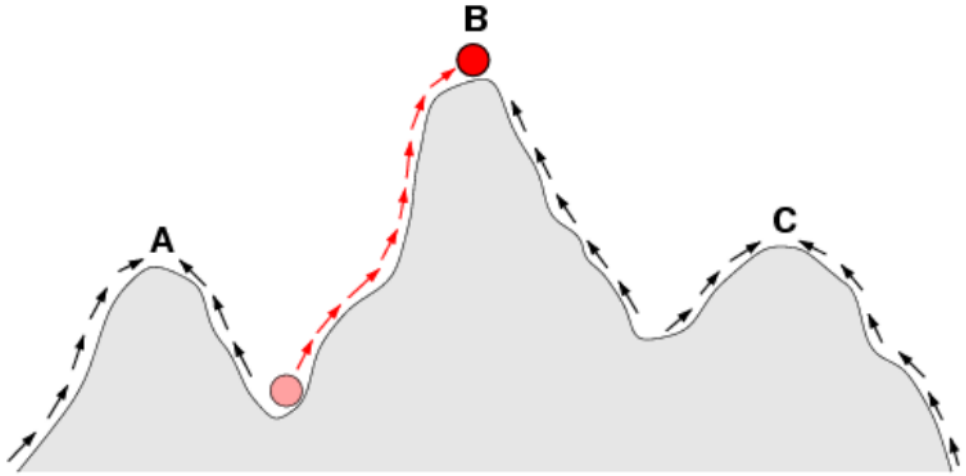
Plateau:
A group of neighboring state that all have identical fitness

Shoulder:
A plateau which has one or more neighboring states with better fitness

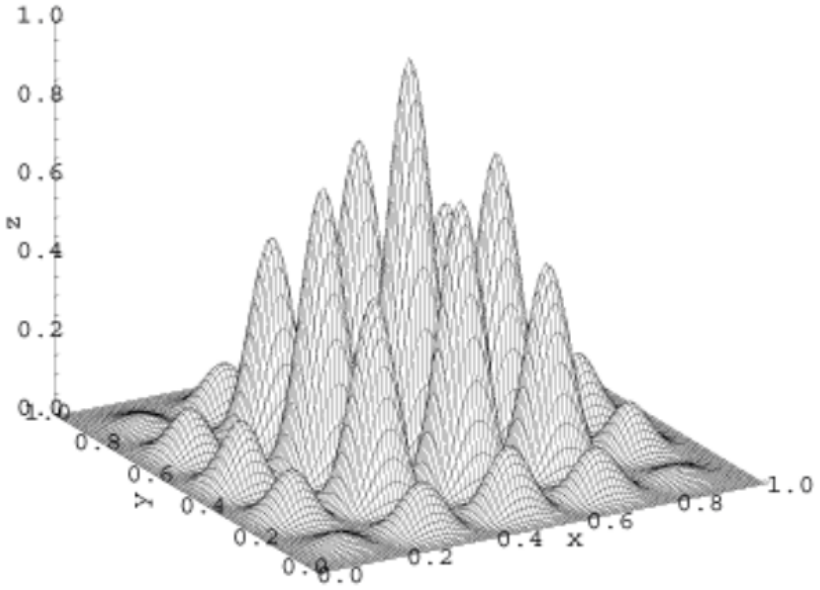
Fitness Function:

Our target metric

Fitness Landscape



1 dimension



2 dimensions

itions

Global Maximum (Minimum):

In our entire landscape, the optimal value for our fitness function

Local Maximum (Minimum

Terminating Condition:

When to stop (threshold, lack of progress, time, steps, etc.)

Neighbor:

States that differ by one increment of one attribute

Basic Algorithm

1. Choose a starting state
2. While terminating condition not met:
    a. Evaluate neighbors
    b. Choose a neighbor
    c. Change current state to chosen neighbor

for our fitness function

Local Maximum (Minimum):

A state whose neighbors all have worse fitness

Plateau:

A group of neighboring state that all have identical fitness

lder:

differ by one
one attribute

A group of neighbo
all have identi

Shoulder:

A plateau which has one or more neighboring states with better fitness

Flavors of Local Search

Steepest-ascent:
Always choose the neighbor with the highest fitness

Random restart (shotgun):
Keep trying

Stochastic:
Most likely choose the neighbor with the highest fitness

Simulated Annealing:
Stochastic with allowed negative progress, threshold decreases over time

