Chapter 3
Solving problems by searching

Introduction

- Simple-reflex agents directly maps states to actions.
- Therefore, they cannot operate well in environments where the mapping is too large to store or takes too much to learn
- Goal-based agents can succeed by considering future actions and desirability of their outcomes
- Problem solving agent is a goal-based agent that decides what to do by finding sequences of actions that lead to desirable states

Outlines

- Problem-Solving Agents
- Problem Types
- Problem Formulation
- Example Problems
- Basic Search Algorithms

Problem solving agents

- Intelligent agents are supposed to maximize their performance measure
- This can be simplified if the agent can adopt a goal and aim at satisfying it
- Goals help organize behaviour by limiting the objectives that the agent is trying to achieve
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving
- Goal is a set of states. The agent's task is to find out which sequence of actions will get it to a goal state
- Problem formulation is the process of deciding what sorts of actions and states to consider, given a goal

Problem solving agents

- An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence
- Looking for such a sequence is called search
- A search algorithm takes a problem as input and returns a solution in the form of action sequence
- Once a solution is found the actions it recommends can be carried out – execution phase

Problem solving agents

- "formulate, search, execute" design for the agent
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do (typically the first action in the sequence)
- Then removing that step from the sequence
- Once the solution has been executed, the agent will formulate a new goal and start all over again

Problem-Solving Agents

Problem-Solving Agents

Restricted form of general agent:

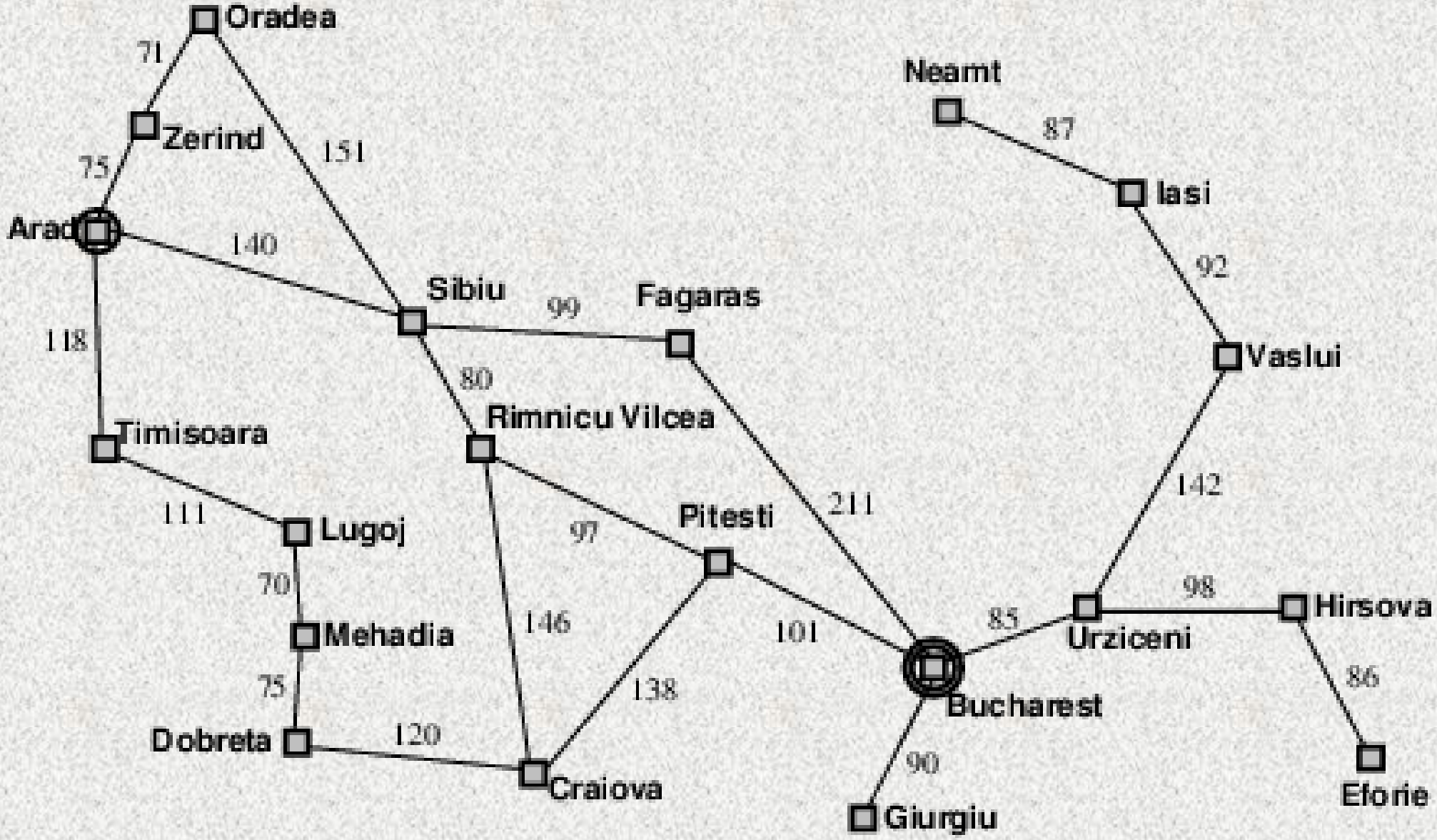
```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept ) returns an action
    persistent: seq , an action sequence, initially empty
    state, some description of the current world state
    goal , a goal, initially null
    problem , a problem formulation
state ← UPDATE-STATE(state, percept )
if seq is empty then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal )
    seq ← SEARCH( problem )
if seq  = failure then return a null action
    action ← FIRST(seq )
    seq ← REST(seq )
return action
```

A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Example: Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- Formulate goal
  - be in Bucharest
- Formulate problem
  - **states**: various cities
  - **actions**: drive between cities
- Find solution
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

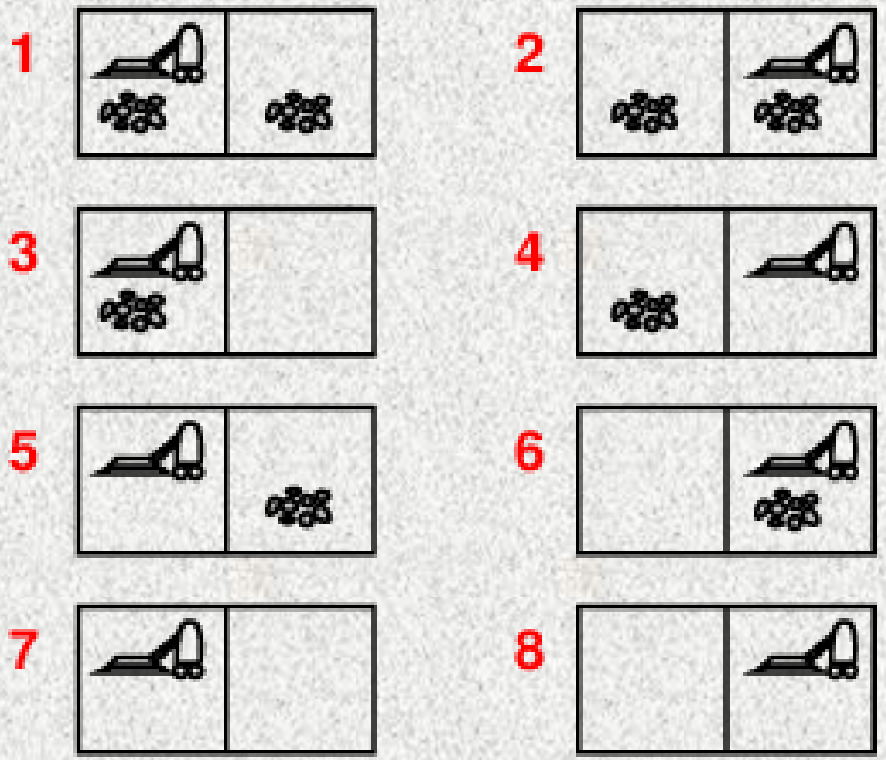
Romania


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
85
Urziceni
86
Bucharest
90
Giurgiu
Eforie

Example: Vacuum World

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

Contingency (may result in more than one state), start in #5.
Murphy's Law: *Suck* can dirty a clean carpet Local sensing: dirt, location only.
Solution?
[*Right,* **if** *dirt* **then** *Suck*]



Problem Formulation

Search Space Definitions

- State
  - A description of a possible state of the world
  - Includes all features of the world that are pertinent to the problem
- Initial state
  - Description of all pertinent aspects of the state in which the agent starts the search
- Goal test
  - Conditions the agent is trying to meet (e.g., have $1M)
- Goal state
  - Any state which meets the goal condition
  - Thursday, have $1M, live in NYC
- Action
  - Function that maps (transitions) from one state to another

Search Space Definitions

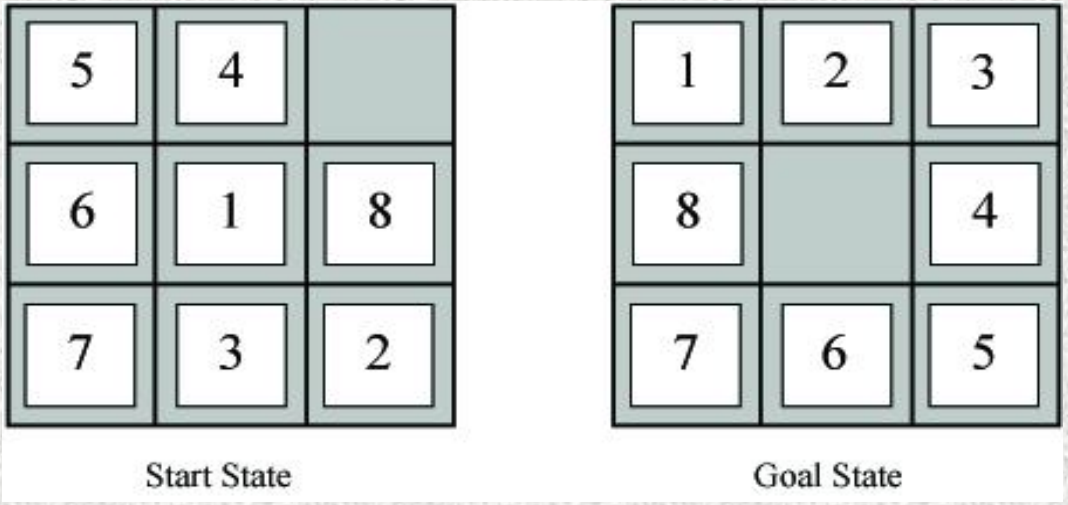
- Problem formulation
  - Describe a general problem as a search problem
- Solution
  - Sequence of actions that transitions the world from the initial state to a goal state
- Solution cost (additive)
  - Sum of the cost of operators
  - Alternative: sum of distances, number of steps, etc.
- Search
  - Process of looking for a solution
  - Search algorithm takes problem as input and returns solution
  - We are searching through a space of possible states
- Execution
  - Process of executing sequence of actions (solution)

Problem Formulation

A search problem is defined by the

1. Initial state (e.g., Arad)
2. Operators (e.g., Arad -> Zerind, Arad -> Sibiu, etc.)
3. Goal State (s) (e.g., at Bucharest)
4. Solution cost (e.g., path cost)

Example Problems – Eight Puzzle



States: tile locations

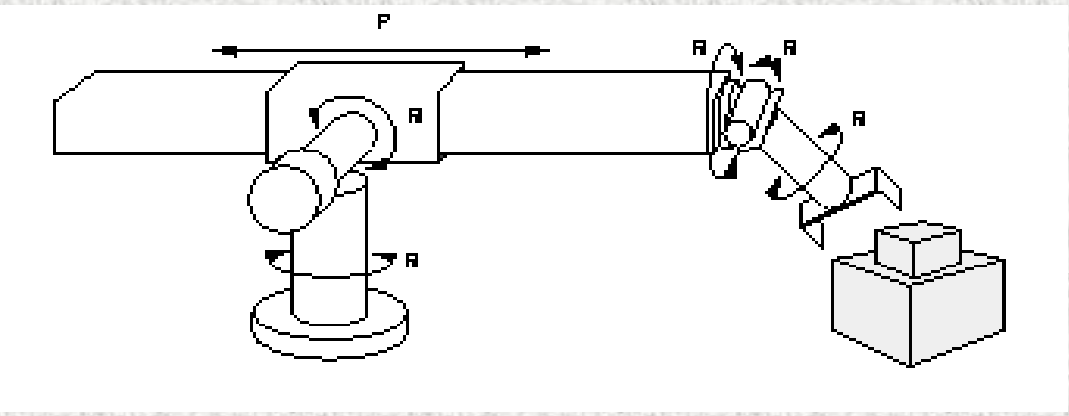
Initial state: one specific tile configuration

Operators: move blank tile left, right, up, or down

Goal: tiles are numbered from one to eight around the square

Path cost: cost of 1 per move (solution cost same as number of most or path length)

Example Problems – Robot Assembly



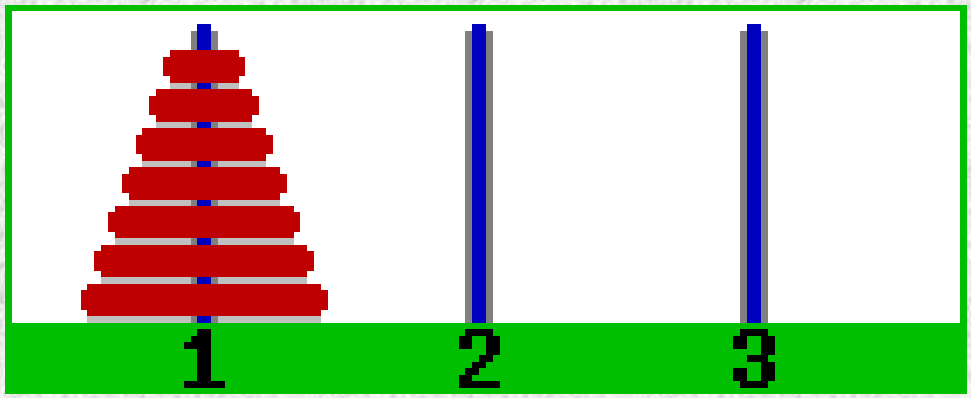
States: real-valued coordinates of
- robot joint angles
- parts of the object to be assembled

Operators: rotation of joint angles

Goal test: complete assembly

Path cost: time (# of steps) to complete assembly

Example Problems – Towers of Hanoi



States: combinations of poles and disks

Operators: move disk x from pole y to pole z subject to constraints

- cannot move disk on top of smaller disk
- cannot move disk if other disks on top

Goal test: disks from largest (at bottom) to smallest on goal pole

Path cost: 1 per move

Example Problems – Rubik's Cube

States: list of colors for each cell on each face

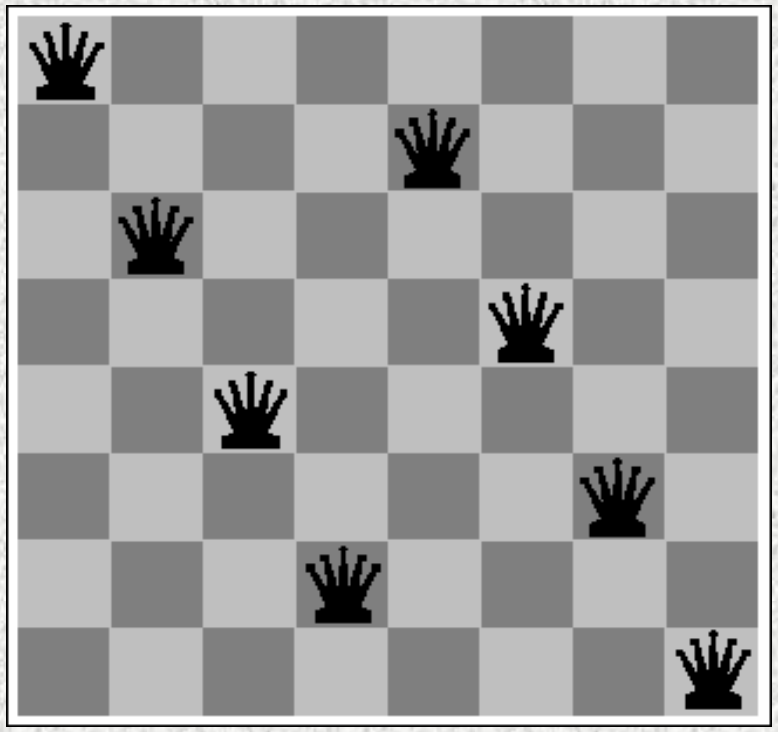
Initial state: one specific cube configuration

Operators: rotate row x or column y on face z direction a

Goal: configuration has only one color on each face

Path cost: 1 per move

Example Problems – Eight Queens



States: locations of 8 queens on chess board

Initial state: one specific queens configuration

Operators: move queen x to row y and column z

Goal: no queen can attack another (cannot be in same row, column, or diagonal)

Path cost: 0 per move

Example Problems – Missionaries and Cannibals

States: number of missionaries, cannibals, and boat on near river bank

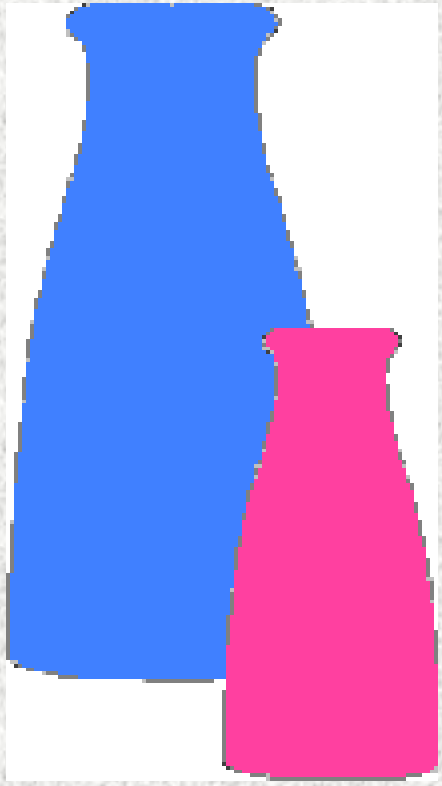
Initial state: all objects on near river bank

Operators: move boat with x missionaries and y cannibals to other side of river
- no more cannibals than missionaries on either river bank or in boat
- boat holds at most m occupants

Goal: all objects on far river bank

Path cost: 1 per river crossing

Example Problems –Water Jug



States: Contents of 4-gallon jug and 3-gallon jug

Initial state: (0,0)

Operators:
- fill jug x from faucet
- pour contents of jug x in jug y until y full
- dump contents of jug x down drain

Goal: (2,n)

