Artificial Intelligence

1. The Intelligent Computer

Definition: **Artificial intelligence** is the study of the computations that make it possible to perceive, reason, and act.

Applications

- Searching, Problem Solving
- Common Sense Reasoning
- Natural Language Processing
- Expert System

Knowledge Representation

- Semantic Net
- Logical, Procedure
- Frame & Script
- Property List

Artificial Intelligence Programming

- Primary symbolic processes
- Heuristic search (solution step implicit)
- Control structure usually separate from domain knowledge
- Usually easy to modify, update, and enlarge
- Some incorrect answers often tolerate
- Satisfactory answers usually acceptable

Conventional Computer Programming

- Often primarily numeric
- Algorithmic (solution steps explicit)
- Information and control integrated
- Difficult to modify
- Correct answers required
- Best possible solution usually sought

2. Semantic Nets and Description Matching

2.1 Semantic Nets

2.1.1 Good Representations Are the Key to Good Problem Solving

The Farmer, Fox, Goose, and Grain

A farmer wants to move himself, a silver fox, a fat goose, and some tasty grain across a river. Unfortunately, his boat is so tiny he can take only one of his possessions across on any trip. An unattended fox will eat a goose, and an unattended goose will eat grain, so the farmer must not leave the fox alone with the goose or the goose alone with the grain.

- English is not a good representation.
- The drawing of states and transition (node-and-link) is a good description because the allowed situations and legal crossings are clearly defined and there are no irrelevant details.
- The farmer and his three possessions each can be on either of the two river banks, there are $2^{1+3} = 16$ arrangements.
- 10 of the states are safe in the sense that nothing is eaten.
- 6 unsafe arrangements place an animal and something the animal likes to eat on one side, with the farmer on the other.
- Because there are 10 safe arrangements, there are ${}^{10}P_2 = 10x9 = 90$ ordered pairs, but only 20 of these pairs satisfy the conditions required for links.

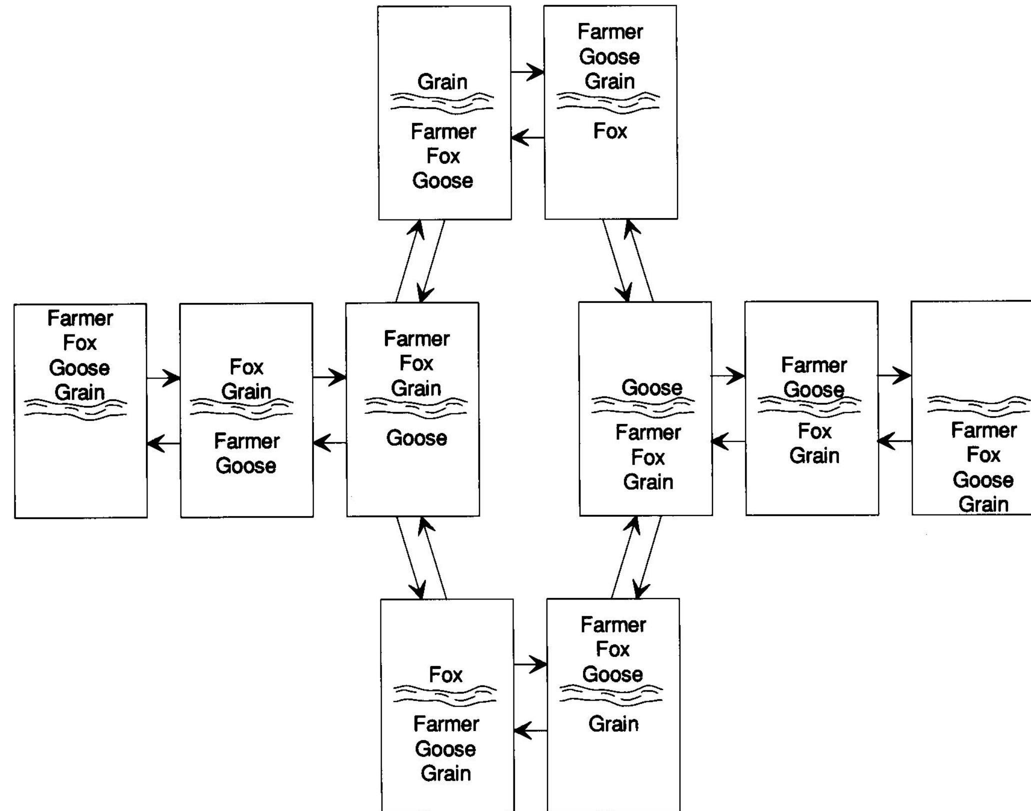

Figure 2.1.1-1 The problem of the farmer, fox, goose, and grain. The farmer must get his fox, goose, and grain across the river, from the arrangement on the left to the arrangement on the right. His boat will hold only him and one of his three possessions.

2.1.2 Good Representations Support Explicit, Constraint-Exposing Description

Good representations have the following characteristics.

- Good representations make the important objects and relations explicit: You can see what is going on at a glance.
- They expose natural constraints: You can express the way one object or relation influences another.
- They bring objects and relations together: You can see all you need to see at one time, as if through a straw.
- They suppress irrelevant detail: You can keep rarely used details out of sight, but still get to them when necessary.
- They are transparent: You can understand what is being said.
- They are complete: You can say all that needs to be said.
- They are concise: You can say what you need to say efficiently.
- They are fast: You can store and retrieve information rapidly.
- They are computable: You can create them with an existing procedure.

2.1.3 A Representation Has Four Fundamental Parts

A representation consists of the following four fundamental parts:

1. A **lexical part** that determines which symbols are allowed in the representation's vocabulary
2. A **structural part** that describes constraints on how the symbols can be arranged
3. A **procedural part** that specifies access procedures that enable you to create descriptions, to modify them, and to answer questions using them
4. A **semantic part** that establishes a way of associating meaning with the descriptions

In the representation used to solve the farmer, fox, goose, and grain problem,

- The lexical part of the representation determines that nodes and links are involved.
- The structural part specifies that links connect node pairs.
- The semantic part establishes that nodes correspond to arrangements of the farmer and his possessions and links correspond to river traversals.
- The procedural part provides constructors that guide pencil and readers that interpret what is seen.

2.1.4 Semantic Nets Convey Meaning

- From the lexical perspective, semantic nets consist of nodes, denoting objects, links, denoting relations between objects, and link labels that denote particular relations.
- From the structural perspective, nodes are connected to each other by labeled links. In diagrams, nodes often appear as circles, ellipses, or rectangles, and links appear as arrows pointing from one node, the tail node, to another node, the head node.
- From the semantic perspective, the meaning of nodes and links depends on the application.
- From the procedural perspective, access procedures are, in general, any one of constructor procedures, reader procedures, writer procedures, or possibly erasure procedures. Semantic nets use constructors to make nodes and links, readers to answer questions about nodes and links, writers to alter nodes and links, and, occasionally, erasers to delete nodes and links.

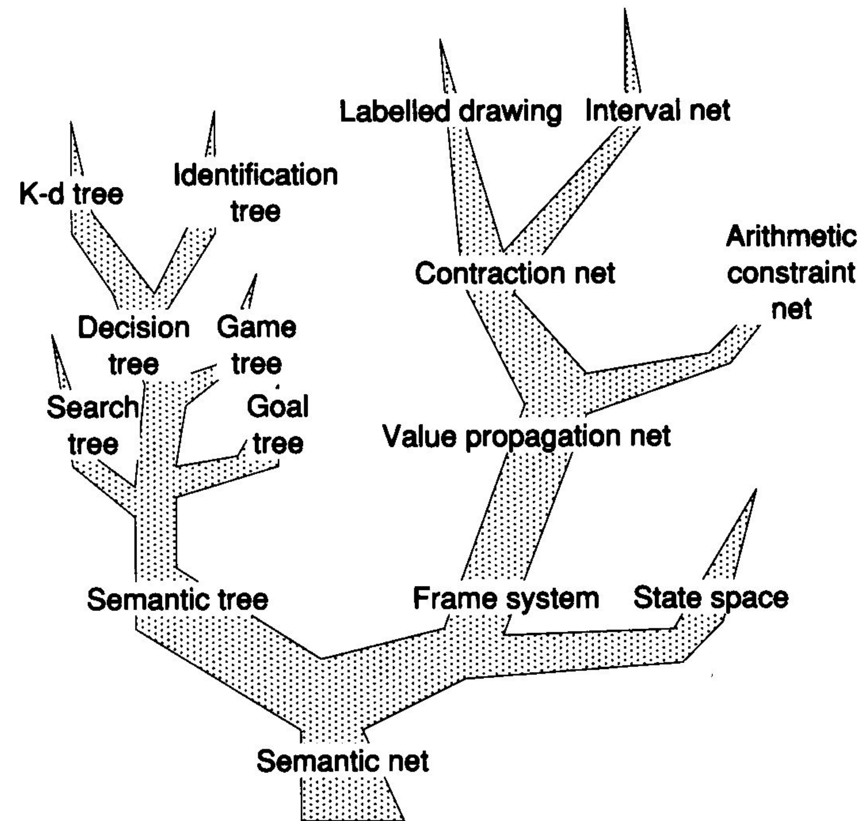

Figure 2.1.4-1 Part of the semantic-net family of representations. Although many programs use one of the family members shown, others use important representations that lie outside of the family.

3. Nets and Basic Search

3.1 Blind Methods

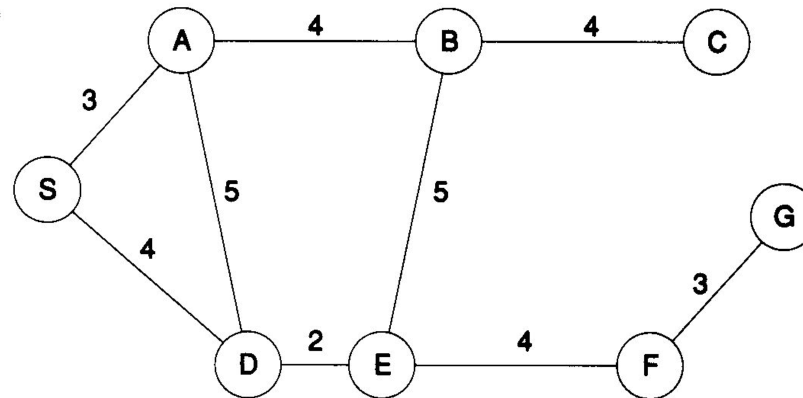

Figure 3.1-1 A basic search problem. A path is to be found from the start node, S, to the goal node, G. Search procedures explore nets such as these, learning about connections and distances as they go.

To find an appropriate path through the highway map, two different costs should be considered:

1. Computation cost
2. Travel cost

3.1.1 Net Search Is Really Tree Search

A search tree, a kind of semantic tree, is a representation in which

- Nodes denote paths.
- Branches connect paths to one-step path extensions.