First Choice:
Always choose the first neighbor with a better fitness

Beam Search:
Track multiple results for greater coverage

Flavors of

Steepest-ascent:

Always choose the neighbor with the highest fitness

Stochastic:

Most likely choose the

Stochastic:

Most likely choose the neighbor with the highest fitness

First Choice:

Always choose the first neighbor with a better fitness

Random restart (shotgun):

Keep trying

Simulated Annealing:

Stochastic with allowed negative progress, threshold decreases over time

Beam Search:

Track multiple results for greater coverage

Genetic Algorithms

State Representation

Algorithm

while threshold not met:
- evaluate population
- select members for reproduction
- create offspring (mate)
- replace all or some of population with new offspring
- mutate

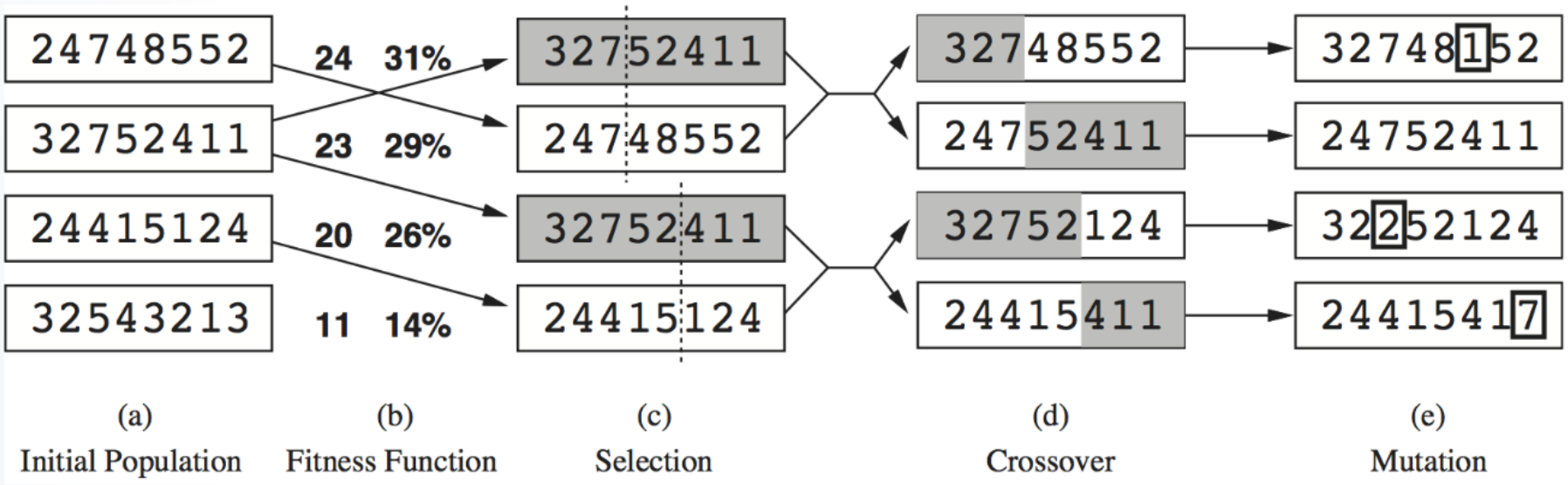
Mutation

Fitness Function

Selection

Mating

State Representation

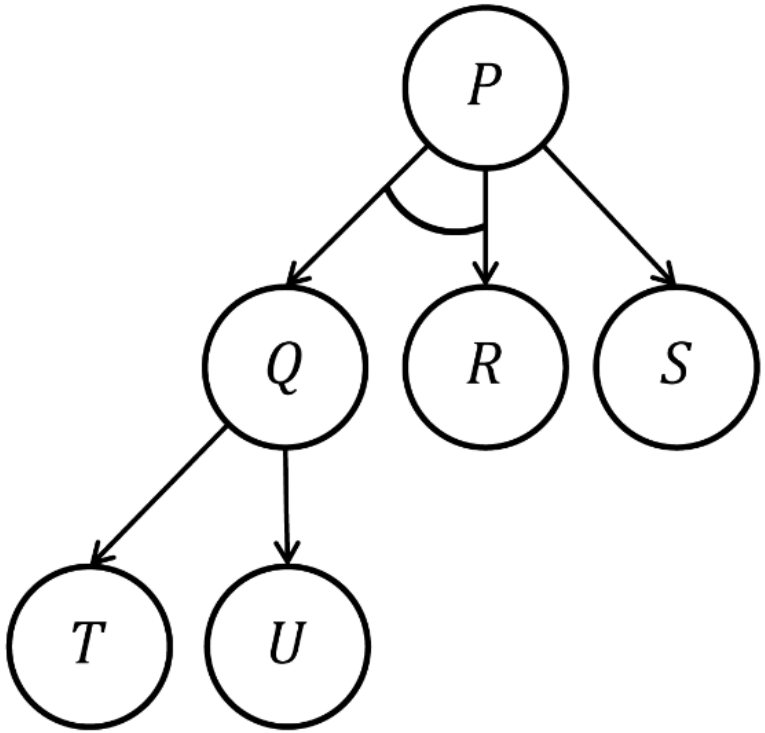


Algorithm

while threshold not met:

- evaluate population