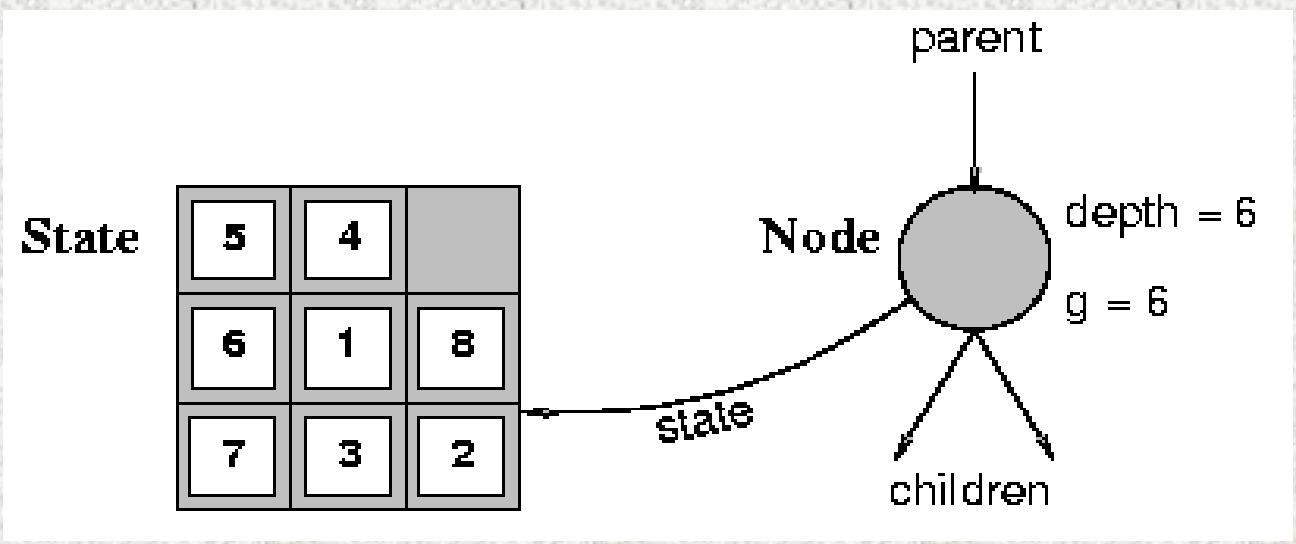
Path cost: 1 per fill

Sample Search Problems

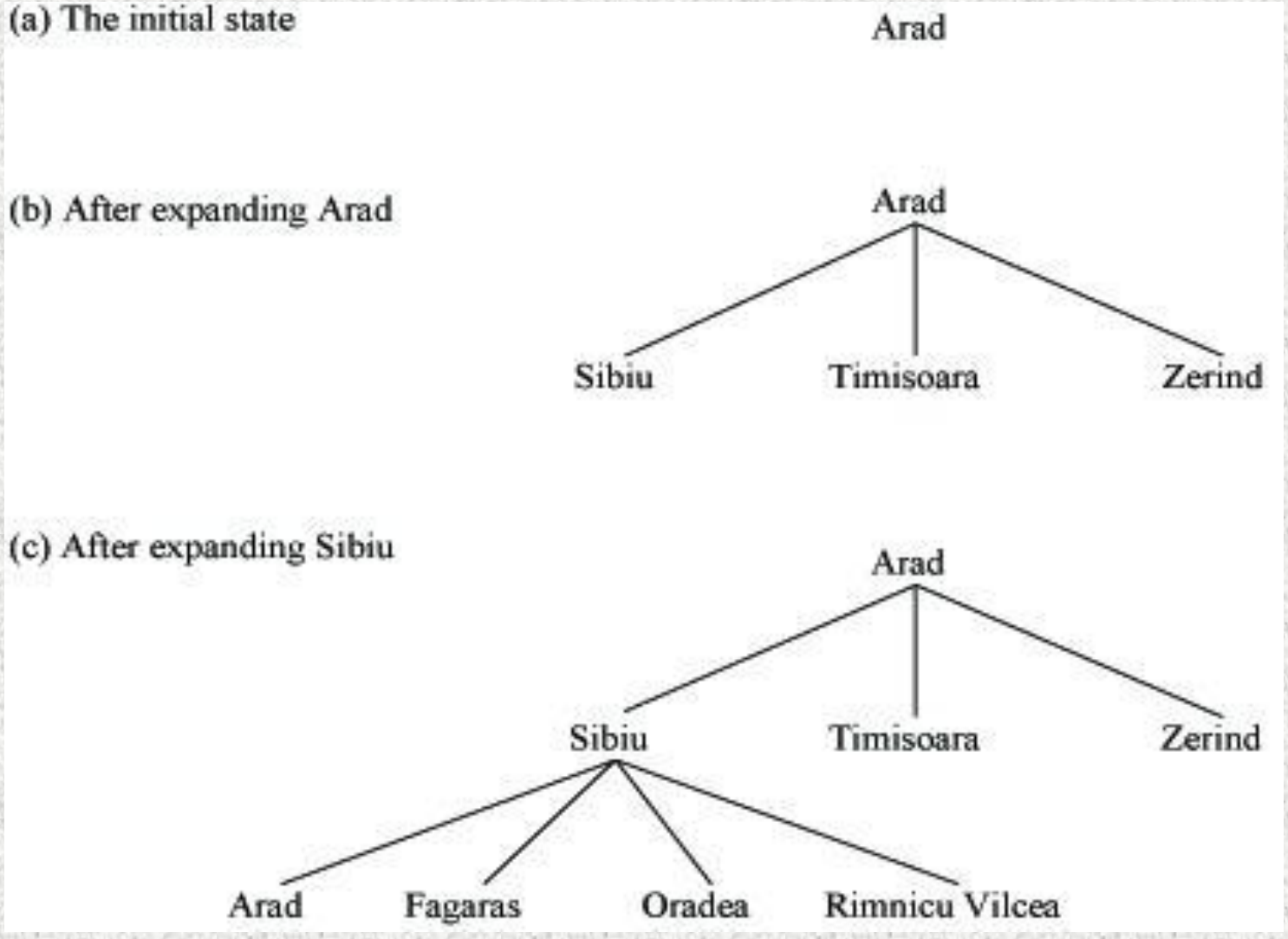
- Graph coloring
- Protein folding
- Game playing
- Airline travel
- Proving algebraic equalities
- Robot motion planning

Visualize Search Space as a Tree

- States are nodes
- Actions are edges
- Initial state is root
- Solution is path from root to goal node
- Edges sometimes have associated costs
- States resulting from operator are children



Search Problem Example (as a tree)



Search Function – Uninformed Searches

```
Open = initial state                                  // open list is all generated states
                                                      // that have not been "expanded"
While open not empty                                  // one iteration of search algorithm
   state = First(open)                                // current state is first state in open
   Pop(open)                                          // remove new current state from open
   if Goal(state)                                     // test current state for goal condition
      return "succeed"                                // search is complete
                                                      // else expand the current state by
                                                      // generating children and
                                                      // reorder open list per search strategy
   else open = QueueFunction(open, Expand(state))
Return "fail"
```

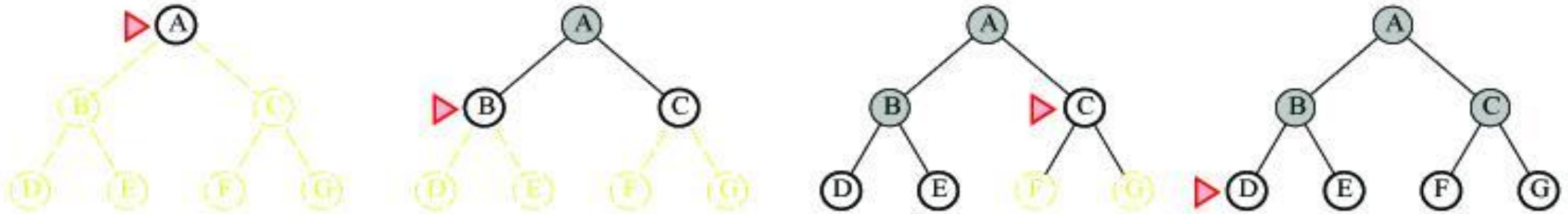
Search Strategies

- Search strategies differ only in Queuing Function
- Features by which to compare search strategies
  - Completeness (always find solution)
  - Cost of search (time and space)
    - Time Complexity (measured in #nodes generated)
    - Space Complexity
  - Cost of solution, optimal solution
  - Make use of knowledge of the domain
    - “uninformed search” vs. “informed search”

Breadth-First Search

- Generate children of a state, QueueingFn adds the children to the end of the open list
- Level-by-level search
- Order in which children are inserted on open list is arbitrary
- In tree, assume children are considered left-to-right unless specified differently
- Number of children is “branching factor” b

BFS Examples



b = 2

Analysis

- Assume goal node at level d with constant branching factor b
- Time complexity (measured in #nodes generated)
  - 1 (1^{st} level) + b (2^{nd} level) + b^2 (3^{rd} level) + … + b^d (goal level) + (b^{d+1} – b) = $O(b^d+1)$
- This assumes goal on far right of level
- Space complexity
  - At most majority of nodes at level d + majority of nodes at level d+1 = $O(b^{d+1})$
  - Exponential time and space

Analysis

- Features
  - Simple to implement
  - Complete
  - Finds shortest solution (not necessarily least-cost unless all operators have equal cost)

Analysis

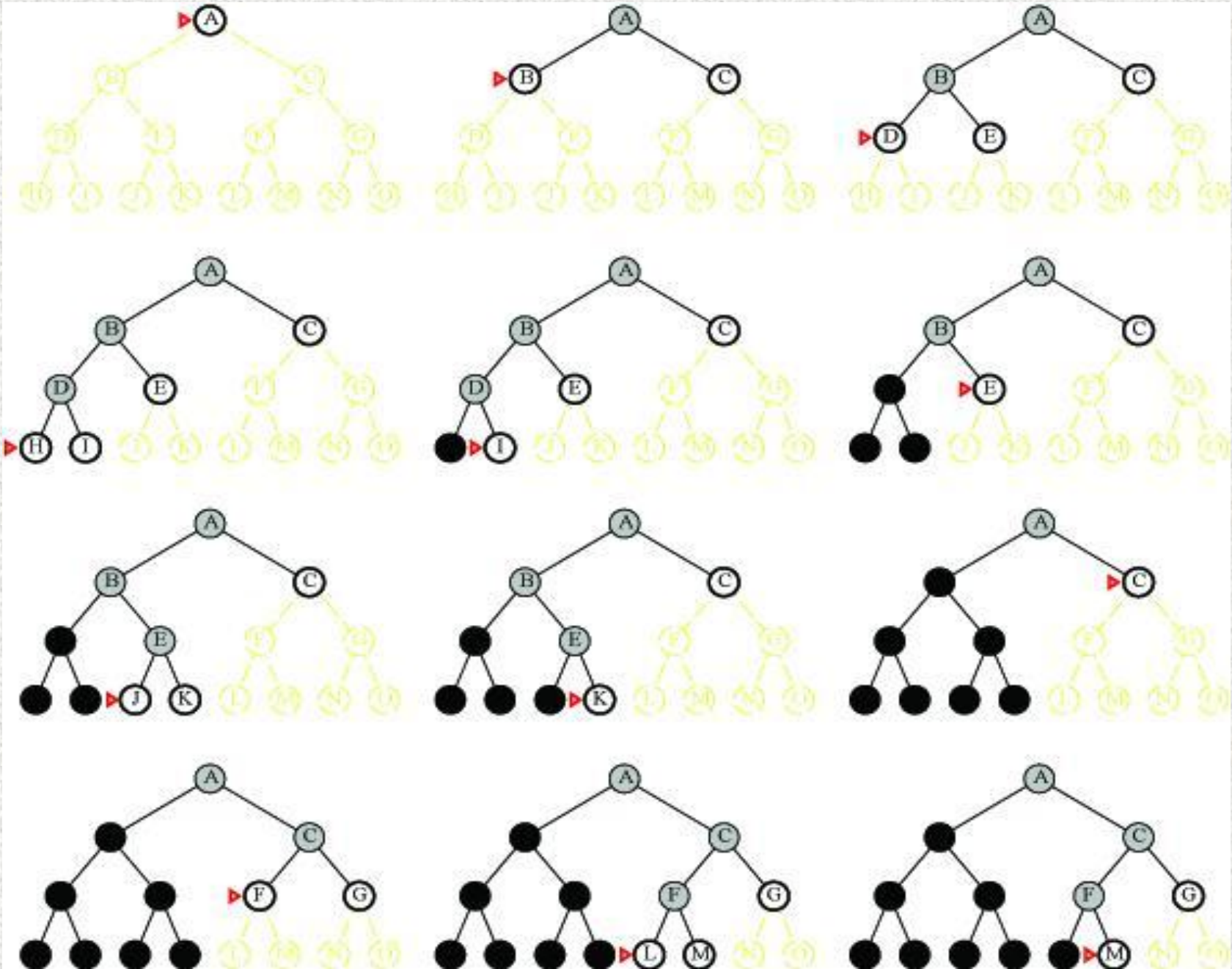
- See what happens with b=10
  - expand 1,000,000 nodes/second
  - 1,000 bytes/node

Depth	Nodes	Time	Memory
2	110	0.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabytes
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabytes
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Depth-First Search

- QueueingFn adds the children to the front of the open list
- BFS emulates FIFO queue
- DFS emulates LIFO stack
- Net effect
  - Follow leftmost path to bottom, then backtrack
  - Expand deepest node first

DFS Examples



Analysis

- Time complexity
  - In the worst case, search entire space
  - Goal may be at level d but tree may continue to level m, m>=d
  - $O(b^m)$
  - Particularly bad if tree is infinitely deep
- Space complexity
  - Only need to save one set of children at each level
  - 1 + b + b + … + b (m levels total) = O(bm)
  - For previous example, DFS requires 117kb instead of 10 petabytes for d=12 (10 billion times less)
- Benefits
  - May not always find solution
  - Solution is not necessarily shortest or least cost
  - If many solutions, may find one quickly (quickly moves to depth d)
  - Simple to implement
  - Space often bigger constraint, so more usable than BFS for large problems

Comparison of Search Techniques

	DFS	BFS
Complete	N	Y
Optimal	N	N
Heuristic	N	N
Time	b^m	b^{d+1}
Space	bm	b^{d+1}

Avoiding Repeated States

Can we do it?

- Do not return to parent or grandparent state
  - In 8 puzzle, do not move-up right after move-down
- Do not create solution paths with cycles
- Do not generate repeated states (need to store and check potentially large number of states)

Variants of DFS

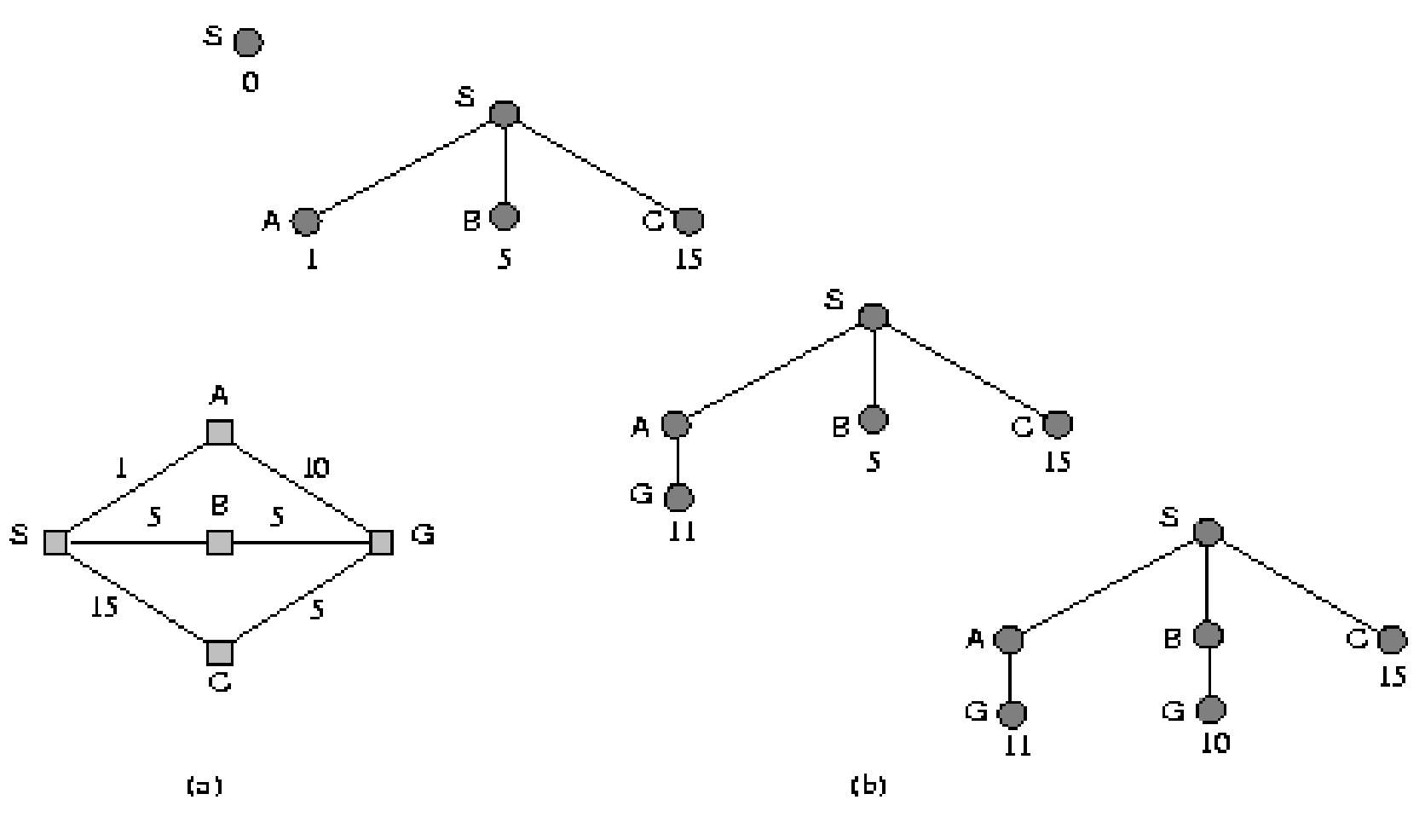
- Backtracking Search: Only one successor is generated instead of all successors. Each partially expanded node remembers which node to generate next. So memory requirement is O(m) instead of O(bm)
- Depth Limiting Search: $O(b\ell)$

  Choosing the right value of ℓ.
- Iterative Deeping

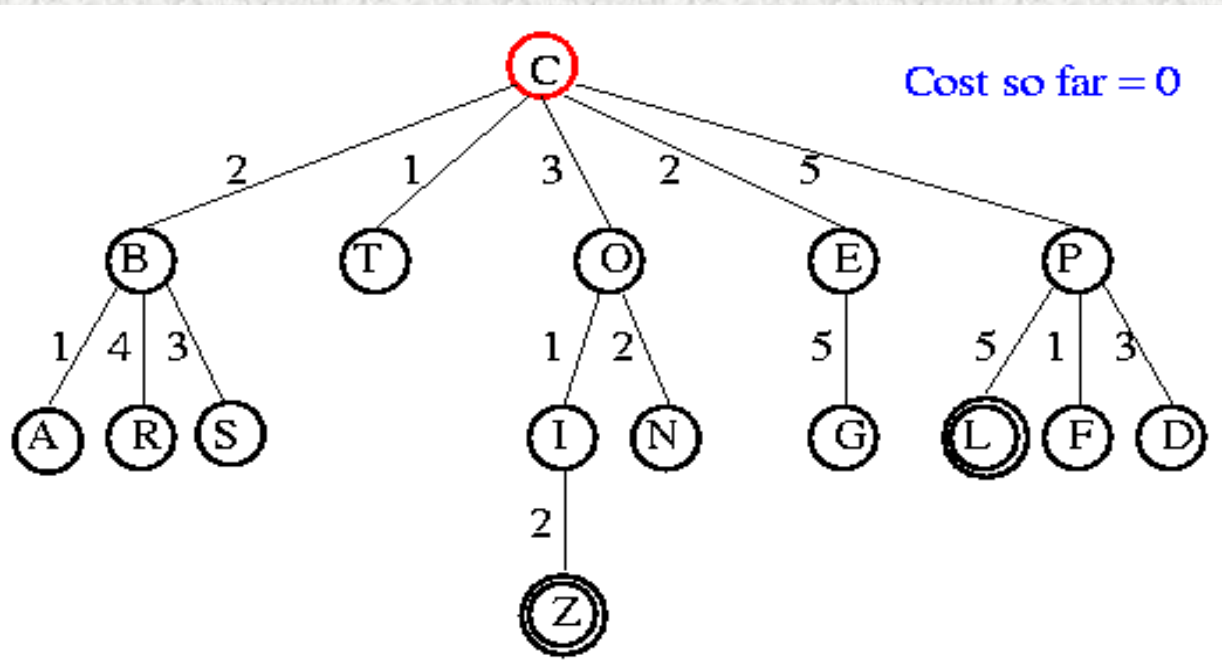
Uniform Cost Search (Branch&Bound)

- QueueingFn is SortByCostSoFar
- Cost from root to current node n is g(n)
  - Add operator costs along path
- First goal found is least-cost solution
- Space & time can be exponential because large subtrees with inexpensive steps may be explored before useful paths with costly steps
- If costs are equal, time and space are $O(b^d)$
  - Otherwise, complexity related to cost of optimal solution

