Artificial Intelligence

Lecture 3 - Problem Solving and Search II

Outline

- Preferred solutions
- Optimal search procedures
- Uniform cost search
- Informed search procedures
- A* search

Problem Definition

- A *search problem* is defined by:
  - a *state space* (i.e., an initial state or set of initial states and a set of operators)
  - a *set of goal states* (listed explicitly or given implicitly by means of a property that can be applied to a state to determine if it is a goal state)
- A *solution* is **any** path in the state space from an initial state to a goal state

Preferred Solutions

- One solution may be *preferable* to another - e.g., we may prefer paths with fewer or less costly actions
- In the route planning problem, we might prefer solutions which
  - minimise the distance travelled
  - the time taken to reach the goal
  - the number of cities (changes) if we are travelling by train
  - the monetary cost (of fuel or train tickets etc)
  - or some *combination* of these and other factors …

Path Cost

- A path cost function, *g(n)*, assigns a cost to a path *n* and can be used to rank alternative solutions
- If all operators have the same cost (e.g, moves in chess) the cost is simply the number of operator applications
- If different operators have different costs (e.g, money, time etc) the path cost is sum of the costs of all the operator applications in the path

Completeness and Optimality

- A search procedure which is guaranteed to find a *solution* (if one exists) is said to be *complete*
- A search procedure which is guaranteed to find a *least cost solution* (if a solution exists) is said to be *optimal*
- A search procedure which expands the *minimum number of nodes* necessary to find an optimal solution (if a solution exists) is said to be *optimally efficient*

Breadth-first Search

- Proceeds level by level down the search tree
- First explores all paths of length 1 from the root node, then all paths of length 2, length 3 etc.
- Starting from the root node (initial state) explores all children of the root node, left to right
- If no solution is found, expands the first (leftmost) child of the root node, then expands the second node at depth 1 and so on …

Properties of Breadth-first Search

- Breadth-first search is *complete* (even if the state space is infinite or contains loops)
- Guaranteed to find an *optimal solution* if cost is a non-decreasing function of the depth of a node - e.g., if all operators have the same cost
- Time and space complexity is $O(b^d)$ where d is the depth of the shallowest solution

Depth-first Search

- Proceeds down a single branch of the tree at a time
- Expands the root node, then the leftmost child of the root node, then the leftmost child of that node etc.
- Always expands a node at the deepest level of the tree
- Only when the search hits a dead end (a partial solution which can’t be extended) does the search *backtrack* and expand nodes at higher levels

Properties of Depth-first Search

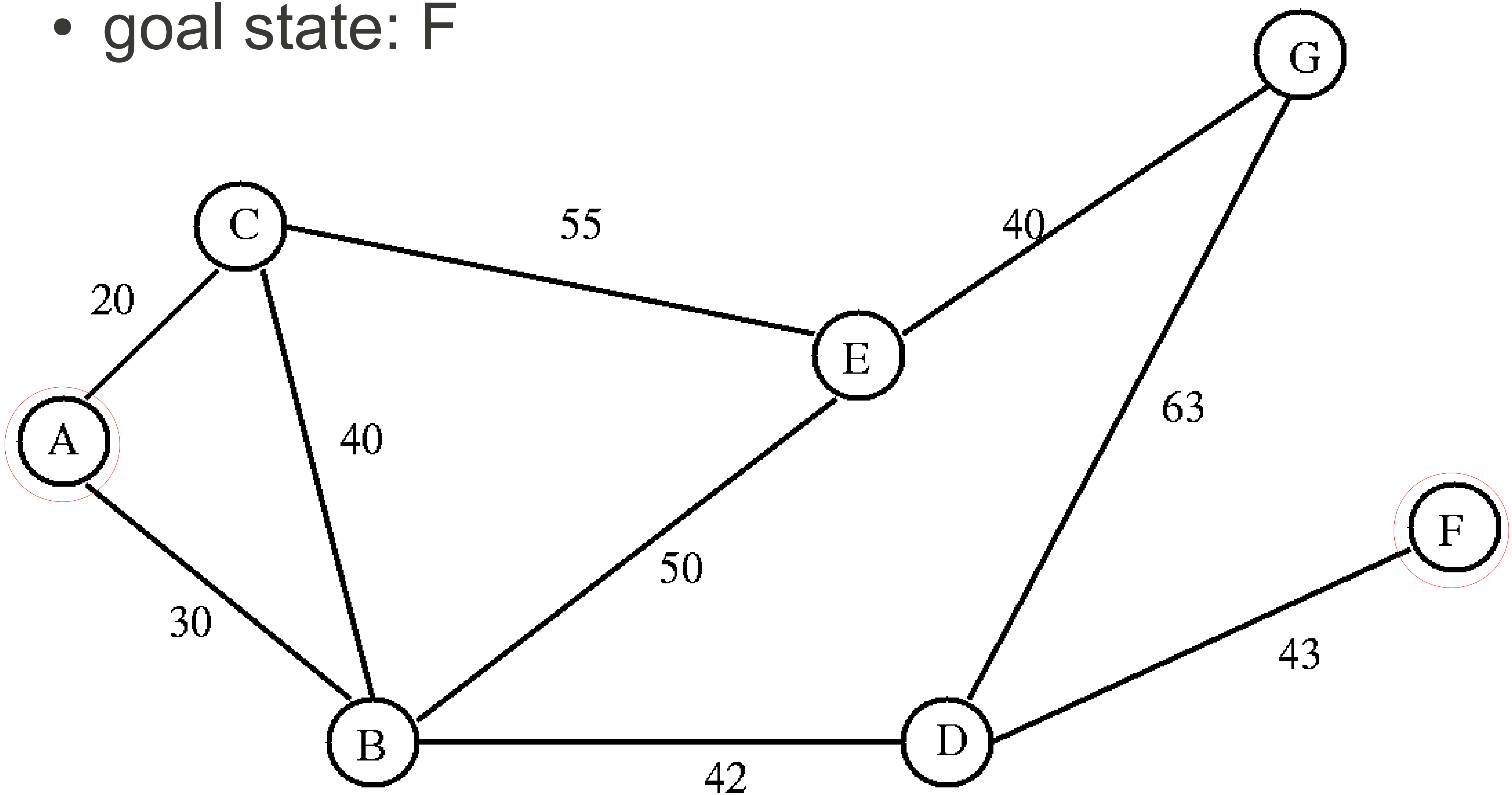
- Depth-first search requires much less memory than breadth-first search - space complexity is $O(bm)$ where m is the maximum depth of the tree
- Time complexity is $O(b^m)$
- However depth-first search is *not complete* (unless the state space is finite and contains no loops)
  - we may get stuck going down an infinite branch that doesn't lead to a solution
- Even if the state space is finite and contains no loops, it is **not** guaranteed to find an optimal solution

Uniform-cost Search

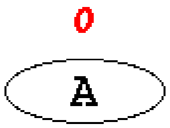
- Breadth-first search finds the *shallowest* goal state - this may not always be the least cost solution for a general path cost function
- *Uniform-cost search* expands leaf nodes in order of cost (as measured by the path cost *g(n)*)
- Expands the root node, then the lowest cost child of the root node, then the lowest cost unexpanded node etc.