- select members for reproduction
- create offspring (mate)
- replace all or some of population with new offspring
- mutate

Stochastic Environments



An and-or tree represents multiple possible outcomes using connected branches.

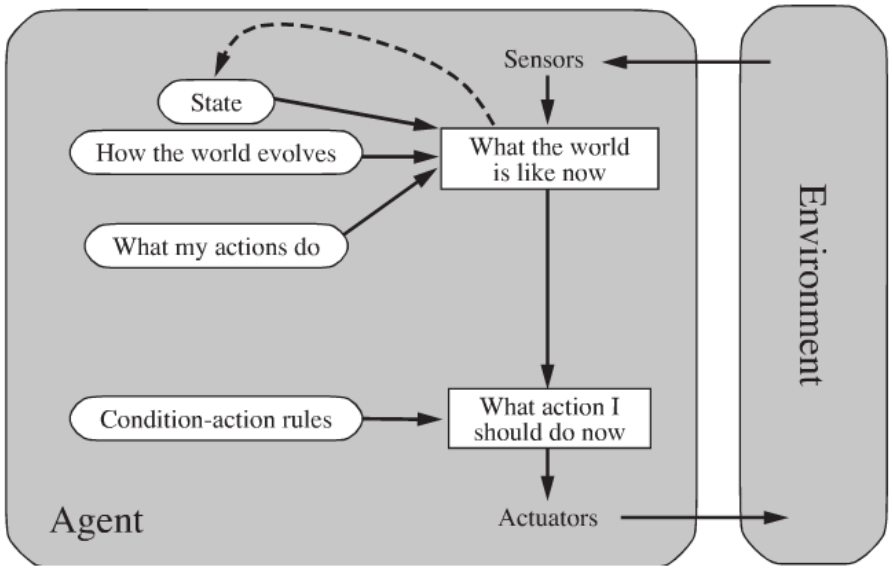
Continuous Environments

Two approaches:

1. Treat the search space as a discrete environment by partitioning it.

2. Examine slope of change of evaluation function and use to guide increments.

Partially Observable Environments



Incremental belief state search allows us to consider multiple possible worlds simultaneously

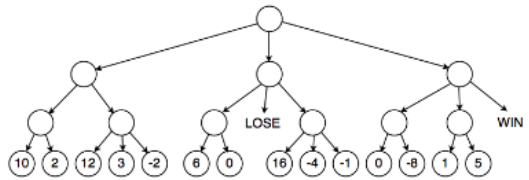
Multi-Agent Environments

Simplifying Assumptions

We'll primarily be concerned with environments that are:

- discrete
- observable
- turn-based
- not solved
- zero-sum

Minimax



Weaknesses

We may spend time exploring branches that are unlikely to be productive.
Cutoff test

We may stop immediately before an important event occurs.
Horizon effect
Quiescence