UCS Example

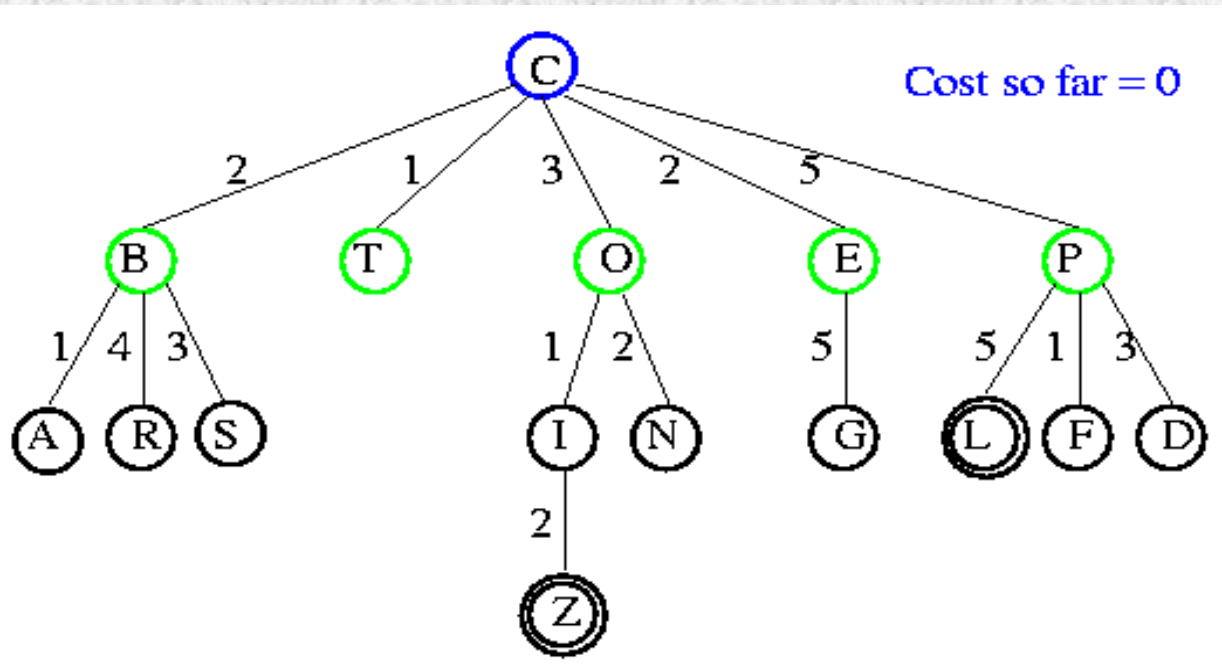


UCS Example



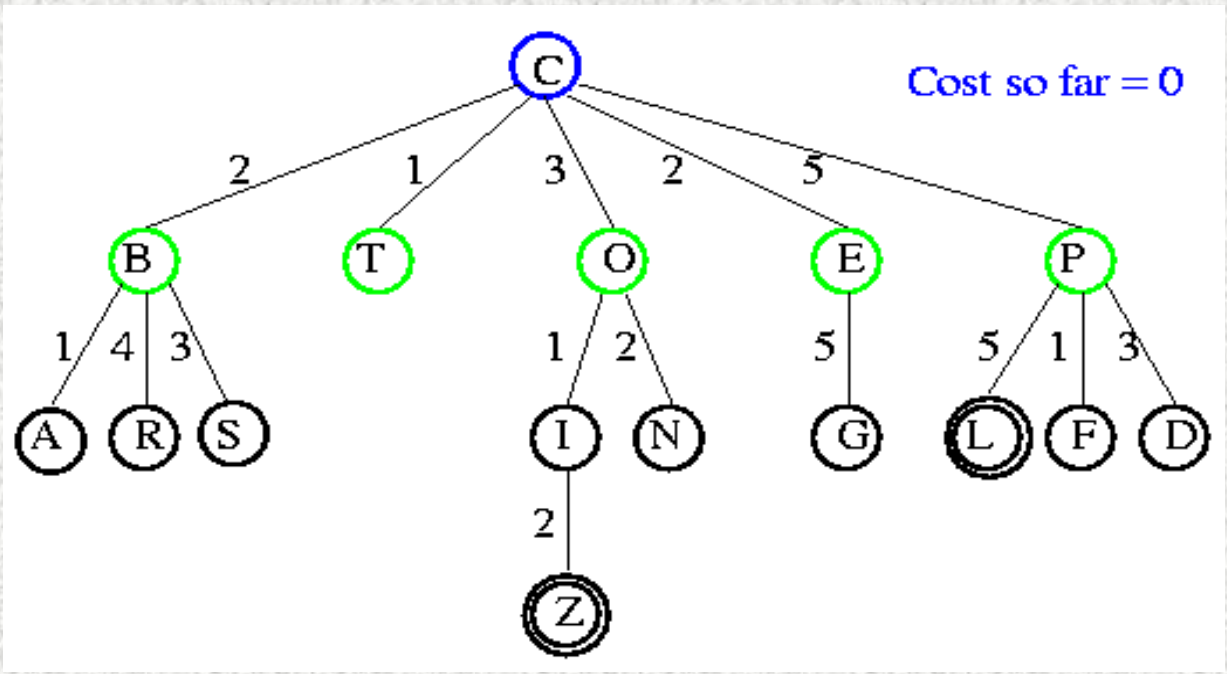
Open list: C

UCS Example



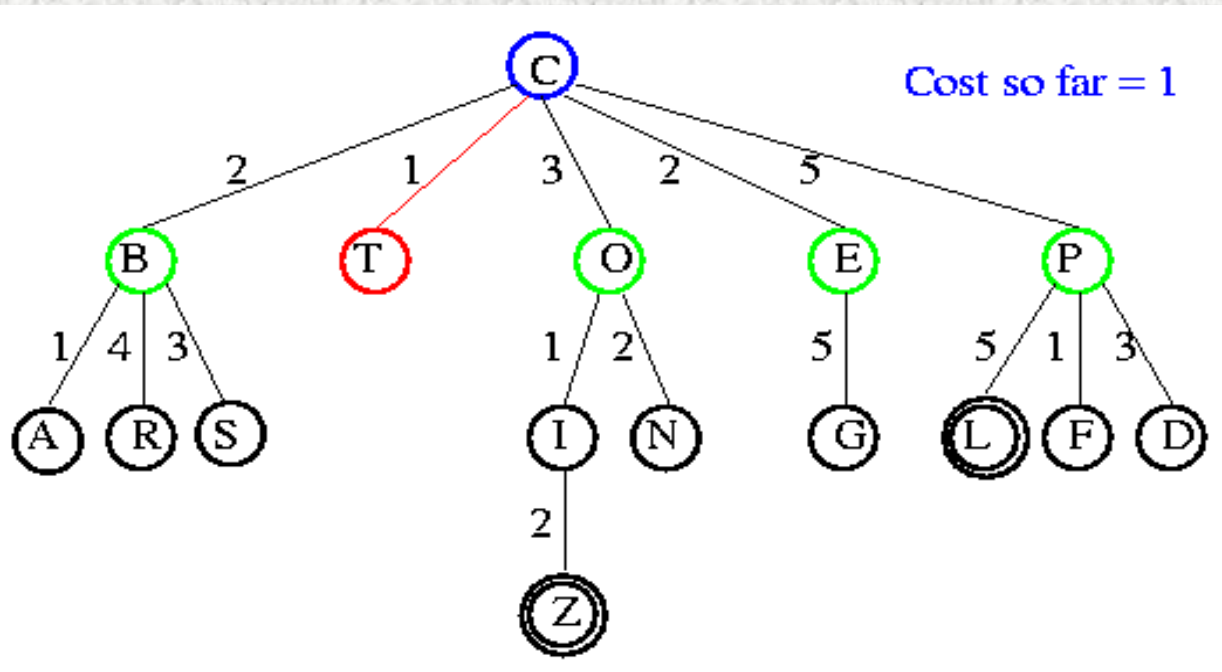
Open list: B(2) T(1) O(3) E(2) P(5)

UCS Example



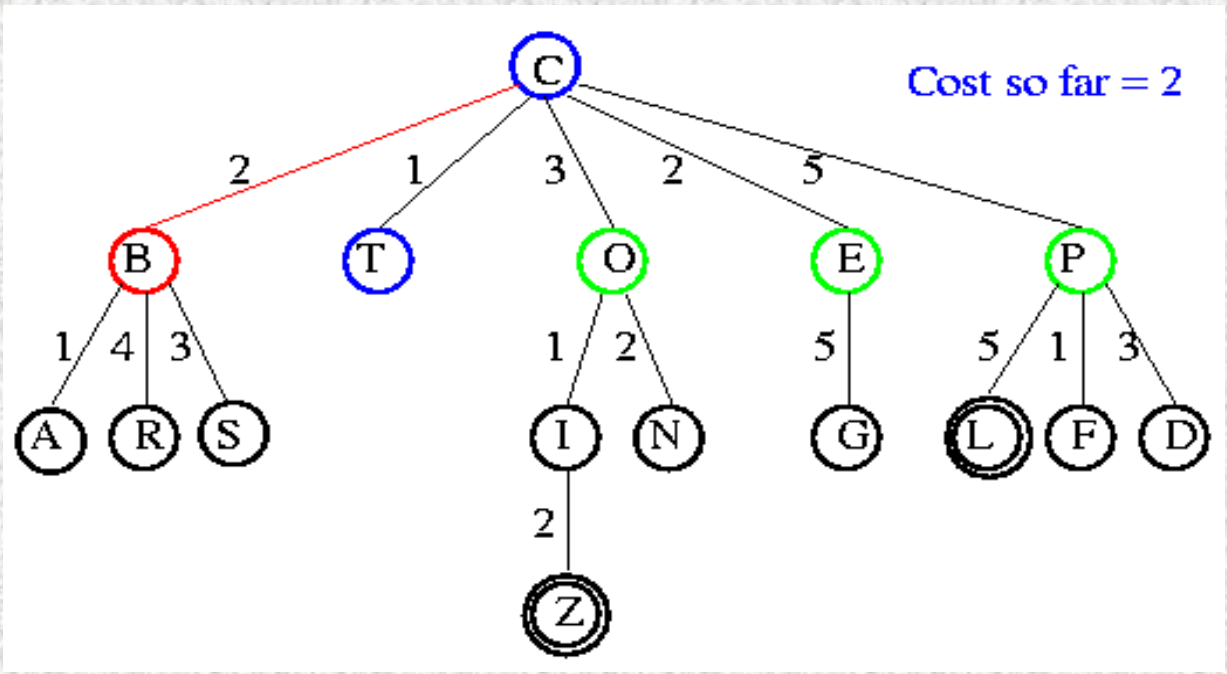
Open list: T(1) B(2) E(2) O(3) P(5)

UCS Example



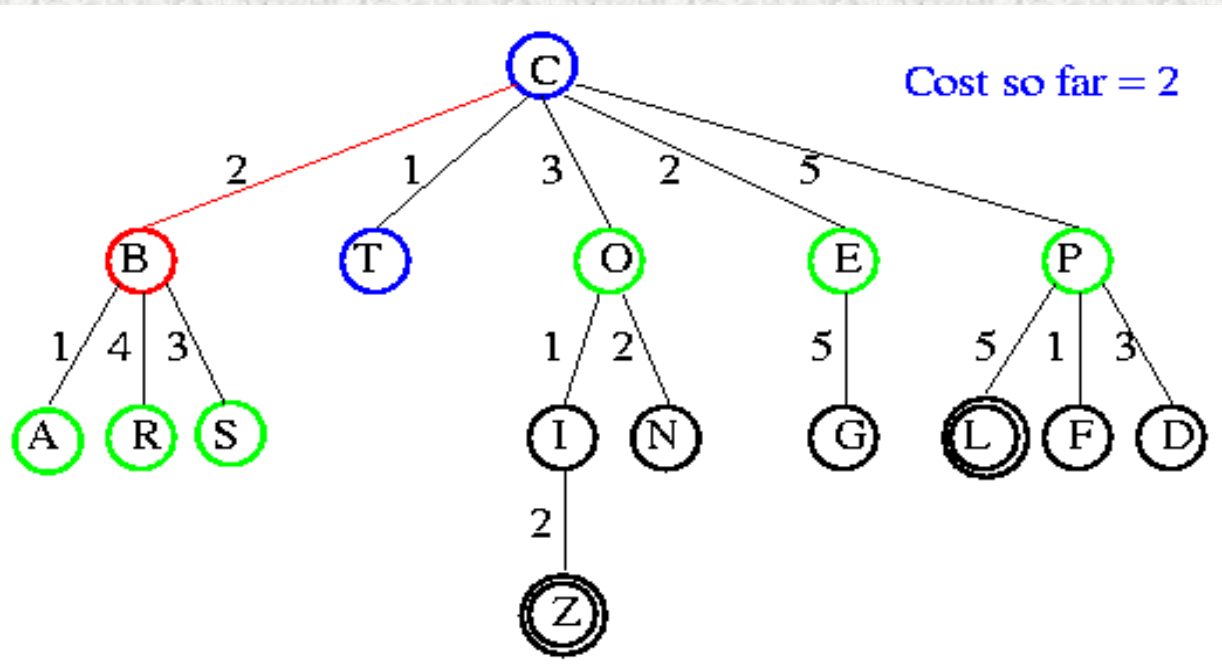
Open list: B(2) E(2) O(3) P(5)

UCS Example



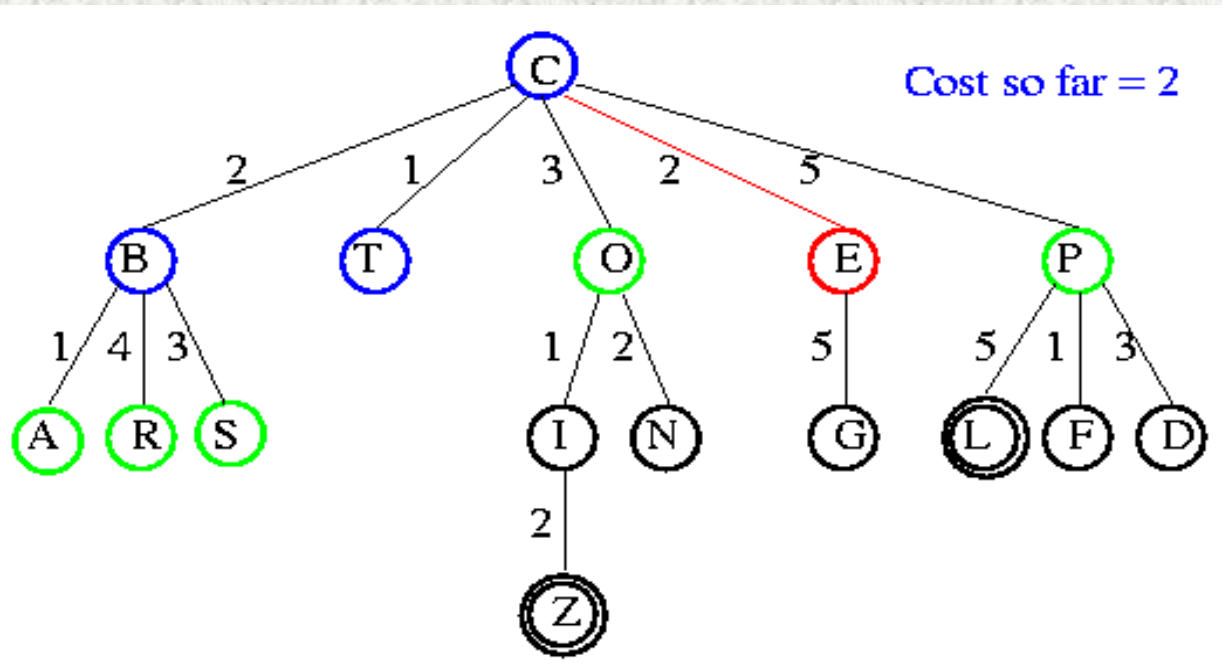
Open list: E(2) O(3) P(5)

UCS Example



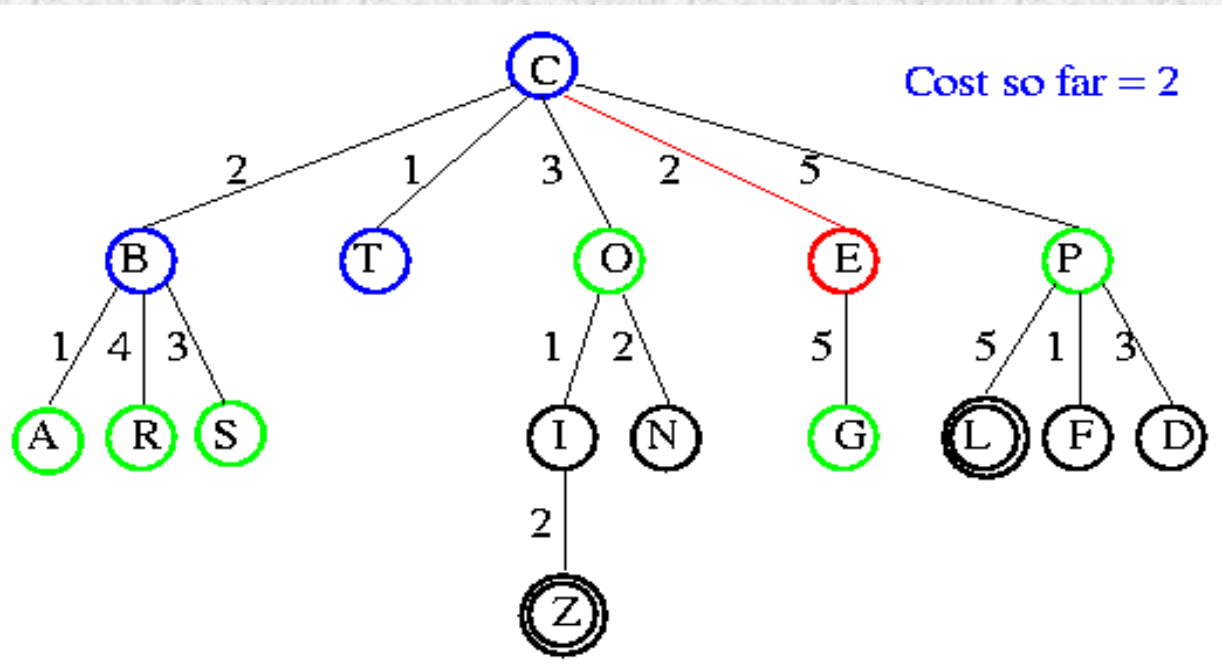
Open list: E(2) O(3) A(3) S(5) P(5) R(6)

UCS Example



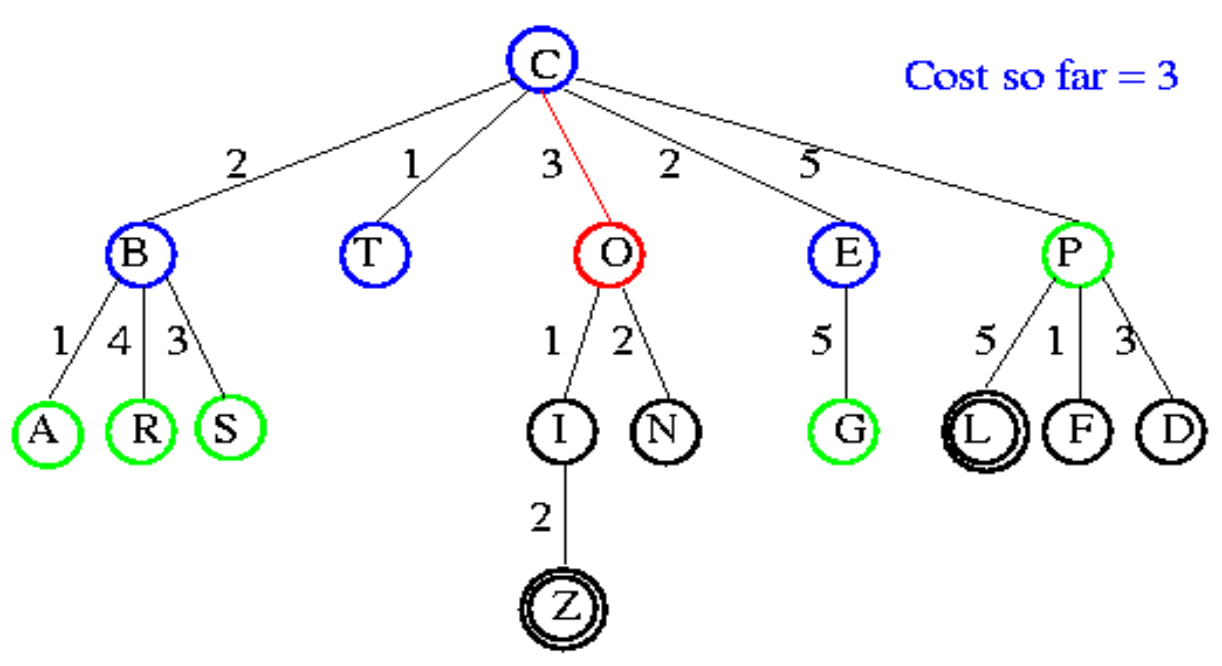
Open list: O(3) A(3) S(5) P(5) R(6)