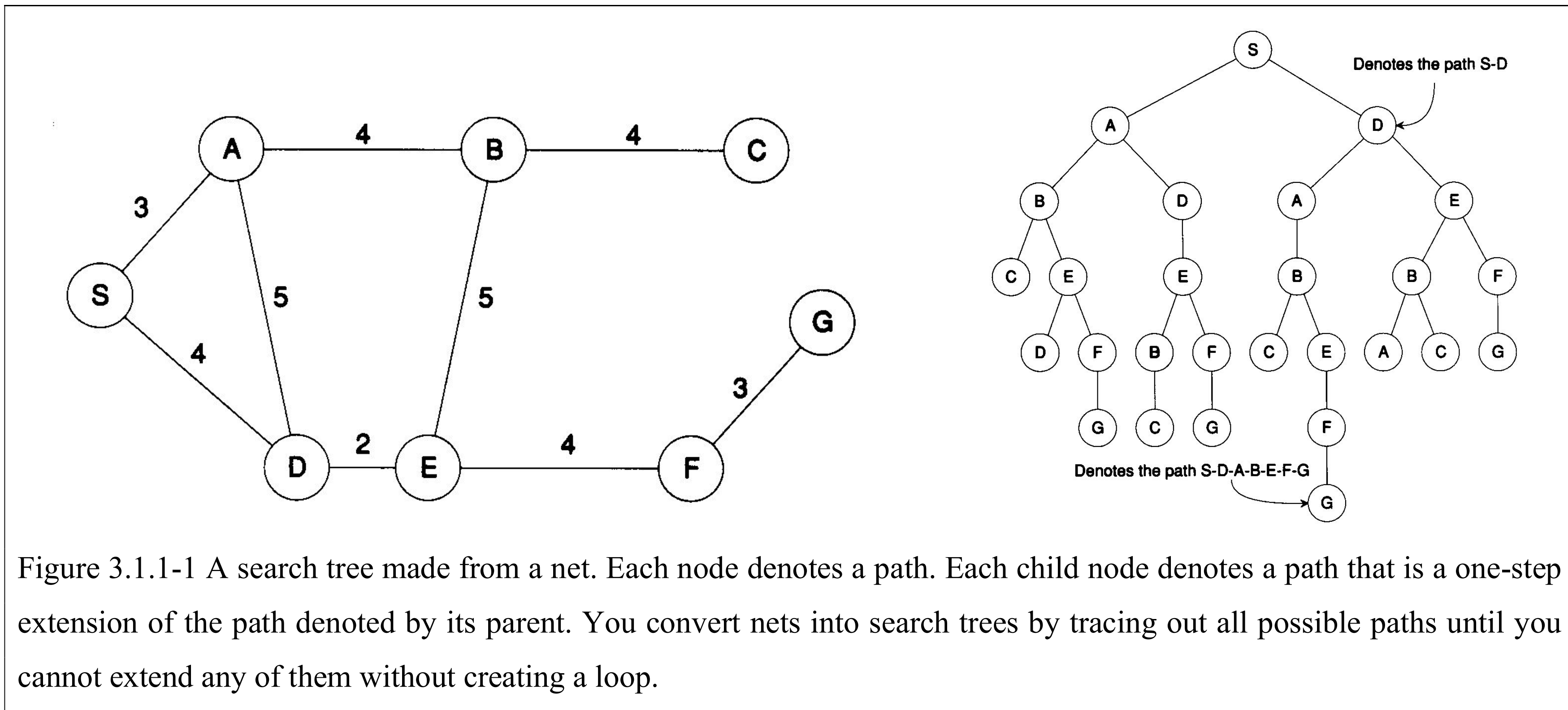

Figure 3.1.1-1 A search tree made from a net. Each node denotes a path. Each child node denotes a path that is a one-step extension of the path denoted by its parent. You convert nets into search trees by tracing out all possible paths until you cannot extend any of them without creating a loop.

- Each node is labeled with only the **terminal node** of the path it denotes.
- Each **child** denotes a path that is a one-city extension of the path denoted by its **parent**.
- **Loop** is not allowed in the search tree.
- The node at the top of a semantic tree, the node with no parent, is called the **root node**.
- The nodes at the bottom, the ones with no children, are called **leaf nodes**.
- One node is the **ancestor** of another, a **descendant**.
- If a node has b children, it is said to have a **branching factor** of b.
- If the number of children is always b for every non-leaf node, then the tree is said to have a branching factor of b.
- Each path that does not reach the goal is called a **partial path**.
- Each path that does reach the goal is called a **complete path**, and the corresponding node is called a **goal node**.
- Determining the children of a node is called **expanding** the node.
- Nodes are said to be **open** until they are expanded, whereupon they become **closed**.

3.1.2 Search Trees Explode Exponentially

- The total number of paths in a tree with branching factor b and depth d is b^d.
- The number of paths is said to explode exponentially as the depth of the search tree increases.

3.1.3 Depth-First Search Dives into the Search Tree

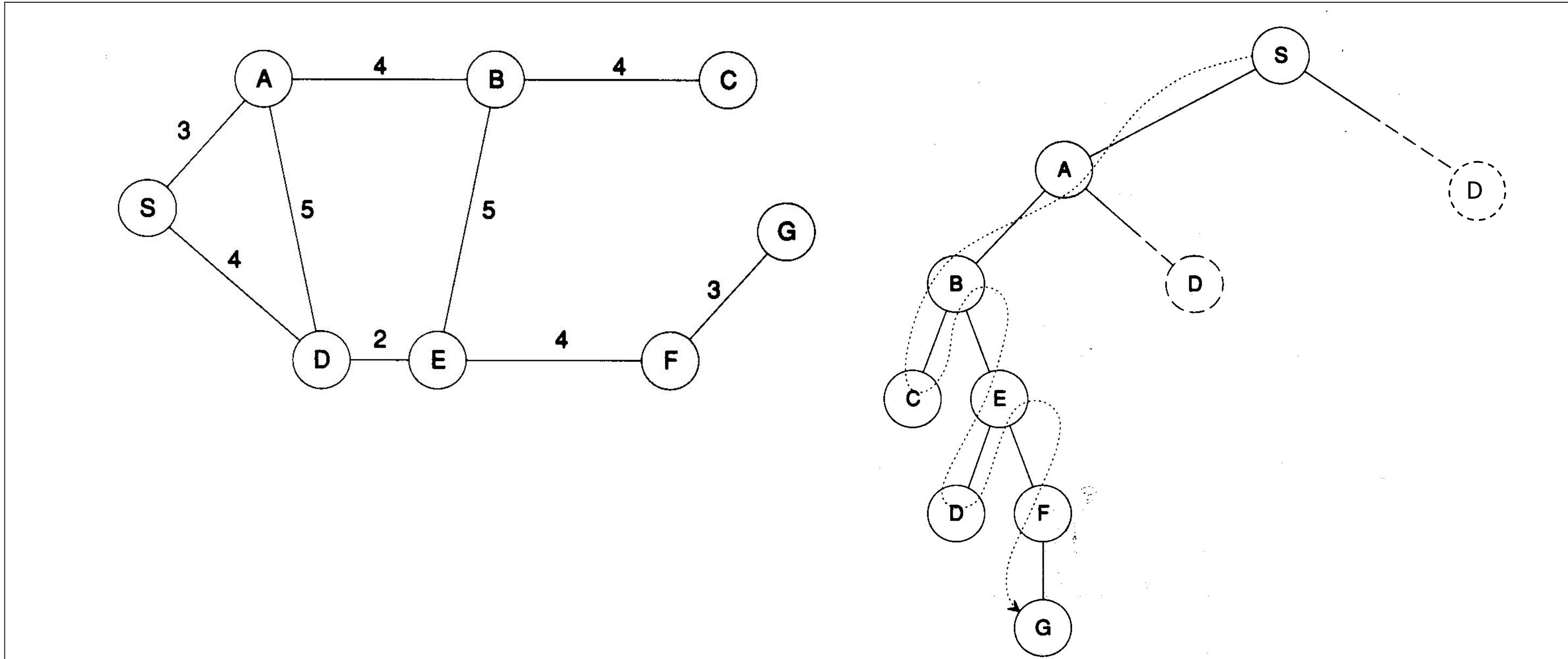

Figure 3.1.3-1 An example of depth-first search. One alternative is selected and pursued at each node until the goal is reached or a node is reached where further downward motion is impossible. When further downward motion is impossible, search is restarted at the nearest ancestor node with unexplored children.

To conduct a depth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
    - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - Reject all new paths with loops.
    - Add the new paths, if any, to the front of the queue.
- If the goal node is found, announce success; otherwise, announce failure.

3.1.4 Breadth-First Search Pushes Uniformly into the Search Tree

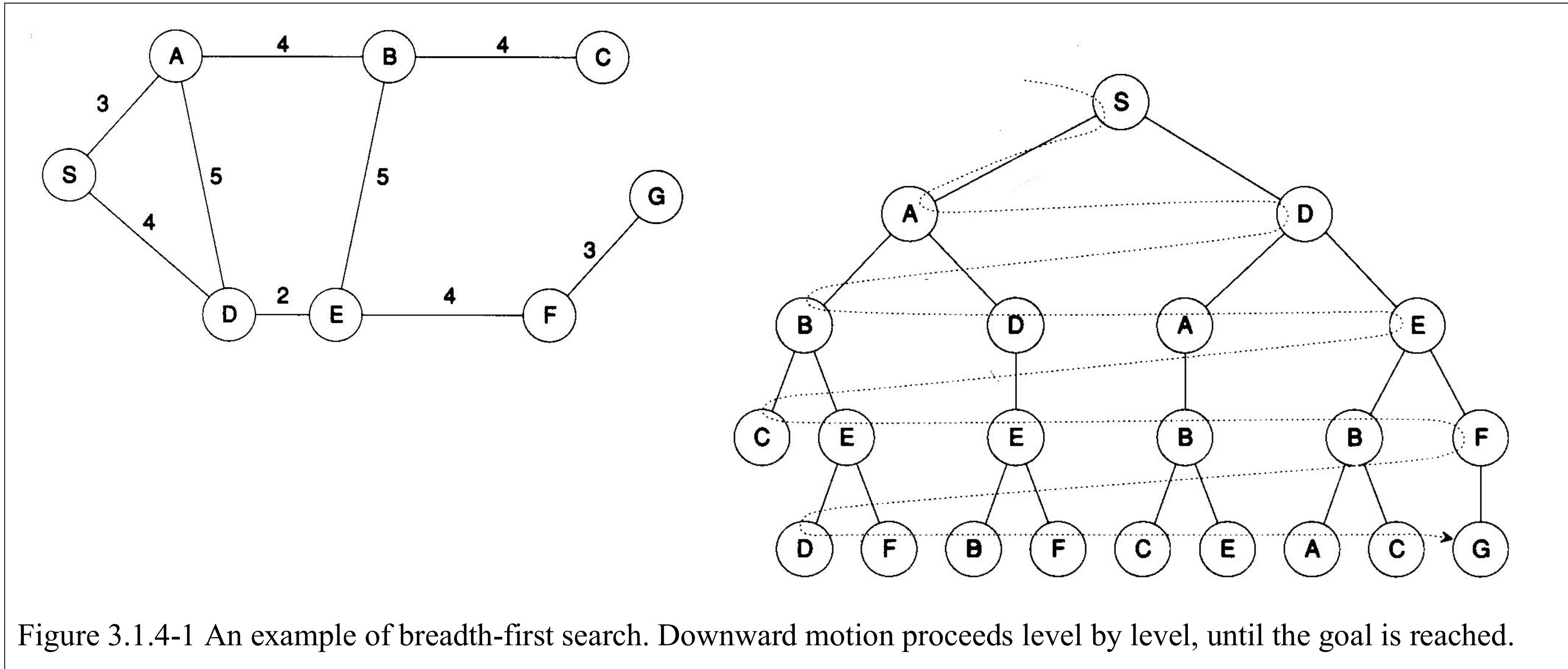

Figure 3.1.4-1 An example of breadth-first search. Downward motion proceeds level by level, until the goal is reached.

To conduct a breadth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the back of the queue.
  - If the goal node is found, announce success; otherwise, announce failure.

3.1.5 The Right Search Depends on the Tree

- Depth-first search is a good idea when all partial paths either reach dead ends or become complete paths after a reasonable number of steps.
- Breadth-first search works even in trees that have infinitely long paths, that neither reach dead ends nor become complete paths.
- Breath-first search is a bad idea if the branching factor is large or infinite, because of exponential explosion.

3.1.6 Nondeterministic Search Moves Randomly into the Search Tree

- If there is no information about a search problem that it is either a large branching factor or long useless paths, nondeterministic search can be chosen.
- In nondeterministic search, an open node is chosen to be expanded at random.

To conduct a nondeterministic search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
    - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - Reject all new paths with loops.
    - Add the new paths at random places in the queue.
    - If the goal node is found, announce success; otherwise, announce failure.

3.2 Heuristically Informed Methods

- Search efficiency may improve spectacularly if there is a way to order the choices so that the most promising are explored earliest.

3.2.1 Quality Measurements Turn Depth-First Search into Hill Climbing

- To move through a tree of paths using hill climbing, the process is similar in depth-first search, except that the choices are ordered according to some heuristic measure, eg. the remaining distance to the goal.

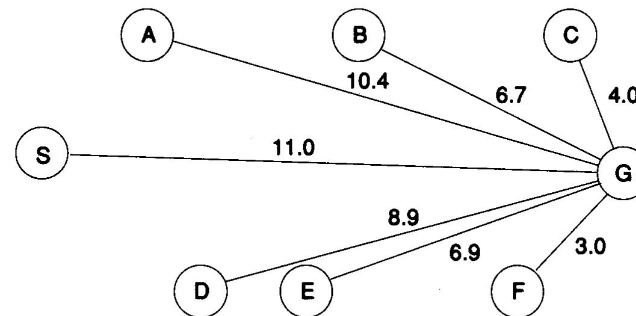

Figure 3.2.1-1 Here you see the distances between each city and the goal. If you wish to reach the goal, it is usually better to be in a city that is close, but not necessarily.