Utility Function

Estimate of our advantage in a given game state

Simplifying Assumptions

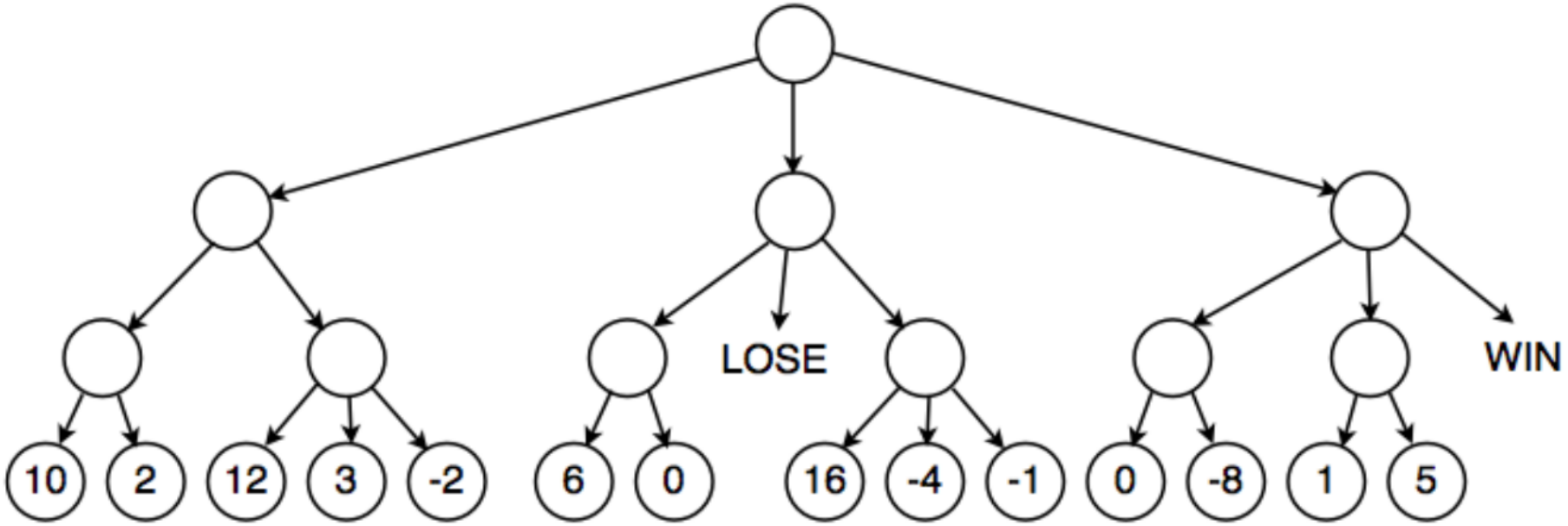
We'll primarily be concerned with environments that are:

