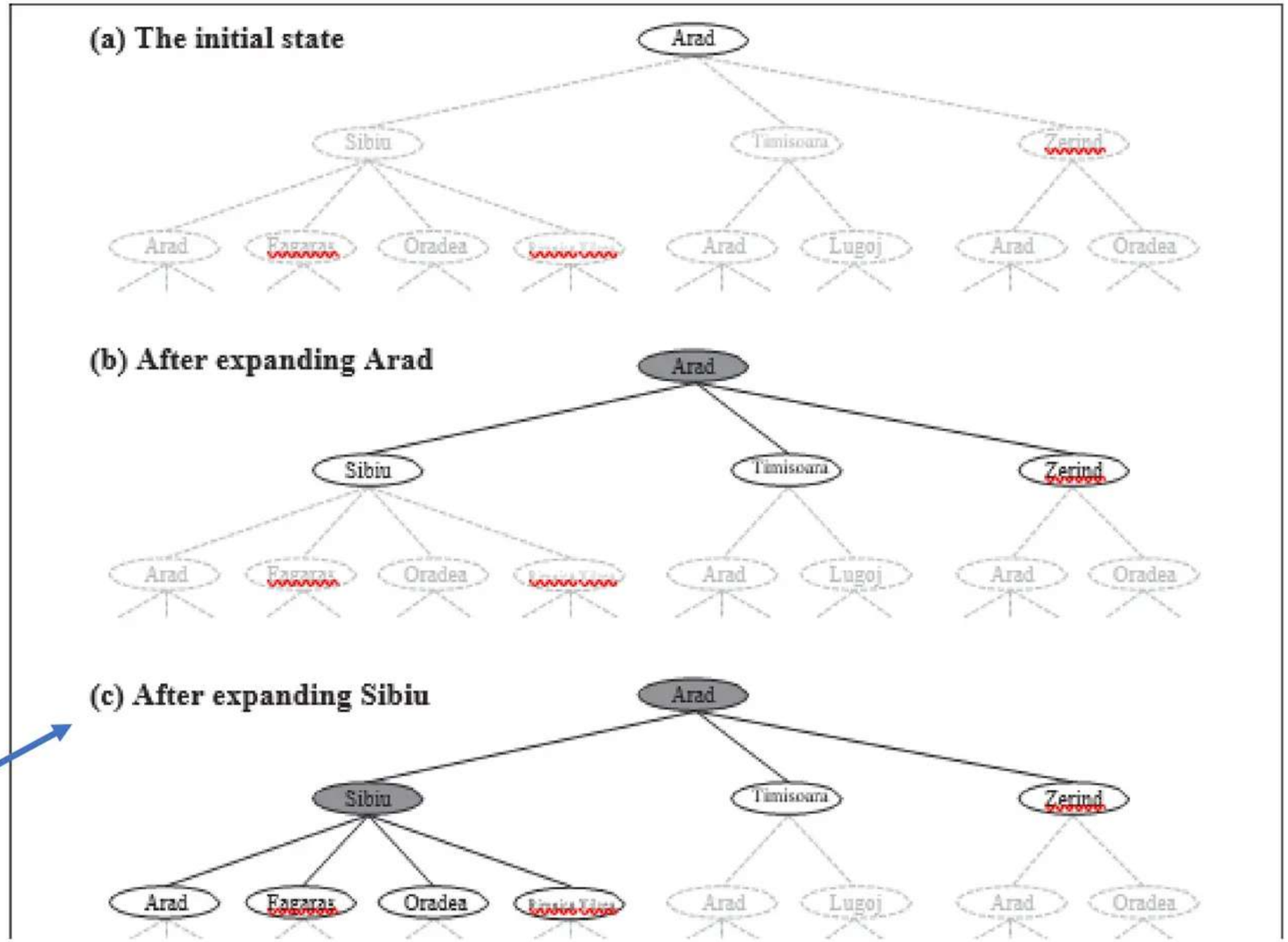
Solving Problems by searching

Searching for Solutions

search algorithms work by considering various possible action sequences

The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem



Partial search trees for finding a route from Arad to Bucharest

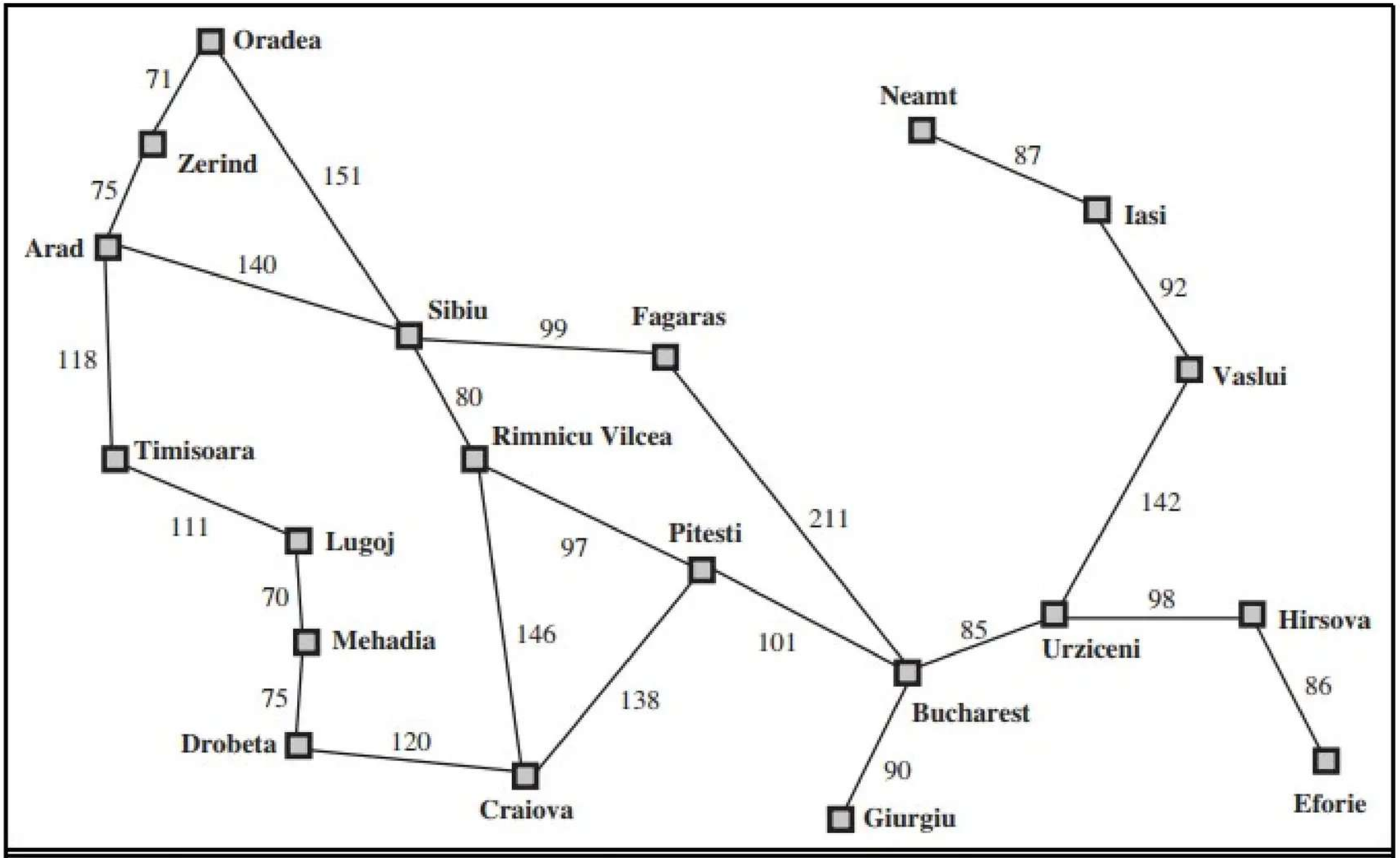


Figure 3.2 A simplified road map of part of Romania.

The Concepts and Terminology of Searching

- applying each legal action to a state – Expanding the State
- The set of all leaf nodes available for expansion - Frontier / Open List
- Search Strategy - The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand
- **redundant paths**, multiple Paths that exist from one state to another

Ex: Arad–Sibiu (140 km long) & Arad–Zerind–Oradea–Sibiu (297 km long)

- Loopy Paths are a special case of these Redundant Paths (Like Arad – Sibiu – Arad)

following redundant paths can cause a tractable problem to become intractable

General tree-search and graph-search algorithms

```
function TREE-SEARCH( problem ) returns a solution, or failure
   initialize the frontier using the initial state of problem
   loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

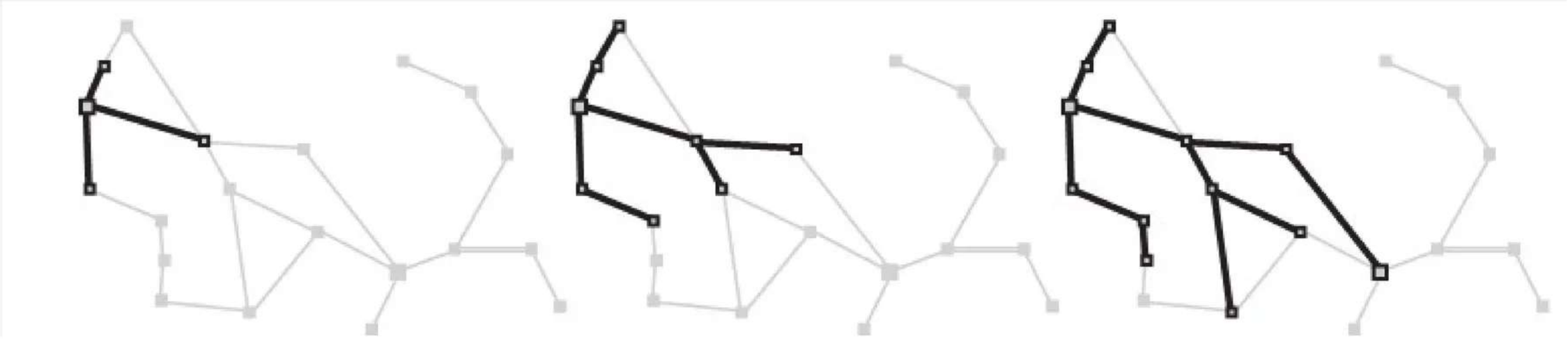
```
function GRAPH-SEARCH( problem ) returns a solution, or failure
   initialize the frontier using the initial state of problem
   initialize the explored set to be empty
   loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
         only if not in the frontier or explored set
```

Also known as **Open List**

Also known as **Closed List**

GRAPH-SEARCH algorithm

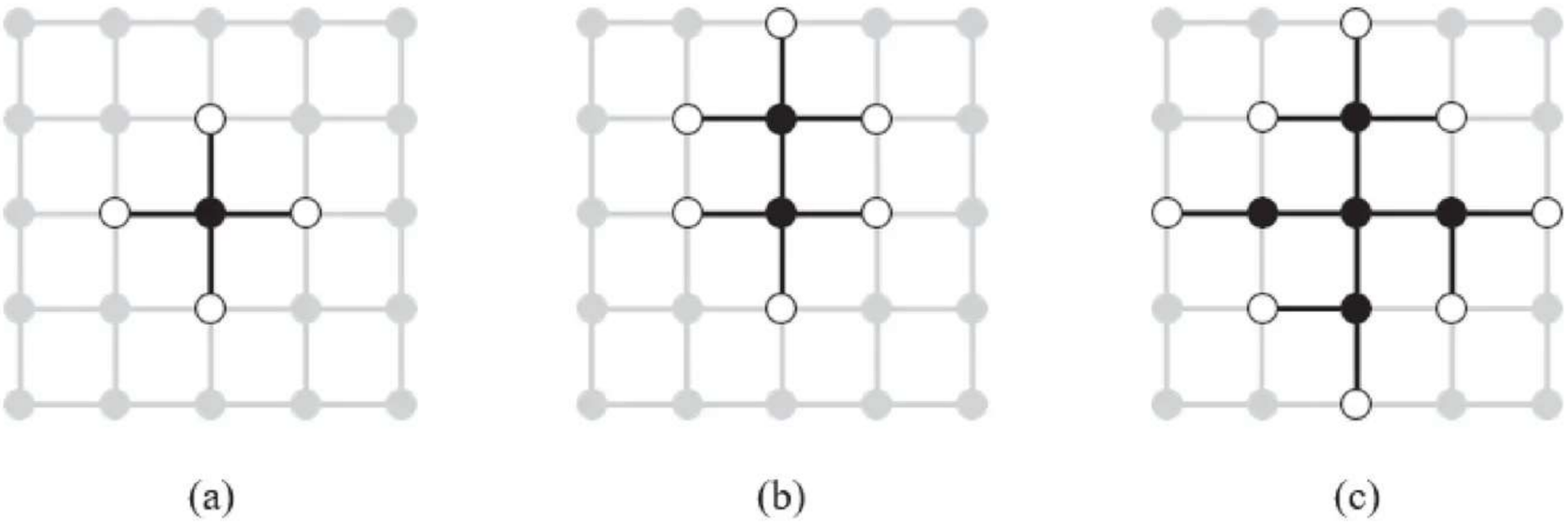
- we can think Graph-Search as growing a tree directly on the state-space graph



- i.e. the GRAPH-SEARCH algorithm contains at most one copy of each state

GRAPH-SEARCH algorithm

- the frontier **separates** the state-space graph into the explored region and the unexplored region



- The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes)

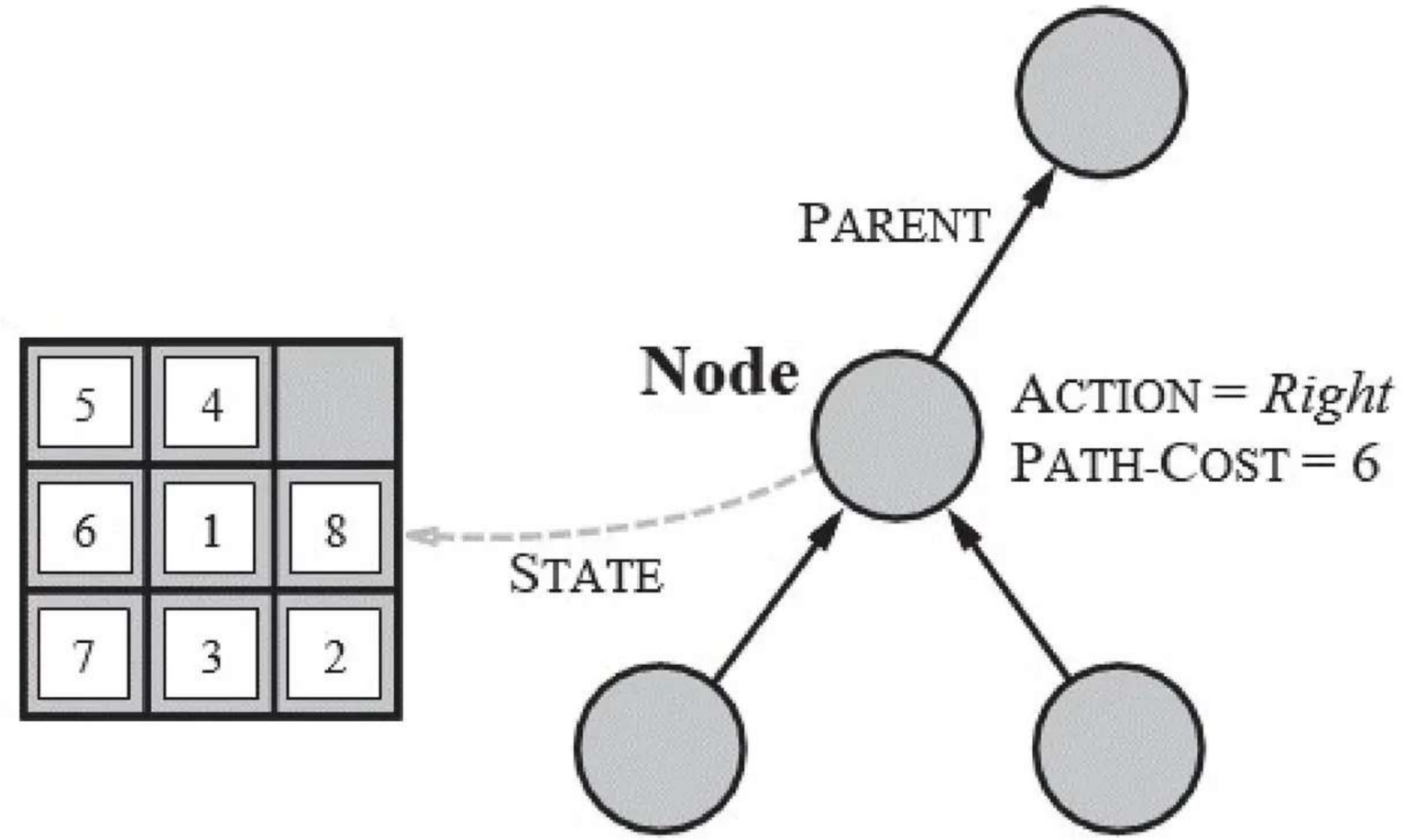
Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node ***n*** of the tree, we have a structure that contains four components:

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

Node Data Structure

- Nodes are the data structures from which the search tree is constructed



Child-Node Computation

- The function CHILD-NODE takes a parent node and an action and returns the resulting child node

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root

Data Structure for Frontier

- The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.
- The appropriate data structure for this is a **queue** (FIFO/LIFO /Priority)
- EMPTY?(*queue*) returns true only if there are no more elements in the queue.
- POP(*queue*) removes the first element of the queue and returns it.
- INSERT(*element* , *queue*) inserts an element and returns the resulting queue.