UCS Example



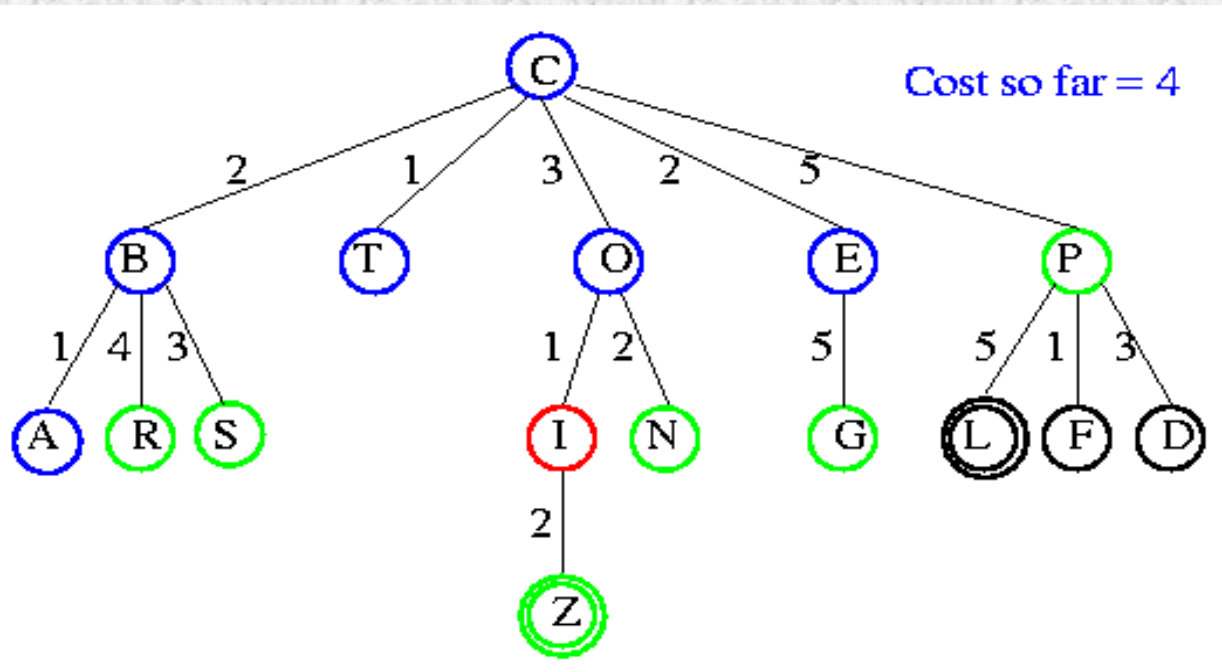
Open list: O(3) A(3) S(5) P(5) R(6) G(7)

UCS Example



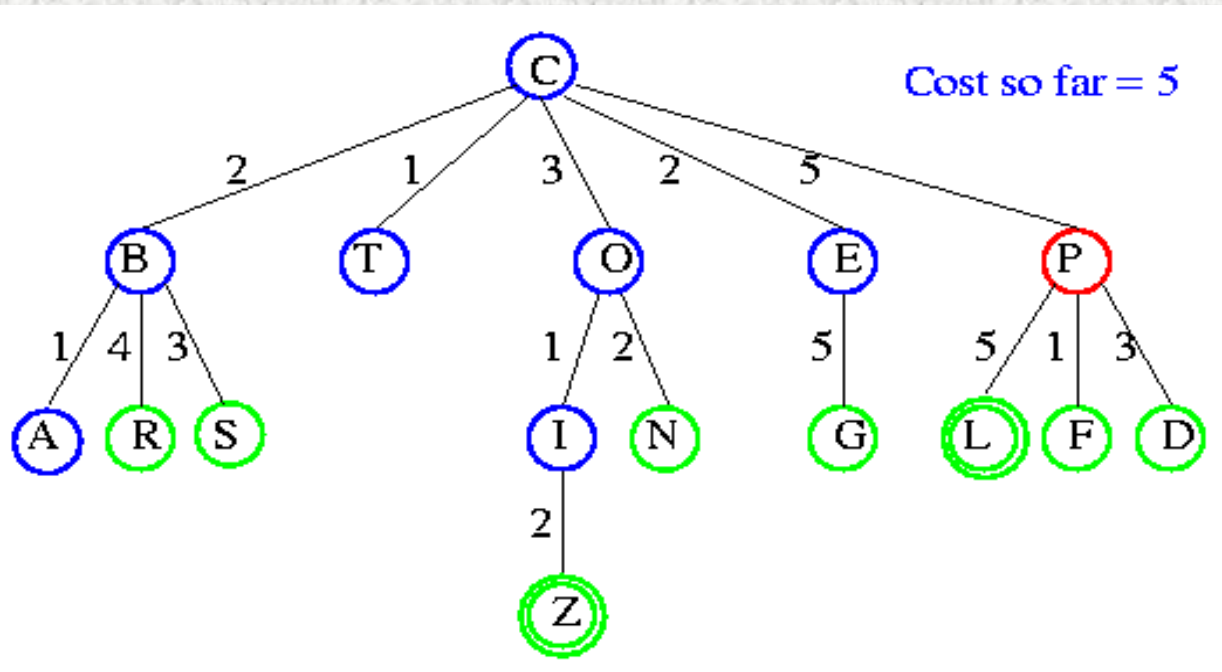
Open list: A(3) S(5) P(5) R(6) G(7)

UCS Example



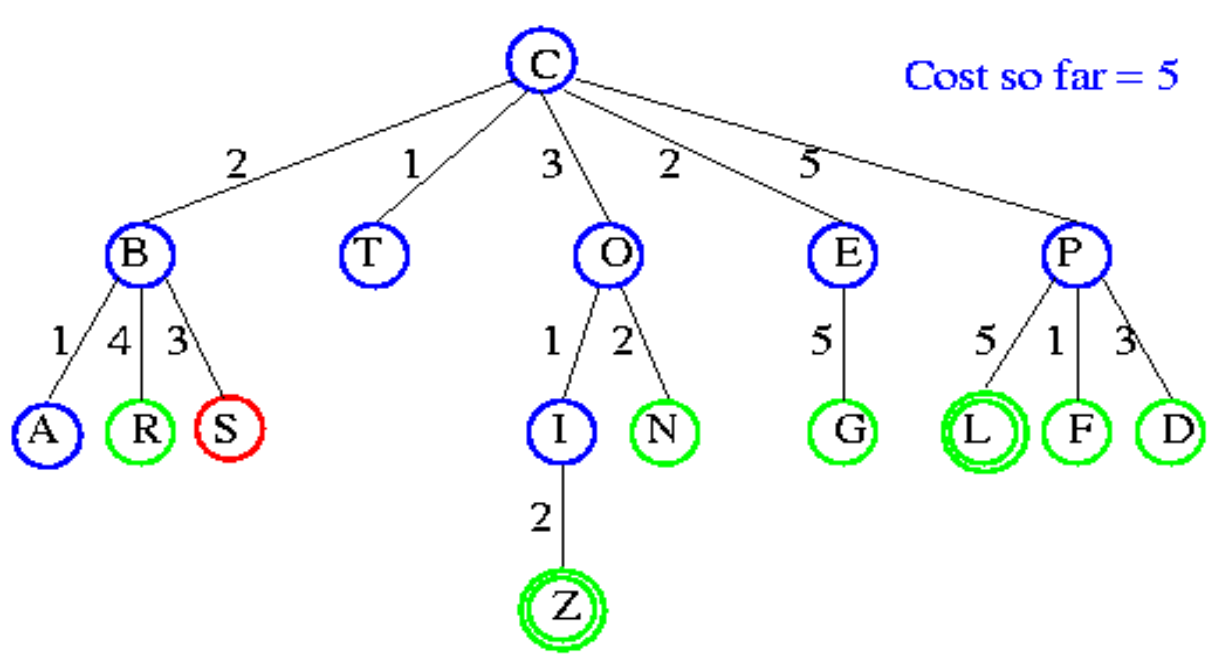
Open list: P(5) S(5) N(5) R(6) Z(6) G(7)

UCS Example



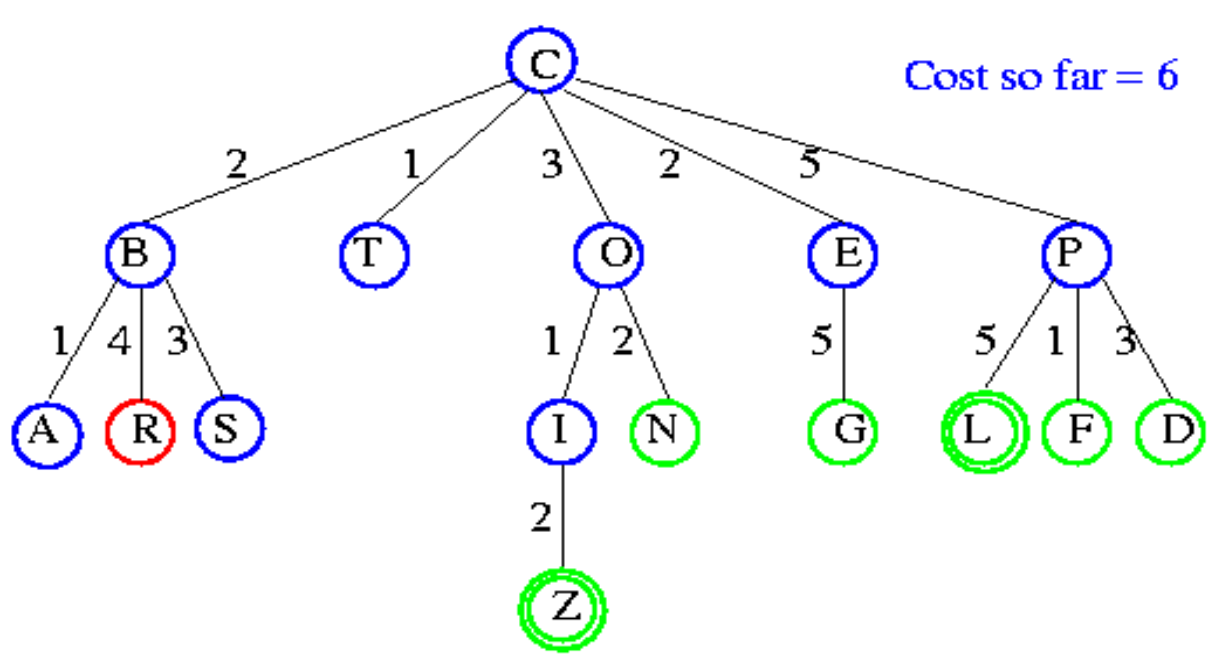
Open list: S(5) N(5) R(6) Z(6) F(6) G(7) D(8) L(10)

UCS Example



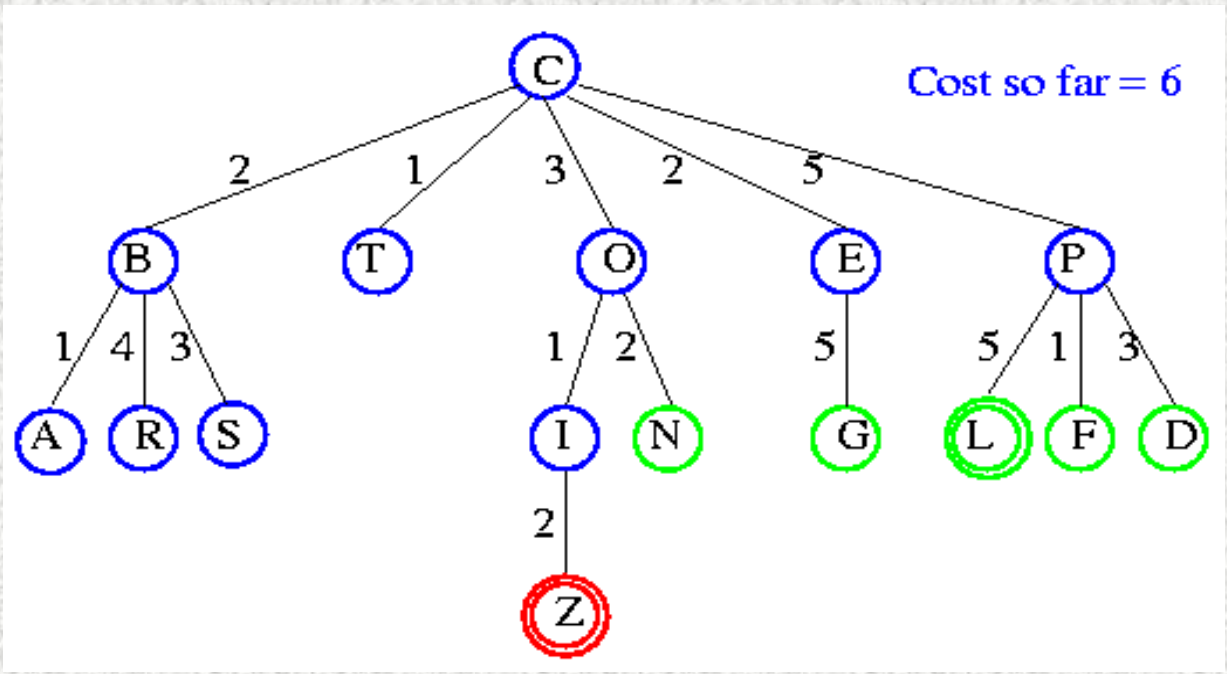
Open list: N(5) R(6) Z(6) F(6) G(7) D(8) L(10)

UCS Example



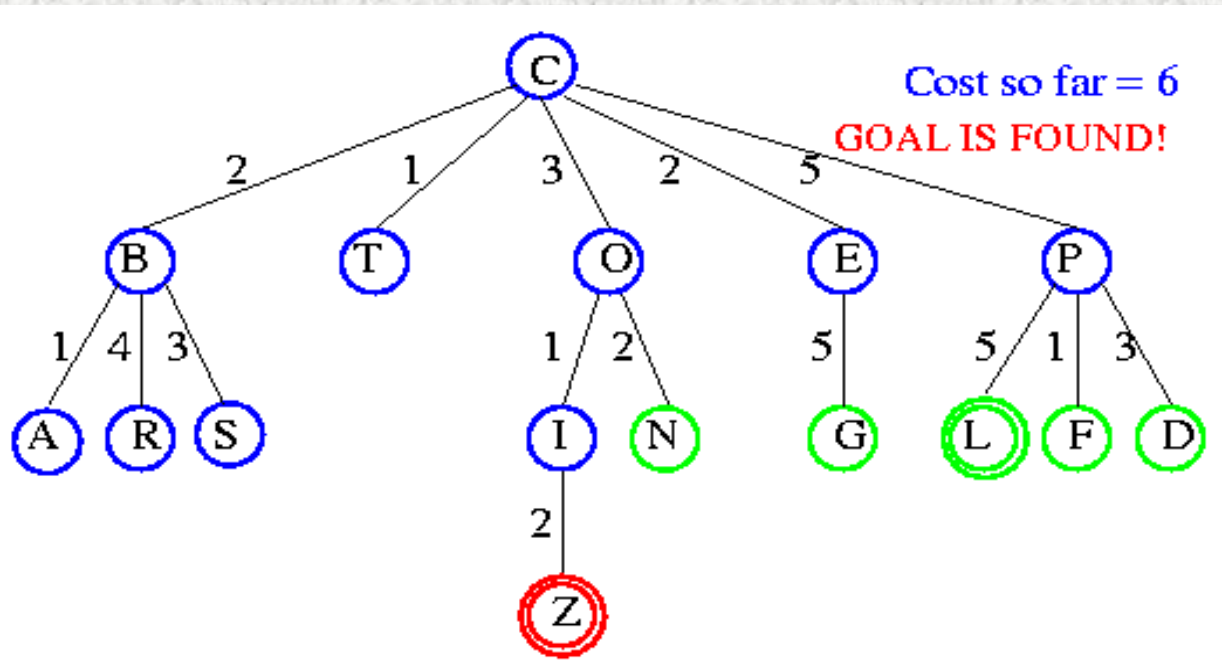
Open list: Z(6) F(6) G(7)D(8) L(10)

UCS Example



Open list: F(6) D(8) G(10) L(10)

UCS Example


C
Cost so far = 6
GOAL IS FOUND!
2
1
3
2
5
B
T
O
E
P
1
4
3
1
2
5
5
1
3
A
R
S
I
N
G
L
F
D
2
Z

Comparison of Search Techniques

	DFS	BFS	UCS
Complete	N	Y	Y
Optimal	N	N	Y
Heuristic	N	N	N
Time	b^m	b^{d+1}	b^m
Space	bm	b^{d+1}	b^m

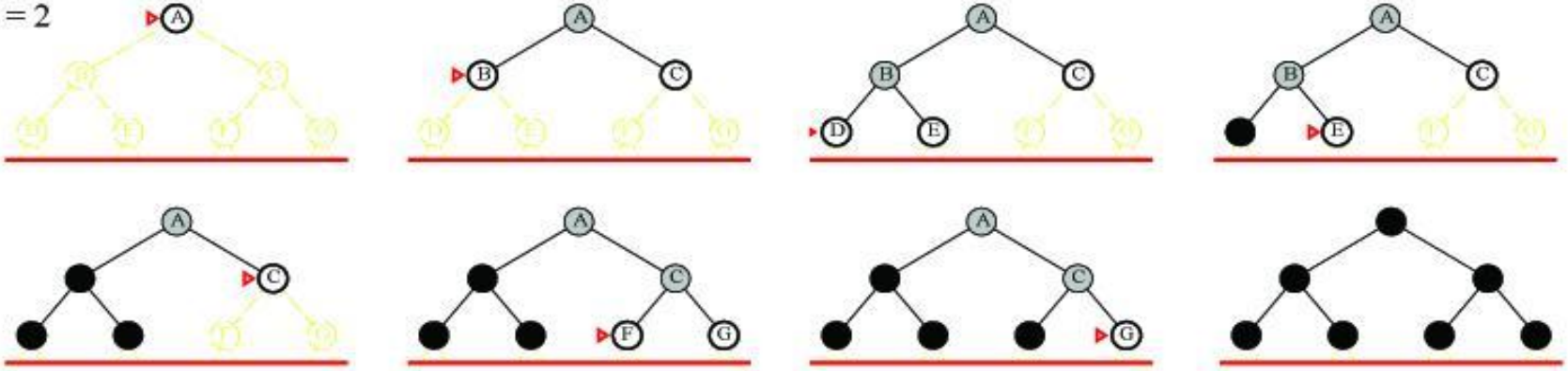
Iterative Deepening Search

- DFS with depth bound
- QueuingFn is enqueue at front as with DFS
  - Expand(state) only returns children such that depth(child) <= threshold
  - This prevents search from going down infinite path
- First threshold is 1
  - If do not find solution, increment threshold and repeat

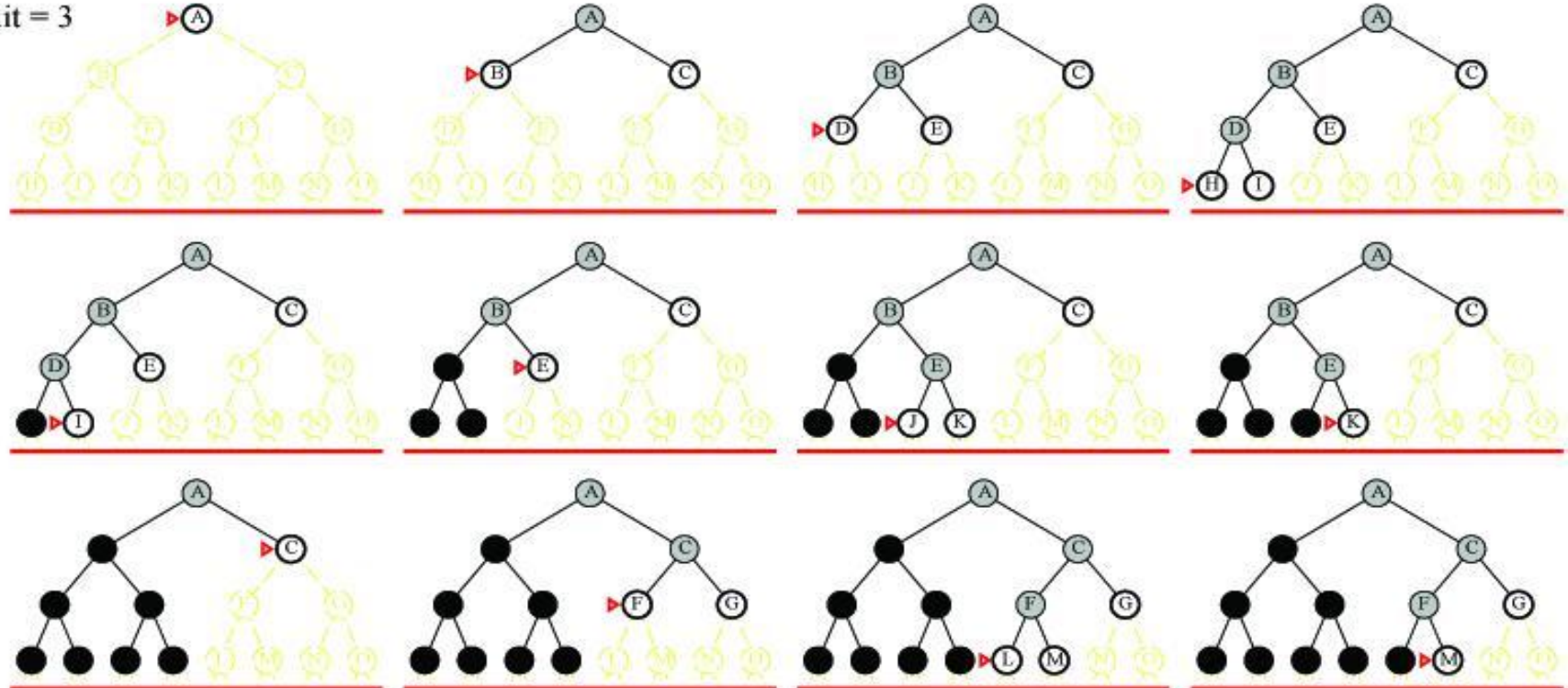
Limit = 0

Limit = 1

Limit = 2



Limit = 3



Analysis

- What about the repeated work?
- Time complexity (number of generated nodes)
  - $[b] + [b + b^2] + .. + [b + b^2 + .. + b^d]$
  - $(d)b + (d-1)\ b^{2\ +} \ldots + (1)\ b^d$
  - $O(b^d)$