- discrete
- observable
- turn-based
- not solved
- zero-sum

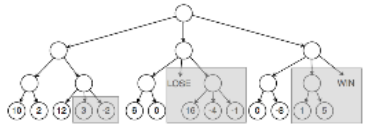
Utility Function

Estimate of our advantage in a given game state

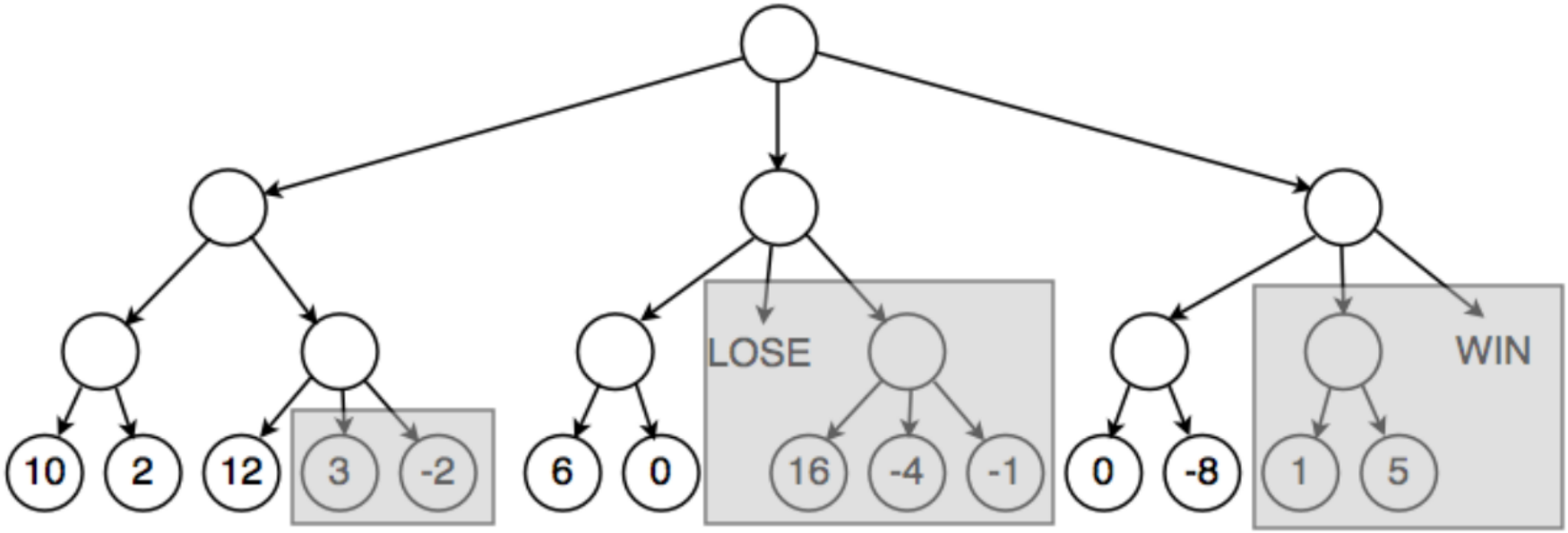
Minimax



Minimax With Alpha-Beta Pruning



Minimax With Alpha-Beta Pruning



Weaknesses

We may spend time exploring branches that are unlikely to be productive.

Cutoff test

We may stop immediately before an important event occurs.

Horizon effect

Quiescence

Constraint Satisfaction

Environment

Factored state representation.

Portions of state have related values (variables are dependent on each other).

Often used for solving scheduling problems.

Components

We represent a CSP with:
- Variables
- Domains
- Constraints

Example

1	2
3	4
5	6

Insert the letters A-F into spaces 1-6, with the following constraints:
- Only one letter per space
- A is not in row 1, column 2
- A and C are not in the same column
- E and F are in the same row
- C and F are not adjacent (horizontally or vertically)
- A and B are adjacent
- F and D are in the same column

Consistency

Consistency is used to propagate constraints.

Node consistency: One variable
Arc consistency: Two variables
Path consistency: Three variables
k-consistency: k variables

Heuristics

Minimum Remaining Values (MRV): Choose the variable with the smallest domain.

Least Restrictive Values (LRV): Choose the value that least restricts other variables' domains.

Algorithm

Alternate variable selection and constraint propagation, backtracking when reaching a dead end.

Environment

Factored state representation.

Portions of state have related values (variables are dependent on each other).

Often used for solving scheduling problems.

Components

We represent a CSP with:

- Variables
- Domains
- Constraints

Consistency

Consistency is used to propagate constraints.

Node consistency: One variable

Arc consistency: Two variables

Path consistency: Three variables

k-consistency: k variables

Algorithm

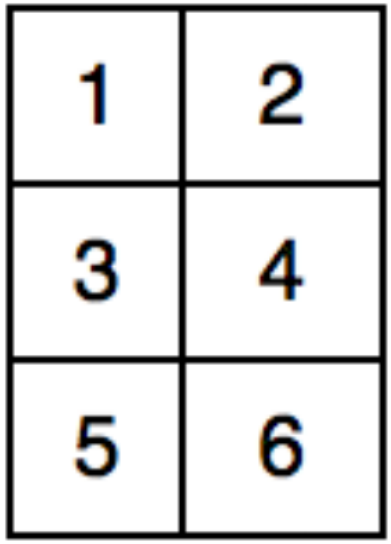
Alternate variable selection and constraint propagation, backtracking when reaching a dead end.

Heuristics

Minimum Remaining Values (MRV): Choose the variable with the smallest domain.

Least Restrictive Values (LRV): Choose the value that least restricts other variables' domains.

Example



Insert the letters A-F into spaces 1-6, with the following constraints:

- Only one letter per space
- A is not in row 1, column 2
- A and C are not in the same column
- E and F are in the same row
- C and F are not adjacent (horizontally or vertically)
- A and B are adjacent
- F and D are in the same column