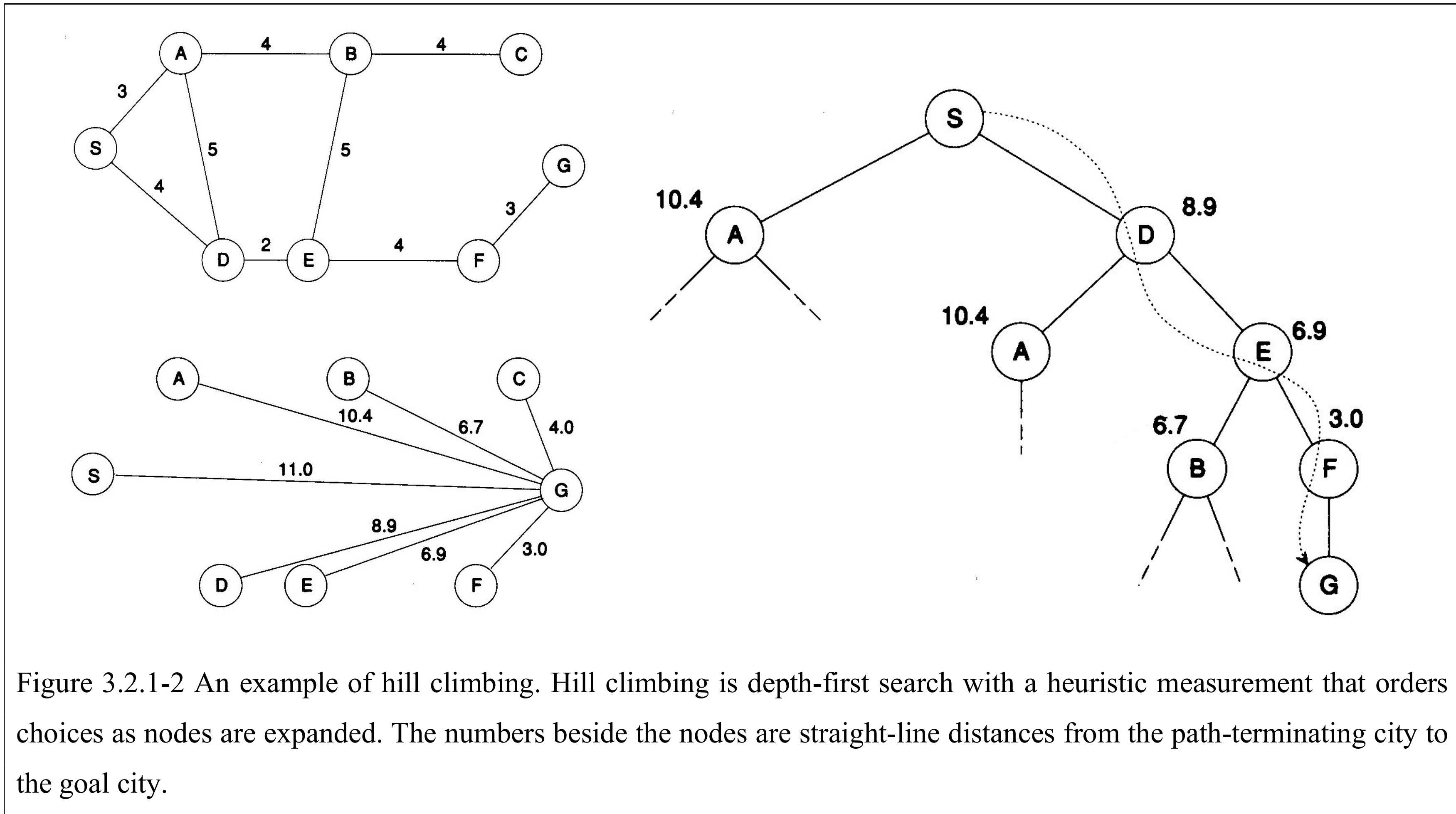

Figure 3.2.1-2 An example of hill climbing. Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded. The numbers beside the nodes are straight-line distances from the path-terminating city to the goal city.

To conduct a hill-climbing search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - *Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal.*
  - Add the new paths, if any, to the front of the queue.
  - If the goal node is found, announce success; otherwise, announce failure.

- *Whenever faced with a search problem, note that, more knowledge generally leads to reduced search time.*
- *When you think you need a better search method, find another space to search instead.*

3.2.2 Foothills, Plateaus, and Ridges Make Hills Hard to Climb

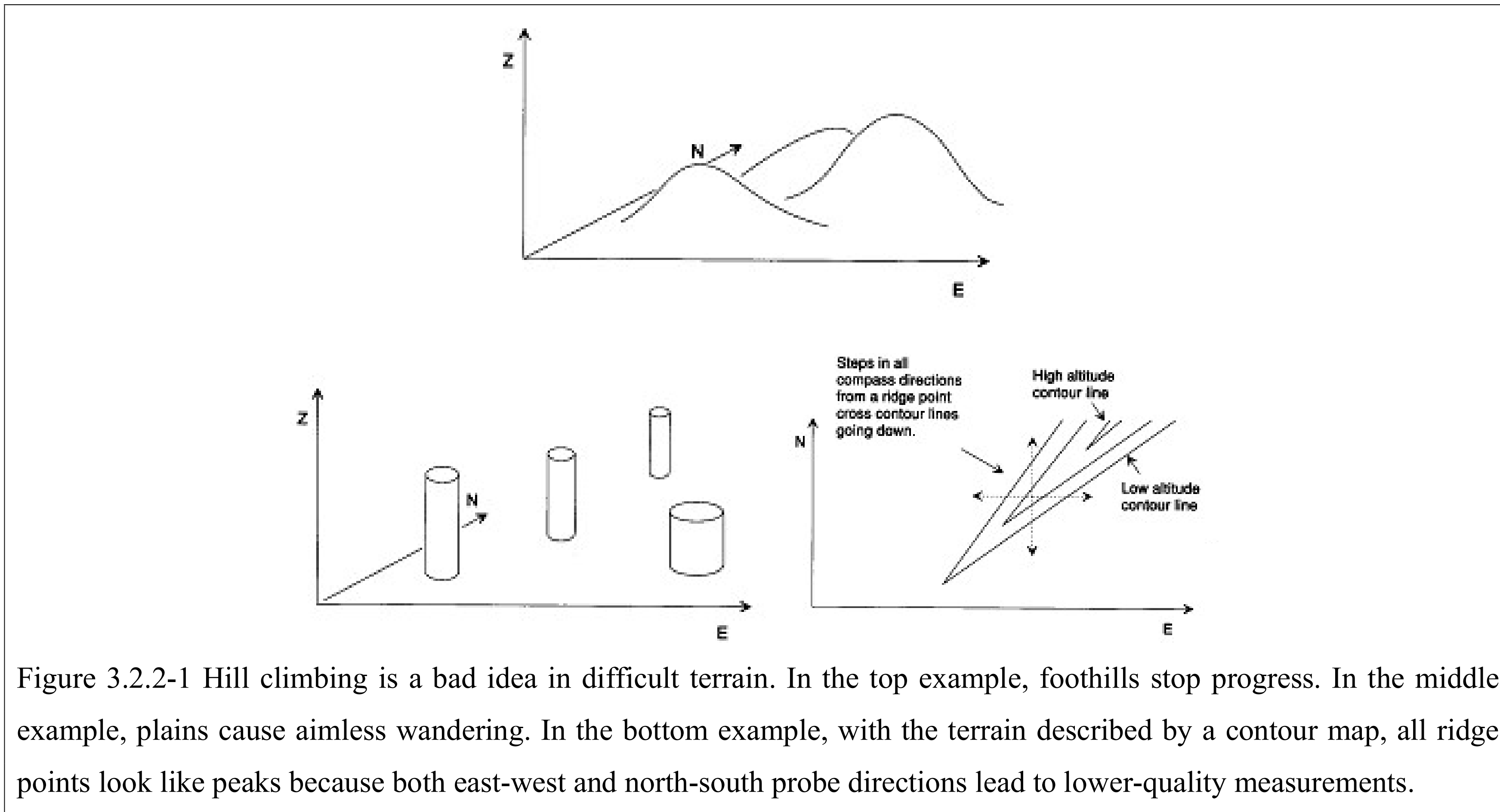

Figure 3.2.2-1 Hill climbing is a bad idea in difficult terrain. In the top example, foothills stop progress. In the middle example, plains cause aimless wandering. In the bottom example, with the terrain described by a contour map, all ridge points look like peaks because both east-west and north-south probe directions lead to lower-quality measurements.

Severe problems with parameter-oriented hill climbing:

- The **foothill** problem occurs whenever there are secondary peaks. The secondary peaks draw the hill-climbing procedure like magnets. An optimal point is found, but it is a local maximum, rather than a global maximum.
- The **plateau** problem comes up when there is a mostly flat area separating the peaks. In extreme cases, the peaks may look like telephone poles sticking up in a football field. For all but a small number of positions, all standard-step probes leave the quality measurement unchanged.
- The **ridge** problem is more subtle, and, consequently, is more frustrating. It is like a knife edge contour. A contour map shows that each standard step takes you down, even though you are not at any sort of local or global maximum. Increasing the number of directions used for the probing steps may help.

- Nondeterministic search is useful when a local maximum is detected. The reason for using this strategy is that a random number of steps, of random size, in random directions, may shield you from the magnet-like attraction of the local maximum long enough for you to escape.

3.2.3 Beam Search Expands Several Partial Paths and Purges the Rest

- Beam search is like breadth-first search in that it progresses level by level, however, beam search moves downward only through the best *w* nodes at each level; the other nodes are ignored.

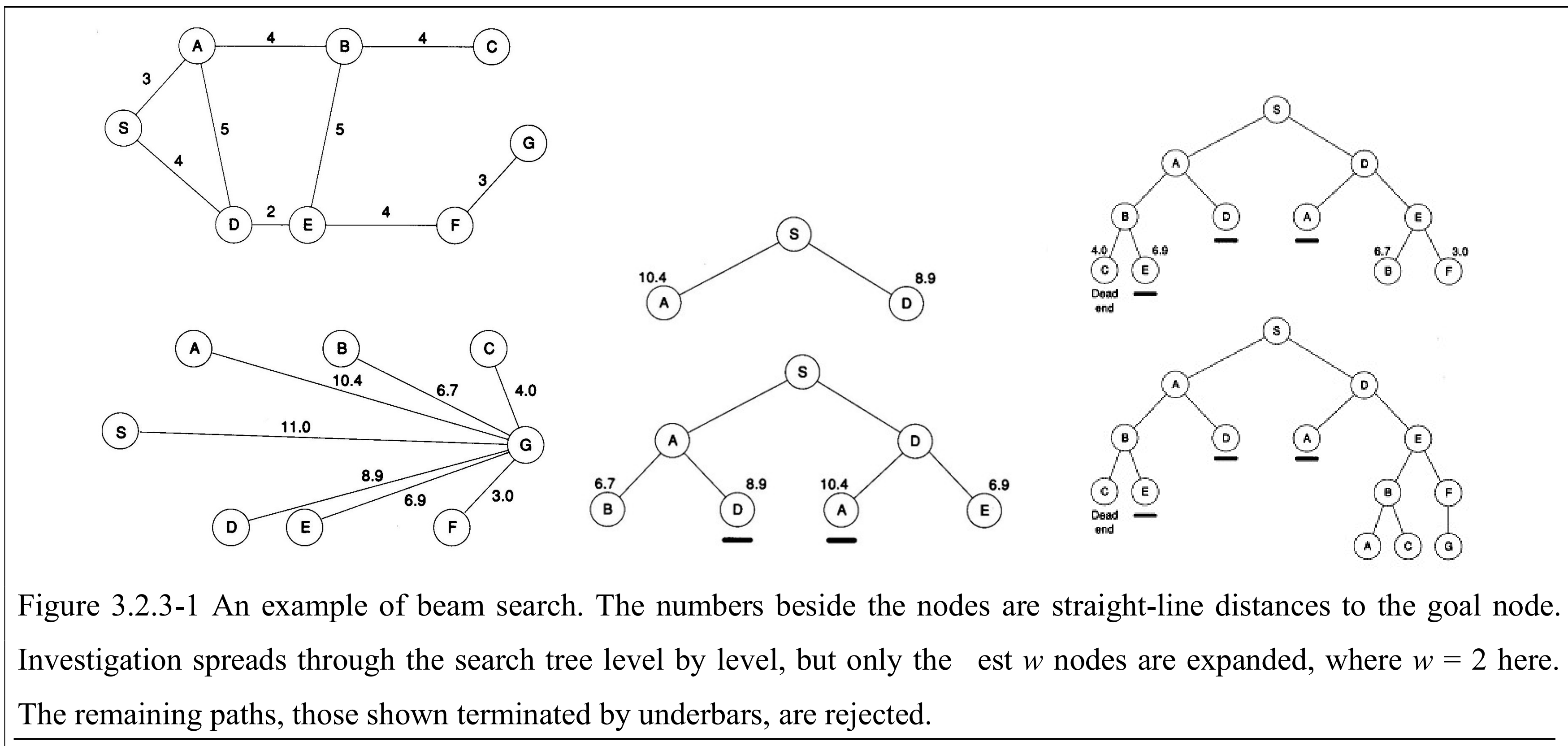

Figure 3.2.3-1 An example of beam search. The numbers beside the nodes are straight-line distances to the goal node. Investigation spreads through the search tree level by level, but only the est *w* nodes are expanded, where $w = 2$ here. The remaining paths, those shown terminated by underbars, are rejected.