- The explored set can be implemented with a hash table to allow efficient checking for repeated states

Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Measuring problem-solving performance

Time and Space Complexity - size of the state space graph, $|V| + |E|$

What if the graph is represented *implicitly* by the initial state, actions, and transition model & Usually an Infinite Graph

- Then complexity is expressed in terms of three quantities:

➢ b, the **branching factor** or maximum number of successors of any node;

➢ d, the **depth** of the shallowest goal node and

➢ m, the **maximum length** of any path in the state space.

Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory

UNINFORMED SEARCH STRATEGIES

BFS and DFS

Breadth-first search

- These strategies have no additional information about states beyond that provided in the problem definition
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- All search strategies are distinguished by the ***order*** in which nodes are **expanded**
- Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic search** strategies

Breadth-first search

- **Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on
- In other words all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded, i.e. the *shallowest* unexpanded node is chosen for expansion
- FIFO queue is used for the frontier

goal test is applied to each node when it is ***generated*** rather than when it is selected for ~~**expansion**~~

Breadth-first search

```
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)
```

Breadth-first search – Time Complexity

- Imagine searching a uniform tree where every state has *b* successors.
- The root of the search tree generates *b* nodes at the first level,
- each of which generates *b* more nodes, for a total of b^2 at the second level.
- Each of *these* generates *b* more nodes, yielding b^3 nodes at the third level, and so on.
- Now suppose that the solution is at depth ***d***. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

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

INFORMED SEARCH STRATEGIES- Heuristics

Hill climbing, A* ,AO*
Algorithms

INFORMED (HEURISTIC) SEARCH STRATEGIES

- These strategies use problem-specific knowledge beyond the definition of the problem itself – Efficient way of finding Solutions
- The general approach is called **best-first search** – which is a Tree/Graph Search mechanism in which a node is selected for expansion based on an evaluation function, f(n)
- Most best-first algorithms include as a component of **f** a heuristic function, denoted h(n):

h(n) = estimated cost of the cheapest path from the state at node **n** to a goal state.

Ex: In Travelling from one city to other problem, straight-line distance from current state to the goal state could be used as Heuristic Estimate

if n is a goal node, then h(n) = 0

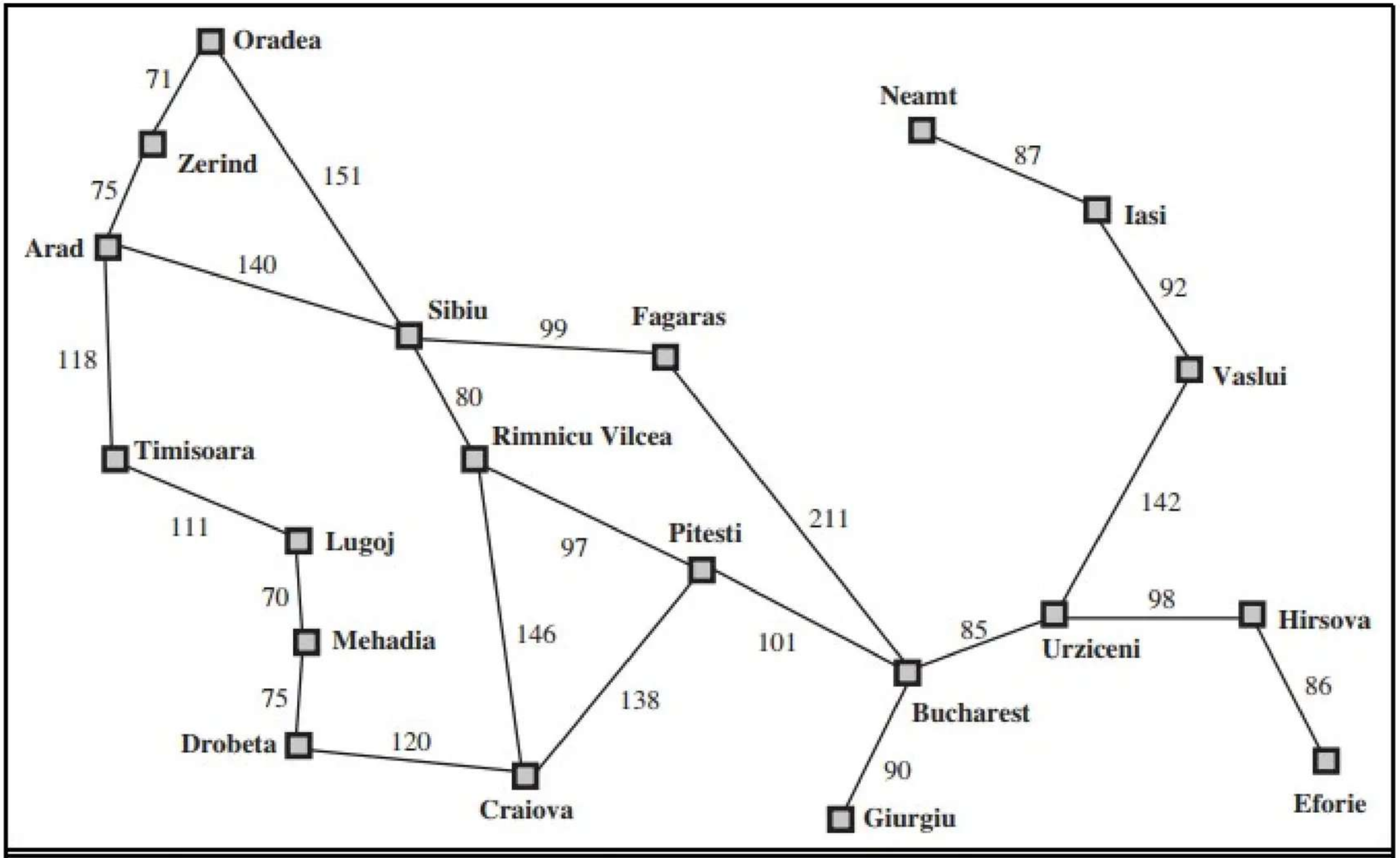
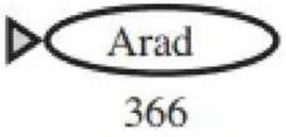


Figure 3.2 A simplified road map of part of Romania.