Example: Simple Route Planning

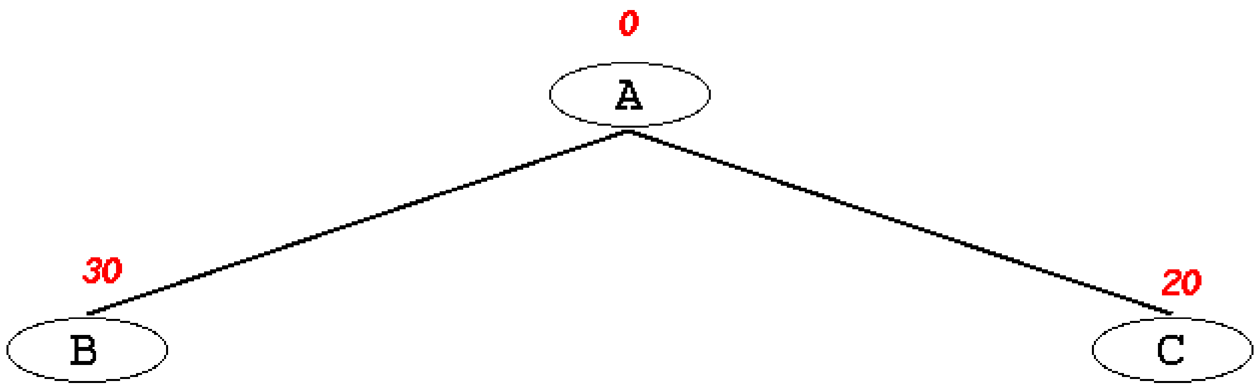
- initial state: A
- goal state: F



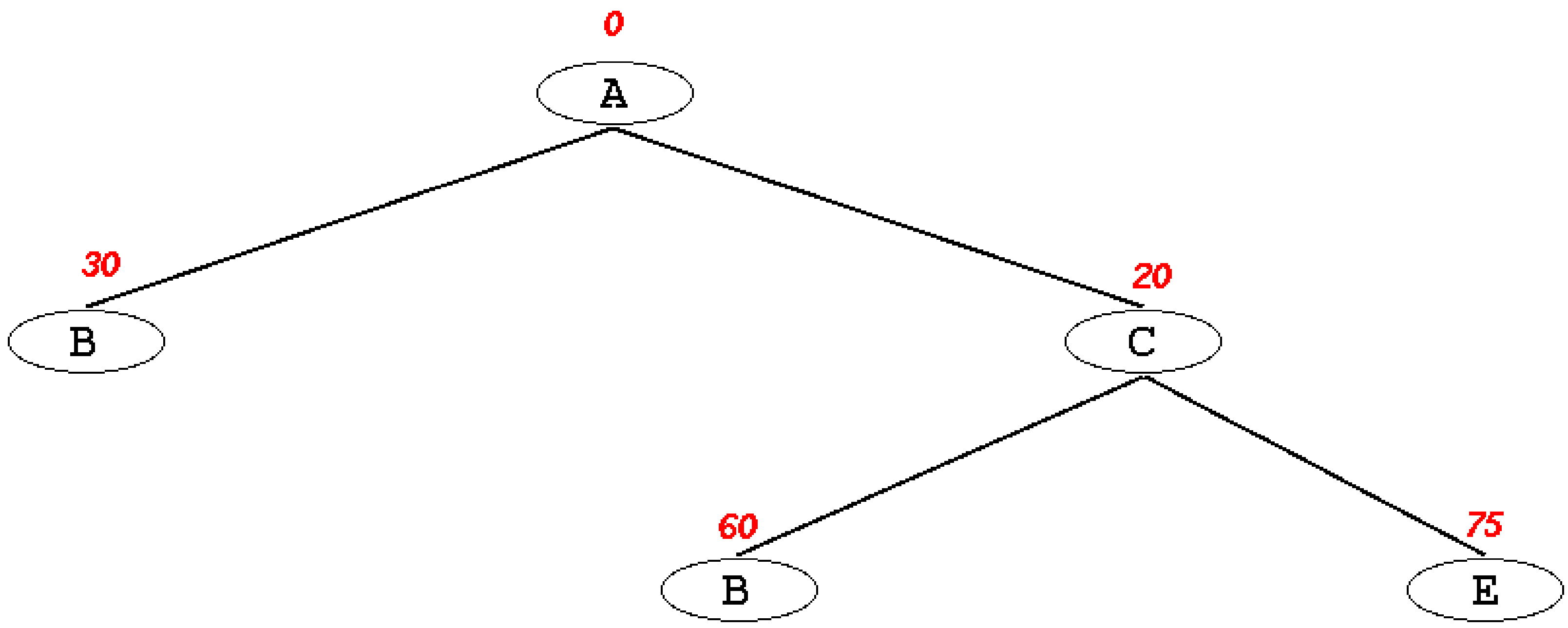
Example: Uniform-cost Search



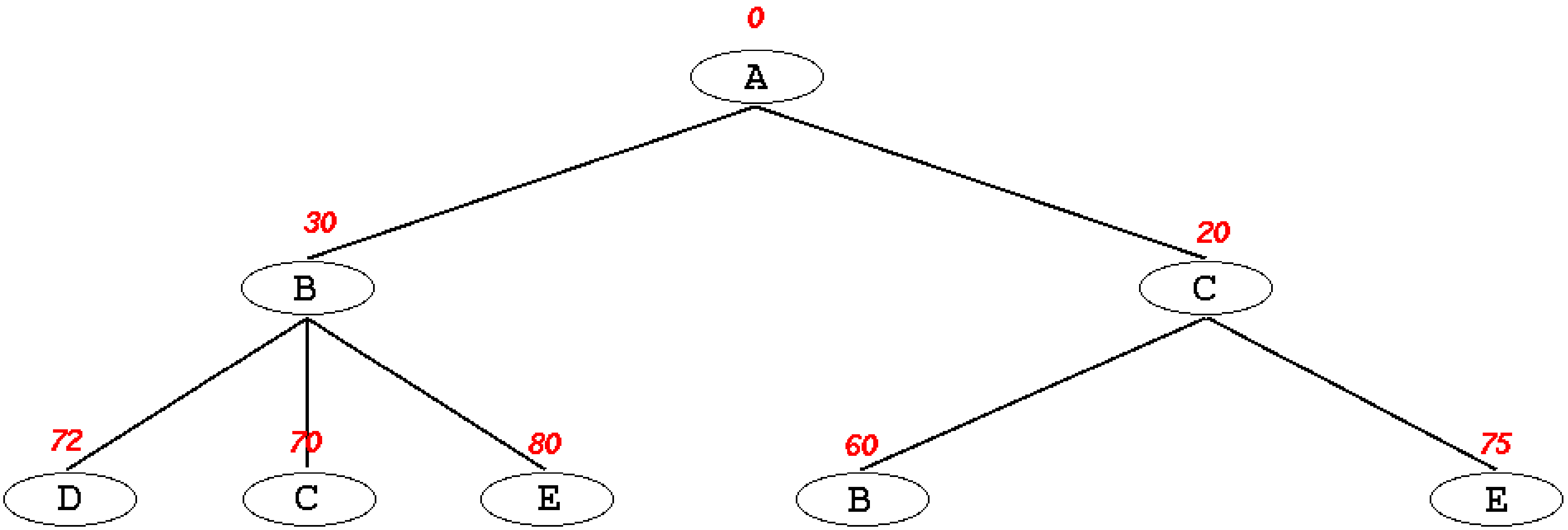
Example: Uniform-cost Search



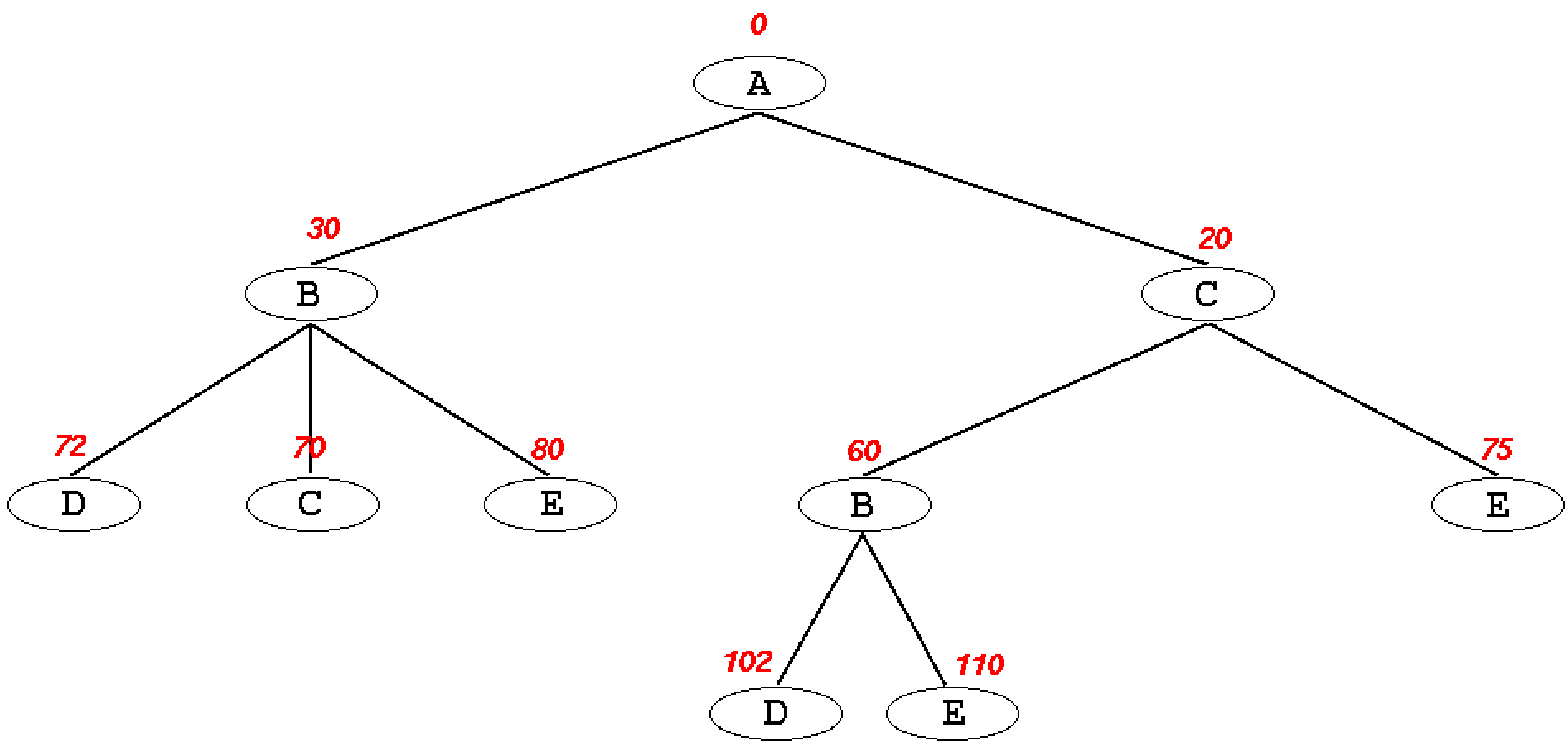
Example: Uniform-cost Search



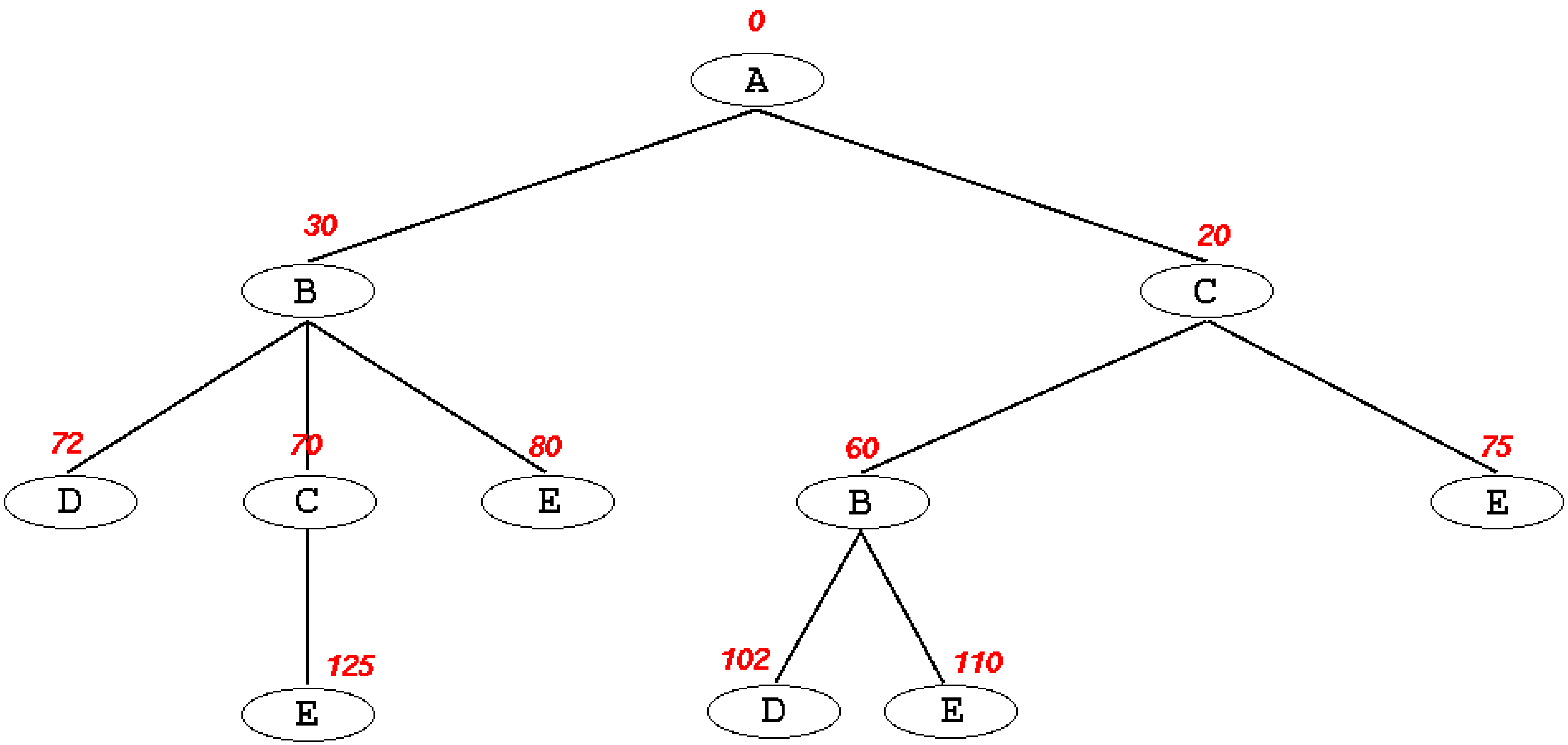
Example: Uniform-cost Search



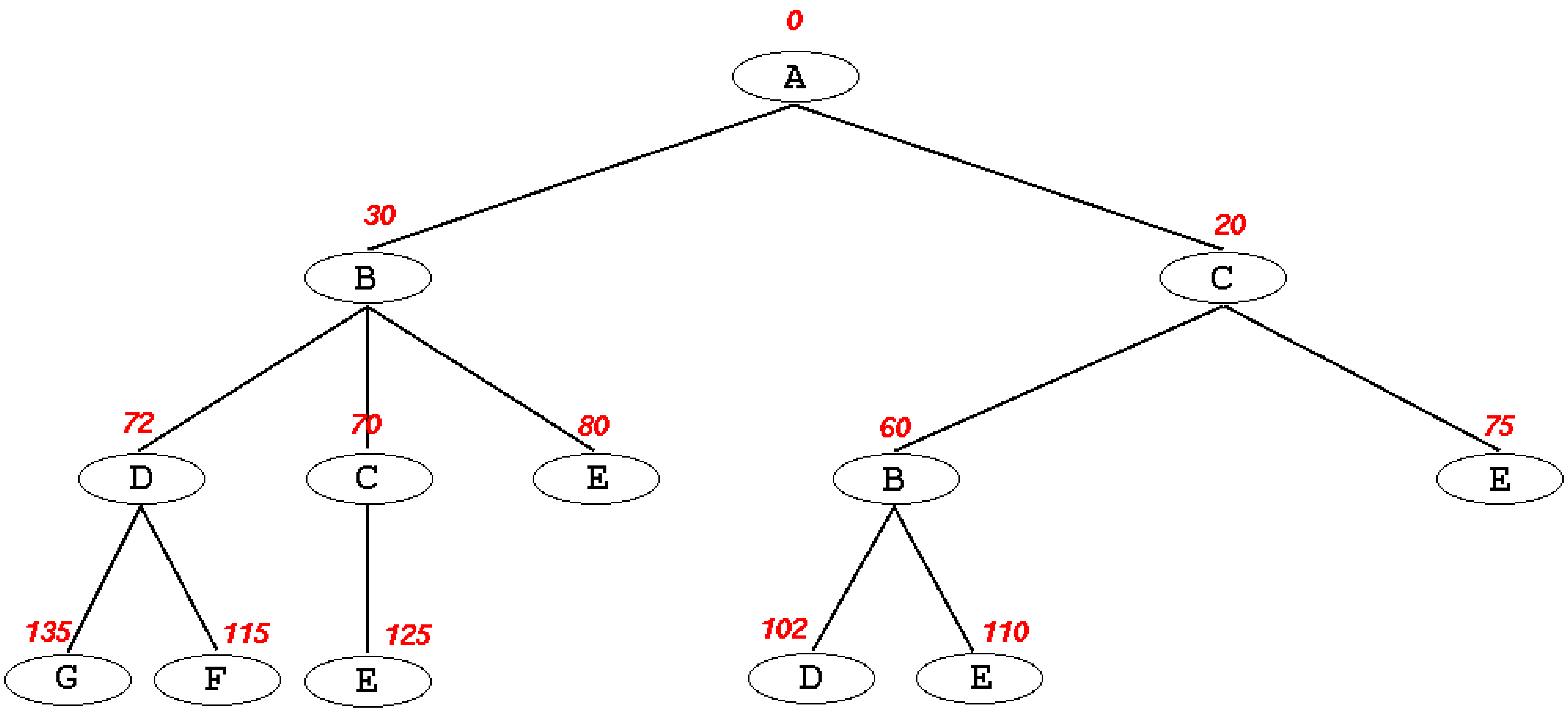
Example: Uniform-cost Search



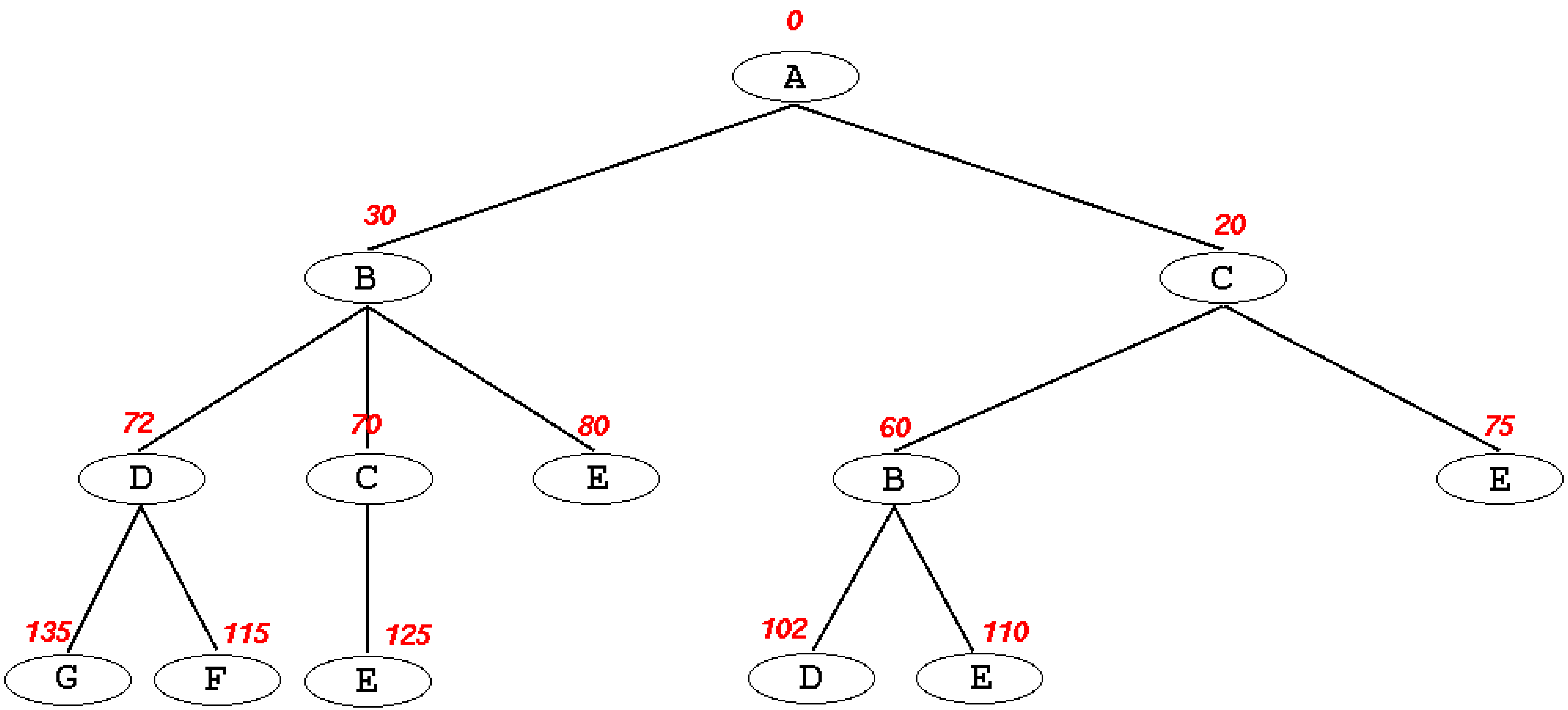
Example: Uniform-cost Search



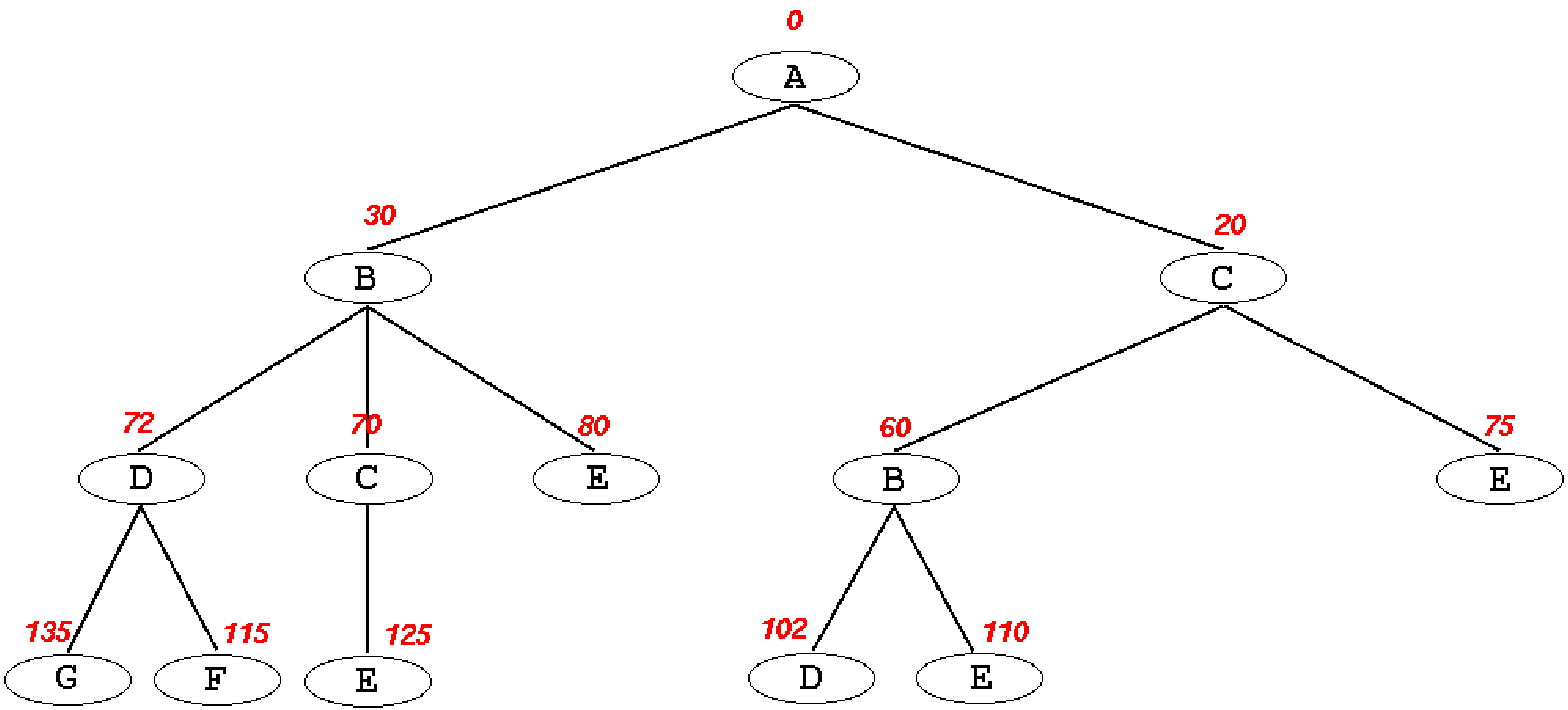
Example: Uniform-cost Search



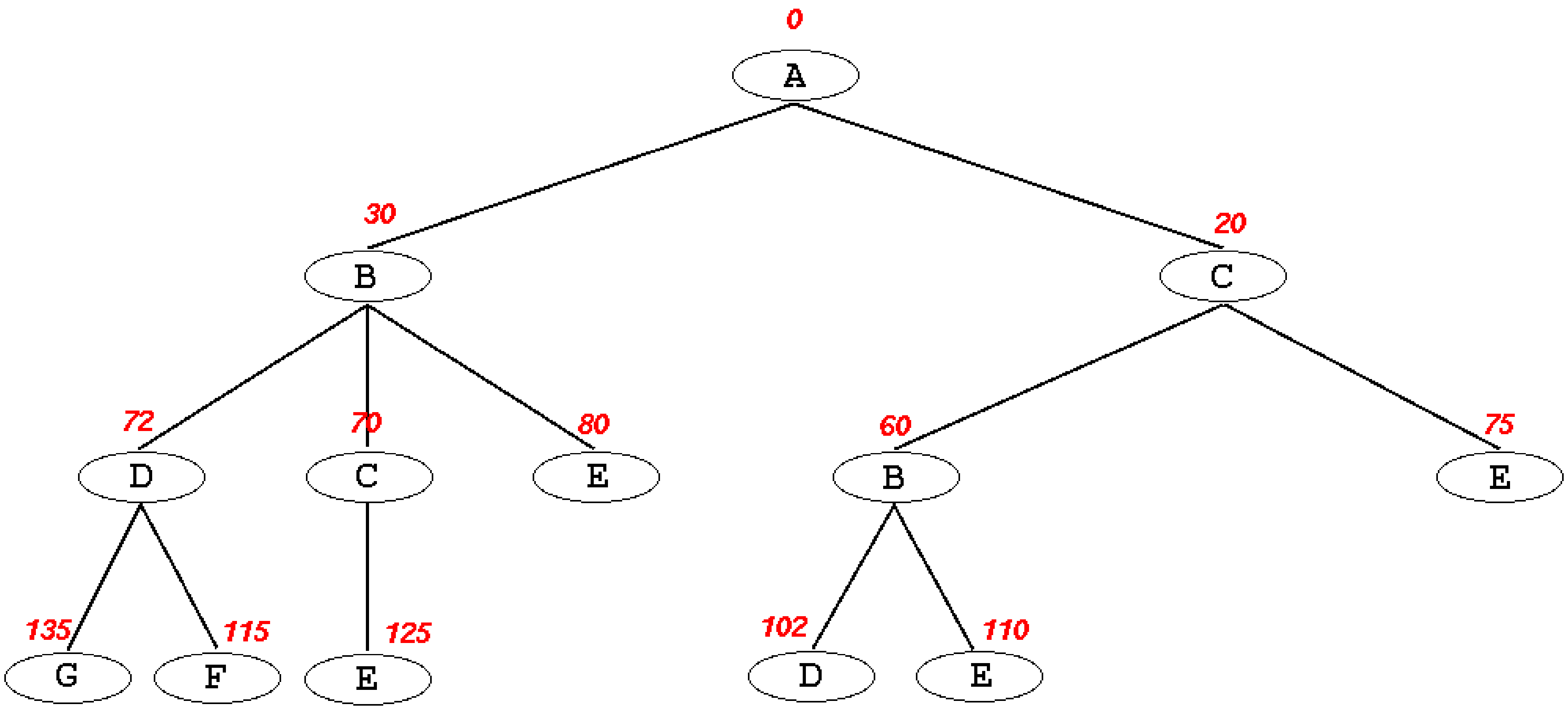
Example: Uniform-cost Search



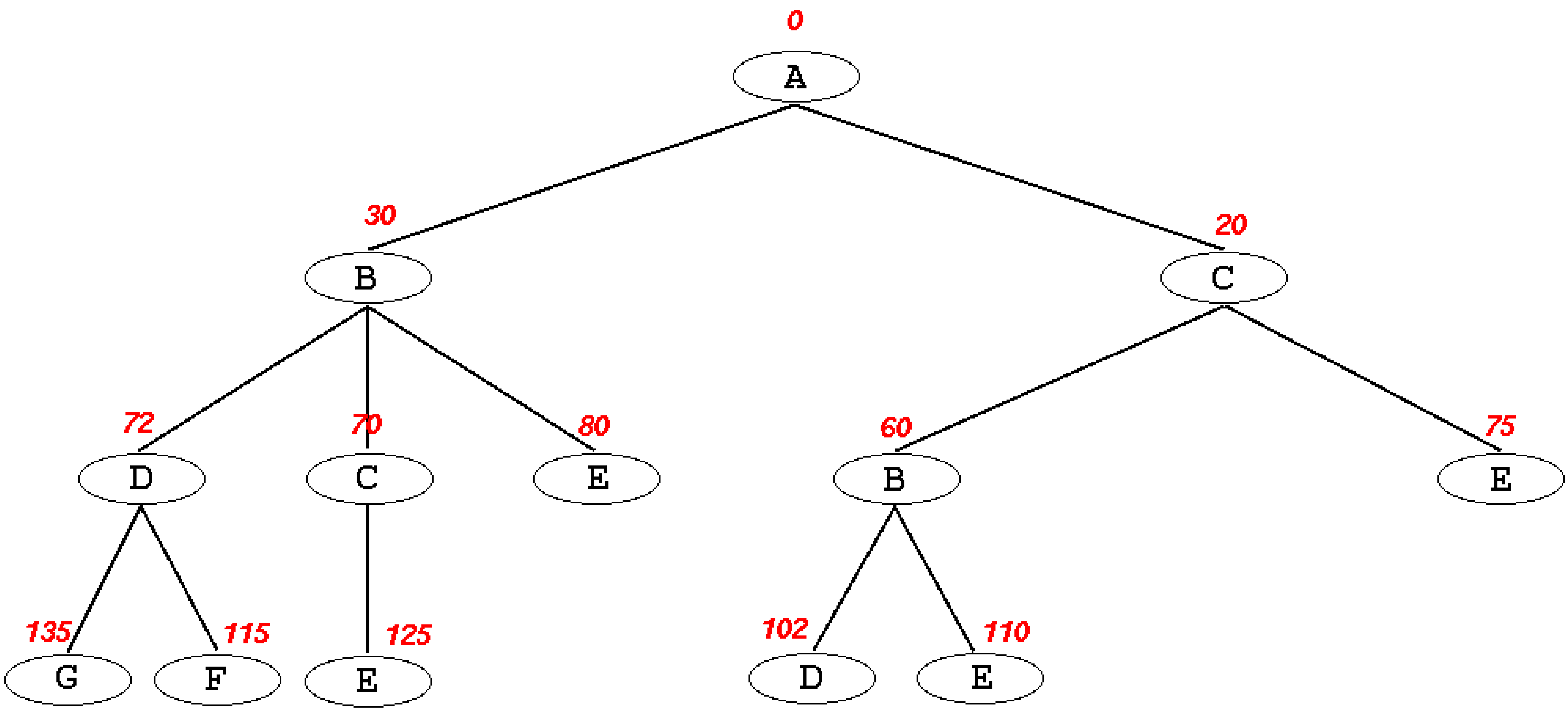
Example: Uniform-cost Search



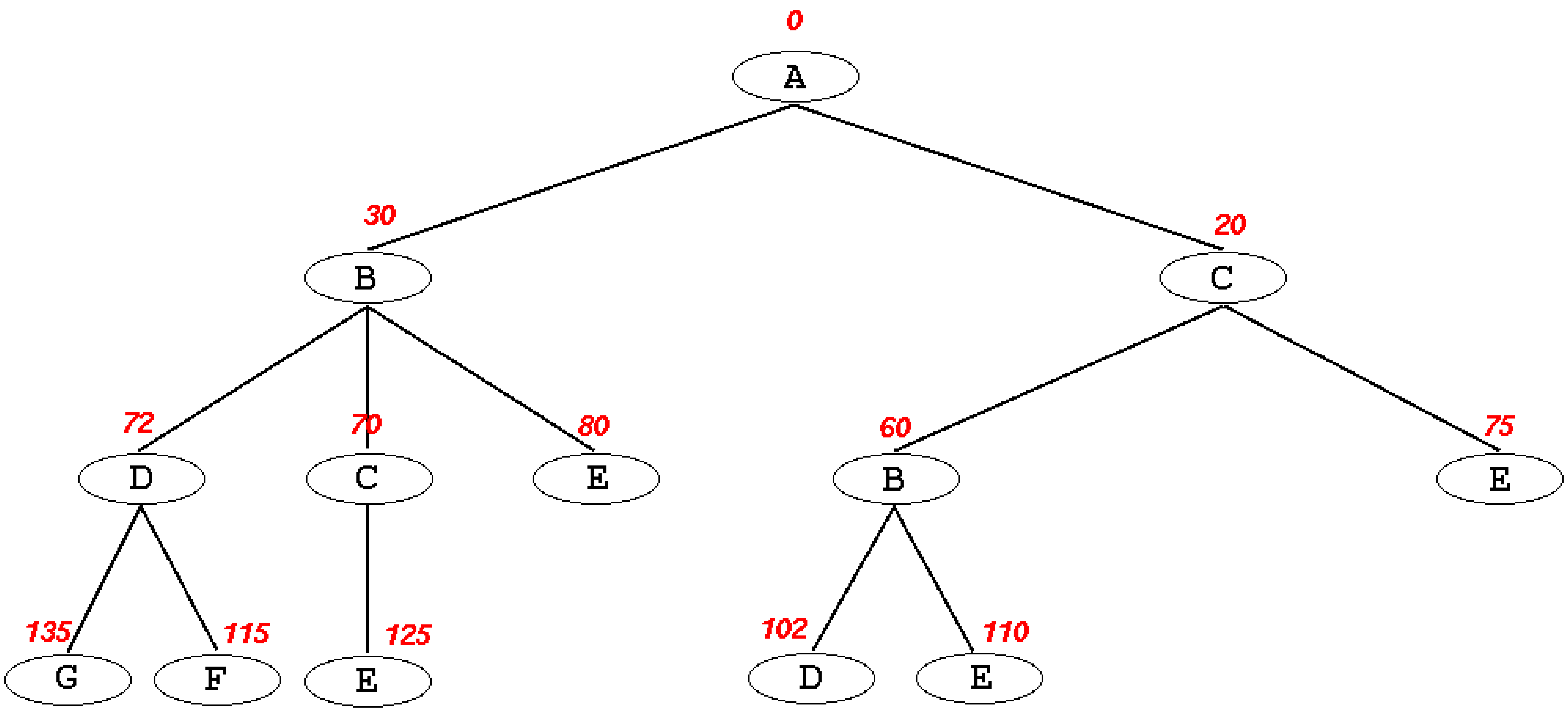
Example: Uniform-cost Search



Example: Uniform-cost Search



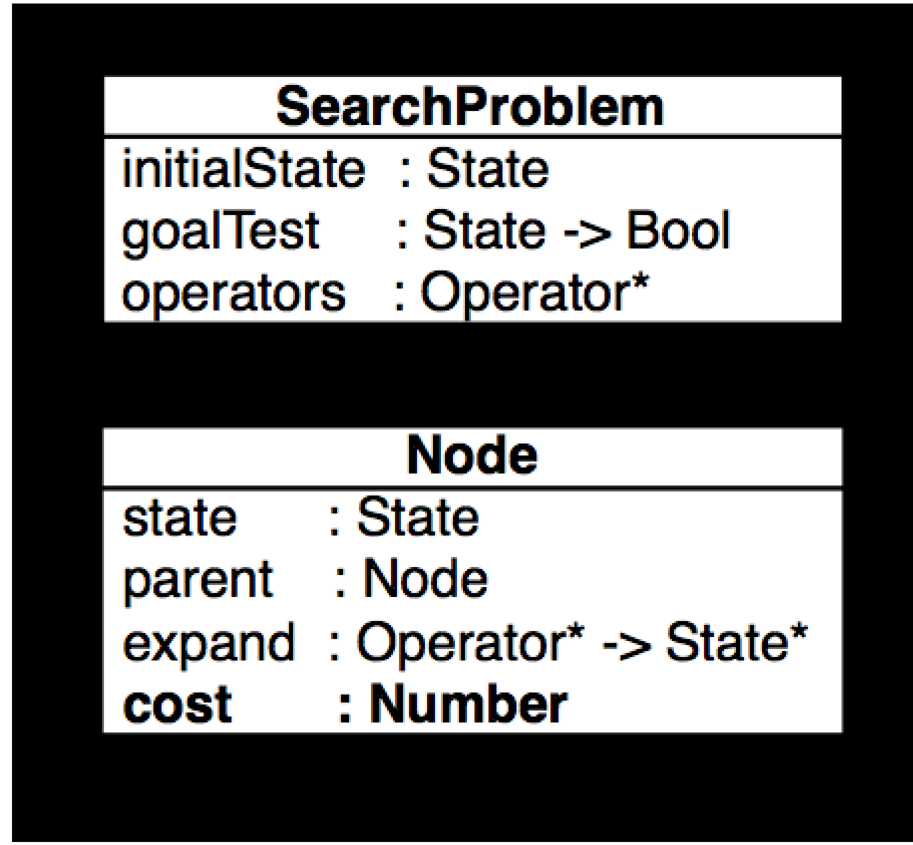
Example: Uniform-cost Search