3.2.4 Best-First Search Expands the Best Partial Path

- In best-first search, forward motion is from the best open node so far, no matter where that node is in the partially developed tree.

3.2.5 Search Alternatives Form a Procedure Family

- Depth-first search is good when unproductive partial paths are never too long.
- Breadth-first search is good when the branching factor is never too large.
- Nondeterministic search is good when it is not sure whether depth-first search or breadth-first search would be better.
- Hill climbing is good when there is a natural measure of distance from each place to the goal and a good path is likely to be among the partial paths that appear to be good at each choice point.
- Beam search is good when there is a natural measure of goal distance and a good path is likely to be among the partial paths that appear to be good at all levels.
- Best-first search is good when there is a natural measure of goal distance and a good partial path may look like a bad option before more promising partial paths are played out.

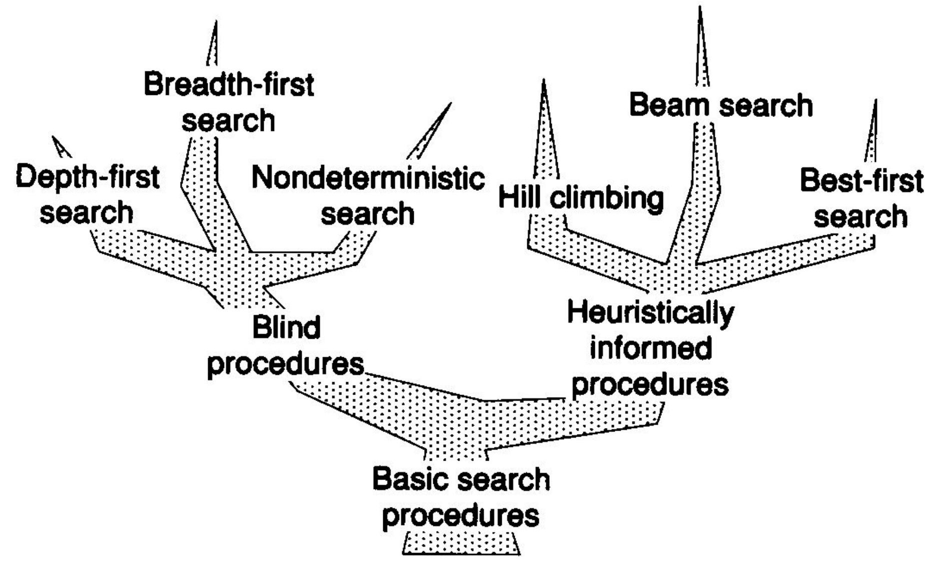

Figure 3.2.5-1 Part of the search family of procedures

4. Nets and Optimal Search

4.1 The Best Path

4.1.1 The British Museum Procedure Looks Everywhere

- In the British Museum procedure, the shortest path through a net is determined from all the possible paths.
- The British Museum procedure is not feasible in a very large tree.
- If the branching factor is b, the number of nodes at depth d must be b^d.

4.1.2 Branch-and-Bound Search Expands the Least-Cost Partial Path

- To find an optimal solution, if a solution is already discovered, there is no need to expand the branches with higher or equal the cost of the discovered solution.

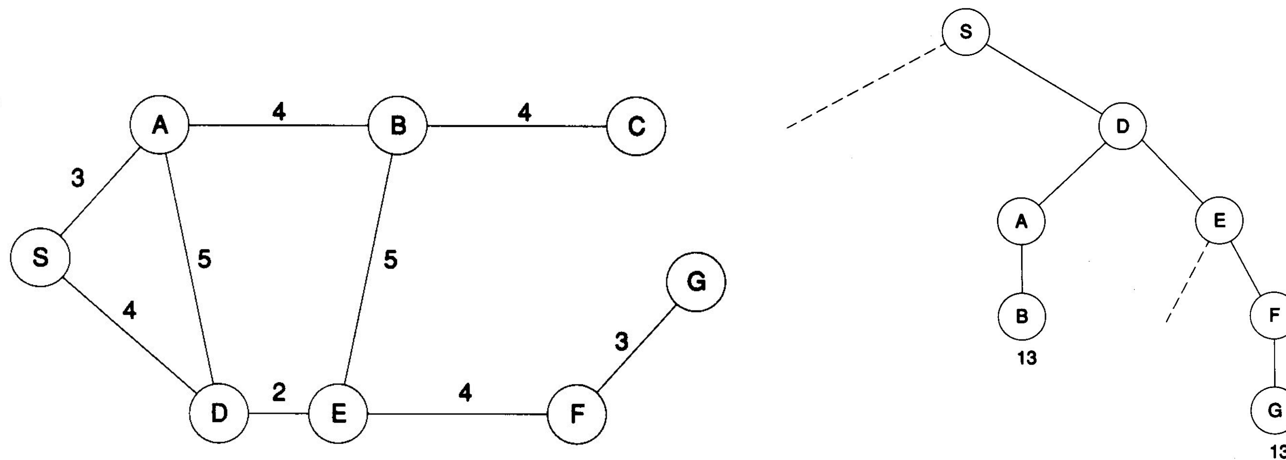

Figure 4.1.2-1 The length of the complete path from S to G, S-D-E-F-G is 13. Similarly, the length of the partial path S-D-A-B also is 13 and any additional movement along a branch will make it longer than 13. Accordingly, there is no need to pursue S-D-A-B any further because any complete path starting with S-D-A-B has to be longer than a complete path already known. Only the other paths emerging from S and from S-D-E have to be considered, as they may provide a shorter path.

To conduct a branch-and-bound search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
    - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - Reject all new paths with loops.
    - *Add the remaining new paths, if any, to the queue.*
    - *Sort the entire queue by path length with least-cost paths in front.*
- If the goal node is found, announce success; otherwise, announce failure.

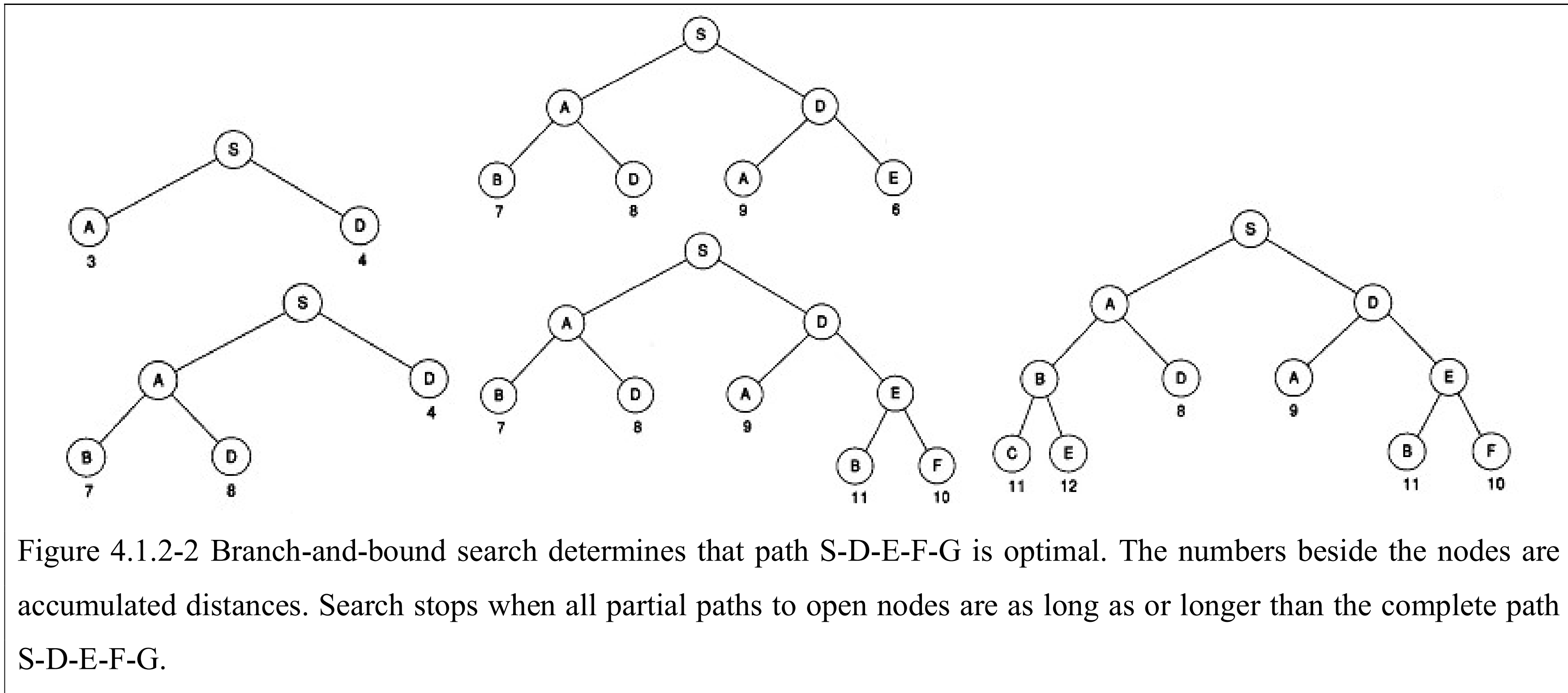

Figure 4.1.2-2 Branch-and-bound search determines that path S-D-E-F-G is optimal. The numbers beside the nodes are accumulated distances. Search stops when all partial paths to open nodes are as long as or longer than the complete path S-D-E-F-G.

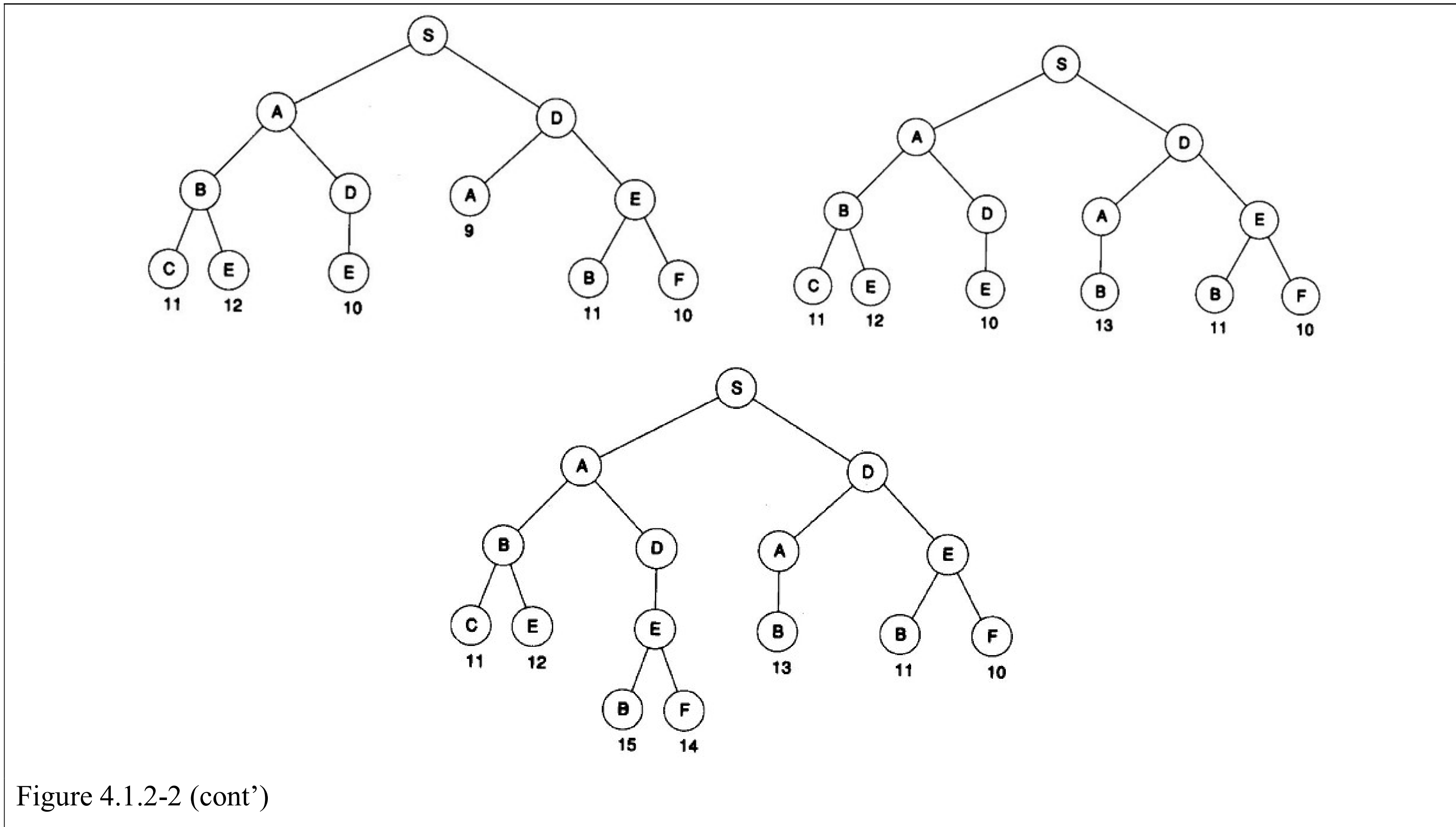

Figure 4.1.2-2 (cont’)