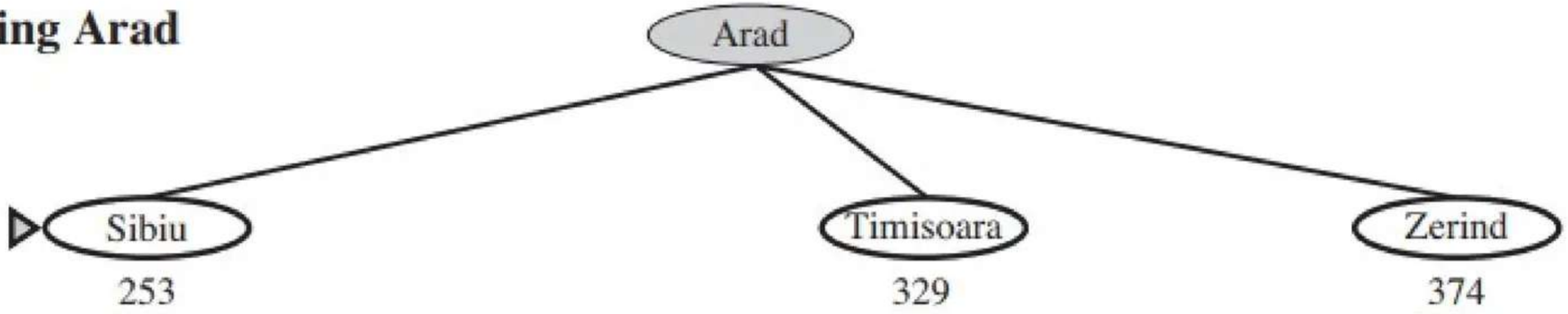
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

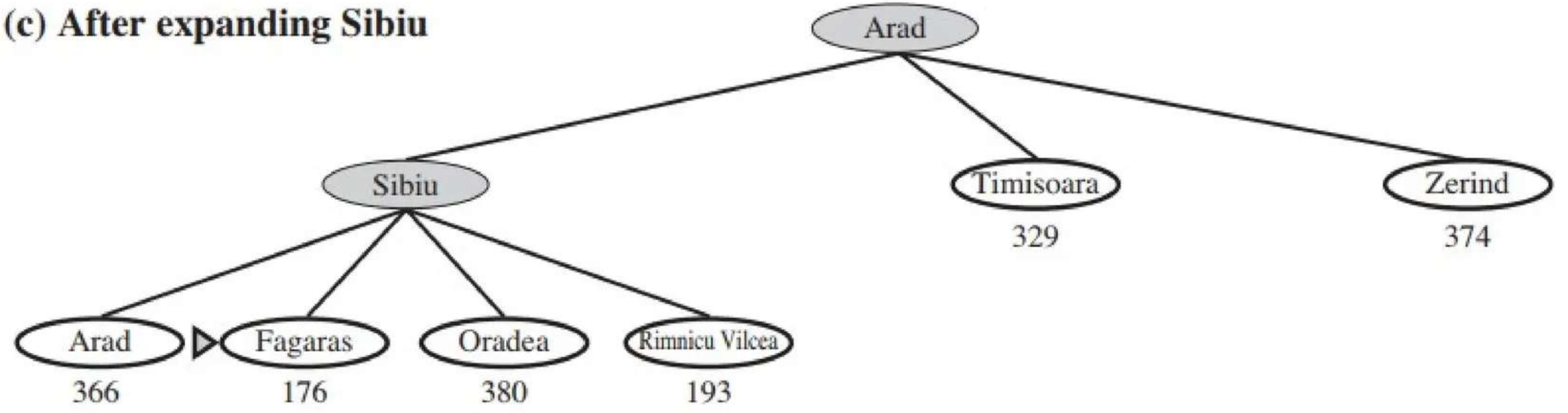
(a) The initial state



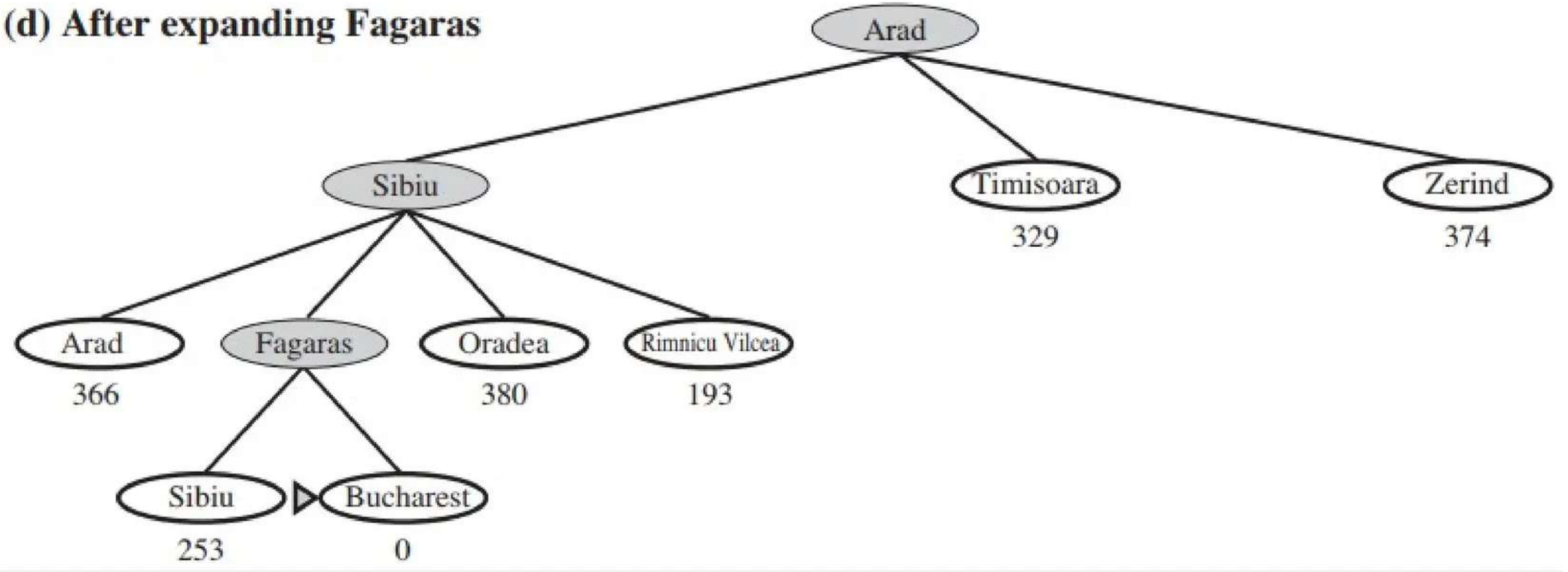
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



Greedy best-first search

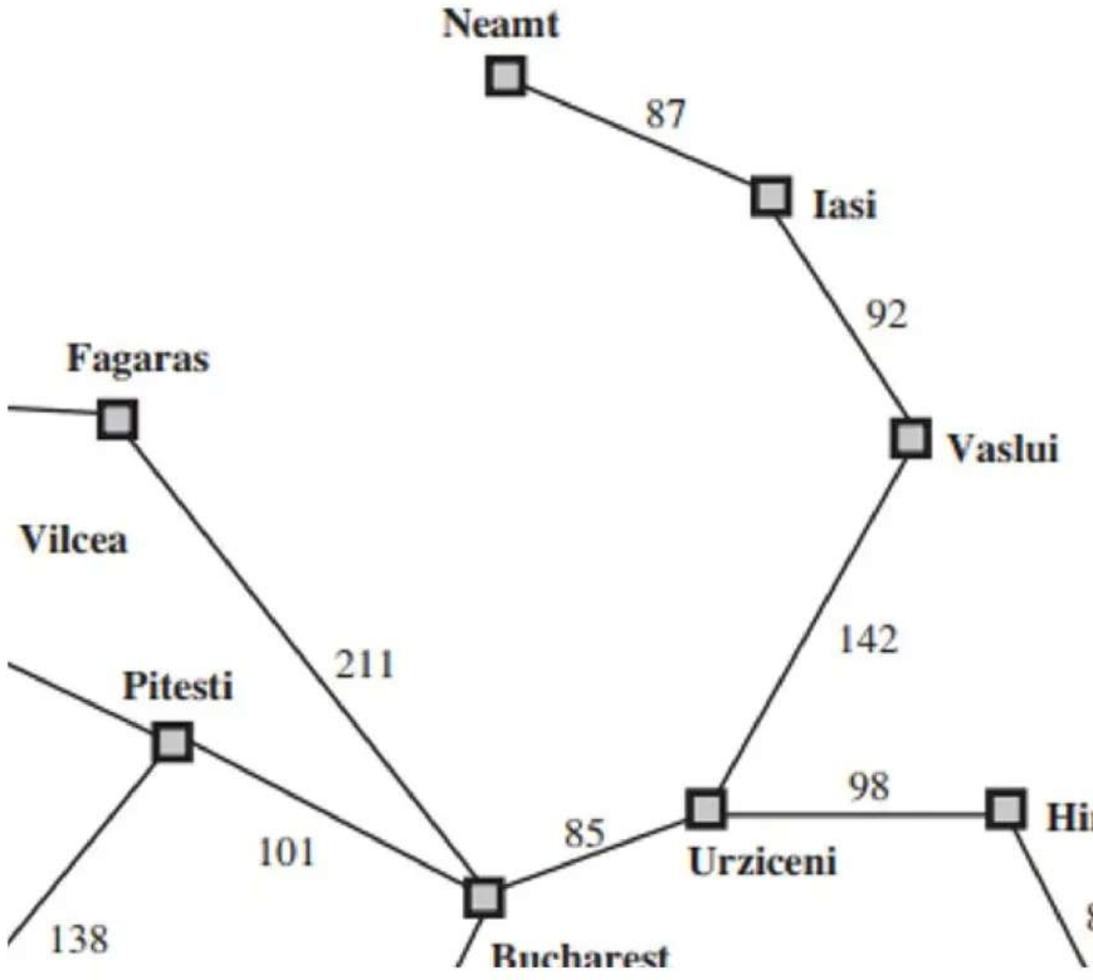
- Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.
- Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$
- we use the **straight line distance** heuristic, which we will call h_{SLD}
- greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not ~~optimal~~
- 32 kilometers longer (Via Fagaras = 99 + 211 = 310) than the path through Rimnicu - Vilcea – Pitesti (80 + 97 + 101 =278)
- "greedy"—at each step it ***tries to get as close to the goal as it can***

Greedy best-first search

- Greedy best-first tree search is incomplete even in a finite state space, much like depth-first search.