State, Search Problem & Node

```
// a search problem
class SearchProblem{
  public State initialState();
  public booleangoalTest(State s);
  public List<Operator> operators();
}

// a search tree node
class Node{
  public State state();
  public Node parent();
  public List<Node> expand(List<Op> ops);
  public int cost();
}
```



Uniform-cost Search Algorithm

```
// pseudocode implementing uniform-cost search
public Node uniformCostSearch( SearchProblem problem) {
  LinkedList<Node> nodes
    = new LinkedList<Node>(new Node(problem.initialState()))

  while(true){
    if (nodes.size() == 0) then { return failure }
    Node node = nodes.removeFirst()
    if (problem.goalTest(node.state()) then { return node }
    nodes.addAll(node.expand(problem.operators())
    // Sort the nodes in order of increasing path cost g(n)
    Collections.sort(nodes, pathCostComparator)
  }
}
```

Properties of Uniform-cost Search

- Uniform-cost search is *complete*
- Guaranteed to find an *optimal solution* if every operator costs at least ε > 0, i.e, if the cost of a path never decreases
    - if operators can have negative cost an exhaustive search of all nodes is required to find an optimal solution
- Time and space complexity is $O(b^{\lceil C^*/\varepsilon \rceil})$ where *C** is the cost of the optimal solution

Exponential Complexity Is Bad

- The eight-puzzle has about 10^5 states and can easily be solved using uninformed search
- Typical solution is about 20 steps long and the average branching factor is about 3, which gives $3^{20} = 3.5 \times 10^9$ states, but we can reduce this to about 3.5×10^5 by eliminating duplicate states
- The fifteen-puzzle (only one tile larger in each direction) has about 10^{13} states without duplicates and cannot be solved using uninformed search on current computers (10,000 GB at one byte per state)
- To solve larger problems, some domain specific knowledge must be added to improve search efficiency

Focusing the Search

- Using the path cost *g(n)* allows us to find an optimal solution
- However it does not direct search toward the goal
- In order to focus the search, we need an *evaluation function* which incorporates some *estimate* of the cost of a path *from a state to the closest goal state*

Informed Search

- *Informed* (or heuristic) search procedures use some form of (often inexact) information to guide the search towards more promising partial solutions
- The *cost* of a partial solution, *n*, is defined as

$$f(n) = g(n) + h(n)$$

  where $g(n)$ is the path cost from the start state to *n* and $h(n)$ is an *estimate* of the cost of going from state *n* to a goal state
- $h(n)$ is often called the *heuristic function* - the more accurate the heuristic function, the more efficient the search