Analysis

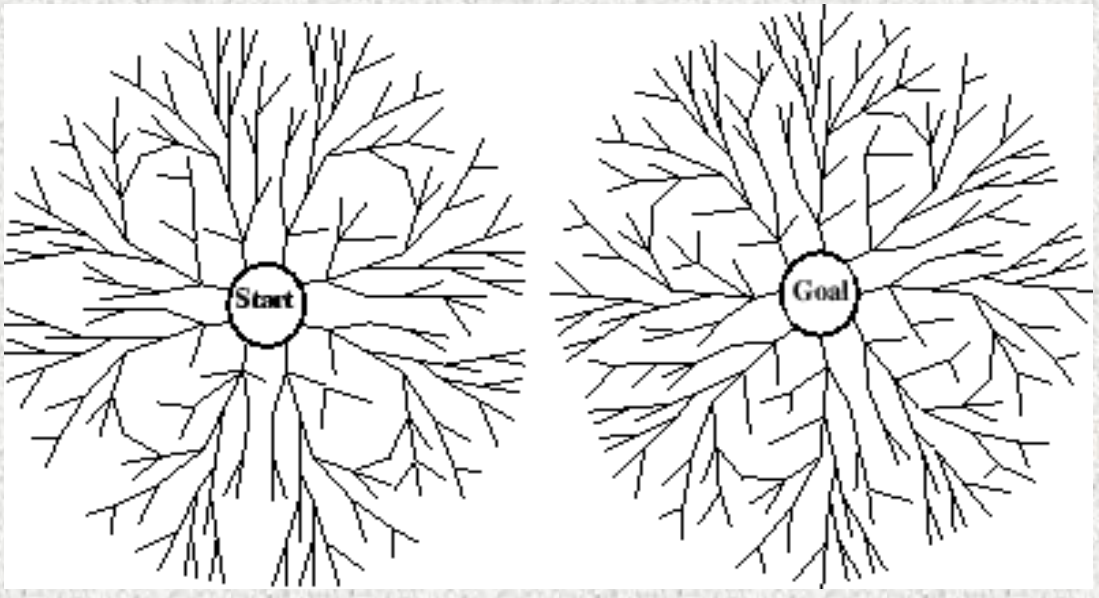
- Repeated work is approximately 1/b of total work
  - Negligible
  - Example: b=10, d=5
  - N(BFS) = 111,110
  - N(IDS) = 123,450
- Features
  - Shortest solution, not necessarily least cost
  - Is there a better way to decide threshold? (IDA*)

Comparison of Search Techniques

	DFS	BFS	UCS	IDS
Complete	N	Y	Y	Y
Optimal	N	N	Y	N
Heuristic	N	N	N	N
Time	b^m	b^{d+1}	b^m	b^d
Space	bm	b^{d+1}	b^m	bd

Bidirectional Search

- Search forward from initial state to goal AND backward from goal state to initial state
- Can prune many options
- Considerations
  - Which goal state(s) to use
  - How to determine when searches overlap
  - Which search to use for each direction
  - Here, two BFS searches
- Time and space is $O(b^{d/2})$



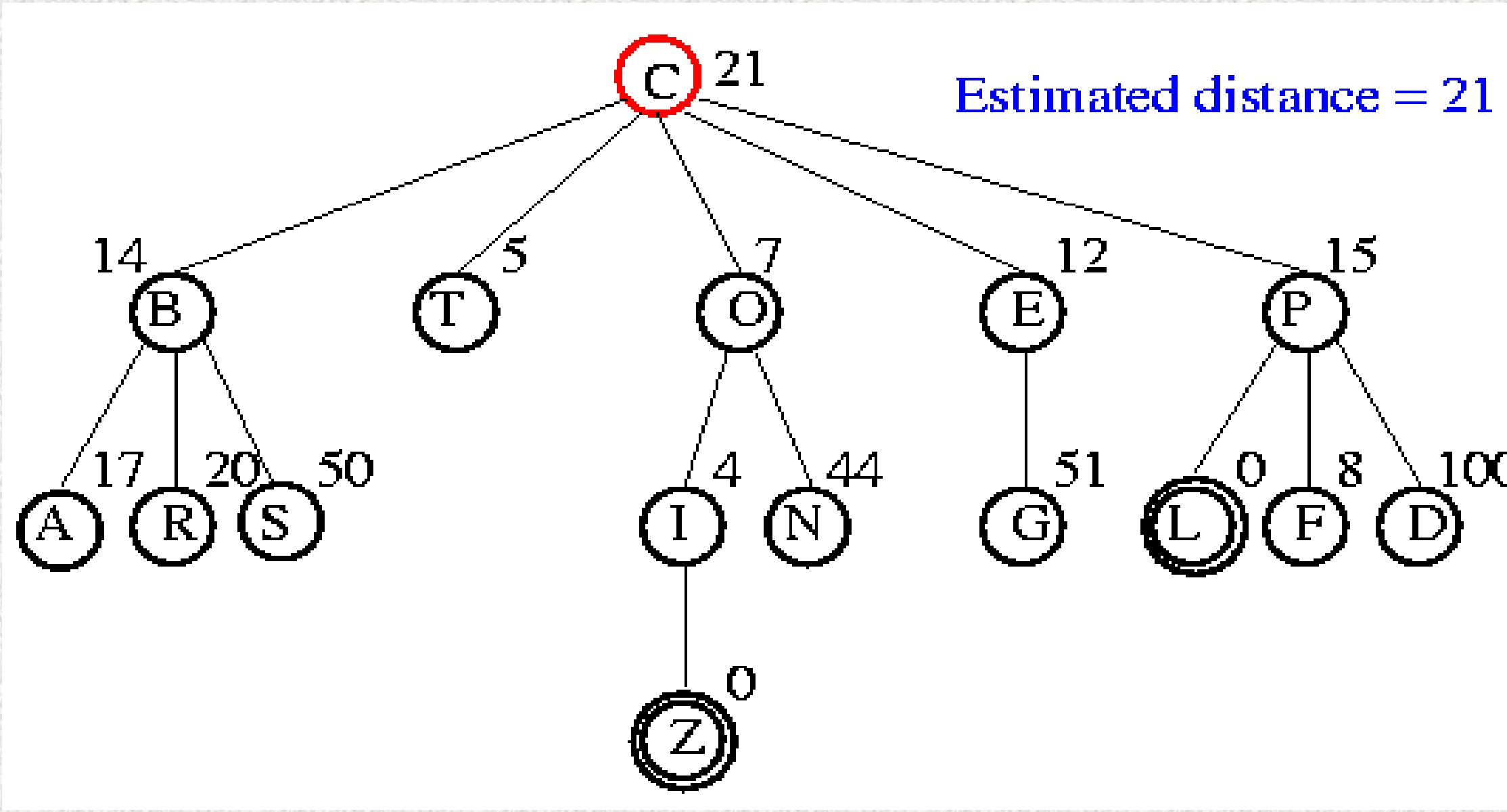
Informed Searches

- Best-first search, Hill climbing, A*
- New terms
  - Heuristics
  - Optimal solution
  - Informedness
  - Admissibility
- New parameters
  - g(n) = cost from initial state to state n
  - h(n) = estimated cost (distance) from state n to closest goal
  - h(n) is our heuristic
    - Robot path planning, h(n) could be Euclidean distance
    - 8 puzzle, h(n) could be #tiles out of place
- Search algorithms which use h(n) to guide search are heuristic search

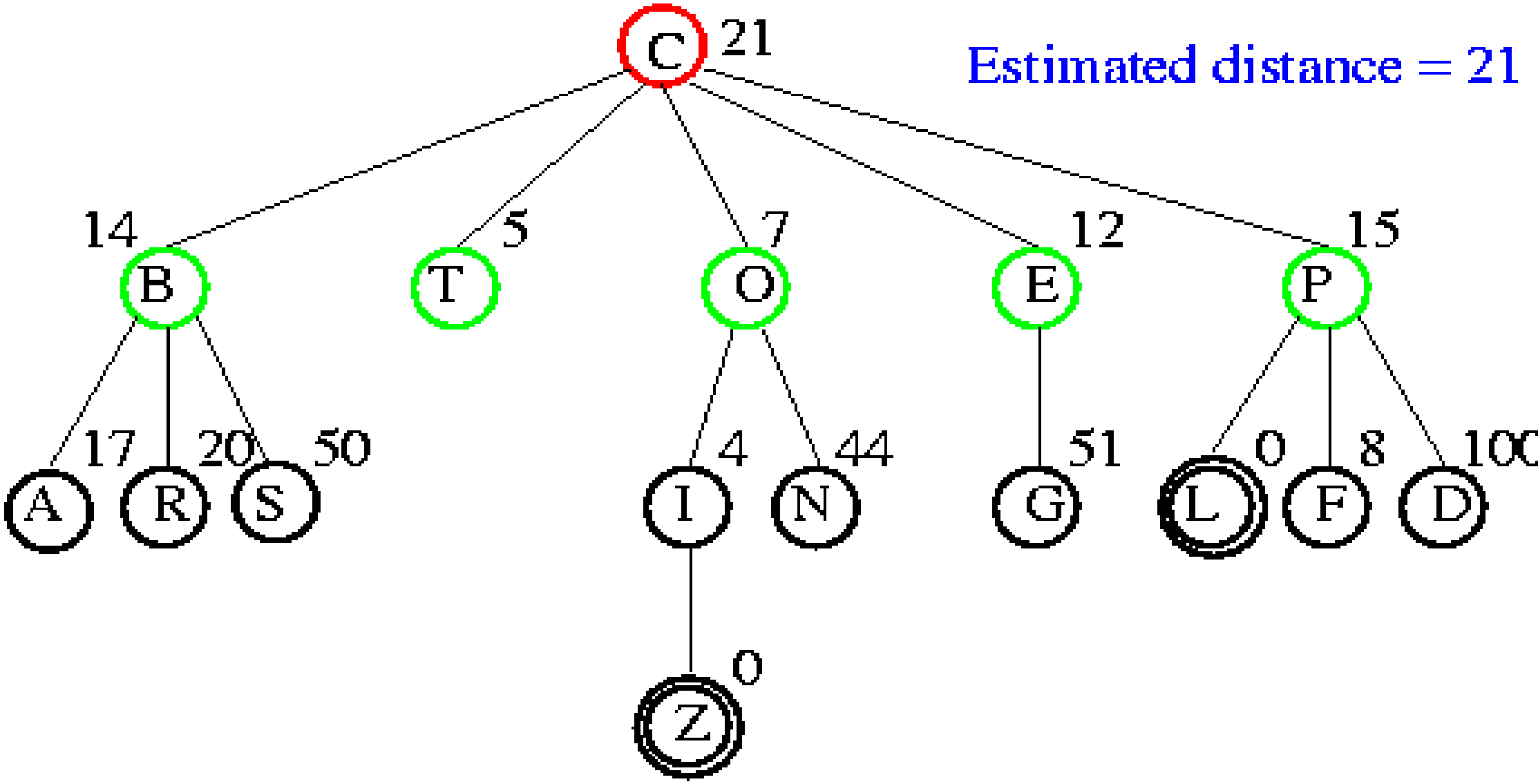
(Greedy) Best-First Search

- QueueingFn is sort-by-h
- Best-first search only as good as heuristic
  - Example heuristic for 8 puzzle:
    Manhattan Distance

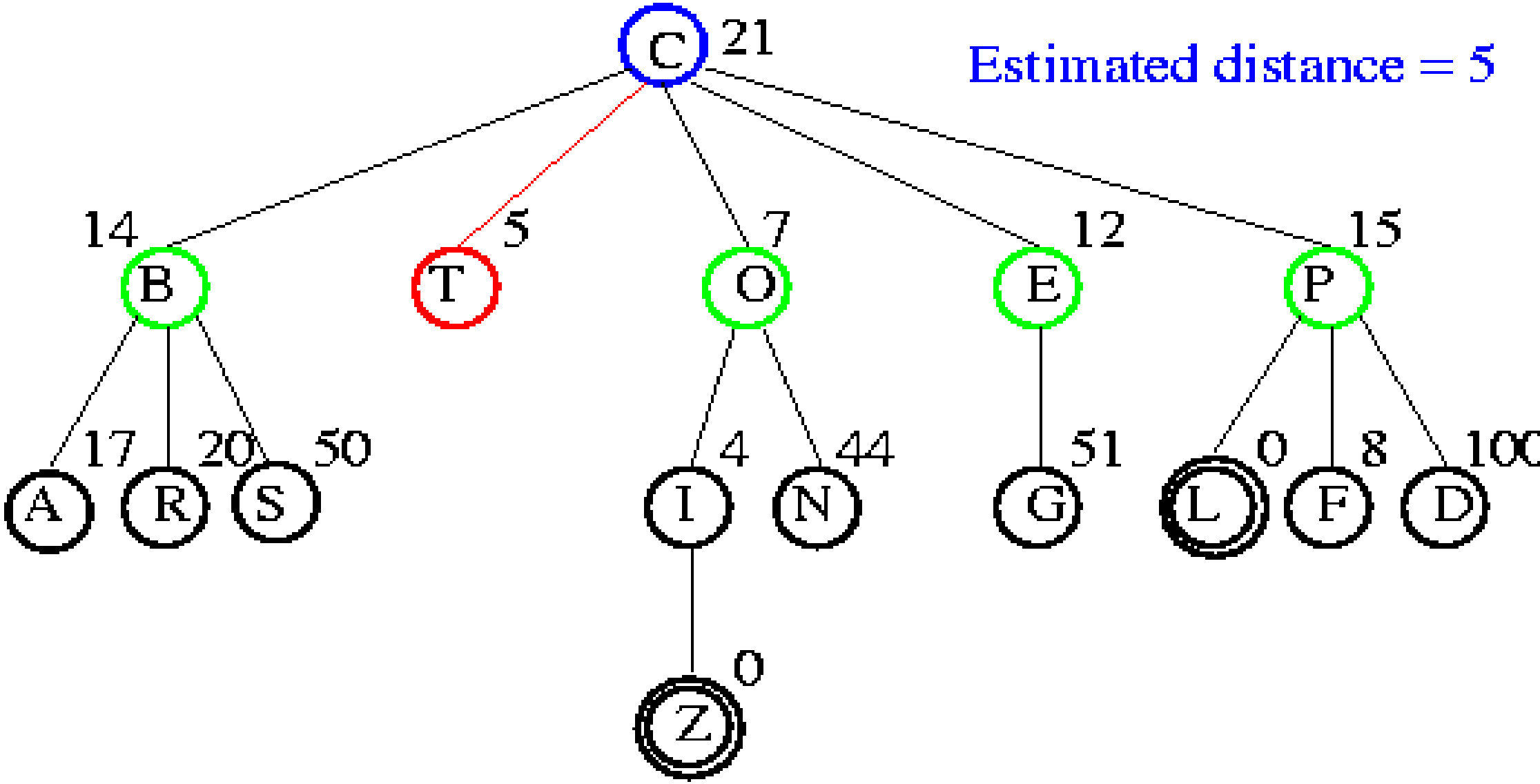
Example


C
21
Estimated distance = 21
14
B
T
5
O
7
E
12
P
15
17
A
20
R
50
S
4
I
44
N
51
G
0
L
8
F
100
D
Z
0

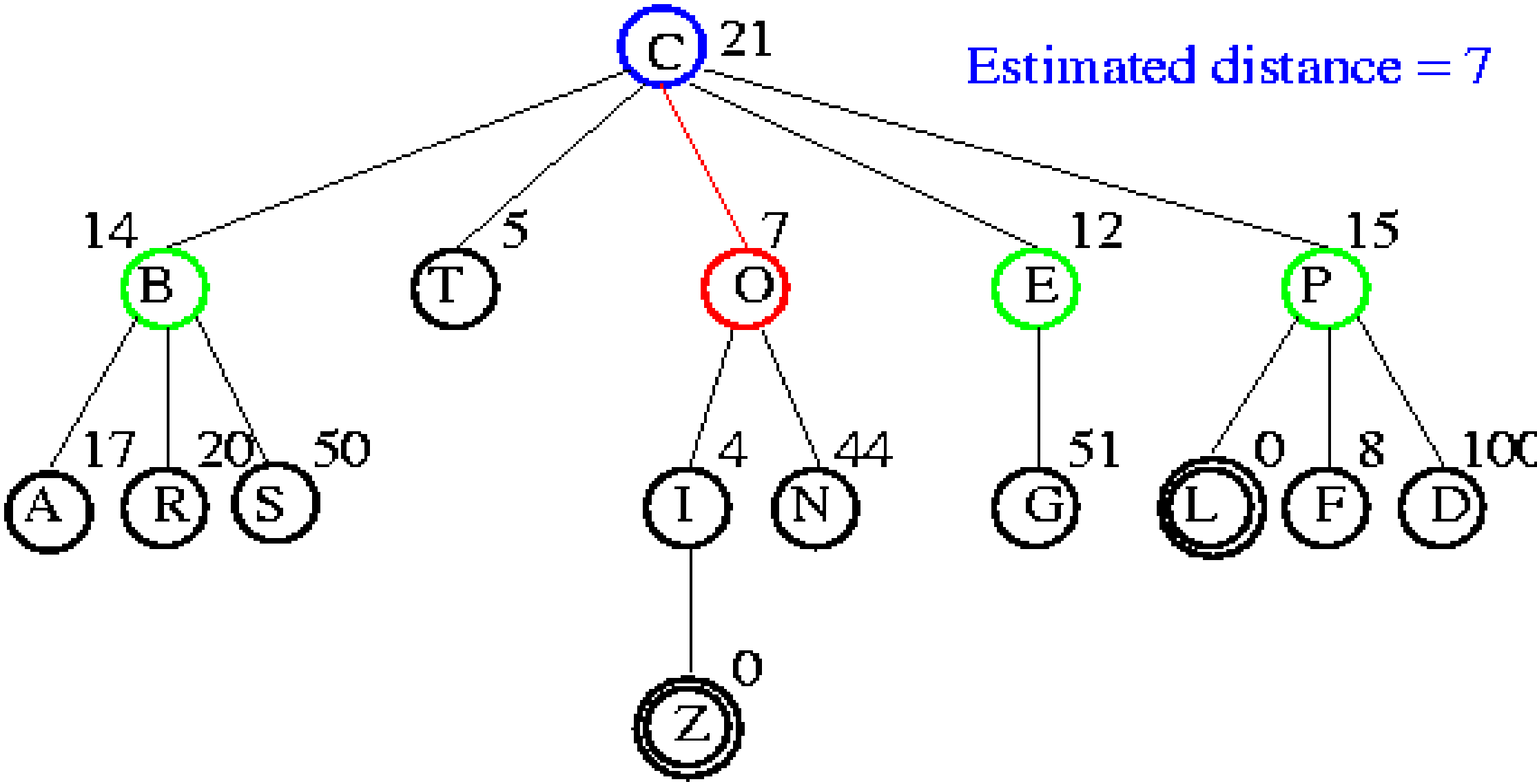
Example


C
21
Estimated distance = 21
14
B
T
5
7
O
E
12
15
P
A
17
R
20
S
50
I
4
N
44
G
51
L
0
F
8
D
100
Z
0