- Consider the problem of getting from

Iasi to **Fagaras**

The algorithm will never find this solution



A* search: Minimizing the total estimated solution cost

- The most widely known form of best-first search is called A∗ search
- It evaluates nodes by combining

  g(n), the cost to reach the node, and

  h(n), the ESTIMATED cost to get from the node to the goal:

f(n) = g(n) + h(n) , estimated cost of the cheapest solution through n

- A∗ search is both complete and optimal

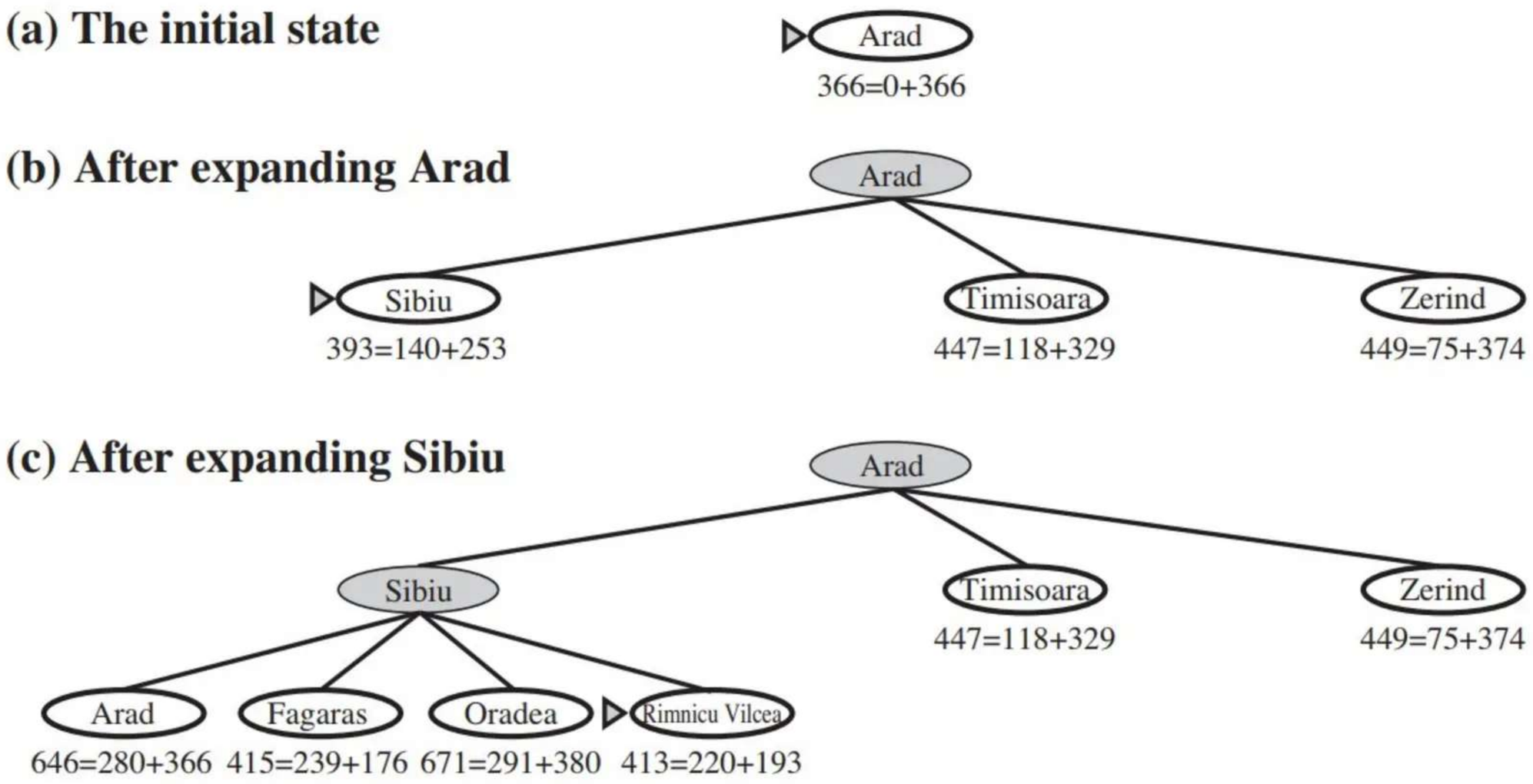
Conditions for optimality: Admissibility and consistency (Monotonicity)

- An admissible heuristic is one that never overestimates the cost to reach the goal, thus making f(n) such that it never overestimates the true cost of a solution along the current path through n
- Admissible heuristics are by nature optimistic because they think ***the cost of solving the problem is less than it actually is*** - Ex: h_{SLD}
- A heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

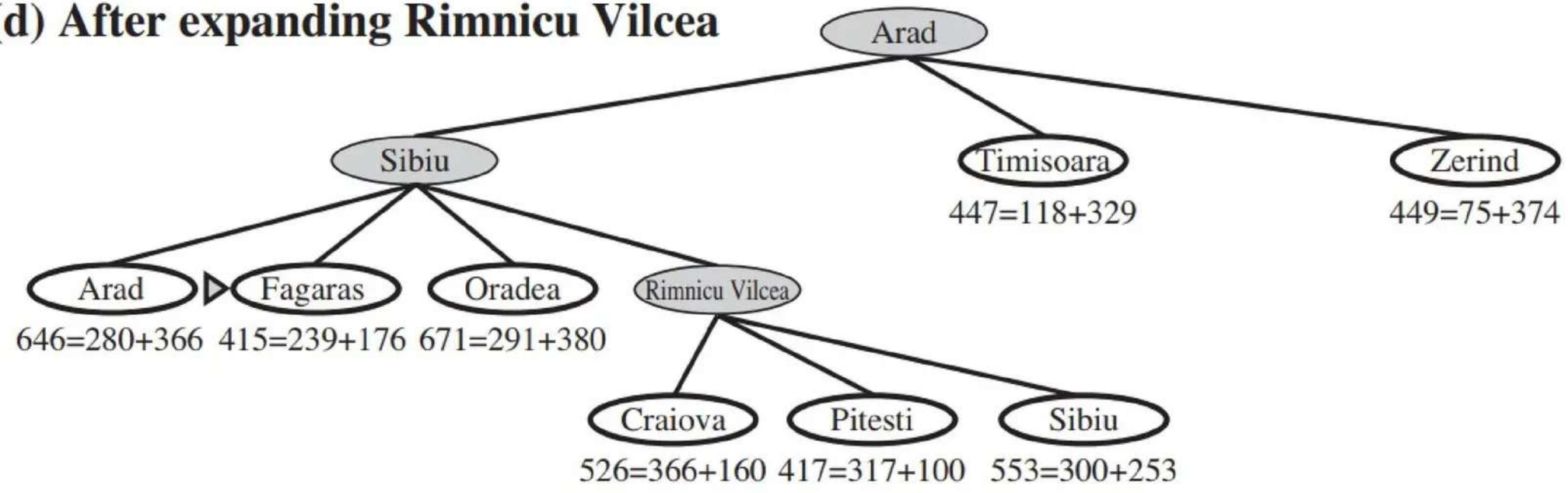
$$h(n) \leq c(n, a, n') + h(n')$$

Consistency is therefore a stricter requirement than admissibility

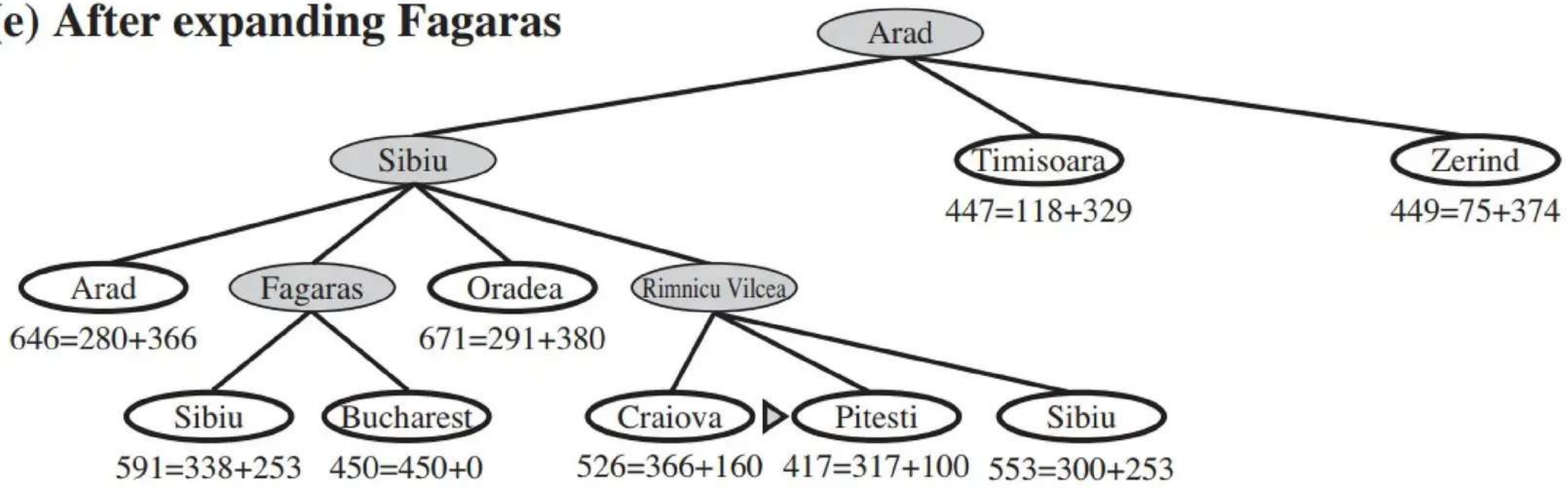
A* in Action for Arad to Bucharest



(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti

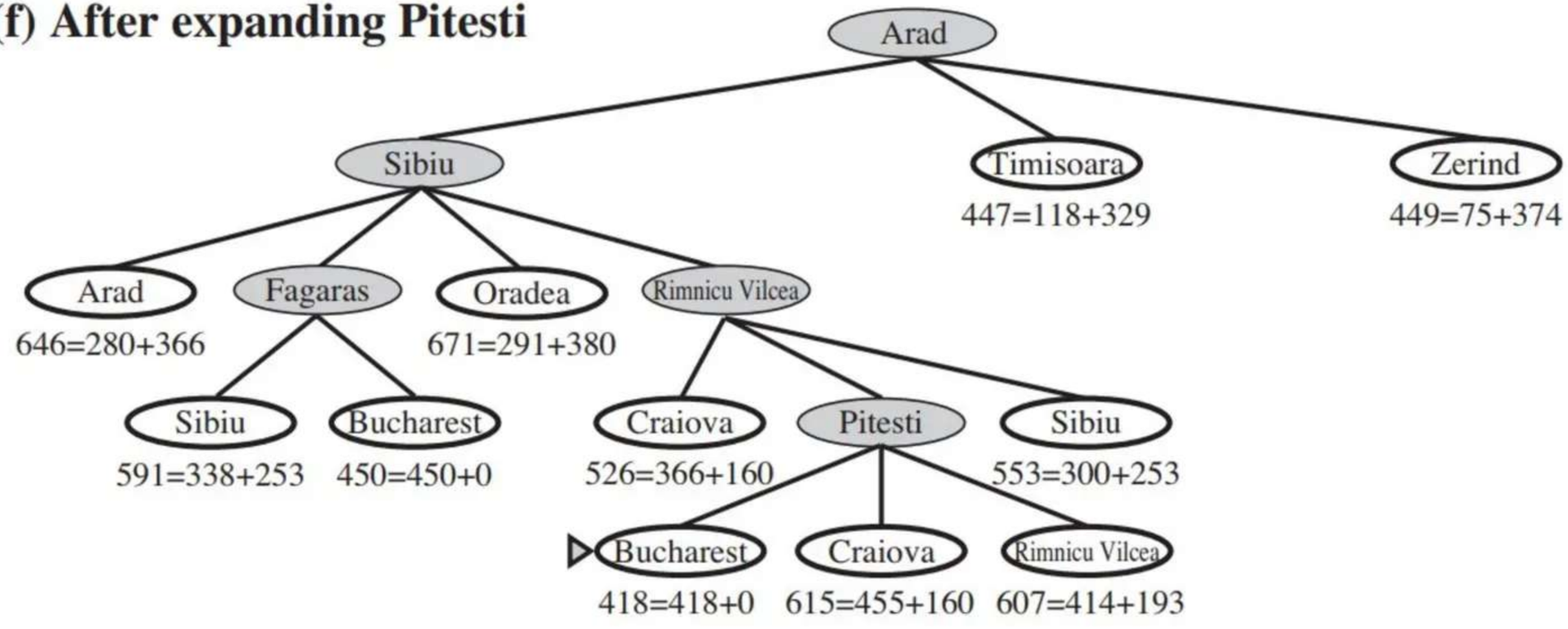


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Optimality of A*

- The tree-search version of A* is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent

1. *if* h(n) *is consistent, then the values of* f(n) *along any path are Increasing*. The proof follows directly from the definition of consistency. Suppose n' is a successor of n;

   then g(n') = g(n) + c(n, a,n') for some action a, and we have

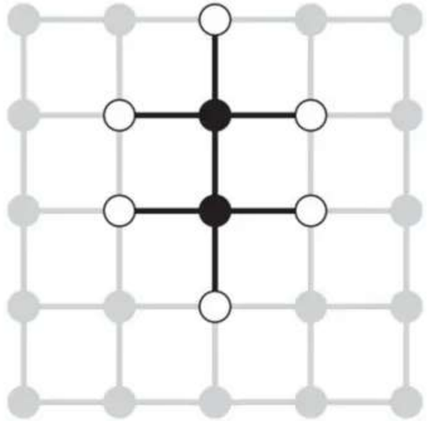
$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

2. The next step is to prove that whenever A* selects a node **n** for expansion, the optimal path to that node has been found

Optimality of A*

- Were this not the case, there would have to be another frontier node **n'** on the optimal path from the start node to **n**, by the graph separation property **f** is nondecreasing along any path, **n'** would have lower f-cost than **n** and would have been selected first

The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes)

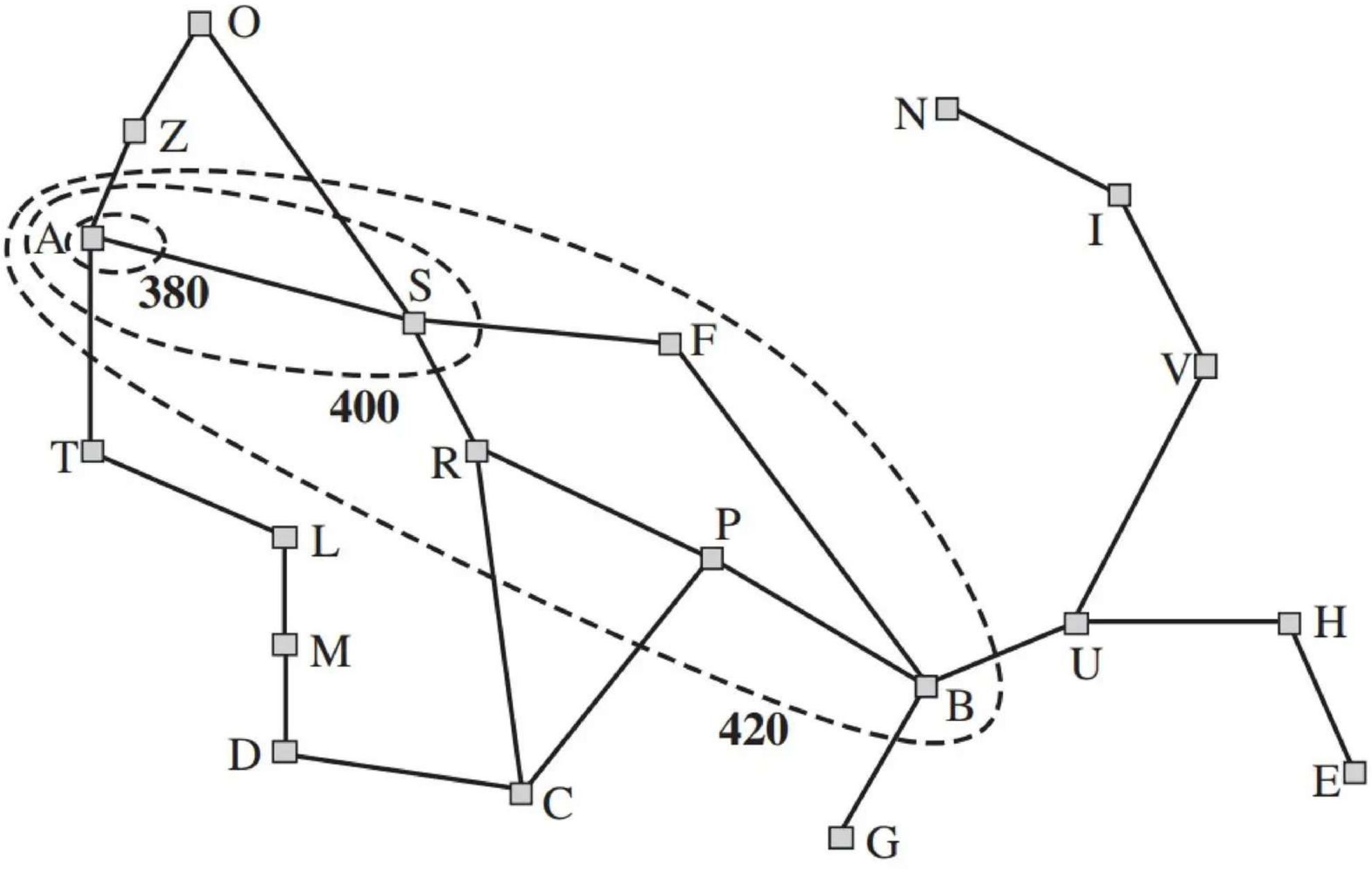