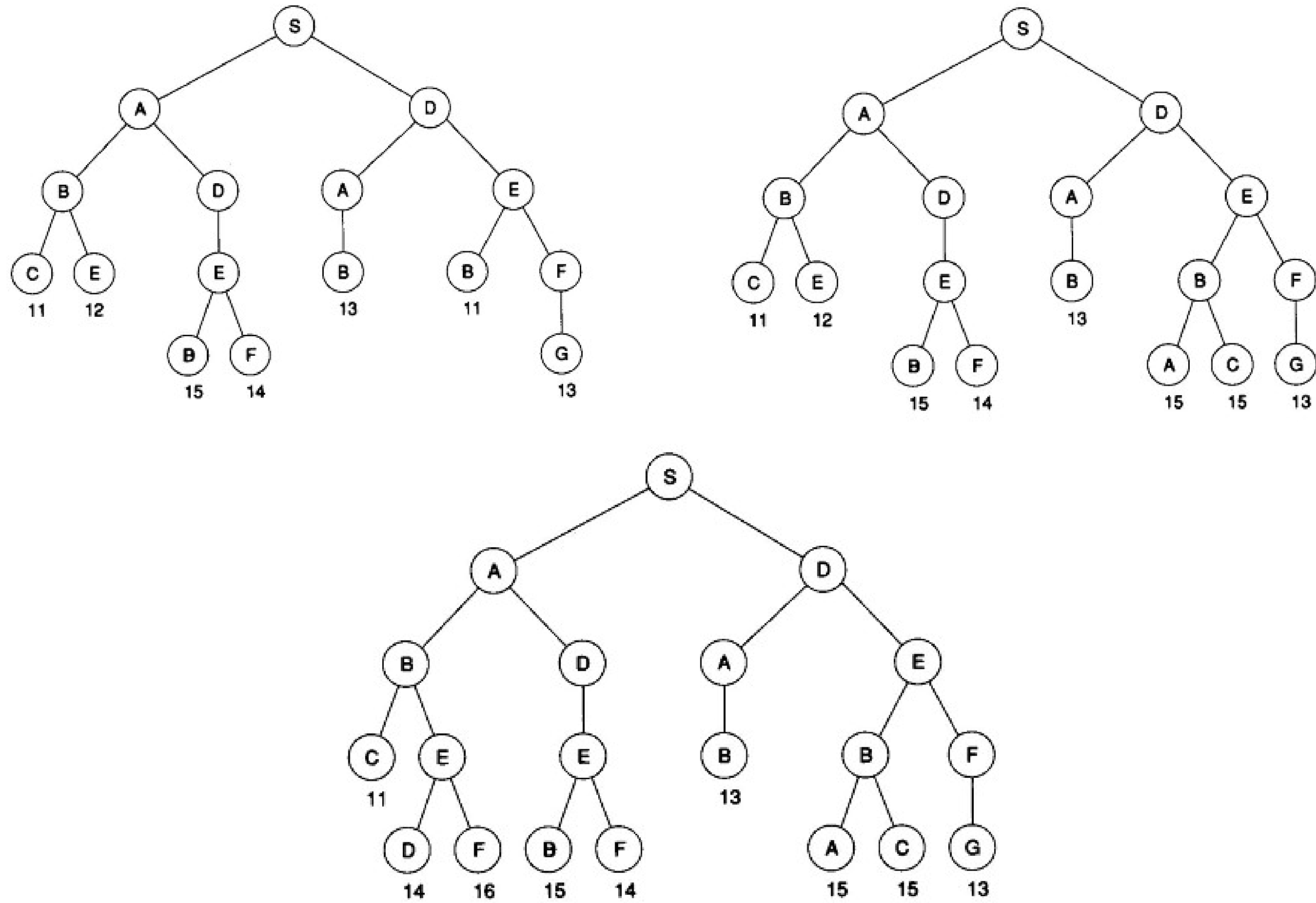

Figure 4.1.2-2 Cont’

4.1.3 Adding Underestimates Improves Efficiency

$$e \text{ (total path length)} = d \text{ (already traveled)} + e \text{ (distance remaining)} \quad (4.1.3\text{-}1)$$

d (already traveled): the known distance already traveled

e (distance remaining): an estimate of the distance remaining

$$u \text{ (total path length)} = d \text{ (already traveled)} + u \text{ (distance remaining)} \quad (4.1.3\text{-}2)$$

d (already traveled): the known distance already traveled

u (distance remaining): an underestimate of the distance remaining.

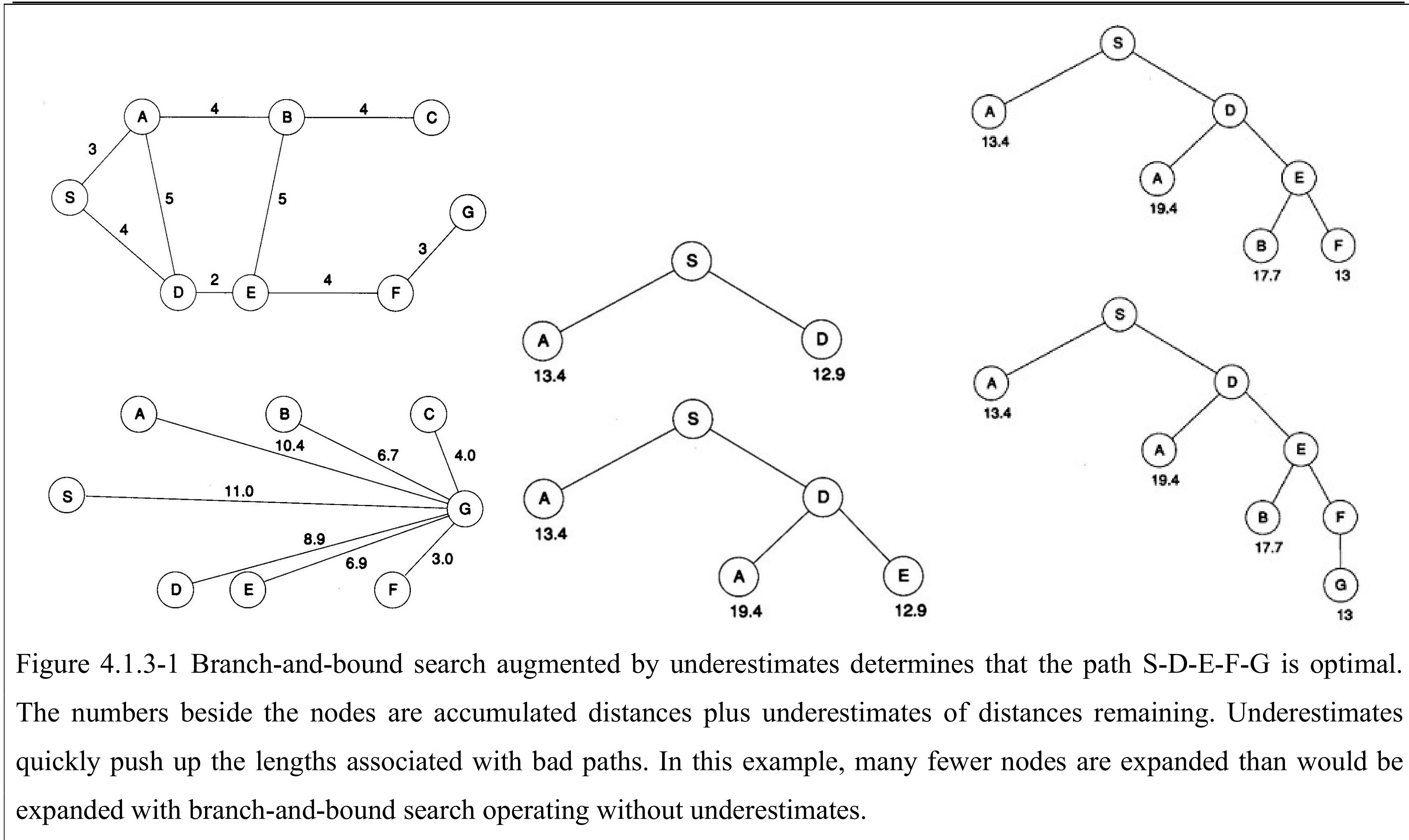

Figure 4.1.3-1 Branch-and-bound search augmented by underestimates determines that the path S-D-E-F-G is optimal. The numbers beside the nodes are accumulated distances plus underestimates of distances remaining. Underestimates quickly push up the lengths associated with bad paths. In this example, many fewer nodes are expanded than would be expanded with branch-and-bound search operating without underestimates.

To conduct a branch-and-bound search with a lower-bound estimate,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
    - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - Reject all new paths with loops.
    - Add the remaining new paths, if any, to the queue.
    - *Sort the entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front.*
- If the goal node is found, announce success; otherwise, announce failure.

- Ignoring estimates of remaining distance altogether can be viewed as the special case in which the underestimate used is uniformly zero.

4.2 Redundant Paths

4.2.1 Redundant Partial Paths Should Be Discarded

The dynamic-programming principle:

The best way through a particular, intermediate place is the best way to it from the starting place, followed by the best way from it to the goal. There is no need to look at any other paths to or from the intermediate place.

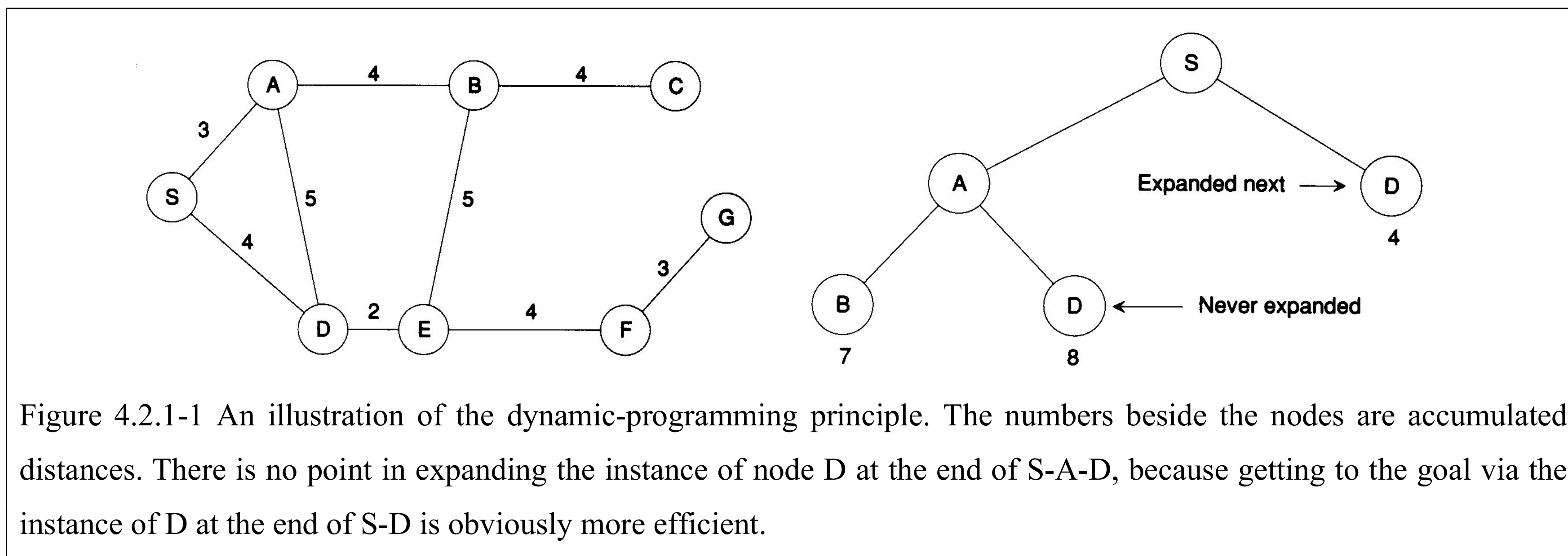

Figure 4.2.1-1 An illustration of the dynamic-programming principle. The numbers beside the nodes are accumulated distances. There is no point in expanding the instance of node D at the end of S-A-D, because getting to the goal via the instance of D at the end of S-D is obviously more efficient.