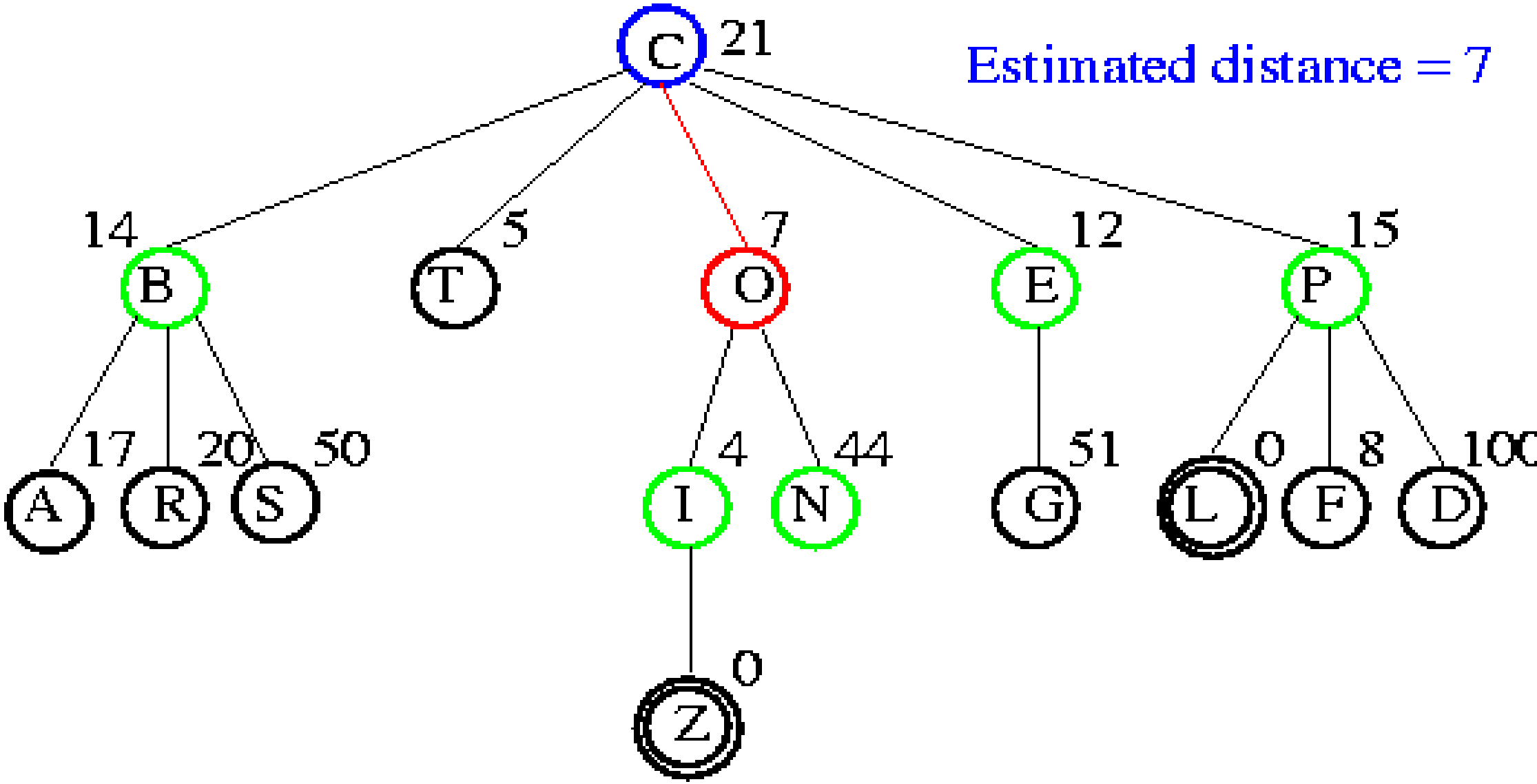
Example


C
21
Estimated distance = 5
14
B
T
5
7
O
E
12
P
15
17
20
50
4
44
51
0
8
100
A
R
S
I
N
G
L
F
D
0
Z

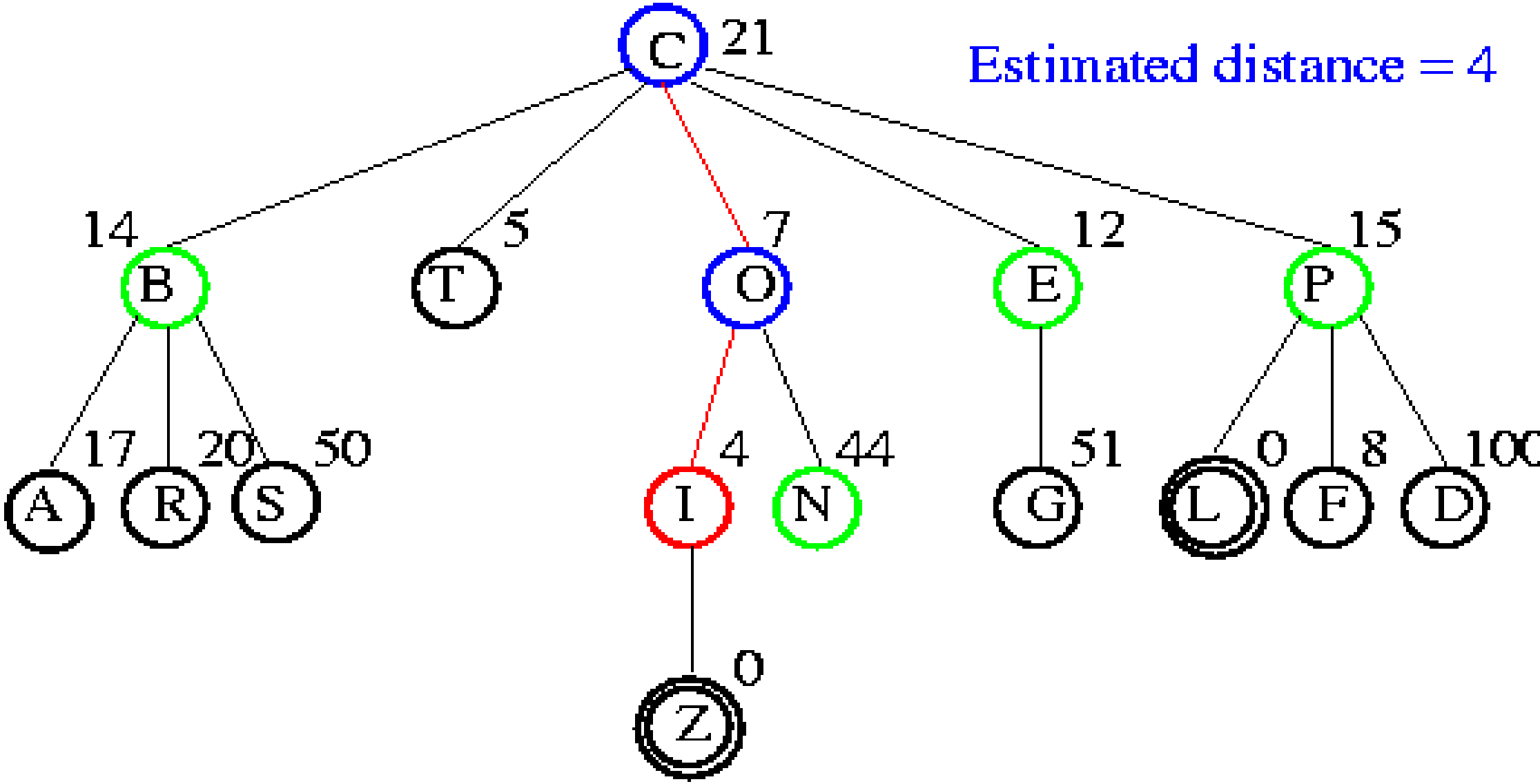
Example


C
21
Estimated distance = 7
14
B
T
5
7
O
E
12
P
15
17
20
50
4
44
51
0
8
100
A
R
S
I
N
G
L
F
D
0
Z

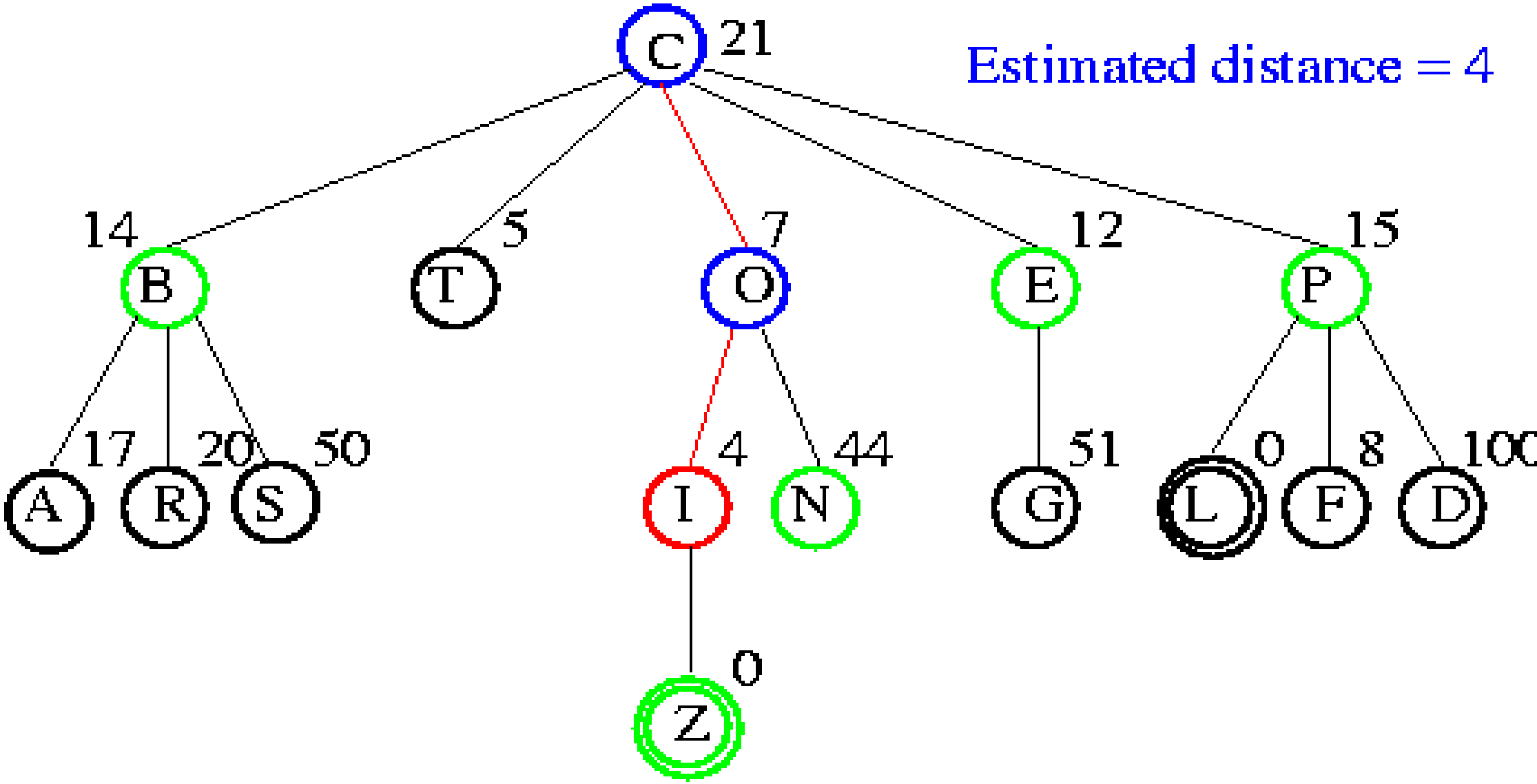
Example


C
21
Estimated distance = 7
14
B
T
5
7
O
E
12
P
15
17
20
50
4
44
51
0
8
100
A
R
S
I
N
G
L
F
D
Z
0

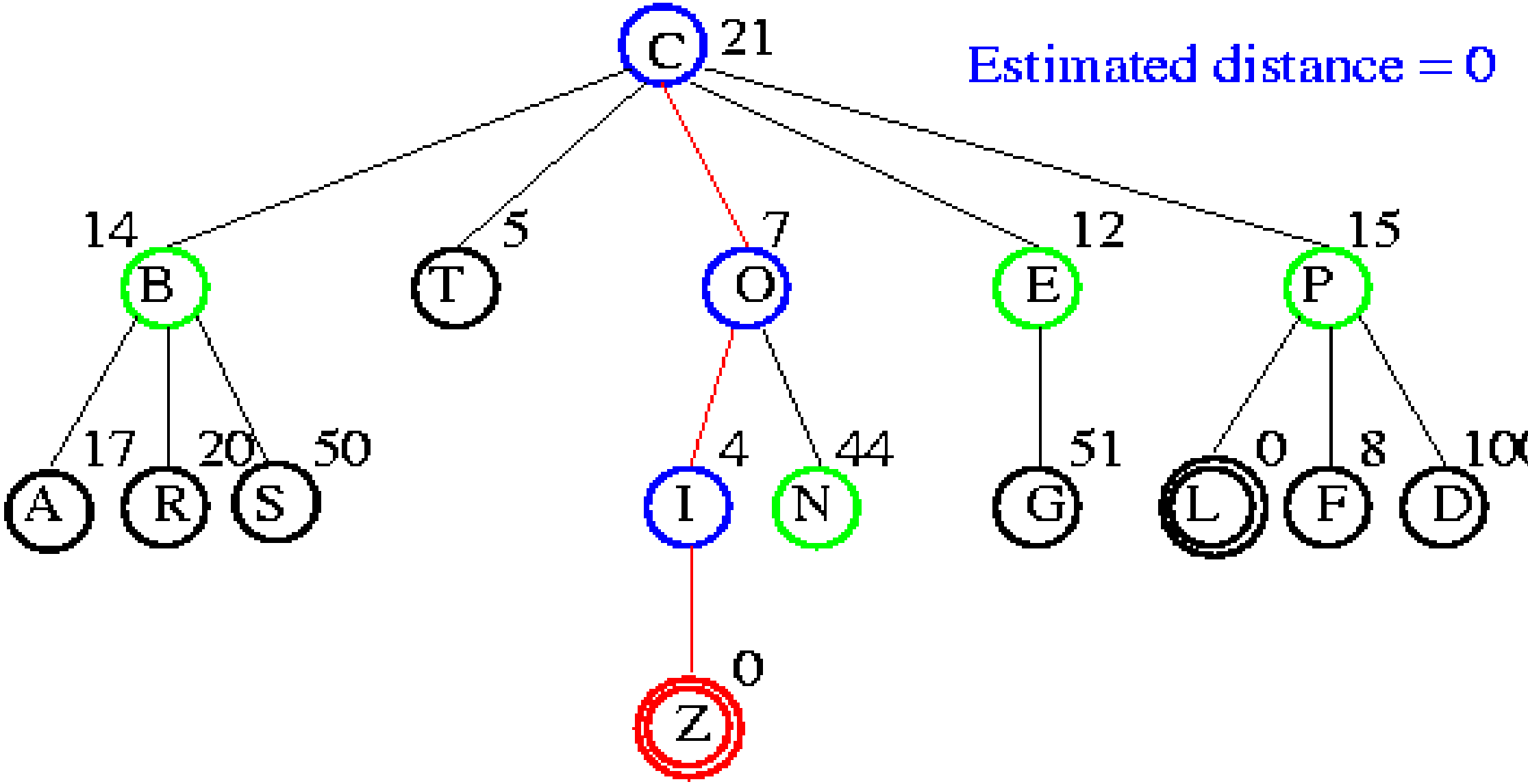
Example


C
21
Estimated distance = 4
14
B
T
5
7
O
E
12
P
15
17
20
50
4
44
51
0
8
100
A
R
S
I
N
G
L
F
D
0
Z

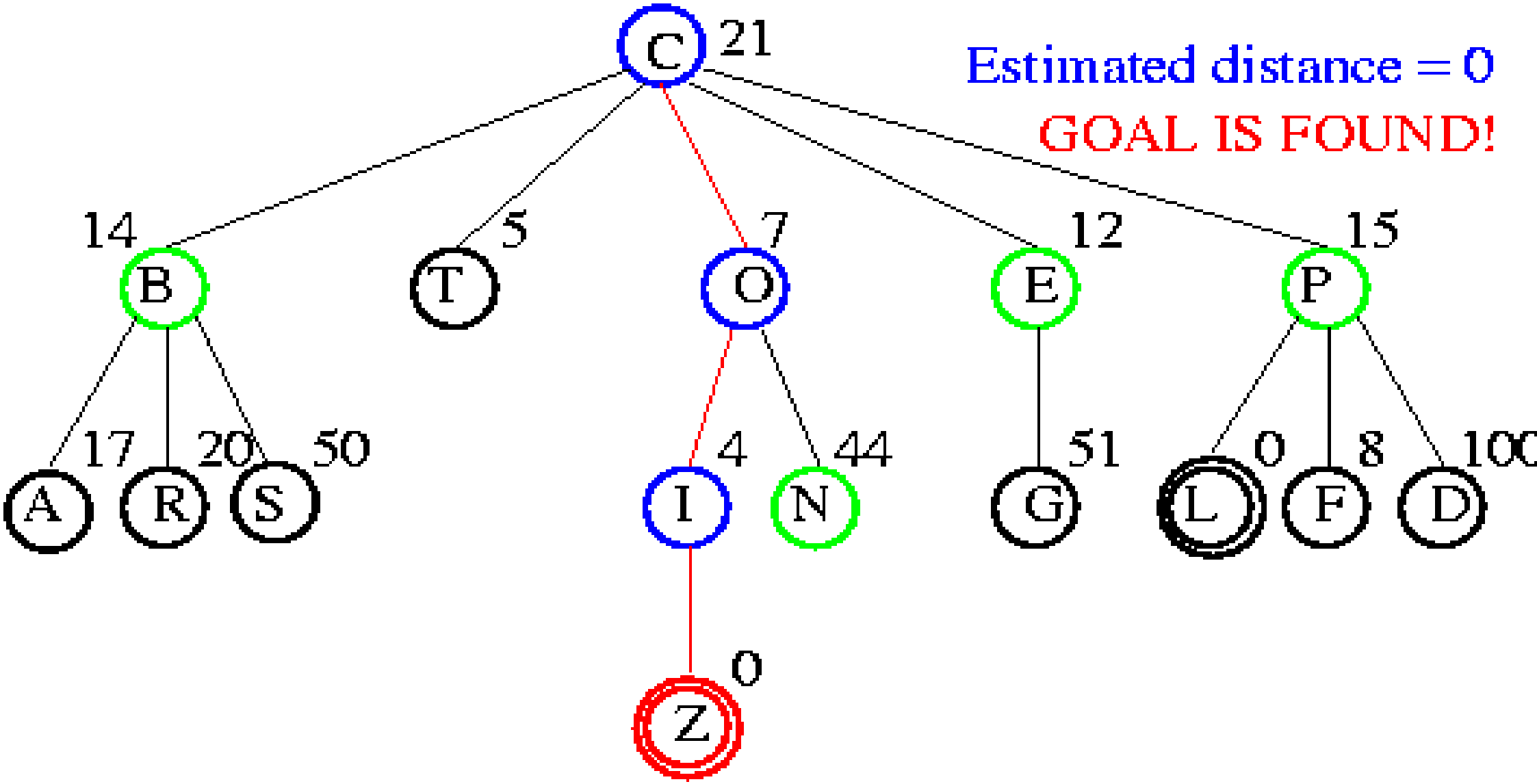
Example


C
21
Estimated distance = 4
14
B
T
5
7
O
E
12
P
15
17
20
50
4
44
51
0
8
100
A
R
S
I
N
G
L
F
D
0
Z

Example


C
21
Estimated distance = 0
14
B
T
5
7
O
E
12
15
P
17
20
50
4
44
51
0
8
10
A
R
S
I
N
G
L
F
D
0
Z

Example


C
21
Estimated distance = 0
GOAL IS FOUND!
14
B
T
5
7
O
E
12
P
15
17
20
50
A
R
S
4
44
I
N
51
G
0
8
100
L
F
D
0
Z

Comparison of Search Techniques

	DFS	BFS	UCS	IDS	Best
Complete	N	Y	Y	Y	N
Optimal	N	N	Y	N	N
Heuristic	N	N	N	N	Y
Time	b^m	b^{d+1}	b^m	b^d	b^m
Space	bm	b^{d+1}	b^m	bd	b^m