From the two preceding observations, it follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have $h = 0$) and all later goal nodes will be at least as expensive.

Contours in the State Space

- The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map
- With uniform-cost search (A∗ search using $h(n) = 0$), the bands will be "circular" around the start state.
- With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path

Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$



Problem Reduction

AND-OR Graphs, AO* Algorithms

Problem Reduction – AND –OR Graphs

- AND-OR graph is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved
- One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution
- OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved

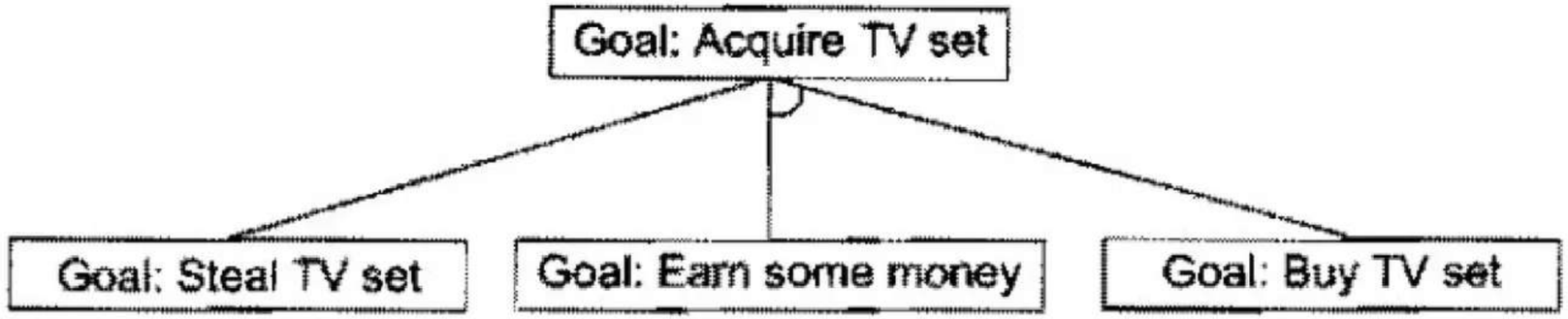


Fig. 3.6 *A Simple AND-OR Graph*

Why best-first search algorithm is not adequate?

- The problem is that the choice of which node to expand next must depend not only on the f' value of that node but also on whether that node is part of the current best path from the initial node

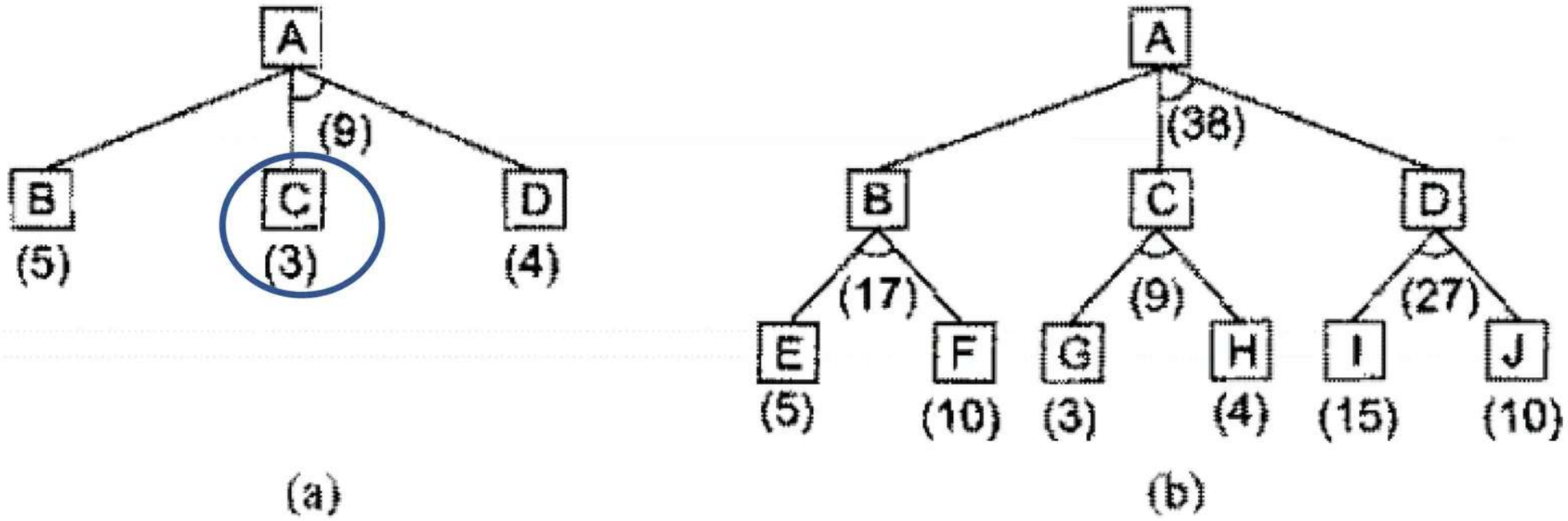


Fig. 3.7 *AND-OR Graphs*

FUTILITY

- If the estimated cost of a solution becomes greater than the value of *FUTILITY*, then we abandon the search.
- *FUTlLITY* should be chosen to correspond to a threshold such that any solution with a cost above it is too expensive to be practical, even if it could ever be found

Algorithm: Problem Reduction

1. Initialize the graph to the starting node,
2. Loop until the starting node is labeled *SOLVED* or until its cost goes above F*UTlLITY*

(a) Traverse the graph, starting at the initial node and following the current best path. and accumulate the set of nodes that are on that path and have not yet been expanded or labelled as solved.

(b) Pick one of these unexpanded nodes and expand it If there are no successors, assign *FUTILITY* as the value of this node, Otherwise, add its successors to the graph and for each of them compute *f'* (use only *h'* and ignore *g*. for reasons we discuss below), If of any node is 0, mark that node as *SOLVED*,

Algorithm: Problem Reduction

(C) (i) Change the f' estimate of the newly expanded node to reflect the new information provided by its successors

(ii) Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as *SOLVED*.

(iii)At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path, This may cause the current best path to change.