To conduct a branch-and-bound search with dynamic programming,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the remaining new paths, if any, to the queue.
  - *If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.*
  - Sort the entire queue by path length with least-cost paths in front.
  - If the goal node is found, announce success; otherwise, announce failure.

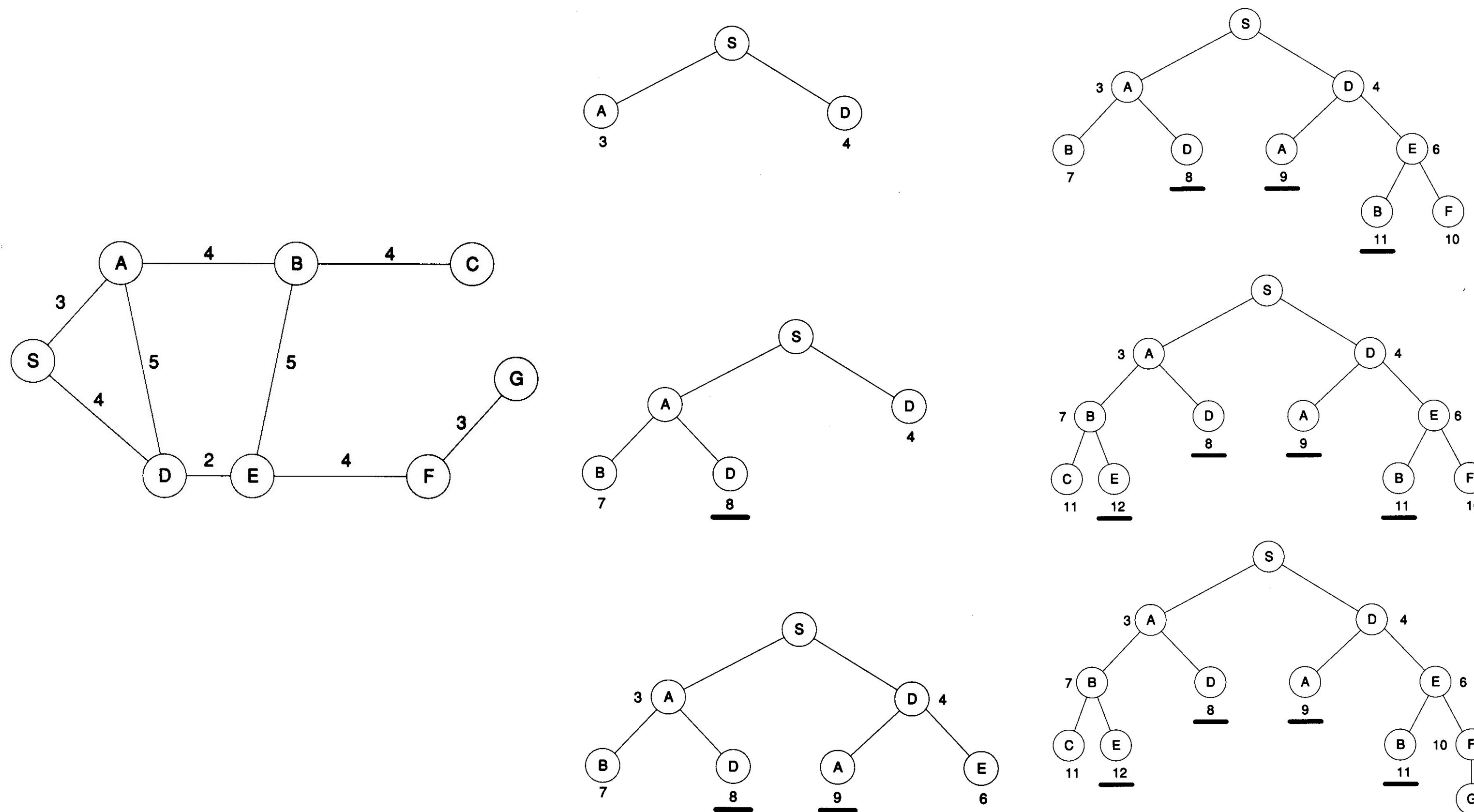

Figure 4.2.1-2 Branch-and-bound search, augmented by dynamic programming, determines that path S-D-E-F-G is optimal. The numbers beside the nodes are accumulated path distances. Many paths, those shown terminated with underbars, are found to be redundant. Thus, dynamic programming reduces the number of nodes expanded.

4.2.2 Underestimates and Dynamic Programming Improve Branch-and-Bound Search

- The A* procedure is branch-and-bound search, with an estimate of remaining distance, combined with the dynamic-programming principle.
- If the estimate of remaining distance is a lower-bound on the actual distance, then A* produces optimal solutions.

To conduct A* search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort the entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front.
- If the goal node is found, announce success; otherwise, announce failure.

4.2.3 Several Search Procedures Find the Optimal Path

- The British Museum procedure is good only when the search tree is small.
- Branch-and-bound search is good when the tree is big and bad paths turn distinctly bad quickly.
- Branch-and-bound search with a guess is good when there is a good lower-bound estimate of the distance remaining to the goal.
- Dynamic programming is good when many paths converge on the same place.
- The A* procedure is good when both branch-and-bound search with a guess and dynamic programming are good.

4.2.4 Robot Path Planning Illustrates Search

Example: Collision-avoidance problem faced by robots. Before a robot begins to move in a cluttered environment, it must compute a collision-free path between where it is and where it wants to be. This requirement holds for locomotion of the whole robot through a cluttered environment and for robot hand motion through a component-filled workspace.

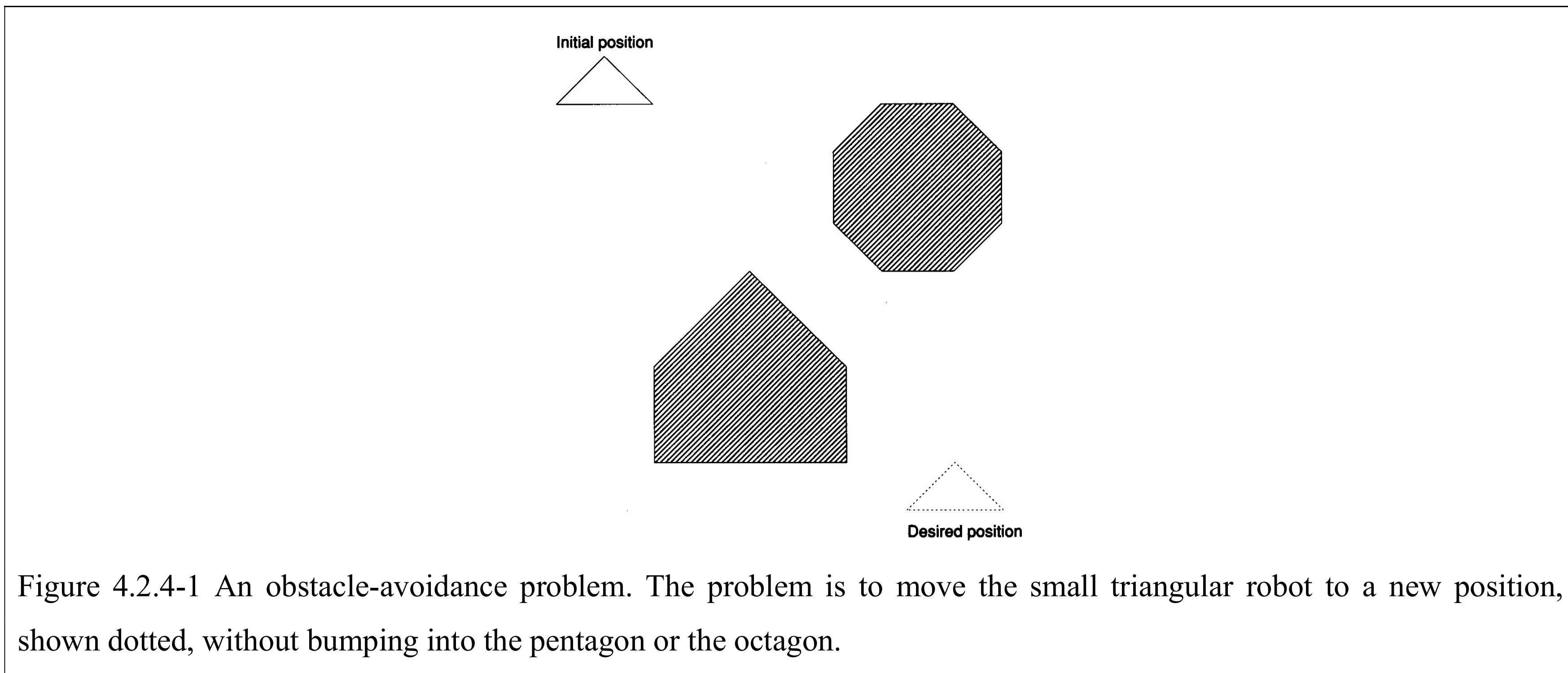

Figure 4.2.4-1 An obstacle-avoidance problem. The problem is to move the small triangular robot to a new position, shown dotted, without bumping into the pentagon or the octagon.

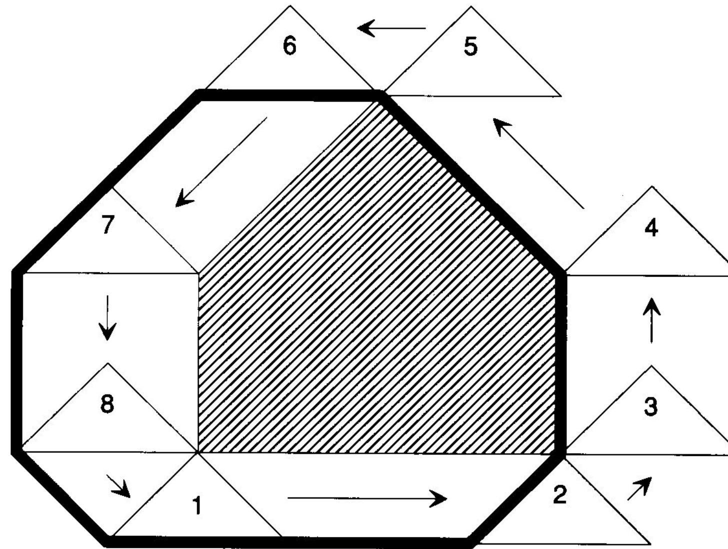

Figure 4.2.4-2 The configuration-space transformation. The heavy line shows the locus of the small triangle's lower-left corner as the small triangle is moved around the big one. Numbered positions are the starting points for each straight-line run. Keeping the lower-left corner away from the heavy line keeps the small triangle away from the pentagon.

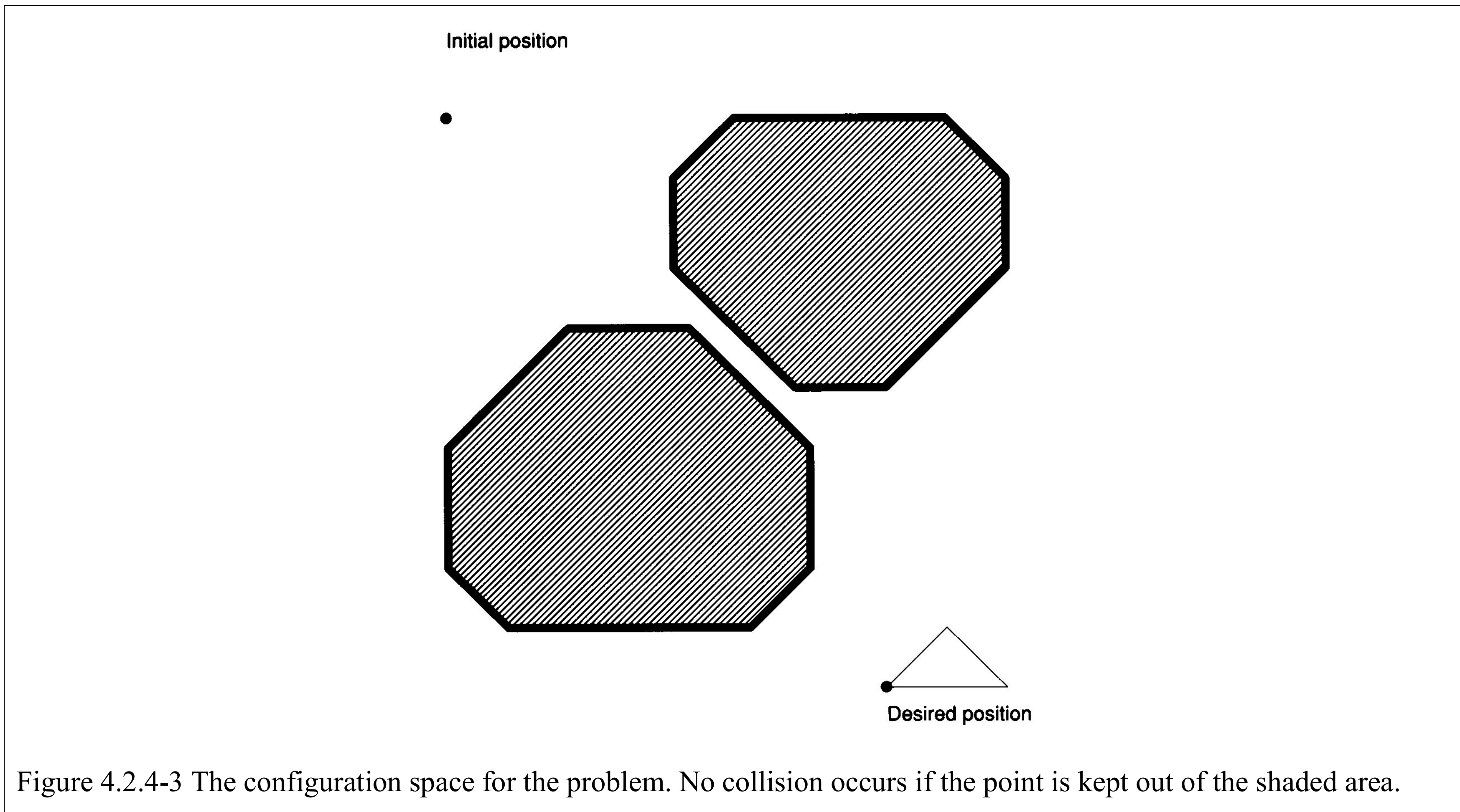

Figure 4.2.4-3 The configuration space for the problem. No collision occurs if the point is kept out of the shaded area.

- Because the links between nodes are placed only when there is an unobstructed line of sight between the nodes, the net is called a **visibility graph**.

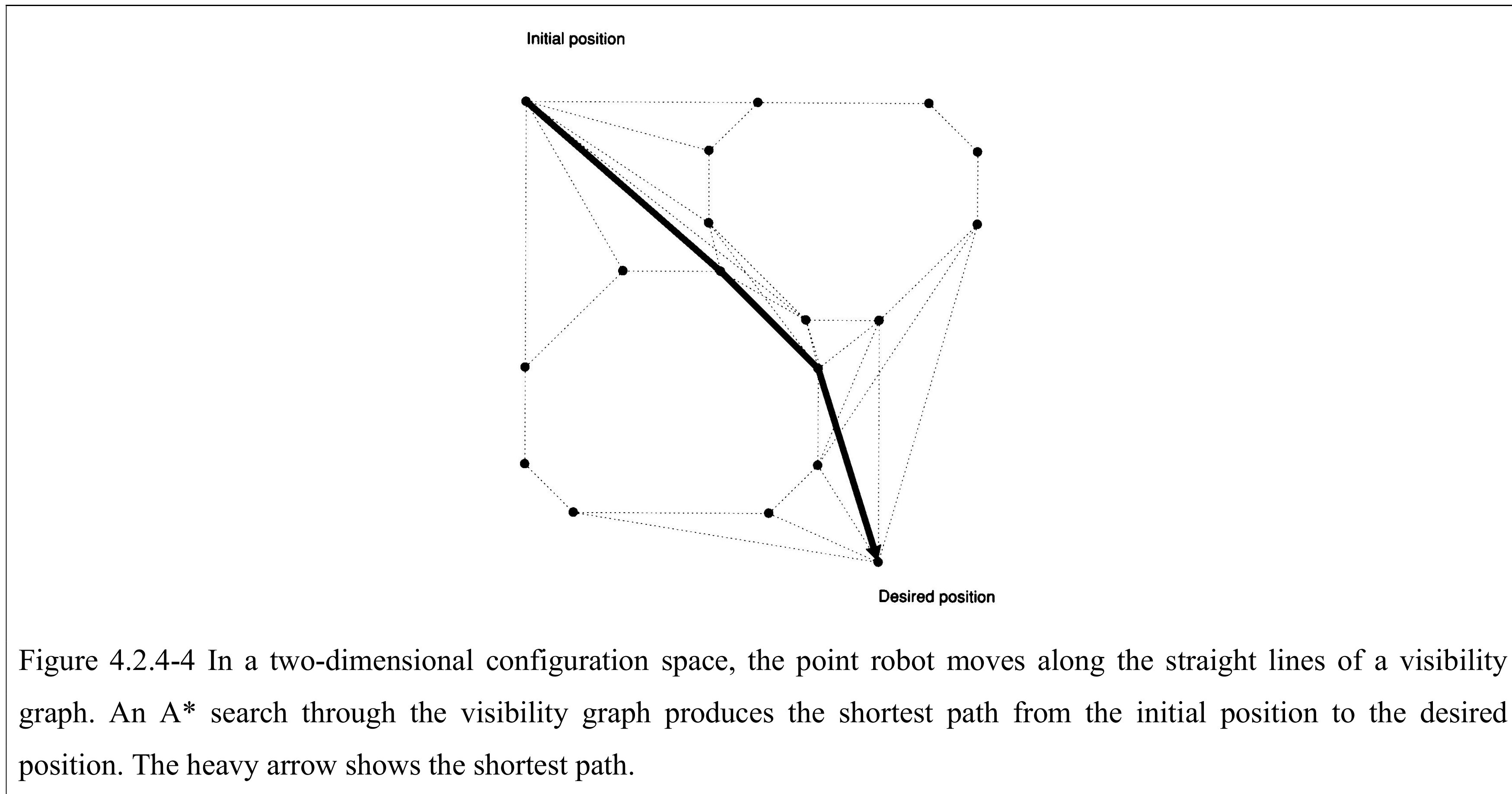

Figure 4.2.4-4 In a two-dimensional configuration space, the point robot moves along the straight lines of a visibility graph. An A* search through the visibility graph produces the shortest path from the initial position to the desired position. The heavy arrow shows the shortest path.

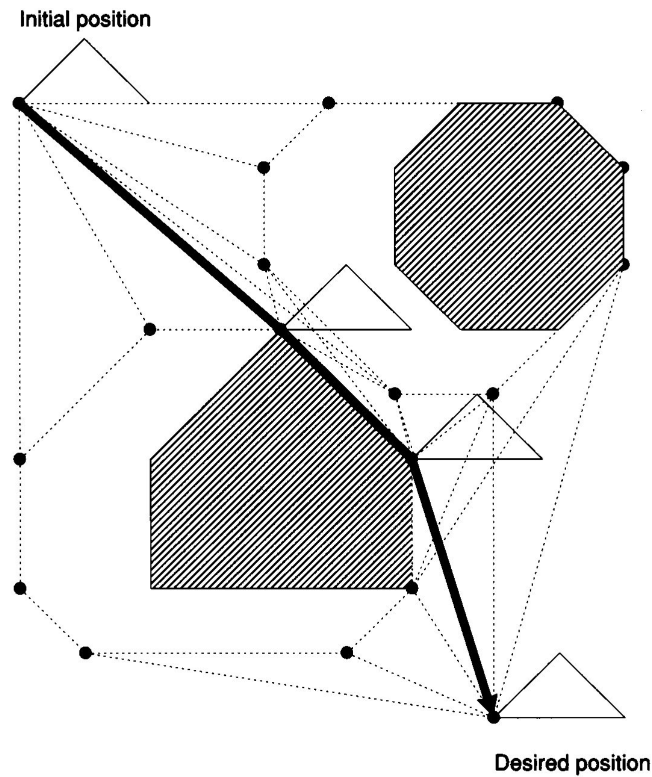

Figure 4.2.4-5 The robot's movement is dictated by the shortest path found in the visibility graph. The lower-left corner of the triangular robot, the one used to produce configuration-space obstacles, is moved along the shortest path. Note that the triangular robot never collides with either the pentagon or the octagon.

5. Trees and Adversarial Search

5.1 Algorithmic Methods

5.1.1 Nodes Represent Board Positions

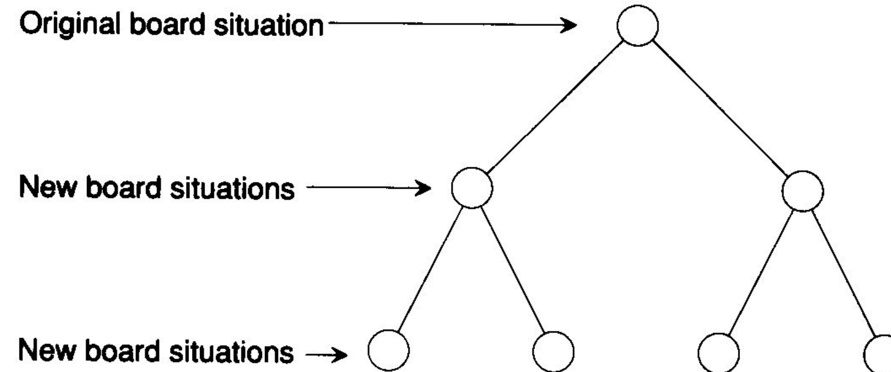

Figure 5.1.1-1 Games raise a new issue: competition. The nodes in a game tree represent board configurations, and the branches indicate how moves can connect them.

- A **game tree**, a special kind of semantic tree in which the nodes denote board configurations, and the branches indicate how one board configuration can be transformed into another by a single move.
- There is special twist in that the decisions are made by two adversaries who take turns making decisions.
- The ply of a game tree, p, is the number of levels of the tree, including the root level. If the depth of a tree is d, then $p = d + 1$.

5.1.2 Exhaustive Search Is Impossible

- Using something like the British Museum procedure to search game trees is definitely out.
- For chess, for example, if we take the effective branching factor to be 16 and the effective depth to be 100, then the number of branches in an exhaustive survey of chess possibilities would be on the order of 10^{120}, a ridiculously large number.

5.1.3 The Minimax Procedure Is a Lookahead Procedure

- All judgments about board situations are converted into a single, overall quality number
- Positive numbers indicate favor to one player, and negative numbers indicate favor to the other.
- The degree of favor increases with the absolute value of the number.
- The process of computing a number that reflects board quality is called **static evaluation**.
- The procedure that does the computation is called a **static evaluator**.
- The number computed is called the **static evaluation score**.
- The player hoping for positive numbers is called the maximizing player or the **maximizer**.
- The player hoping for negative numbers is called the minimizing player or **minimizer**.

A **game tree**, a kind of semantic tree, is a representation in which

- Nodes denote board configurations
- Branches denote moves

The procedure by which the scoring information passes up the game tree is called the **MINIMAX** procedure, because the score at each node is either the minimum or the maximum of the scores at the nodes immediately below:

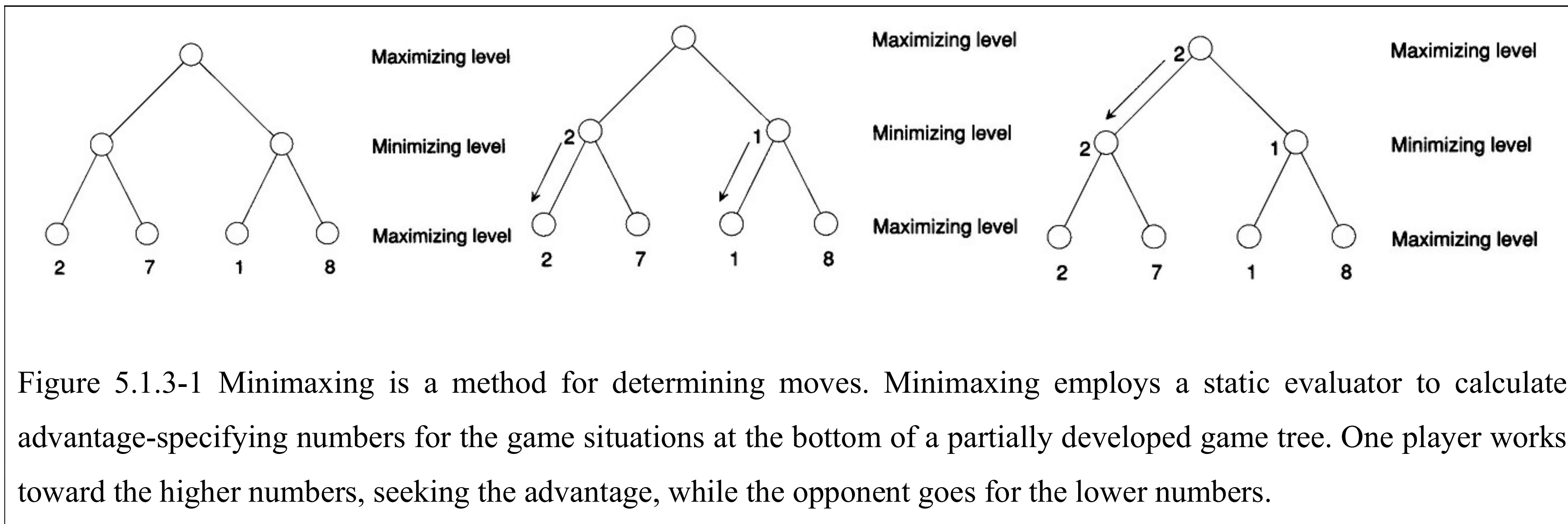

Figure 5.1.3-1 Minimaxing is a method for determining moves. Minimaxing employs a static evaluator to calculate advantage-specifying numbers for the game situations at the bottom of a partially developed game tree. One player works toward the higher numbers, seeking the advantage, while the opponent goes for the lower numbers.

To perform a minimax search using MINIMAX,

- If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
- Otherwise, if the level is a minimizing level, use MINIMAX on the children of the current position. Report the minimum of the results.
- Otherwise, the level is a maximizing level. Use MINIMAX on the children of the current position. Report the maximum of the results.

5.1.4 The Alpha-Beta Procedure Prunes Game Trees

- Alpha-beta procedure is somewhat like the branch-and-bound idea in that some paths are demonstrated to be bad even though not followed to the lookahead limit.

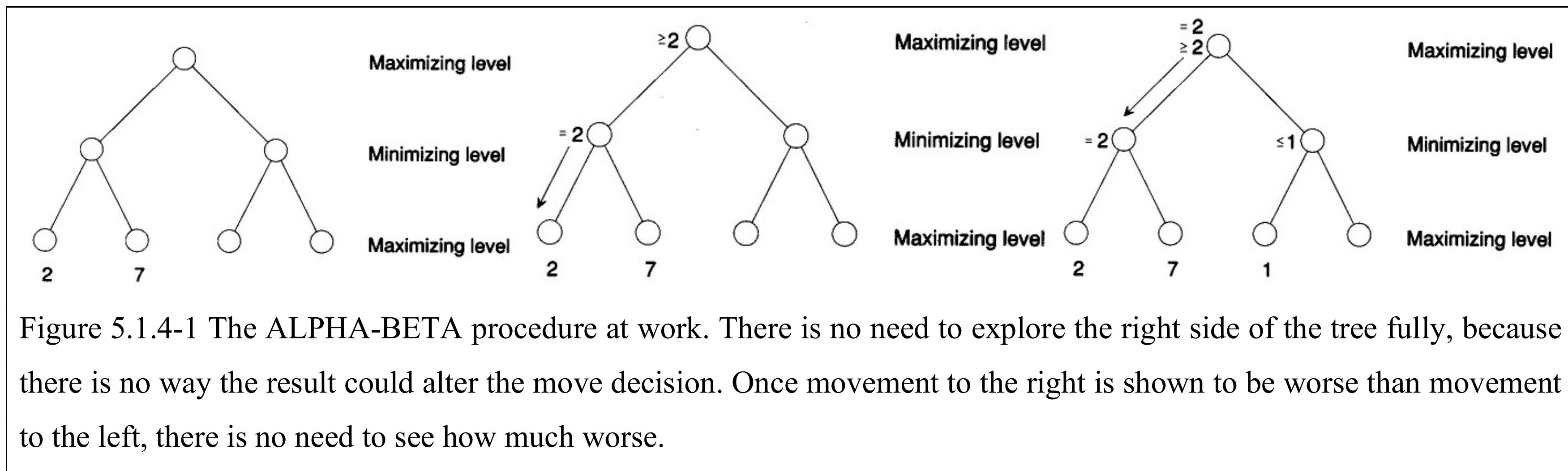

Figure 5.1.4-1 The ALPHA-BETA procedure at work. There is no need to explore the right side of the tree fully, because there is no way the result could alter the move decision. Once movement to the right is shown to be worse than movement to the left, there is no need to see how much worse.

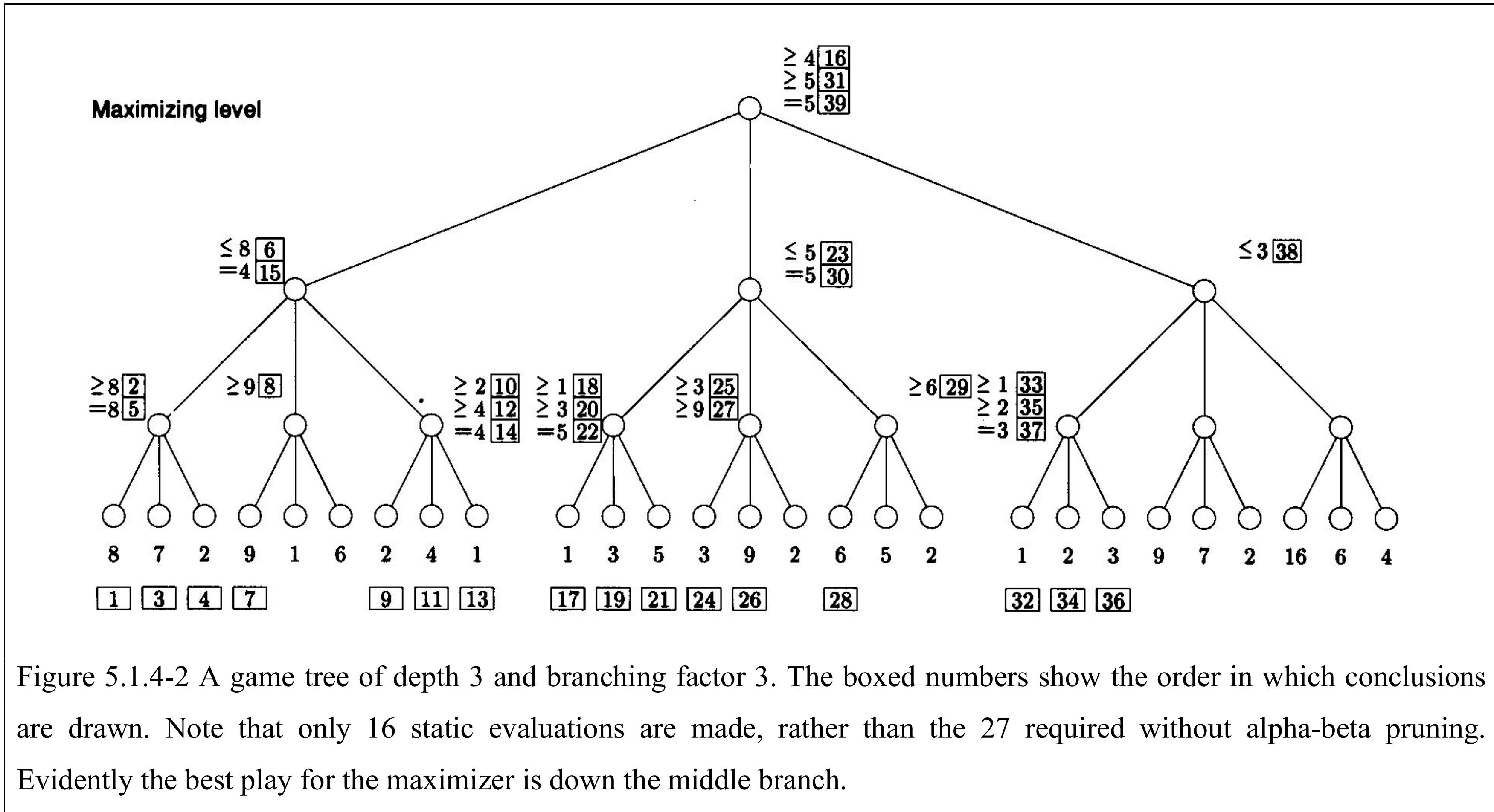

Figure 5.1.4-2 A game tree of depth 3 and branching factor 3. The boxed numbers show the order in which conclusions are drawn. Note that only 16 static evaluations are made, rather than the 27 required without alpha-beta pruning. Evidently the best play for the maximizer is down the middle branch.

To perform minimax search with the ALPHA-BETA procedure,

- If the level is the top level, let alpha be -∞ and let beta be +∞.
- If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
- If the level is a minimizing level,
    - Until all children are examined with ALPHA-BETA or until alpha is equal to or greater than beta,
        - Use the ALPHA-BETA procedure, with the current alpha and beta values, on a child; note the value reported.
        - Compare the value reported with the beta value; if the reported value is smaller, reset beta to the new value.
    - Report beta.
- Otherwise, the level is a maximizing level:
    - Until all children are examined with ALPHA-BETA or alpha is equal to or greater than beta,
        - Use the ALPHA-BETA procedure, with the current alpha and beta value, on a child; note the value reported.
        - Compare the value reported with the alpha value; if the reported value is larger, reset alpha to the new value.
    - Report alpha.

6.1.5 Alpha-Beta May Not Prune Many Branches from the Tree

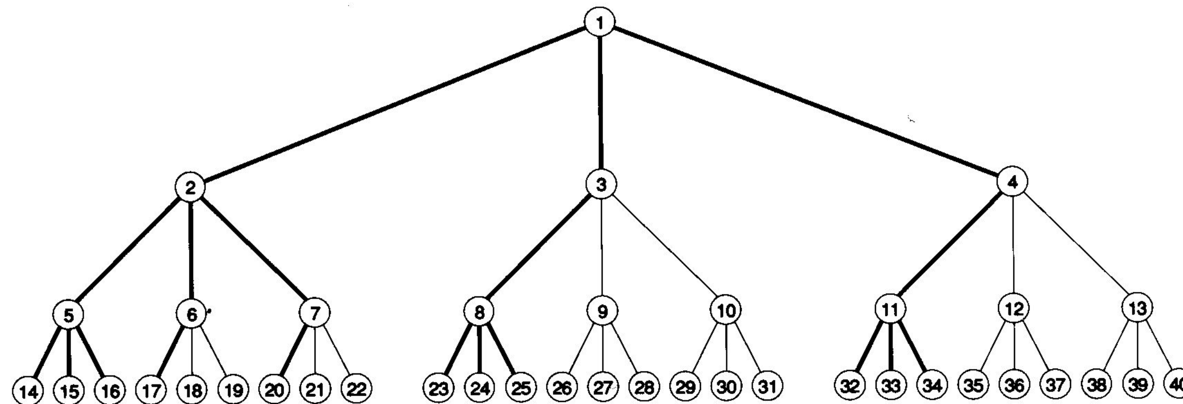

Figure 6.1.5-1 An ideal situation from the perspective of the ALPHA-BETA procedure. The ALPHA-BETA procedure cuts the exponent of exponential explosion in half, because not all the adversary's options need to be considered in verifying the left-branch choices. In a tree with depth 3 and branching factor 3, the ALPHA-BETA procedure can reduce the number of required static evaluations from 27 to 11.

The number of static evaluations, s, needed to discover the best move in an optimally arranged tree,

$$s = 2b^{d/2}-1 \qquad \text{for } d \text{ even;}$$

$$s = b^{(d+1)/2}+b^{(d-1)/2}-1 \qquad \text{for d odd.} \qquad (6.1.5\text{-}1)$$

b: the branching factor, d: the depth of the tree:

6.1.6 Progressive Deepening Keeps Computing Within Time Bounds

- In tournaments, players are required to make a certain number of moves within time limits.
- The way to wriggle out of this dilemma is to analyze each situation to depth 1, then to depth 2, then to depth 3, and so on until the amount of time set aside for the move is used up.
- The choice is determined by the analysis at one level less deep than the analysis in progress when time runs out.
- This method is called **progressive deepening**.
- The number of nodes requiring static evaluation at the bottom of a tree with depth d and effective branching factor b is b^d.
- The number of nodes in the rest of the tree,

$$b^0 + b^1 + \ldots + b^{d-1} = (b^d - 1)/(b-1) \quad (6.1.6\text{-}1)$$

- The ratio of the number of nodes in the bottom level to the number of nodes up to the bottom level,

$$b^d(b-1)/(b^d-1) \approx b-1 \quad (6.1.6\text{-}2)$$