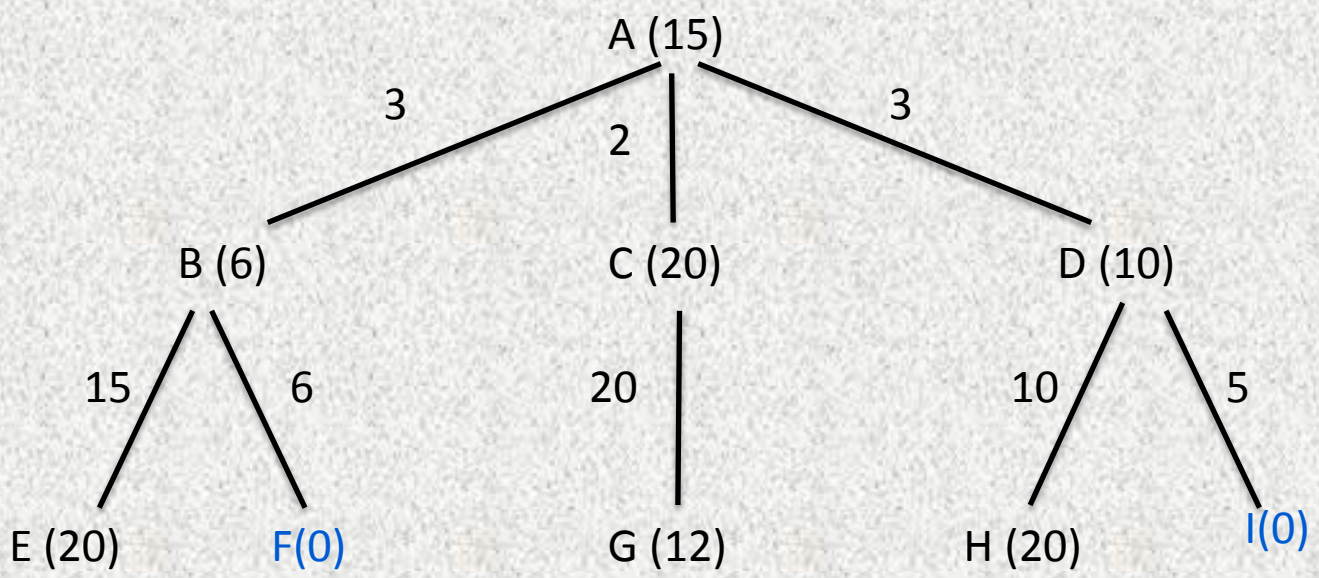
A*

- QueueingFn is sort-by-f
  - f(n) = g(n) + h(n)
- Note that UCS and Best-first both improve search
  - UCS keeps solution cost low
  - Best-first helps find solution quickly
- A* combines these approaches

Power of f(n)

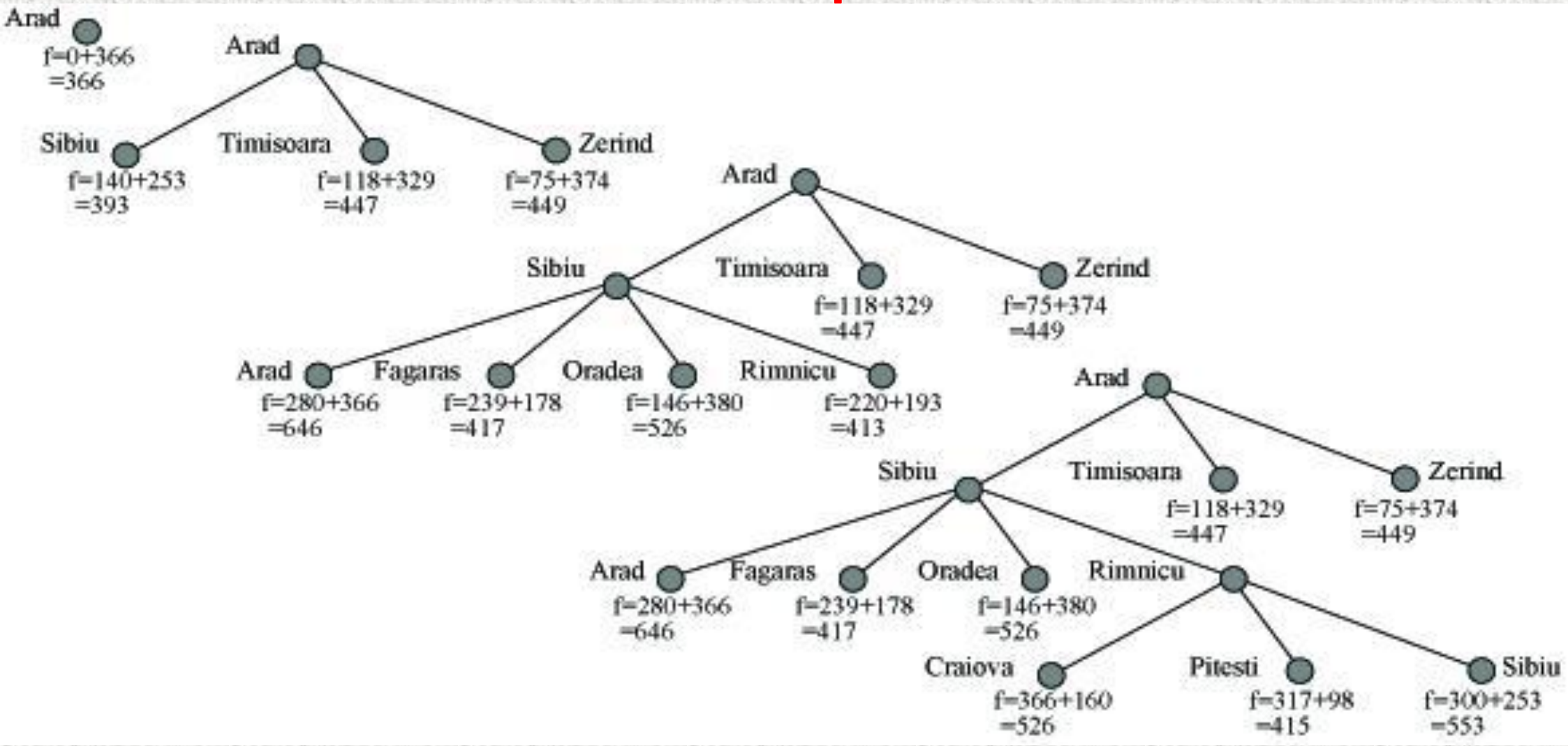
- If heuristic function is wrong it either
  - overestimates (guesses too high)
  - underestimates (guesses too low)
- Overestimating is worse than underestimating
- A* returns optimal solution if h(n) is admissible
  - heuristic function is admissible if never overestimates true cost to nearest goal
  - if search finds optimal solution using admissible heuristic, the search is admissible

Overestimating

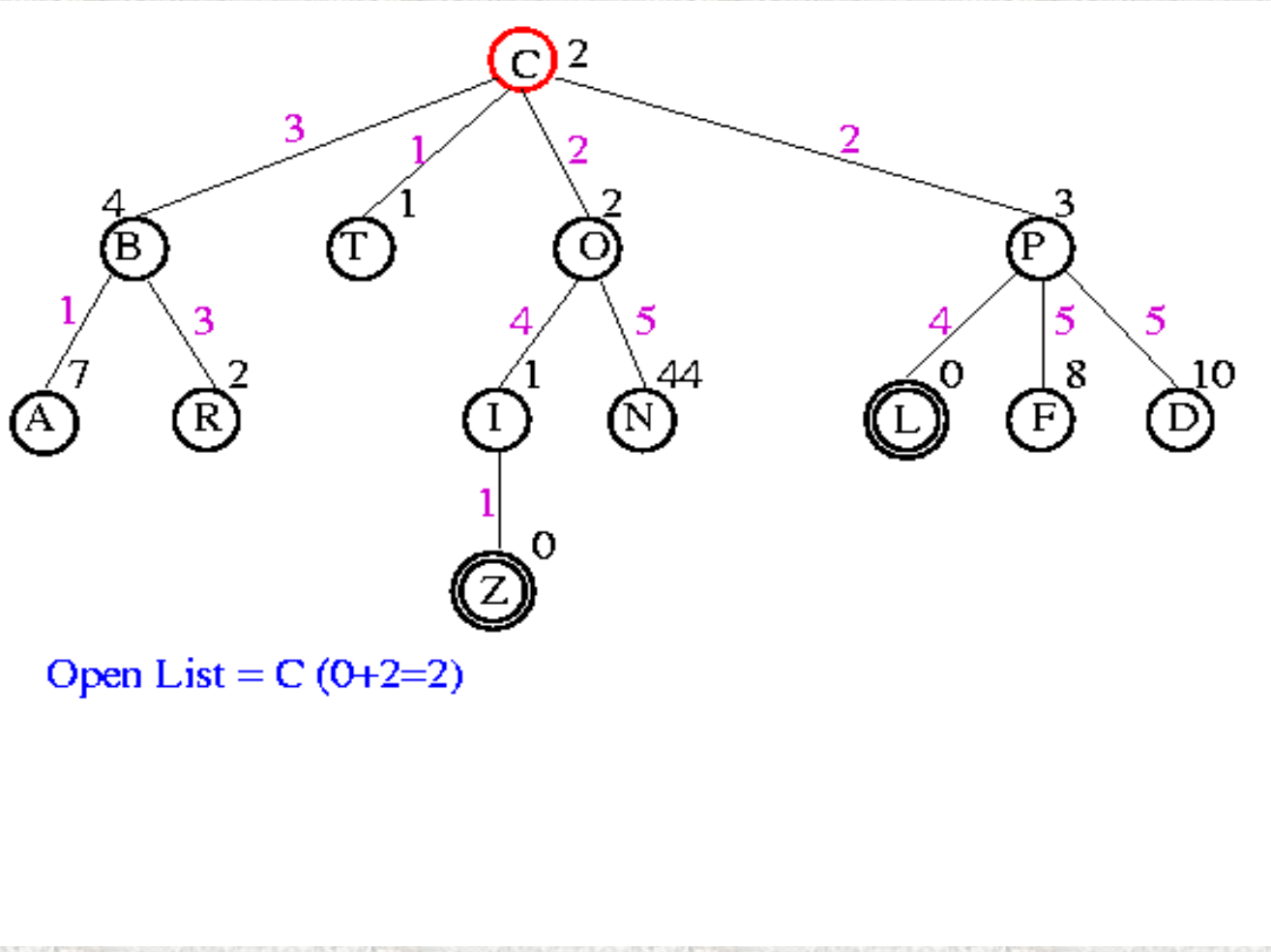


- Solution cost:
  - ABF = 9
  - ADI = 8
- Open list:
  - A (15) B (9) F (9)
- Missed optimal solution

Example

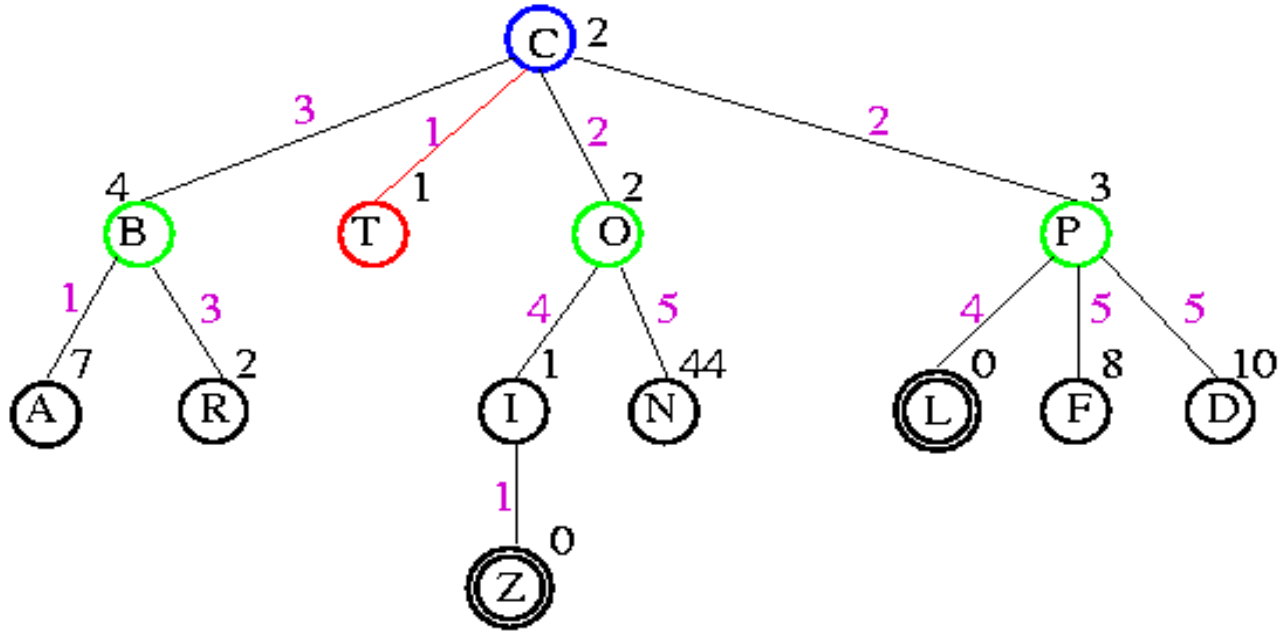

Arad
f=0+366
=366
Arad
Sibiu
f=140+253
=393
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
Sibiu
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
f=280+366
=646
Fagaras
f=239+178
=417
Oradea
f=146+380
=526
Rimnicu
f=220+193
=413
Arad
Sibiu
Timisoara
f=118+329
=447
Zerind
f=75+374
=449
Arad
f=280+366
=646
Fagaras
f=239+178
=417
Oradea
f=146+380
=526
Rimnicu
Craiova
f=366+160
=526
Pitesti
f=317+98
=415
Sibiu
f=300+253
=553

Example



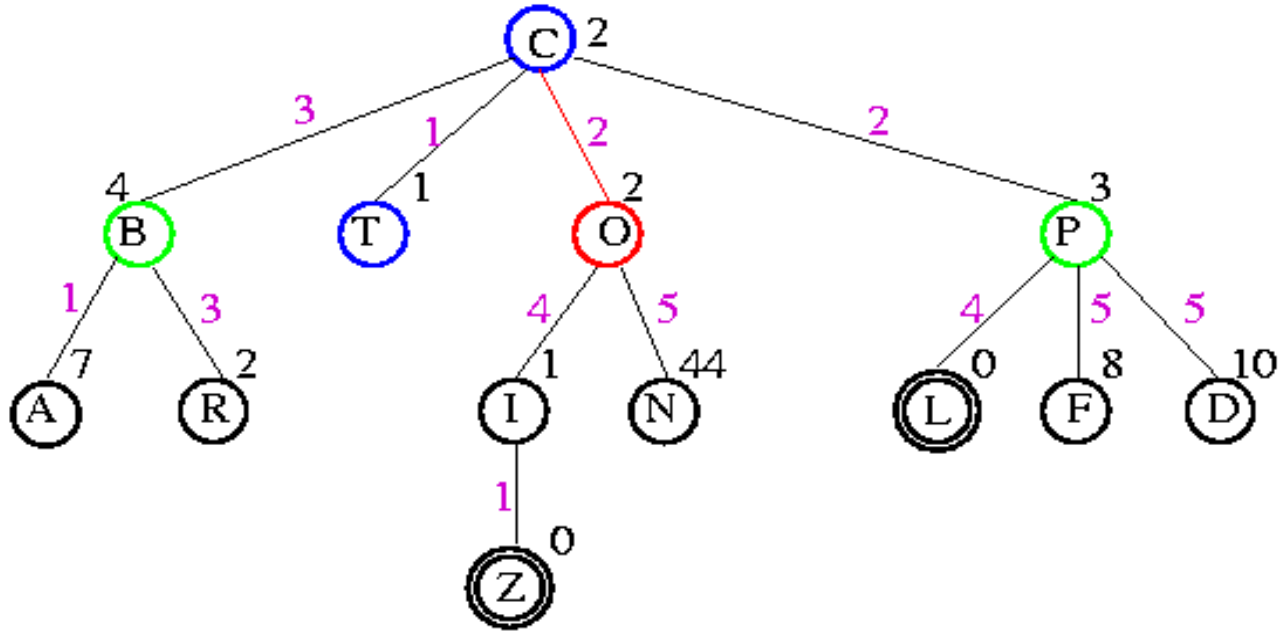
Open List = C (0+2=2)

Example



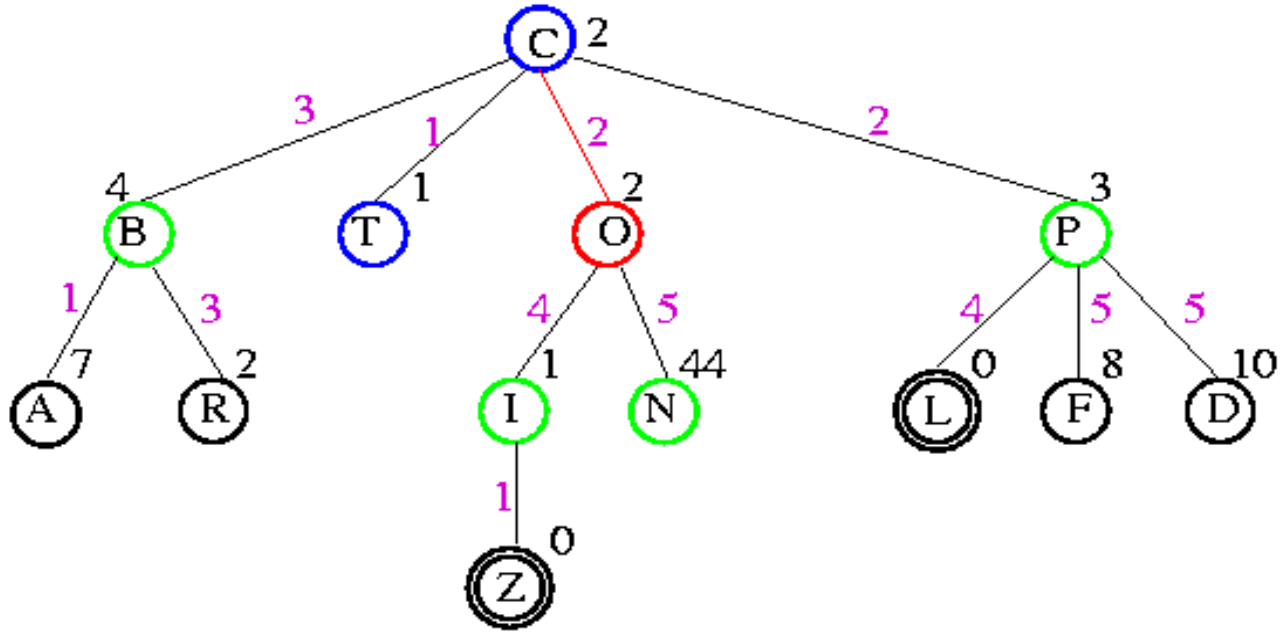
Open List = T (1+1=2), O (2+2=4), P (2+3=5), B(3+4=7)

Example



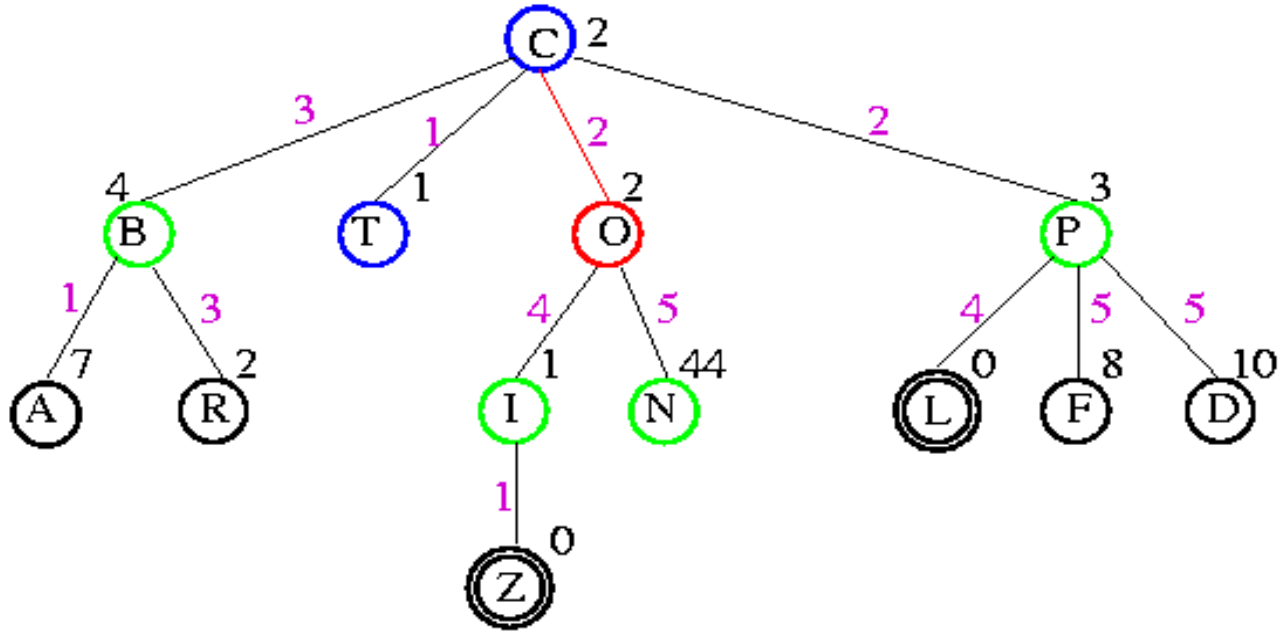
Open List = O (2+2=4), P (2+3=5), B(3+4=7)