(iv) This propagation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their *f'* values be the best estimates available

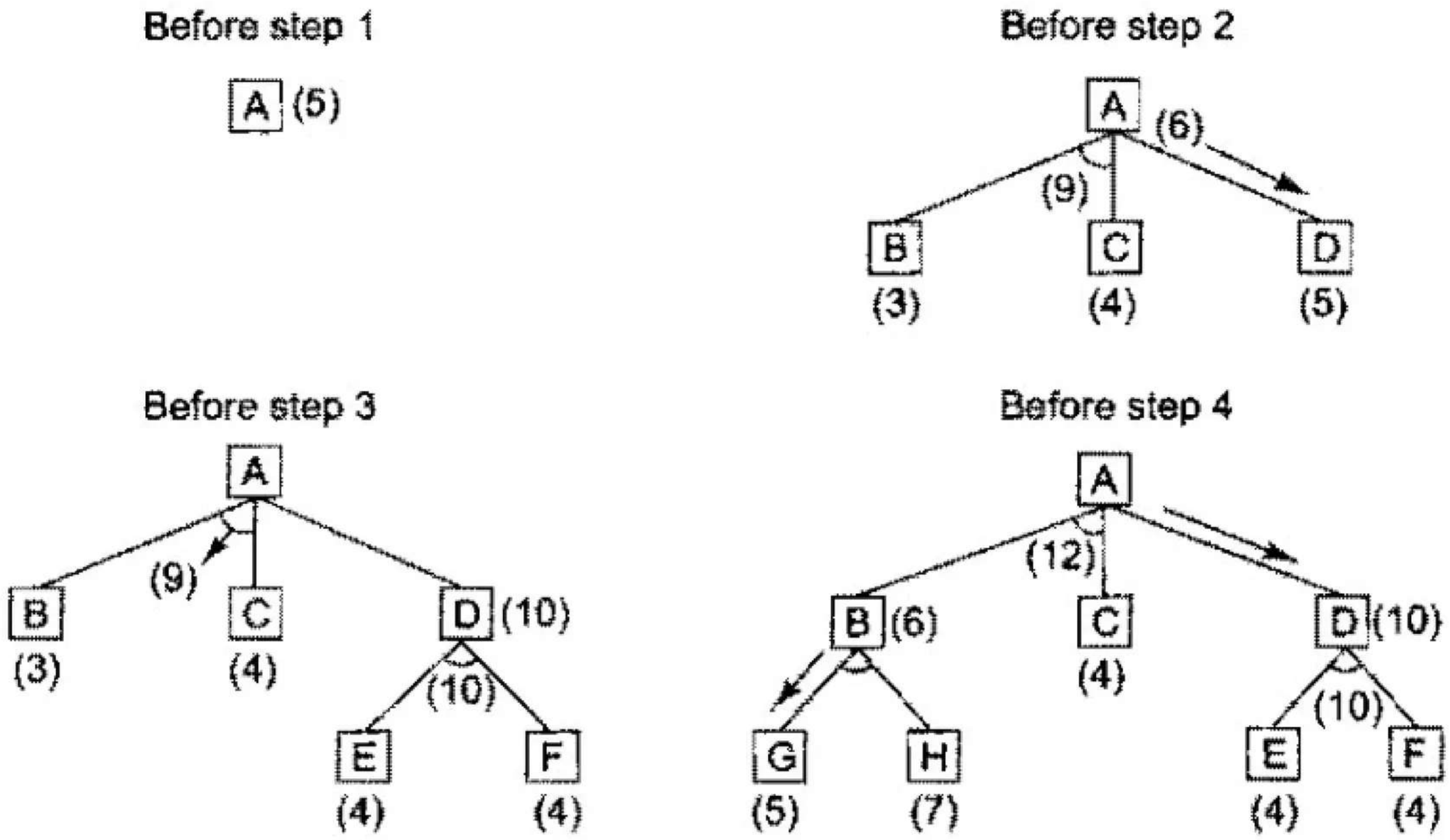


Fig. 3.8 *The Operation of Problem Reduction .*

Difference from Best-First

the desired path from one node to another need not always the one with the lowest cost

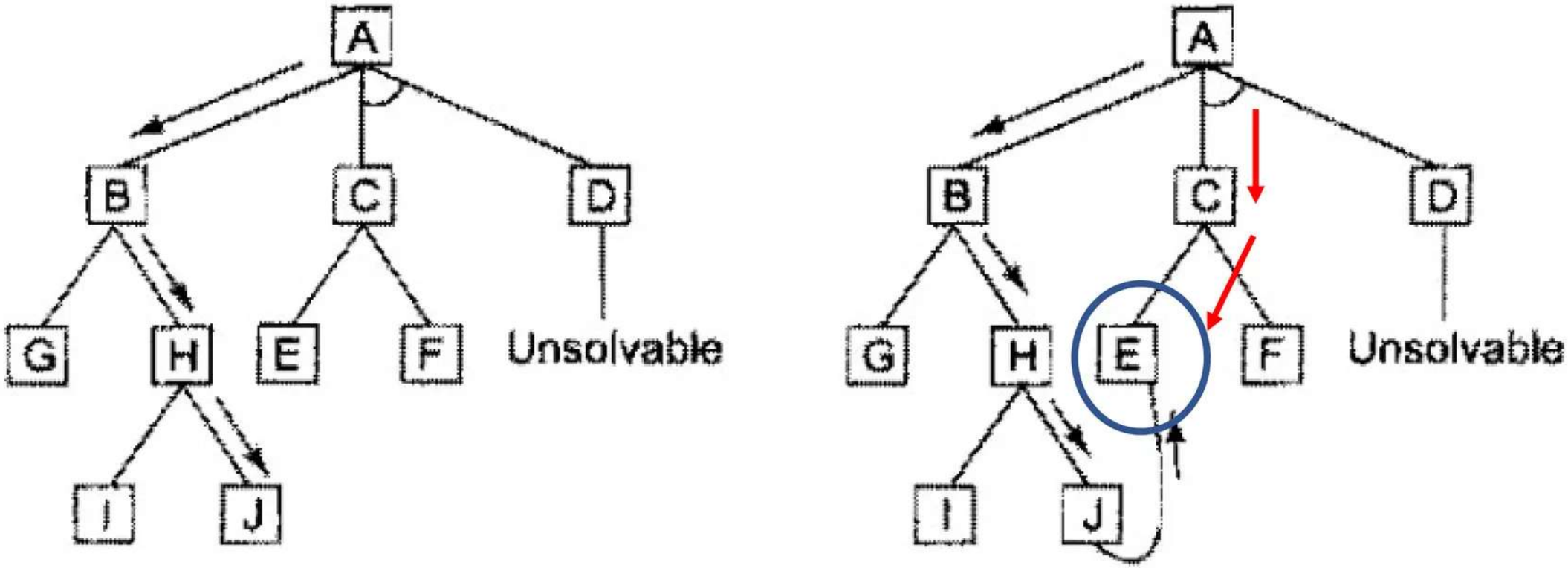


Fig. 3.9 *A Longer Poth May Be Better*

Limitation

No interaction between subgoals

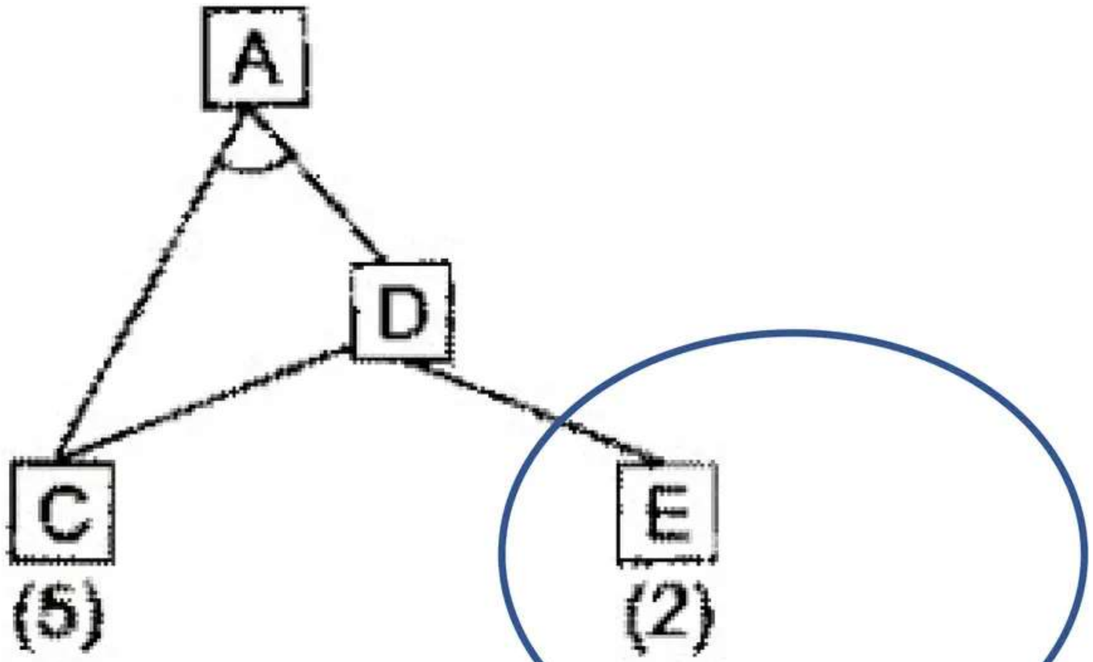


Fig. 3.10 *Interacting Subgoals*

The AO Algorithm (More generic AND/OR Search)*

1. AO* algorithm will use a single structure *GRAPH* Rather than OPEN and CLOSED Lists
2. Each node in the graph will point both down to its immediate successors and up to its immediate predecessors
3. We will not store *g* as we did in the A* algorithm because of the top-down traversing of the best-known path, which guarantees that only nodes that are on the best path will ever be considered for expansion. So *h'* Will serve as the estimate of goodness of a node

Algorithm: AO*

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT.)* Compute $h'(INIT)$
2. Until *INIT* is labeled *SOLVED* or until *INIT's* h' value becomes greater than *FUTILITY,* repeat the following procedure:
   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE.*
   (b) Generate the successors of *NODE.* If there are none, then assign *FUTILITY* as the h' value of *NODE.* This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR)* that is not also an ancestor of *NODE* do the following:
      (i) Add *SUCCESSOR* to *GRAPH.*
      (ii) If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an h' value of 0.
      (iii) If *SUCCESSOR* is not a terminal node, compute its h' value.
   (c) Propagate the newly discovered information up the graph by doing the following: Let S be a set of nodes that have been labeled *SOLVED* or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to *NODE.* Until S is empty, repeat the, following procedure:

(i) If possible, select from S a node none of whose descendants in *GRAPH* occurs in S. If there is no such node, select any node from S. Call this node *CURRENT*, and remove it from S.

(ii) Compute the cost of each of the arcs emerging from *CURRENT*. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT'S* new h' value the minimum of the costs just computed for the arcs emerging from it.

(iii) Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.

(iv) Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED*.

(v) If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to S.

The AO Algorithm Issues*

1. In step 2(c)(v) the ancestors of anode whose cost was altered are added to the set of nodes whose costs must also be revised. As stated, the algorithm will insert all the node's ancestors' into the set. which may result in the propagation of the cost change back up through a large number of paths that are already known not to be very good

- Do We need to Update node B?
- Isn't the work expended on the path through B is wasted ?
- What if the cost of node E is updated as 10 and is not propagated to B ?

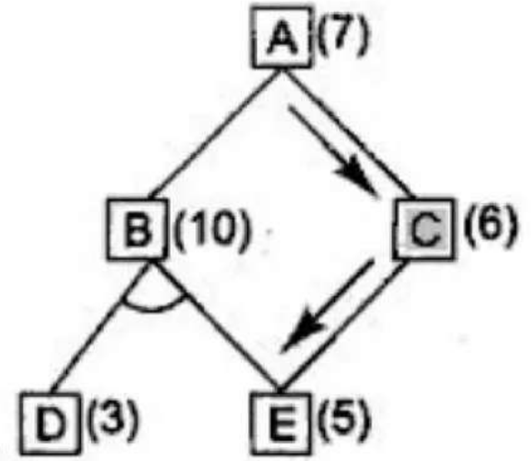


Fig. 3.11 *An Unnecessary Backward Propagation*

The AO Algorithm Issues*

2. if the node whose cost has changed is farther down in the search Graph, the error may never be detected

If the cost of G is revised and if it is not immediately propagated back to *E*, then the change will never be recorded and a nonoptimal solution through B may be discovered

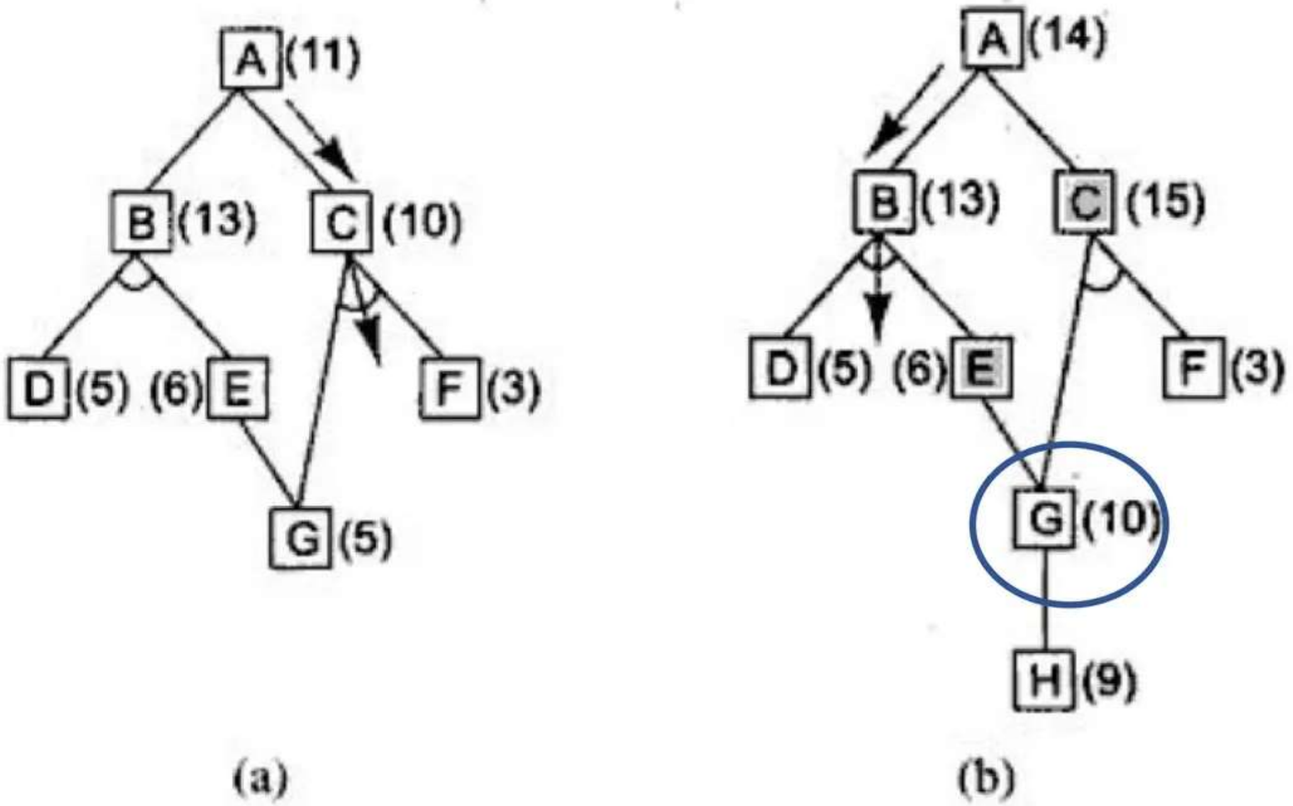


Fig. 3.12 *A Necessary Backward Propagation*

Game Playing-Adversarial Search

Games, mini-max algorithm,
optimal decisions in multiplayer games, Problem in Game playing, Alpha-Beta pruning,
Evaluation functions.