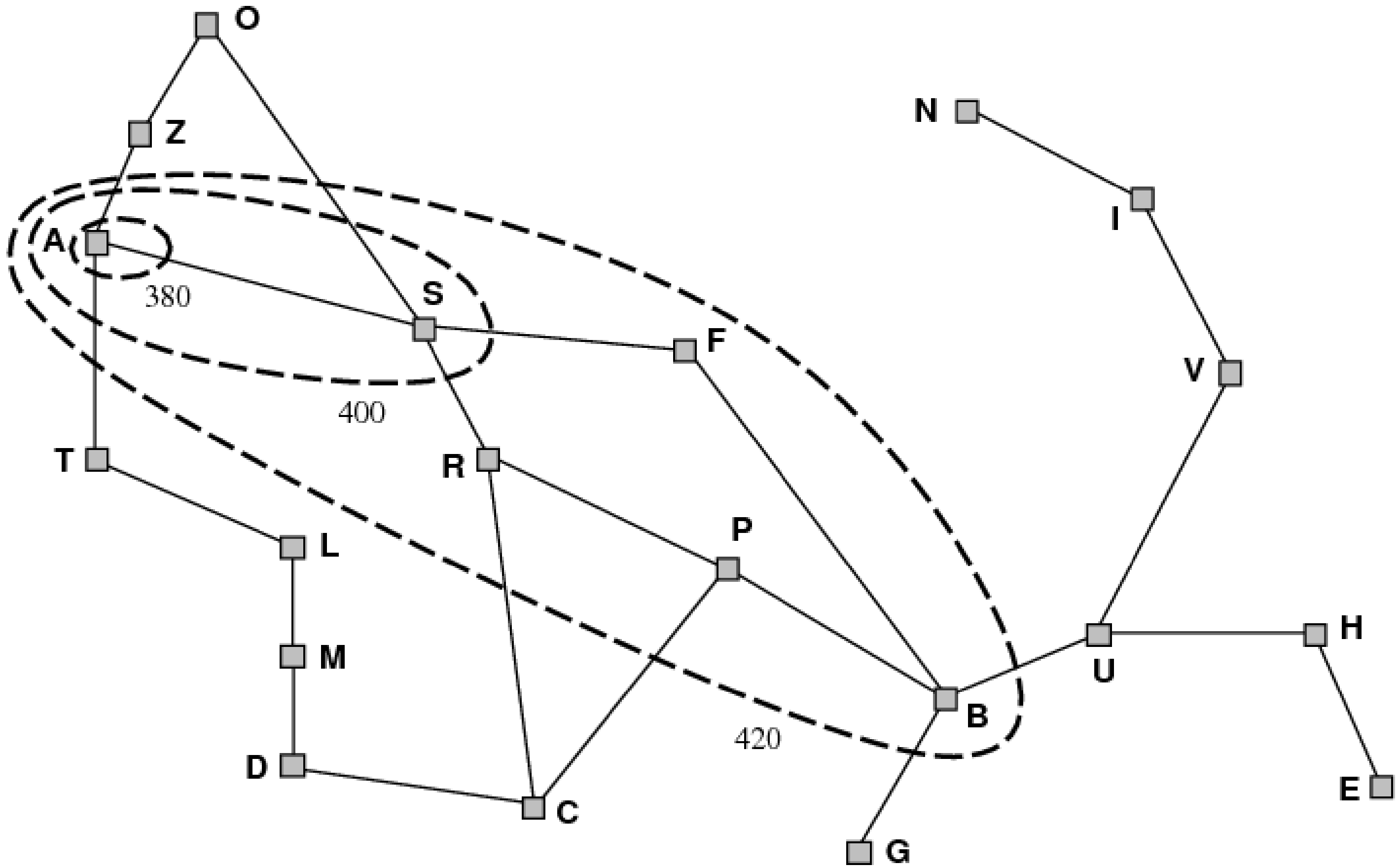
Informed Search Procedures

- Costs are used to order partial solutions so that the most promising (least cost) nodes are expanded first
  - **greedy search** expands the node with the lowest *h(n)* value, i.e., the node which is estimated to be closest to the goal
  - **A* search** expands the node with the lowest *f(n)* value, i.e., the path through *n* with the lowest estimated cost
- In contrast, uniform-cost search expands the node with the lowest *g(n)* value, i.e., the node with the lowest path cost

A* Search

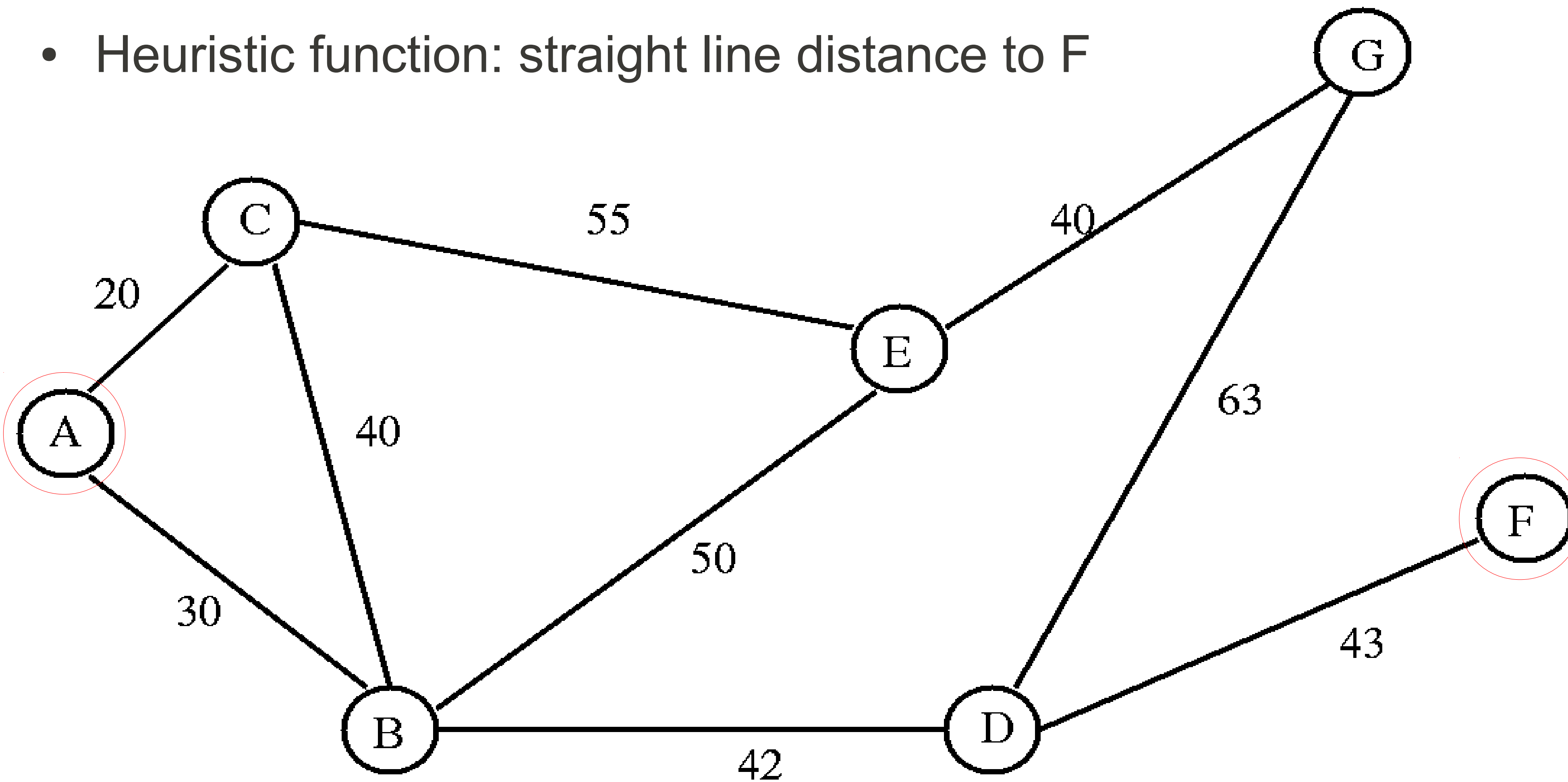
- A* search expands leaf nodes in order of cost (as measured by the cost function *f(n)*)
- Expands the root node, then the lowest cost child of the root node, then the lowest cost unexpanded node etc.
- Fans out from the root node, expanding nodes in bands of increasing *f*-cost
- With uniform-cost search (A* with *h(n)* = 0 for all *n*) the bands are circular around the start state
- With more accurate heuristics, the bands are distorted towards the goal state around the optimal path

Example: A* Search



Example: Simple Route Planning

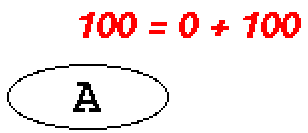
- Initial state: A
- Goal state: F
- Heuristic function: straight line distance to F



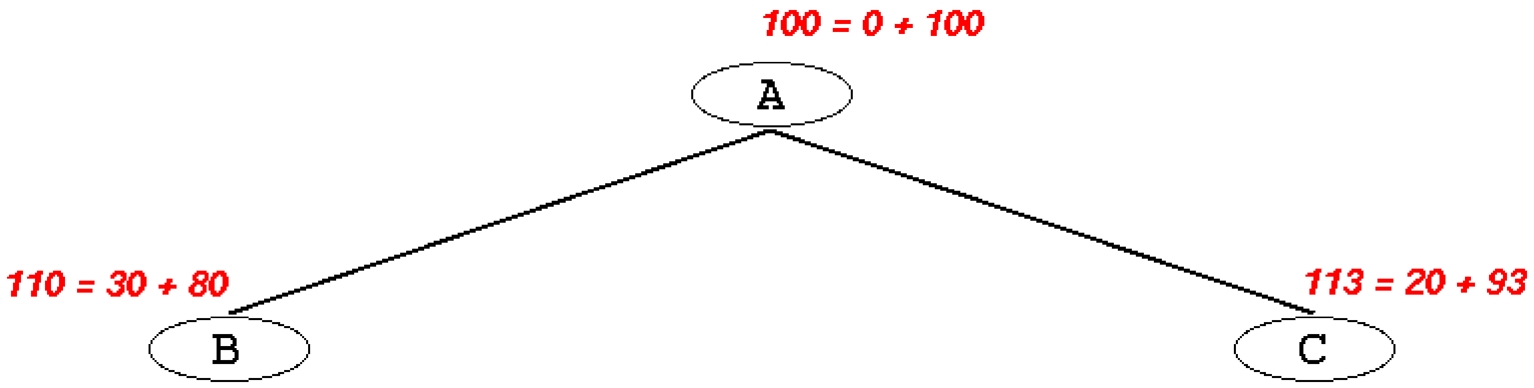
Straight Line Distances from F

City	Distance from F
A	100
B	80
C	93
D	43
E	47
G	37

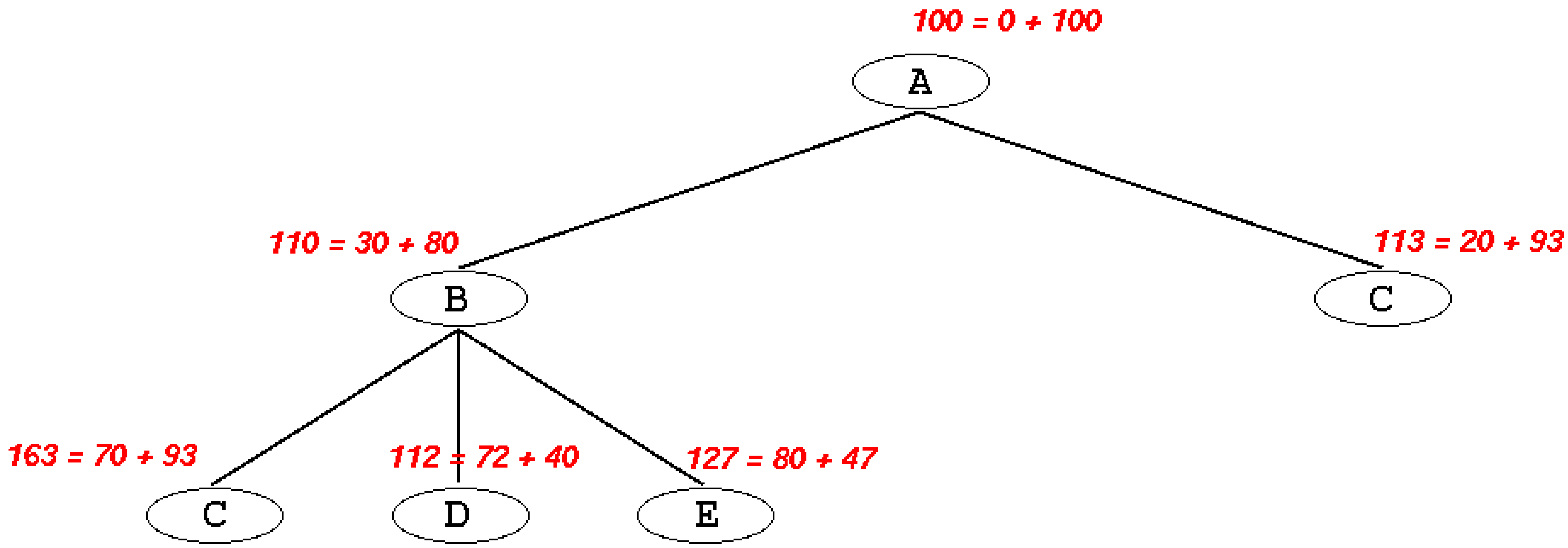
Example: A* Search



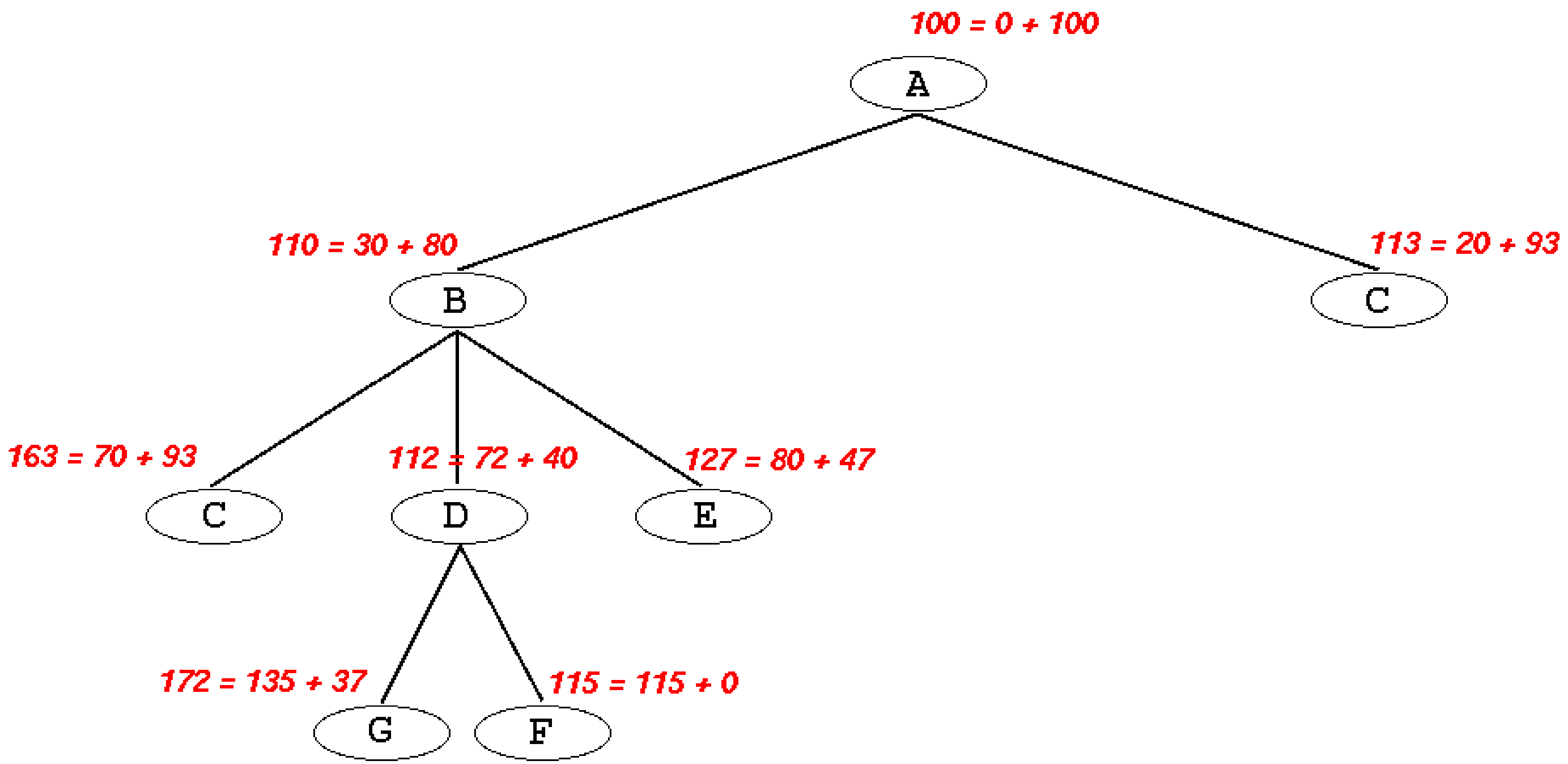
Example: A* Search



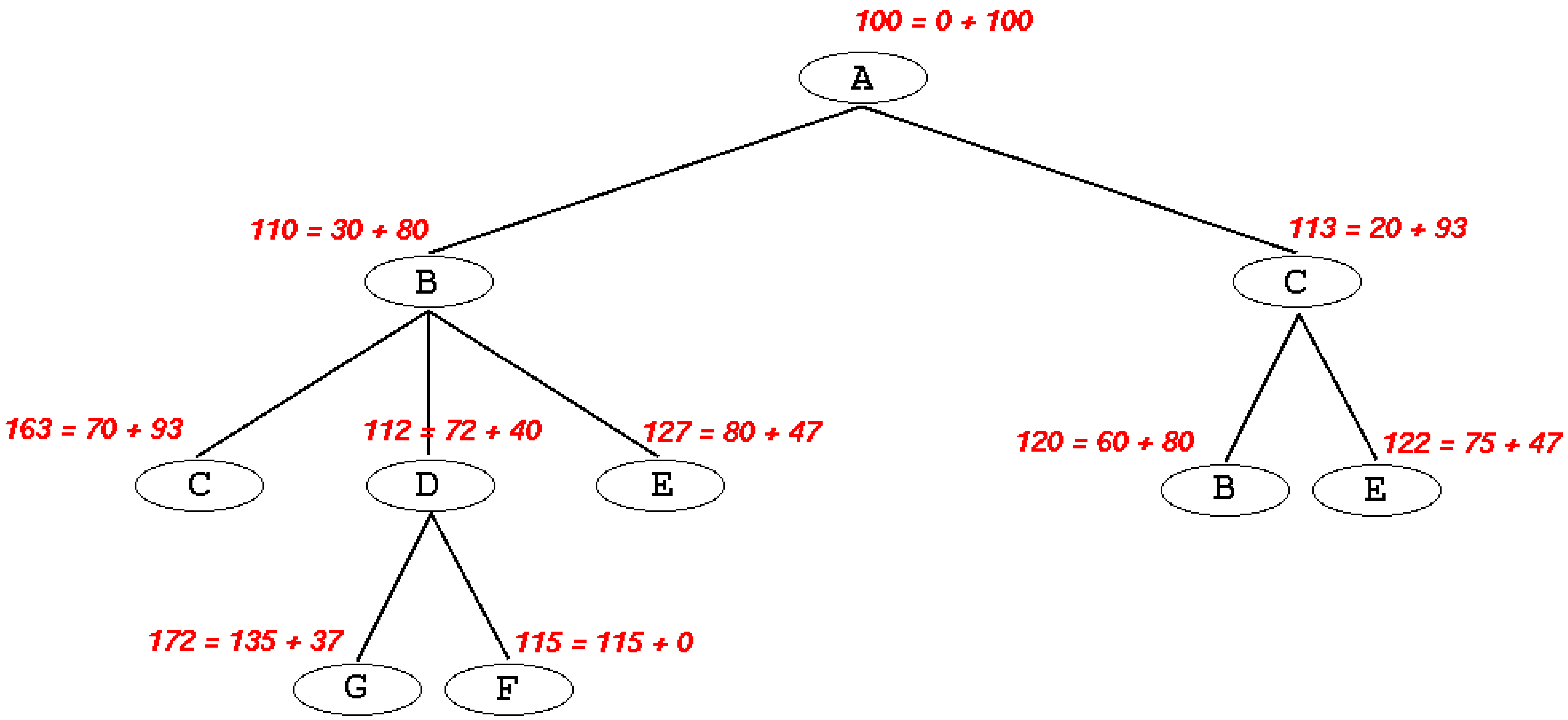
Example: A* Search



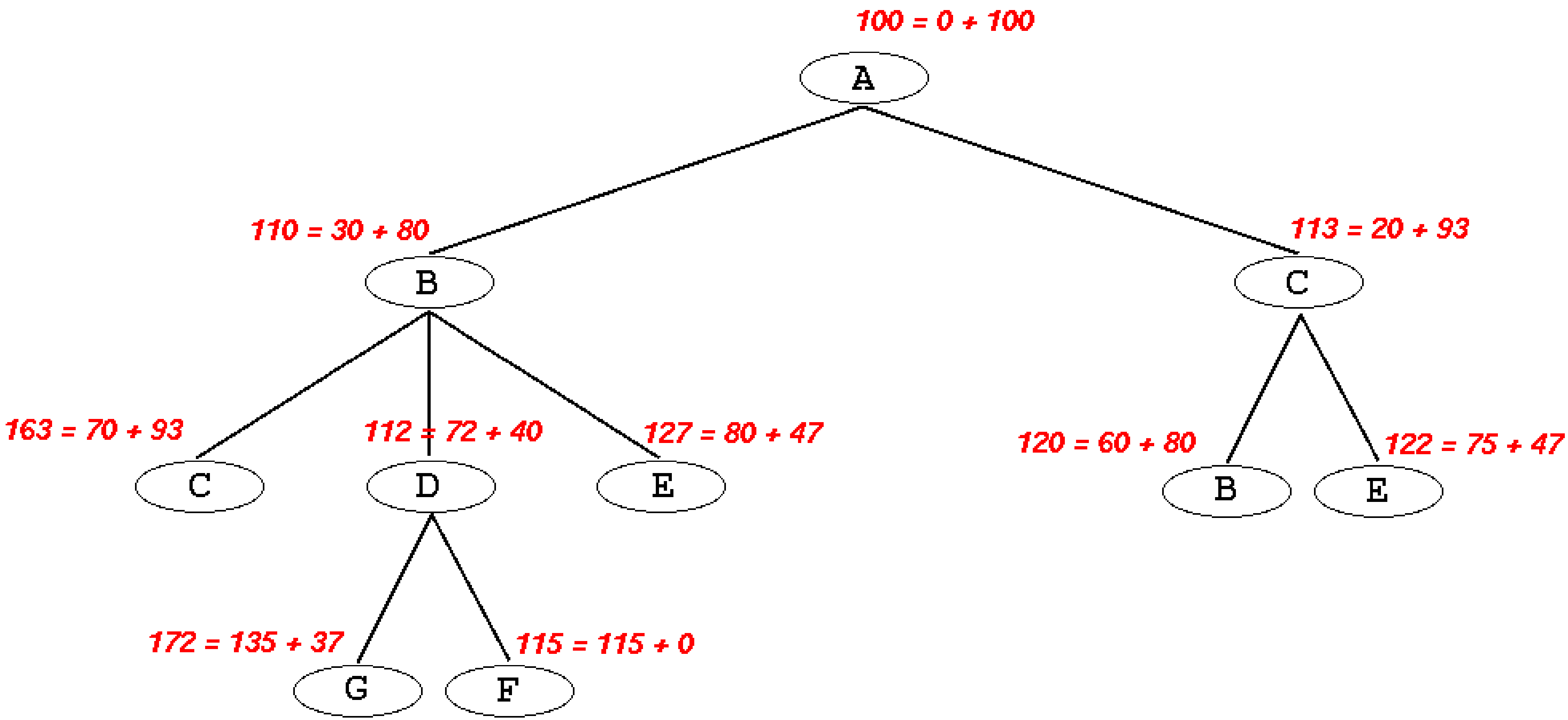
Example: A* Search



Example: A* Search



Example: A* Search



A* Search Algorithm

```
// pseudocode implementing A* search
public Node A*Search(SearchProblemproblem) {
  LinkedList<Node> nodes
      = new LinkedList<Node>(new Node(problem.initialState()))
  while(true) {
    if (nodes.size() == 0) then { return failure }
    Node node = nodes.removeFirst()
    if (problem.goalTest(node.state()) then { return node }
    nodes.addAll(node.expand(problem.operators())
    // Sort the nodes in order of increasing estimated cost f(n)
    Collections.sort(nodes, estimatedCostComparator)
  }
}
```

Properties of A*

- A* is *complete* on locally finite graphs (graphs with a finite branching factor) provided there is some positive constant δ such that each operator costs at least δ
- It is *optimal* if the heuristic function *h* is *admissible*, i.e,. it never *overestimates* the cost of reaching a goal state from the current state
- If *h* is admissible, $f(n)$ never overestimates the actual cost of the best solution through *n*
- Time and space complexity is $O(b^d)$ where *d* is the depth of the solution unless $|h(n) - h^*(n)| \leq O(\log h^*(n))$
- A* is *optimally efficient* for any given heuristic function - no other optimal algorithm is guaranteed to expand fewer nodes than A*

Comparison of Search Procedures

Search Procedure	Complete	Optimal	Optimally Effifient
depth-first	no	no	-
breadth-first	yes	yes*	no
uniform-cost	yes	yes	no
greedy	no	no	-
A*	yes	yes	yes

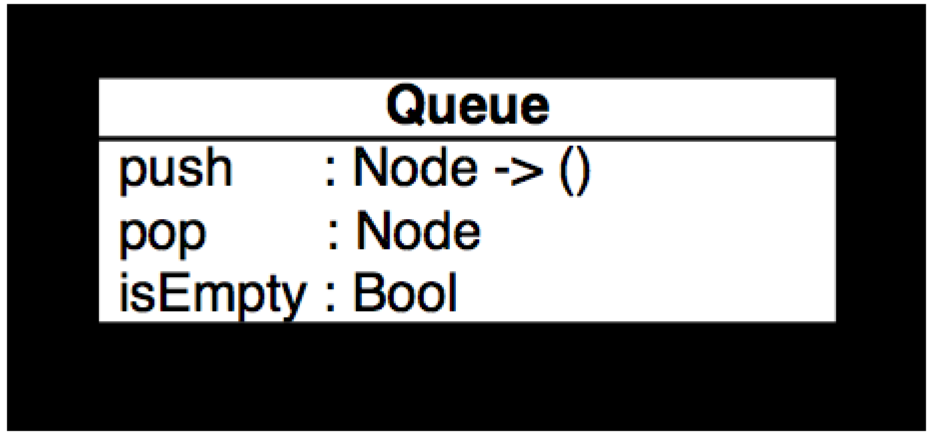
Total Search Cost

- The *search cost* is a function of the *time* and *memory* required to find a solution
- The *total cost* of the search is the sum of the path cost and the search cost
- For large complex problems, there is usually a tradeoff to be made
  - finding a better or an optimal solution (least path cost) usually has a higher search cost
- The relative importance of these two costs determines how much computation we are prepared to do for a given improvement in solution quality

General Search

- All of the search procedures presented (informed and uninformed) follow the same basic pattern
- The difference is in the order in which new states are expanded (e.g., breadth- or depth-first, or in cost order)
- We can write a *general* search method that can be specialised to different search procedures
- To do this, we use a *queue* data-structure which determines the order in which nodes are expanded

Node Queue




```
// An abstract queue data structure
   public interface Queue{
   public void push(List<Node> n);
   public Node pop();
   public boolean isEmpty();
}
```

General Search

```
// pseudocode implementing general search
public Node GeneralSearch(SearchProblem problem, Queue nodes)
{
   nodes.push(new Node(problem.initialState());

   while(true) {
      if (nodes.isEmpty()) then { return failure }
      Node node = nodes.pop()
      if (problem.goalTest(node.state()) then { return node }
      nodes.push(node.expand(problem.operators());
   }
}
```

Search Strategies

```
// breadth-first search
return generalSearch(problem, new FIFOQueue());
// depth-first search
return generalSearch(problem, new LIFOQueue());
// uniform-cost search
return generalSearch(problem, new PrioQueue(g));
// greedy search
return generalSearch(problem, new PrioQueue(h));
// A* search
return generalSearch(problem, new PrioQueue(f));
// In Java: Queue, Stack, PriorityQueue
```

Eliminating Loops

- The psuedocode presented in these slides leaves out several details, in particular elimination of loops, discussed in Lecture 2
- Elimination of loops can be accomplished by at least two approaches:
  - avoid visiting a state we have already visited in this *path* (node);
  - avoid visiting a state we have *ever* visited before (in any path).
- The second option is only ok for algorithms which guarantee that the first encounter of a state will be the shortest path to that state (e.g., uniform cost search), or if we don't care about the path
- In practice, avoiding loops in a node path is often good enough

Keeping Track of All Previous States

```
public Node GeneralSearch(SearchProblem problem, Queue nodes)
{
   // Store all previously visited states: may be large!
   Set<State> visited = new HashSet<State>();
   nodes.push(new Node(problem.initialState());
   while(true) {
      if (nodes.isEmpty()) then { return failure }
      Node node = nodes.pop()
      if (problem.goalTest(node.state()) then { return node }
      for each (newNode in node.expand(problem.operators()) {
         if (visited.contains(newNode.state()) { // skip }
         else {visited.add(newNode.state()); nodes.push(newNode)}
      }
   }
}
```

Eliminating Loops in a Node

```
public Node GeneralSearch(SearchProblem problem, Queue nodes)
{
  nodes.push(new Node(problem.initialState());
    while(true) {
    if (nodes.isEmpty()) then { return failure }
    Node node = nodes.pop()
    if (problem.goalTest(node.state()) then { return node }
    // Push nodes for states not already in this node path
    for (Node newNode : node.expand(problem.operators())
      if (newNode.state() is not in node.path()) {
        nodes.push(newNode)
      }
    }
  }
}
```