Example



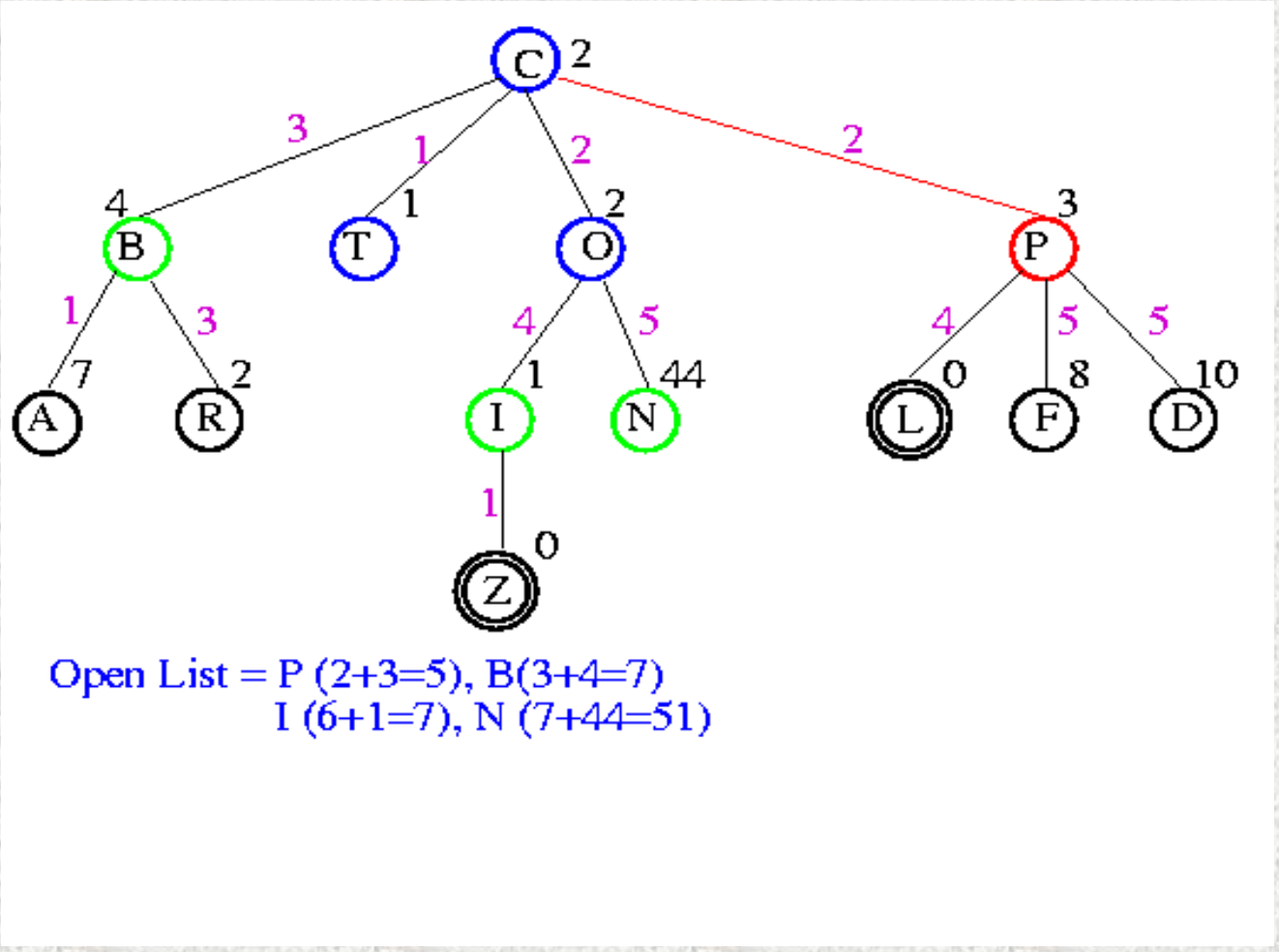
Open List = O (2+2=4), P (2+3=5), B(3+4=7)

Example



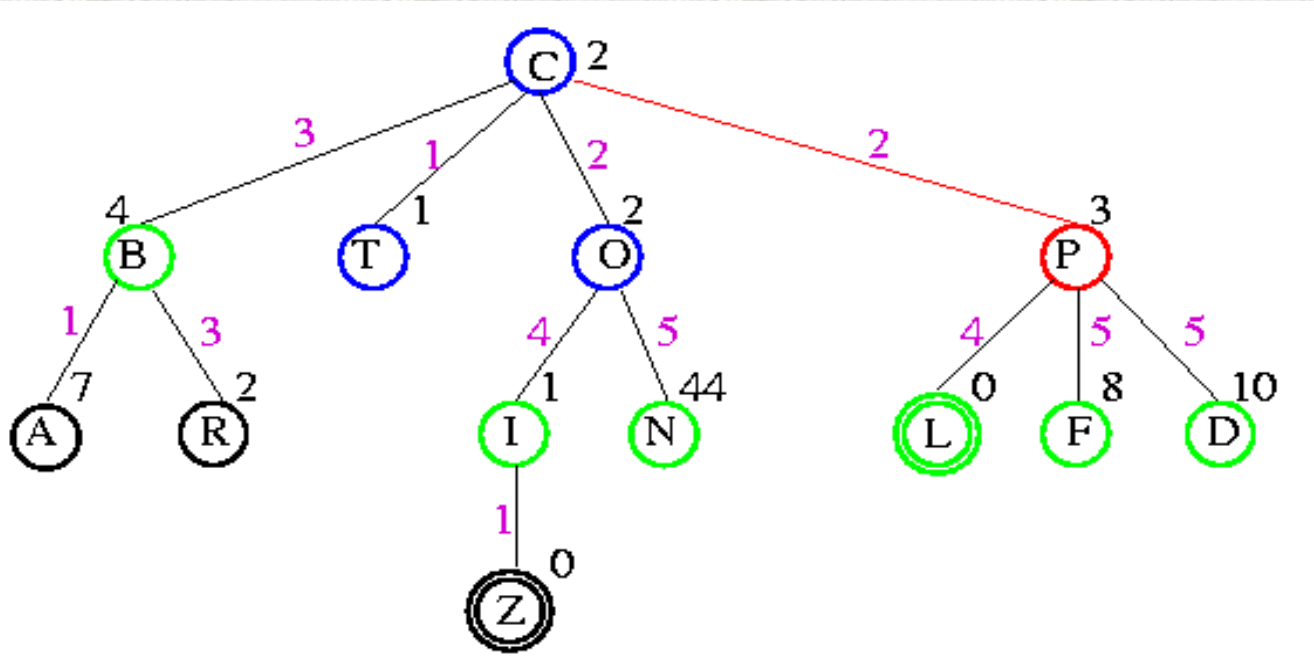
Open List = O (2+2=4), P (2+3=5), B(3+4=7)
I (6+1=7), N (7+44=51)

Example



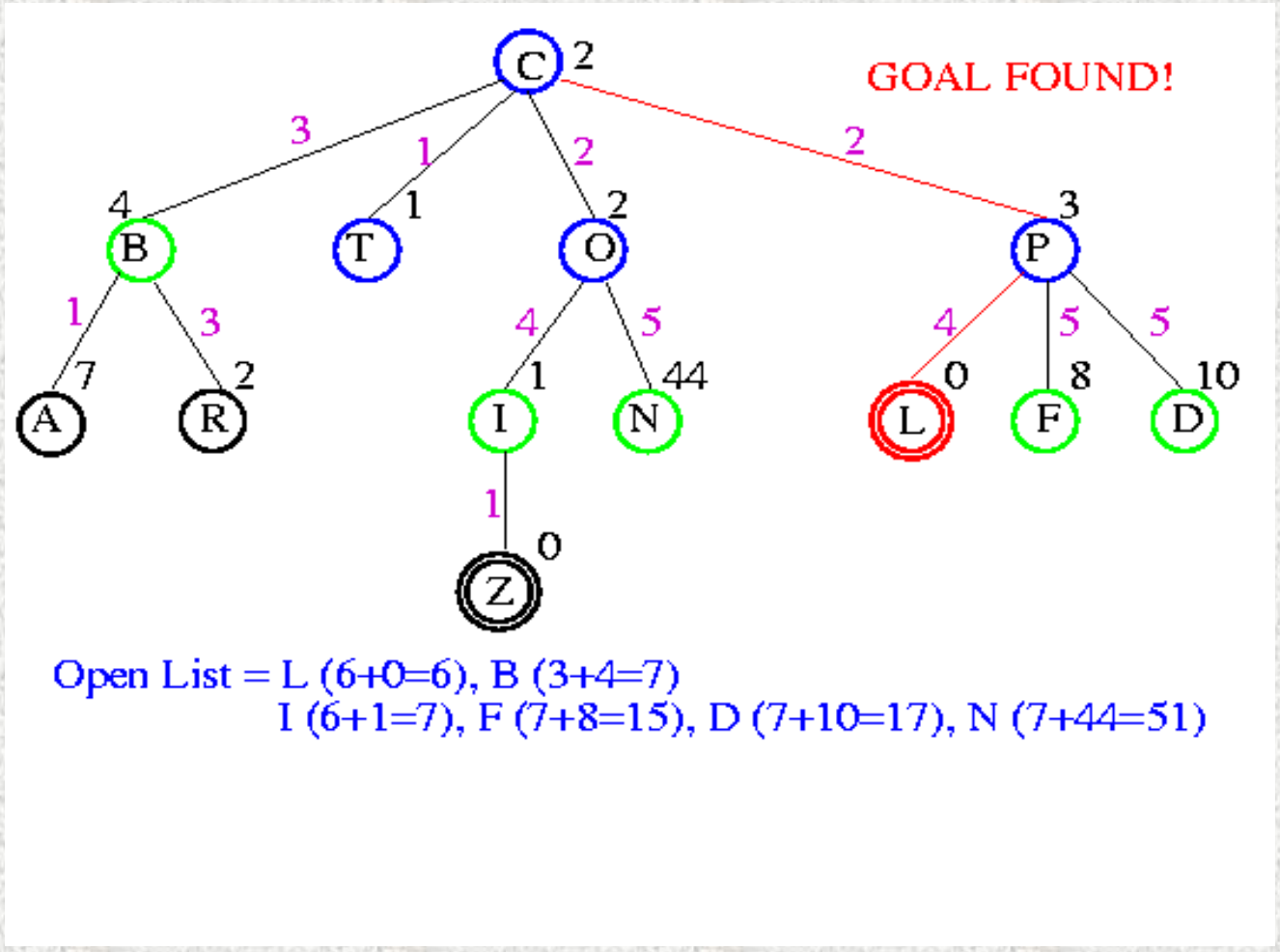
Open List = P (2+3=5), B(3+4=7)
I (6+1=7), N (7+44=51)

Example



Open List = P (2+3=5), L (6+0=6), B (3+4=7)
I (6+1=7), F (7+8=15), D (7+10=17), N (7+44=51)

Example



Open List = L (6+0=6), B (3+4=7)
I (6+1=7), F (7+8=15), D (7+10=17), N (7+44=51)

Comparison of Search Techniques

	DFS	BFS	UCS	IDS	Best	A*
Complete	N	Y	Y	Y	N	Y
Optimal	N	N	Y	N	N	Y
Heuristic	N	N	N	N	Y	Y
Time	b^m	b^{d+1}	b^m	b^d	b^m	b^m
Space	bm	b^{d+1}	b^m	bd	b^m	b^m

Affect of h on A*

The heuristic can be used to control A*'s behavior. At one extreme,

if h(n) is 0, then only g(n) plays a role, and A* turns into UCS. And is guaranteed to find a shortest path.

If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower.

If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else, making it very fast. It's nice to know that given perfect information, A* will behave perfectly.

If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.

At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into Greedy Best-First-Search.

Admissible Heuristics

- A* search is optimal when using an admissible heuristic function *h*.
- How do we devise good heuristic functions for a given problem? Typically, that depends on the problem domain.
- However, there are some general techniques that work reasonably well across several domains.

Heuristic Functions

- Q: Given that we will only use heuristic functions that do not overestimate, what type of heuristic functions (among these) perform best?
- A: Those that produce higher h(n) values.

Reasons

- Higher h value means closer to actual distance
- Any node n on open list with
  - $f(n) < f^*(goal)$
  - will be selected for expansion by A*
- This means if a lot of nodes have a low underestimate (lower than actual optimum cost)
  - All of them will be expanded
  - Results in increased search time and space

Informedness

- If h1 and h2 are both admissible and
- For all x, h1(x) > h2(x), then h1 “dominates” h2
  - Can also say h1 is “more informed” than h2
- Example
  - h1(x): $|x_{goal} - x|$
  - h2(x): Euclidean distance $\sqrt{(x_{goal} - x)^2 + (y_{goal} - y)^2}$
  - h2 dominates h1

Effect on Search Cost

- If h2(n) >= h1(n) for all n (both are admissible)
  - then h2 dominates h1 and is better for search
- Typical search costs
  - d=14, IDS expands 3,473,941 nodes
    - A* with h1 expands 539 nodes
    - A* with h2 expands 113 nodes
  - d=24, IDS expands ~54,000,000,000 nodes
    - A* with h1 expands 39,135 nodes
    - A* with h2 expands 1,641 nodes

Examples of Admissible Heuristics

- Consider the 8-puzzle problem:
  - $h_1(n)$ = number of tiles in the wrong position at state n
  - $h_2(n)$ = sum of the Manhattan distances of each tile from its goal position.

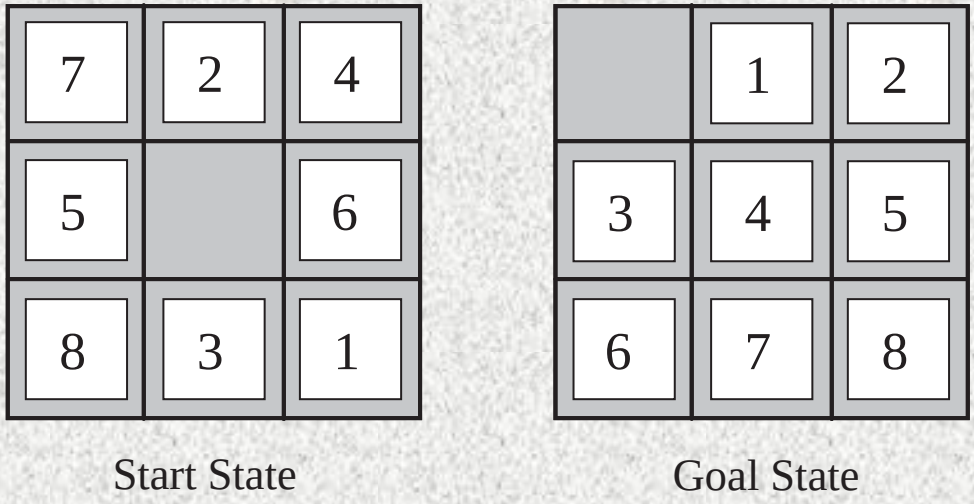


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

h_1(Start) = 8

h_2(Start) = 3+1+2+2+2+3+3+2 = 18

Dominance

Definition: A heuristic function h_2 dominates a heuristic function h_1 for the same problem if $h_2(n) \geq h_1(n)$ for all nodes n.

- For the 8-puzzle, h_2 = *total Manhattan distance* dominates h_1 = *number of misplaced tiles*.
- With A* search, a heuristic function h2 is always better for search than a heuristic function h1, if h2 is admissible and dominates h1.

  The reason is that A* with h_1 is guaranteed to expand at least all as many nodes as A* with h_2.

  What if neither of h_1, h_2 dominates the other? If both h_1, h_2 are admissible, use $h(n) = max(h_1(n), h_2(n))$.

Effectiveness of Heuristic Functions

Definition Let

h be a heuristic function for A*,
N the total number of nodes expanded by one A* search with *h*,
d the depth of the found solution.

The effective branching Factor (EBF) of *h* is the value b^* that solves the equation

$$x^d + x^{d-1} + \cdots + x^2 + x + 1 = N$$

(the branching factor of a uniform tree with *N* nodes and depth *d*).

A heuristics *h* for A* is effective in practice if its average EBF is close to 1.

Note: If h_2 dominates some h_1, its EFB is never greater than h_1's.

Dominance and EFB: The 8-puzzle

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

Average values over 1200 random instances of the problem Search cost = no. of expanded nodes

IDS = iterative deepening search

$A^*(h_1)$ = A* with h_1 = number of misplaces tiles

$A^*(h_2)$ = A* with h_2 = total Manhattan distance

Generating Heuristic Functions

- Generate heuristic for simpler (relaxed) problem
  - Relaxed problem has fewer restrictions
  - Eight puzzle where multiple tiles can be in the same spot
  - Cost of optimal solution to relaxed problem is an admissible heuristic for the original problem
- Pattern Databases
- Learn heuristic from experience

Devising Heuristic Functions

A relaxed problem is a search problem in which some restrictions on the applicability of the next-state operators have been lifted.

Example

- original-8-puzzle: "A tile can move from position A to position B if A is adjacent to B and B is empty."
- relaxed-8-puzzle-1: "A tile can move from A to B if A is adjacent to B."
- relaxed-8-puzzle-2: "A tile can move from A to B if B is empty."
- relaxed-8-puzzle-3: "A tile can move from A to B."

The exact solution cost of a relaxed problem is often a good (admissible) heuristics for the original problem.

Key point: the optimal solution cost of the relaxed problem is no greater than the optimal solution cost of the original problem.

Relaxed Problems: Another Example

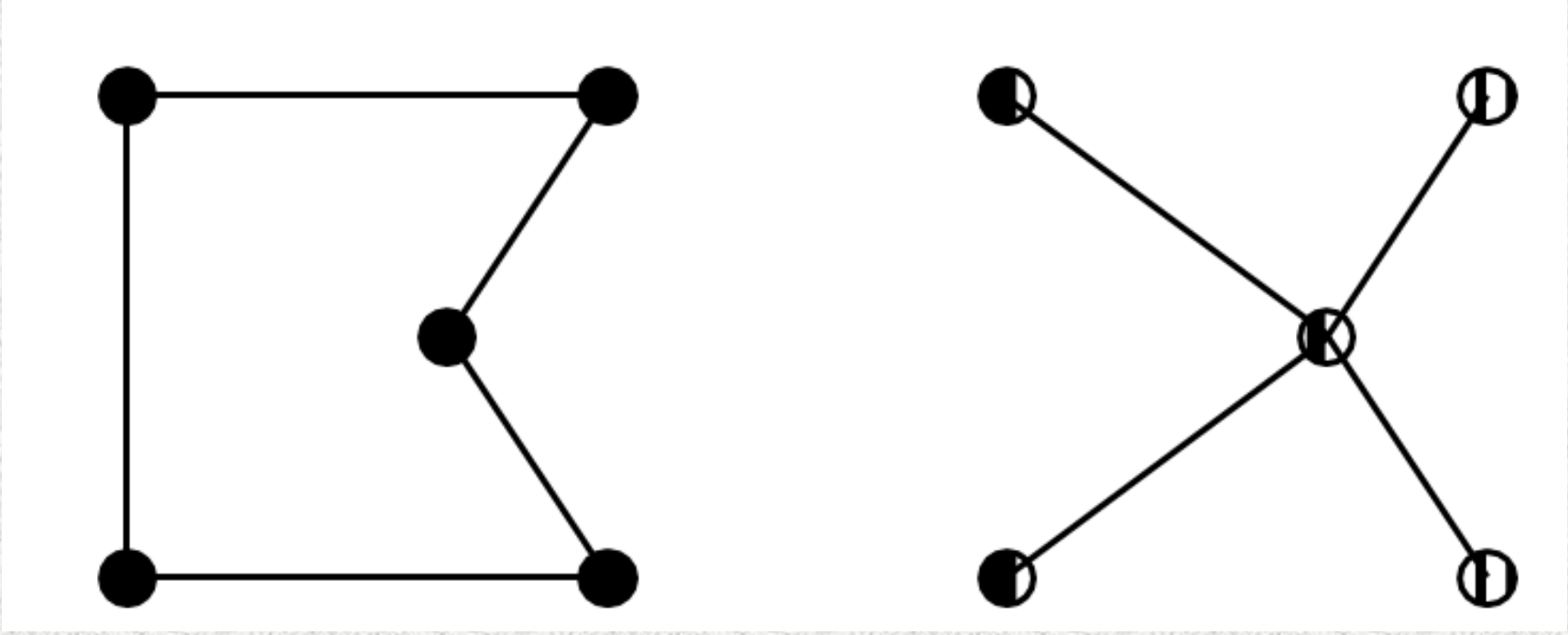
Original problem Traveling salesperson problem: Find the shortest tour visiting n cities exactly once.

Complexity: NP-complete.

Relaxed problem Minimum spanning tree: Find a tree with the smallest cost that connects the n cities.

Complexity: $O(n^2)$

Cost of tree is a lower bound on the shortest (open) tour.



End of Chapter 3