Adversarial Search (games)

- **competitive** environments, in which the agents goals are in conflict, giving rise to **adversarial search** problems
- In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, **turn-taking**, two-player, zero-sum games of perfect information (such as chess).
- In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.
- For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

Why Games?

1. Games are interesting *because* they are too hard to solve.

For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes

2. Games, like the real world, therefore require the ability to make *some* decision even when calculating the *optimal* decision is infeasible.

3. Games also penalize inefficiency severely.

A∗ search that is half as efficient will simply take twice as long to run to completion, a chess program that is half as efficient in using its available time probably will be beaten into the ground

What we learn

Techniques for choosing a good move when time is limited

1. **Pruning** allows us to ignore portions of the search tree that make no difference to the final choice
2. Heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete search.
3. Some games like **bridge**, which includes elements of **imperfect information** because not all cards are visible to each player

Two player Games - formulation

We first consider games with two players, whom we call

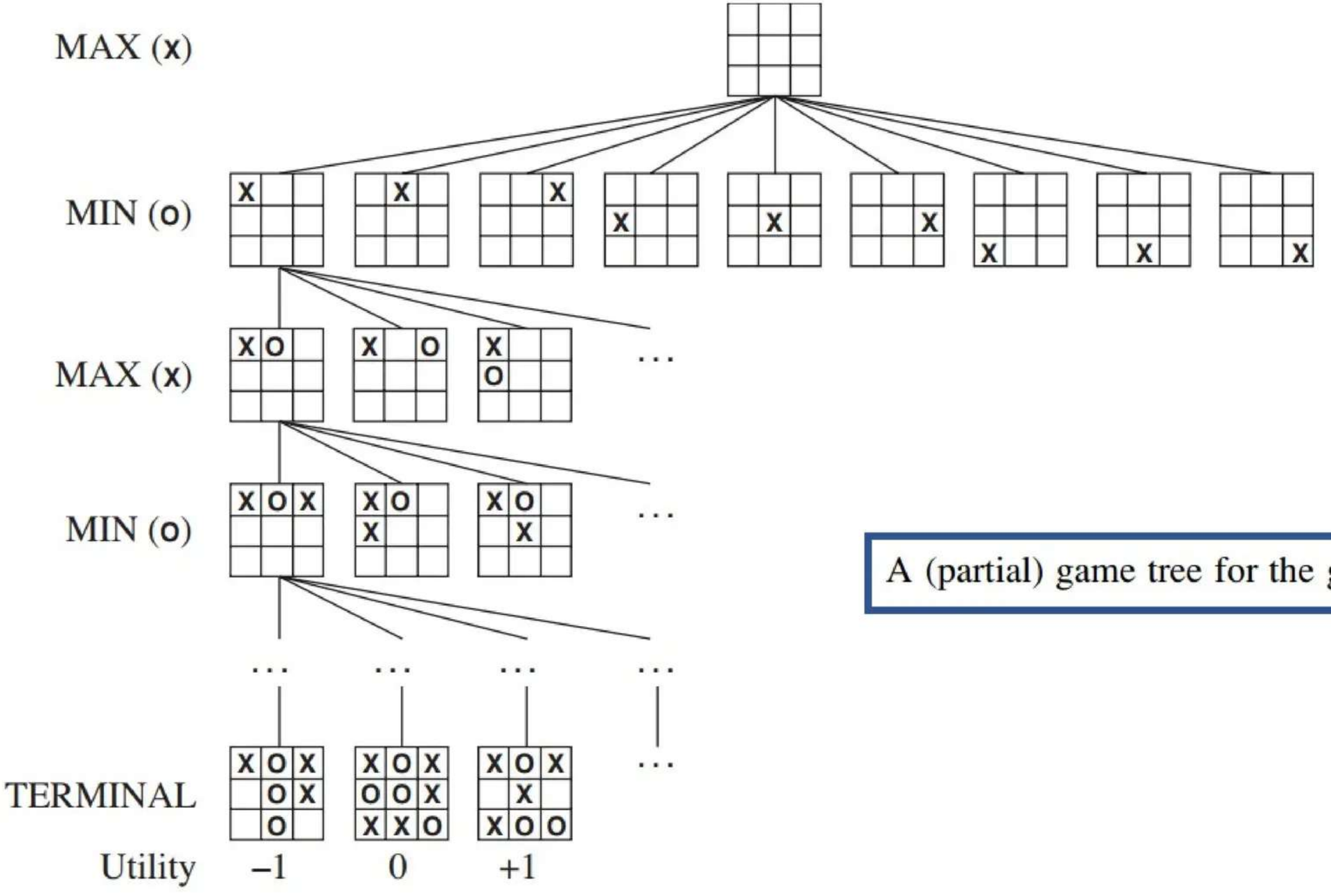
- MAX and MIN MAX moves first, and then they take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.

A game can be formally defined as a kind of search problem with the following elements:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.

Two player Games - formulation

- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values $+1$, 0, or $\frac{1}{2}$. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to $+192$. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.



A (partial) game tree for the game of tic-tac-toe

The minimax algorithm

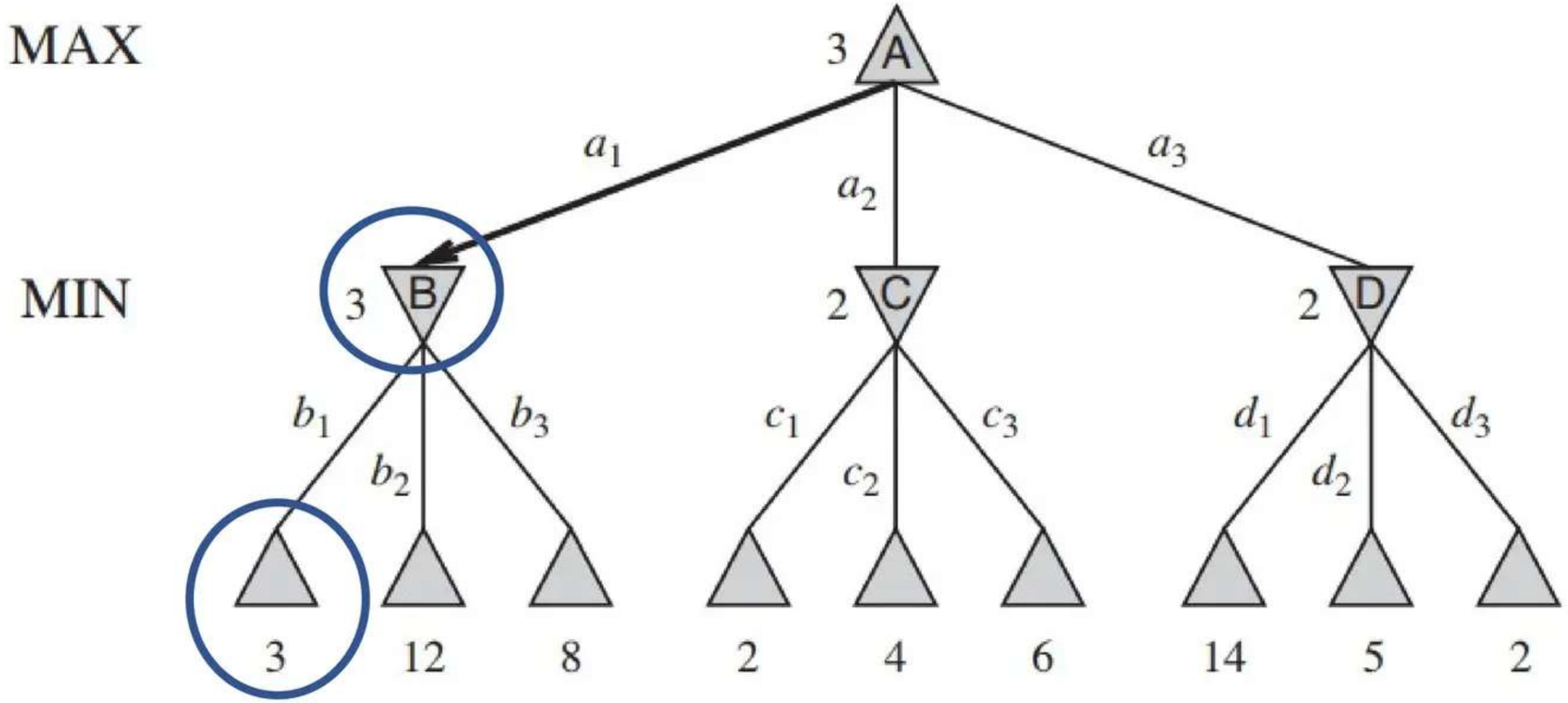


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

MiniMax Value

Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(n)

- Given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.
- So we have the following:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

MiniMax Complexity

The minimax algorithm performs a complete depth-first exploration of the game tree.

If the maximum depth of the tree is **m** and there are **b** legal moves at each point, then the time complexity of the minimax algorithm is $O(\mathbf{b}^m)$.

The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time

Optimal decisions in multiplayer games

When we extend the minimax idea to multiplayer games some interesting new conceptual issues arise

- we need to replace the single value for each node with a *vector* of values.

For example, in a three-player game with players A, B, and C, a vector $<v_A,v_B,v_C>$ is associated with each node.

For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.)

The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Optimal decisions in multiplayer games

- Multiplayer games usually involve **alliances**, whether formal or informal, among the players.
- Alliances are made and broken as the game proceeds.

suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behavior.

Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement.

a social stigma attaches to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy.

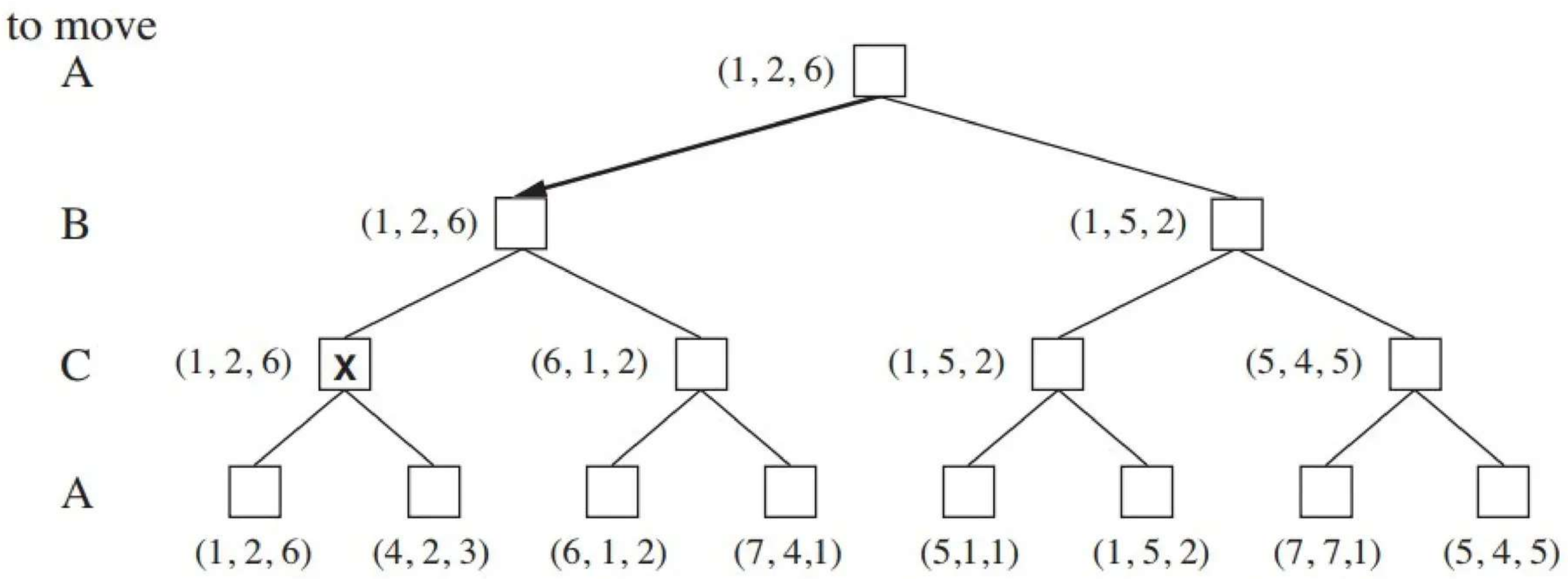


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

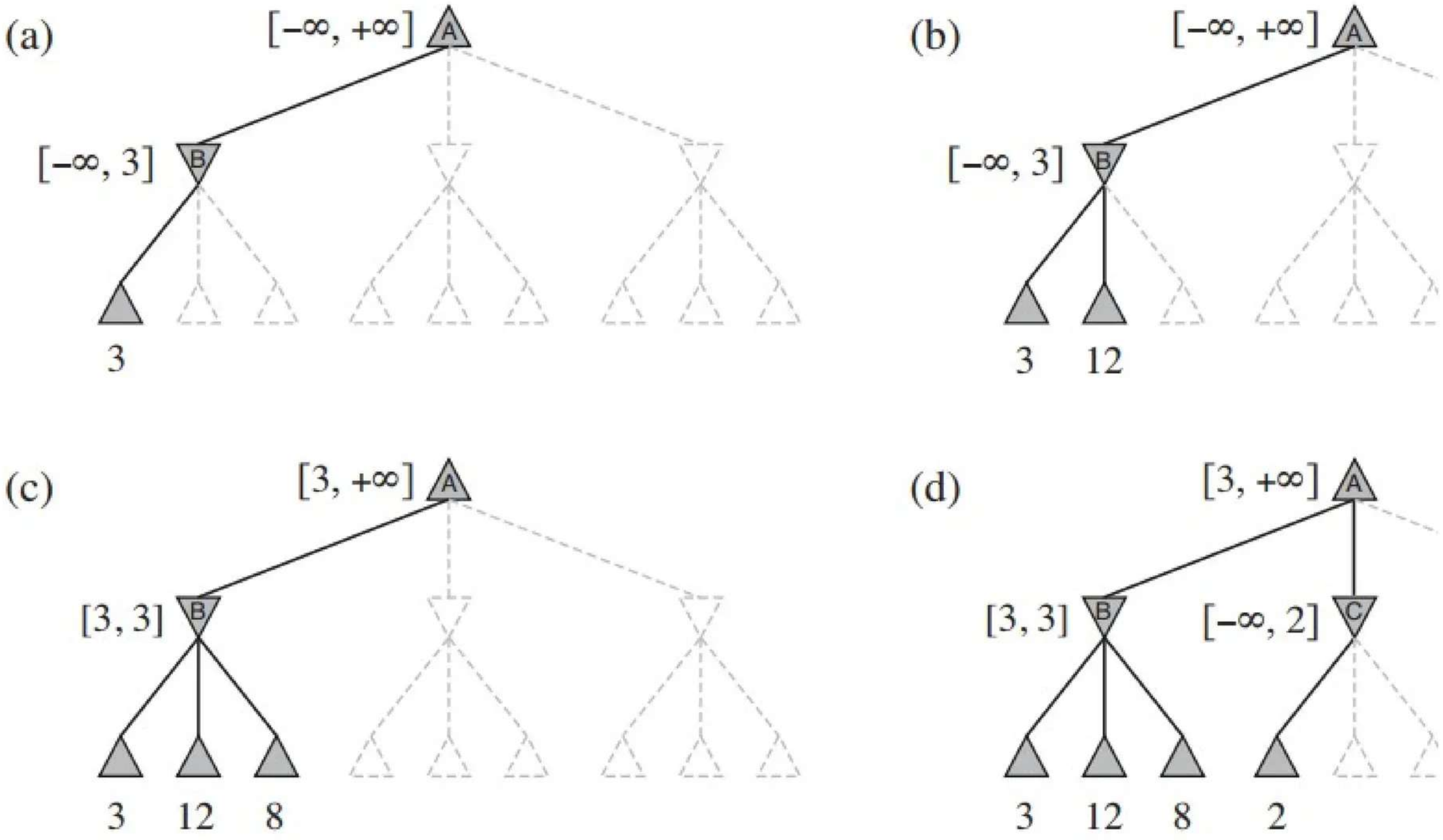
Alpha-Beta Pruning

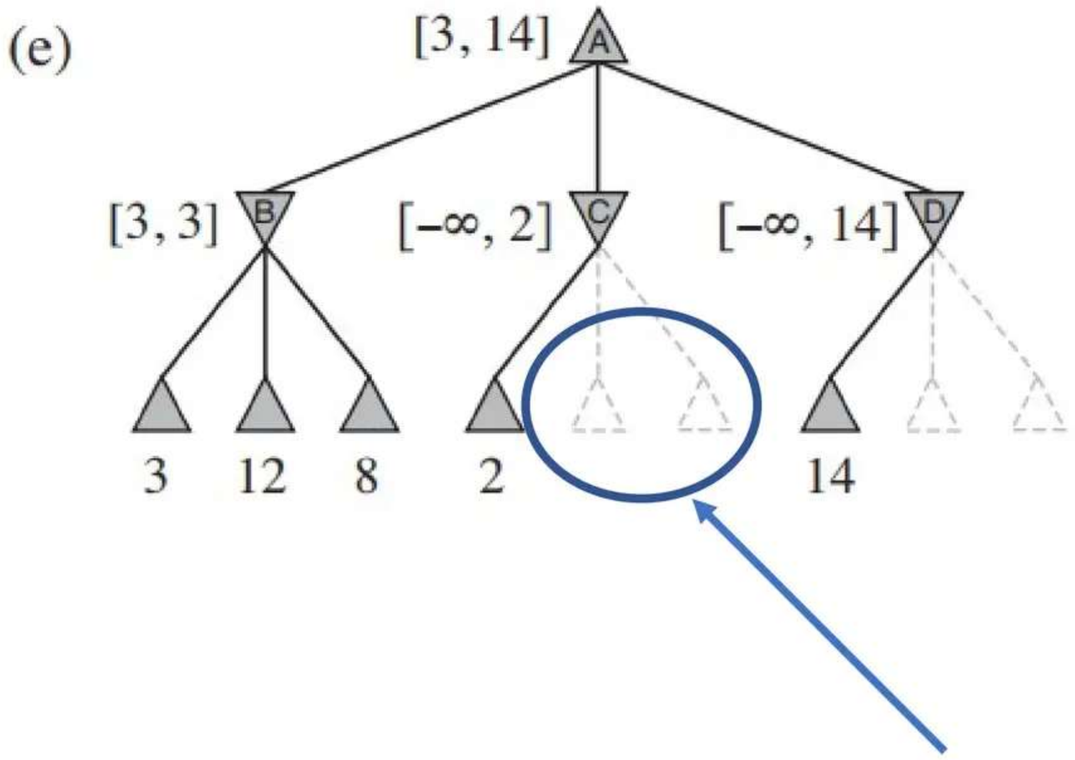
- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half.
- **alpha–beta pruning**, When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
- Let the two unevaluated successors of node C have values x and y

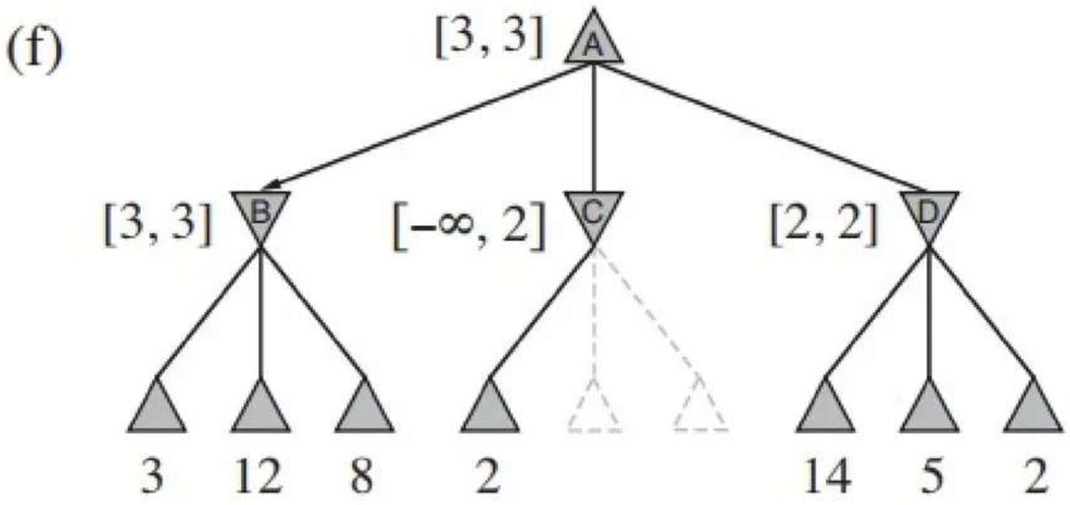
$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}
$$

Alpha-Beta Pruning

- minimax decision are *independent* of the values of the pruned leaves x and y
- Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2





This Need not be expanded

Evaluation functions

- An evaluation function returns an *estimate* of the expected utility of the game from a given position (Just like the heuristic functions of which return an estimate of the distance to the goal)
- the performance of a game-playing program depends strongly on the quality of its evaluation function.
- An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.
- How exactly do we design good evaluation functions?

1. states that are wins must evaluate better than draws, which in turn must be better than losses - order the *terminal* states
2. the computation of Evaluation Function must not take too long!

Evaluation functions

3. for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning

- Most evaluation functions work by calculating various **features** of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on
- The features, taken together, define various *categories* or *equivalence classes* of states: the states in each category have the same values for all the features

Ex: one category contains all two-pawn vs. one-pawn endgames

- Any given category will contain some states that lead to wins, some that lead to draws, and some that lead to losses

Evaluation functions

- The evaluation function cannot know which states are which (Loss / Win / Draw leading), but it can return a single value that reflects the *proportion* of states with each outcome.

For example, suppose our experience suggests that 72% of the states encountered in the two-pawns vs. one-pawn category lead to a win (utility +1); 20% to a loss (0), and 8% to a draw (1/2).

Then a reasonable evaluation for states in the category is the **expected value**: $(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$

As with terminal states, the evaluation function need not return actual expected values as long as the *ordering* of the states is the same

Evaluation functions

- In principle, the expected value can be determined for each category, resulting in an evaluation function that works for any state.
- In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning

Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value

Evaluation functions

- For example, introductory chess books give an approximate **material value** for each piece:

each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the **queen 9**. Other features such as "good pawn structure" and "king safety" might be worth half a pawn. These feature values are then simply added up to obtain the evaluation of the position.

A secure **advantage** equivalent to a pawn gives a substantial likelihood of winning

and

a secure **advantage** equivalent to three pawns should give almost certain victory

weighted linear function

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where each w_i is a weight and each f_i is a feature of the position

Disadv:- involves a strong assumption that the contribution of each feature is *independent* of the values of the other features

Hence, the current programs for chess and other games also use *nonlinear* combinations of features.

For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame