Artificial Intelligence
Chapter 9
Heuristic Search

Biointelligence Lab
School of Computer Sci. & Eng.
Seoul National University

Outline

- Using Evaluation Functions
- A General Graph-Searching Algorithm
- Algorithm A*
- Iterative-Deepening A*
- Heuristic Functions and Search Efficiency

9.1 Using Evaluation Functions

- Best-first search (BFS) = Heuristic search
  - proceeds preferentially using heuristics
  - Basic idea
    - Heuristic evaluation function $\hat{f}$: based on information specific to the problem domain
    - Expand next that node, *n*, having the smallest value of $\hat{f}(n)$
    - Terminate when the node to be expanded next is a goal node
- Eight-puzzle
  - The number of tiles out of places: measure of the goodness of a state description

$$\hat{f}(n) = \text{number of tiles out of place (comapred with goal)}$$

9.1 Using Evaluation Functions (Cont'd)

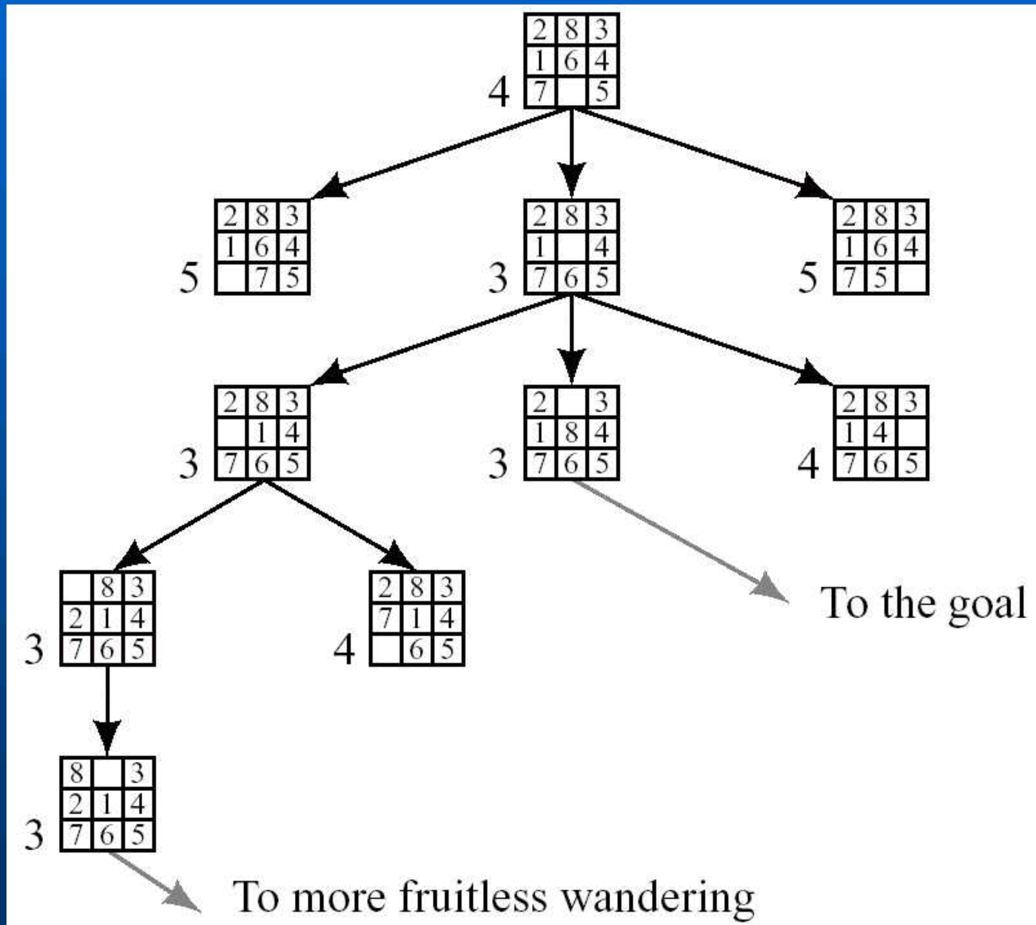

Figure 9.1 A Possible Result of a Heuristic Search Procedure

9.1 Using Evaluation Functions (cont'd)

- Preference to early path: add "depth factor" → Figure 9.2

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

$\hat{g}(n)$: estimate of the depth of n

: the length of the shortest path from the start to n

$\hat{h}(n)$: heuristic evaluation of node n

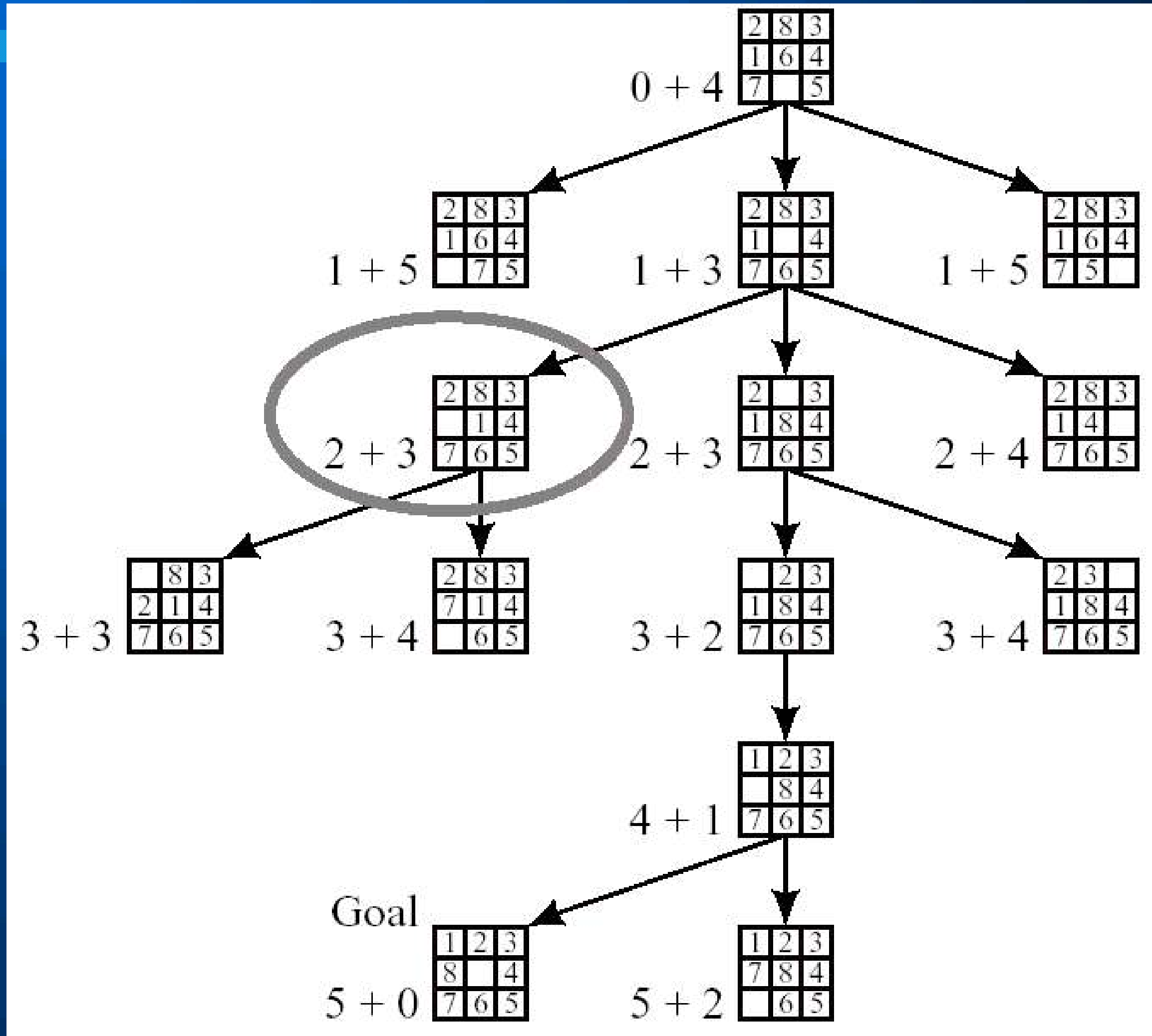

Figure 9.2 Heuristic Search Using $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$

9.1 Using Evaluation Functions (cont'd)

- Questions
  - How to settle on evaluation functions for guiding BFS?
  - What are some properties of BFS?
  - Does BFS always result in finding good paths to a goal node?

9.2 A General Graph-Searching Algorithm

- GRAPHSEARCH: general graph-searching algorithm
  1. Create a search tree, *Tr,* with the start node n_0 → put n_0 on ordered list ***OPEN***
  2. Create empty list ***CLOSED***
  3. If ***OPEN*** is empty, exit with failure
  4. Select the first node *n* on ***OPEN*** → remove it → put it on ***CLOSED***
  5. If *n* is a goal node, exit successfully: obtain solution by tracing a path backward along the arcs from *n* to n_0 in *Tr*
  6. Expand *n*, generating a set *M* of successors + install *M* as successors of *n* by creating arcs from *n* to each member of *M*
  7. Reorder the list ***OPEN***: by arbitrary scheme or heuristic merit
  8. Go to step 3

9.2 A General Graph-Searching Algorithm (Cont'd)

- Breadth-first search
  - New nodes are put at the end of OPEN (FIFO)
  - Nodes are not reordered
- Depth-first search
  - New nodes are put at the beginning of OPEN (LIFO)
- Best-first (heuristic) search
  - OPEN is reordered according to the heuristic merit of the nodes

9.2.1 Algorithm A*

- Algorithm A*
  - Reorders the nodes on OPEN according to increasing values of $\hat{f}$
- Some additional notation
  - $h(n)$: the *actual cost* of the minimal cost path between n and a goal node
  - $g(n)$: the cost of a minimal cost path from n_0 to n
  - $f(n) = g(n) + h(n)$: the cost of a minimal cost path from n_0 to a goal node over all paths via node n
  - $f(n_0) = h(n_0)$: the cost of a minimal cost path from n_0 to a goal node
  - $\hat{h}(n)$: estimate of $h(n)$
  - $\hat{g}(n)$: the cost of the lowest-cost path found by A* so far to n

9.2.1 Algorithm A* (Cont'd)

- Algorithm A*
  - If $\hat{h} = 0$: uniform-cost search
  - When the graph being searched is not a tree?
    - more than one sequence of actions that can lead to the same world state from the starting state
  - In 8-puzzle problem
    - Actions are reversible: implicit graph is not a tree
    - Ignore loops in creating 8-puzzle search tree: don't include the parent of a node among its successors
    - Step 6
      - Expand n, generating a set M of successors that are not already parents (ancestors) of n + install M as successors of n by creating arcs from n to each member of M

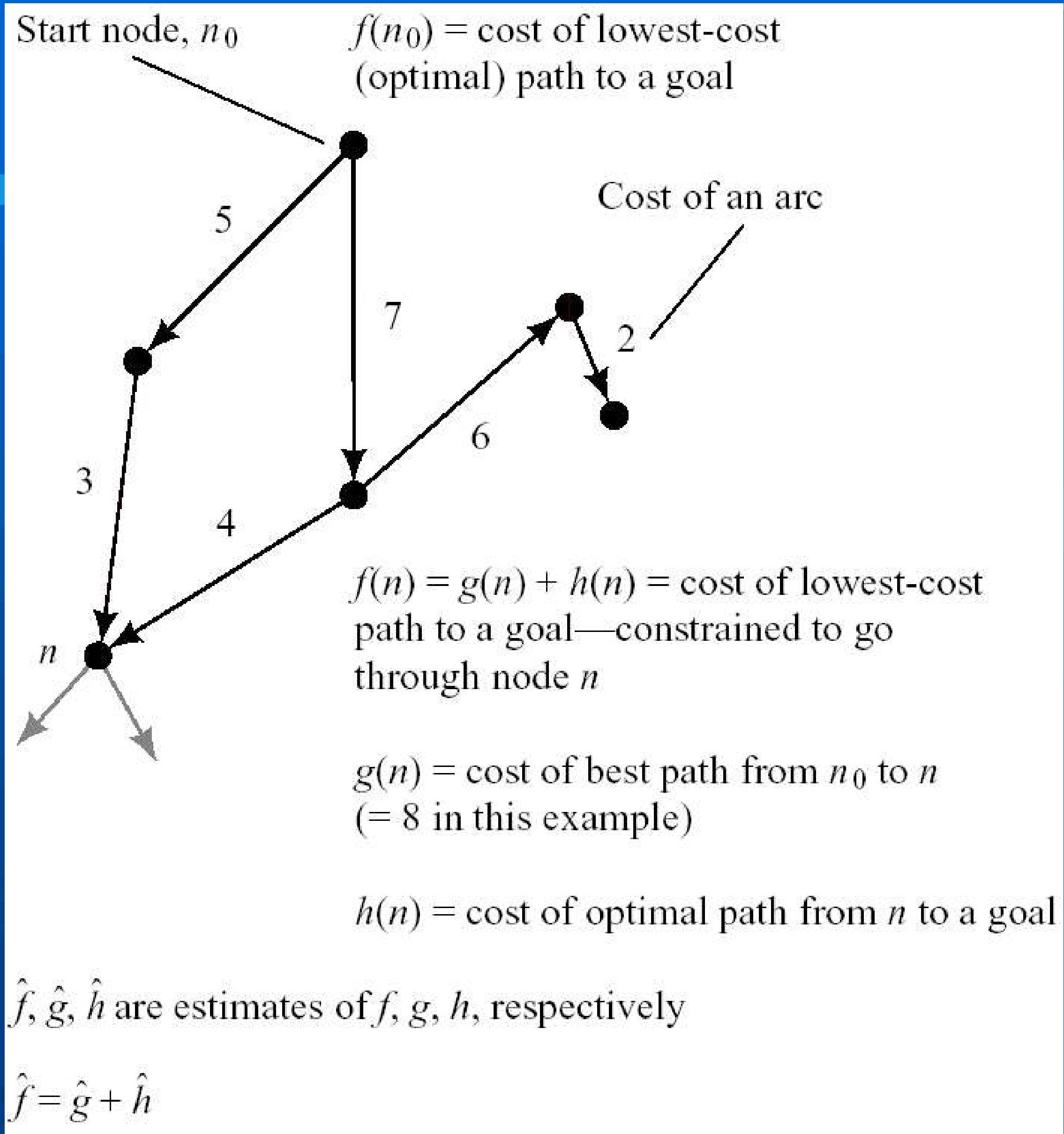

Figure 9.3 Heuristic Search Notation

9.2.1 Algorithm A* (Cont'd)

- Modification of A* to prevent duplicate search effort
  - *G*
    - search graph generated by A*
    - structure of nodes and arcs generated by A*
  - *Tr*
    - subgraph of *G*
    - tree of best (minimal cost) paths
  - Keep the search graph
    - subsequent search may find shorter paths
    - the paths use some of the arcs in the earlier search graph, not in the earlier search tree

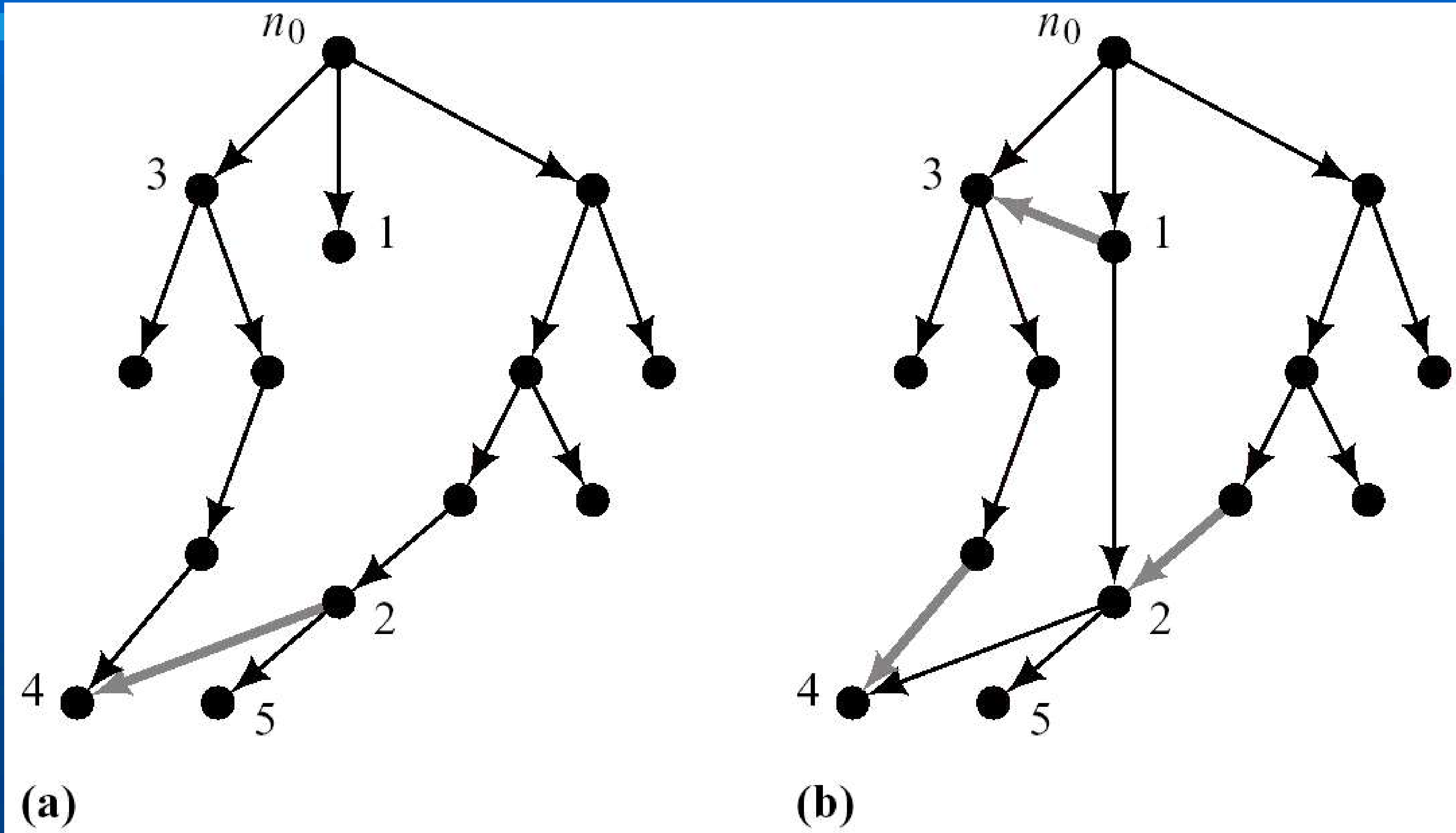

Figure 9.4 Search Graphs and Trees Produced by a Search Procedure

9.2.1 Algorithm A* (Cont'd)

- A* that maintains the search graph
  1. Create a search graph, G, consisting solely of the start node, n_0 → put n_0 on a list ***OPEN***
  2. Create a list ***CLOSED***: initially empty
  3. If ***OPEN*** is empty, exit with failure
  4. Select the first node on OPEN → remove it from ***OPEN*** → put it on ***CLOSED***: node n
  5. If n is a goal node, exit successfully: obtain solution by tracing a path along the pointers from n to n_0 in G
  6. Expand node n, generating the set, M, of its successors that are not already ancestors of n in G →install these members of M as successors of n in G

9.2.1 Algorithm A* (Cont'd)

7. Establish a pointer to n from each of those members of M that were not already in G → add these members of M to ***OPEN***
   → for each member, m, redirect its pointer to n if the best path to m found so far is through n → for each member of M already on ***CLOSED***, redirect the pointers of each of its descendants in G
8. Reorder the list OPEN in order of increasing $\hat{f}$ values
9. Go to step 3

- Redirecting pointers of descendants of nodes
  - Save subsequent search effort

9.2.2 Admissibility of A*

- Conditions that guarantee A* always finds minimal cost paths
  - Each node in the graph has a finite number of successors
  - All arcs in the graph have costs greater than some positive amount ε
  - For all nodes in the search graph, $\hat{h}(n) \le h(n)$
- Theorem 9.1
  - Under the conditions on graphs and on $\hat{h}$, and providing there is a path with finite cost from n_0 to a goal node, algorithm A* is guaranteed to terminate with a minimal-cost path to a goal

9.2.2 Admissibility of A* (Cont'd)

- Lemma 9.1
  - At every step before termination of A*, there is always a node, *n**, on ***OPEN*** with the following properties
    - *n** is on an optimal path to a goal
    - A* has found an optimal path to *n**
    - $\hat{f}(n^*) \le f(n_0)$
  - Proof : by mathematical induction
    - Base case
      - at the beginning of search, n_0 is on ***OPEN*** and on an optimal path to the goal
      - A* has found this path
      - $\hat{f}(n_0) \le f(n_0)$ because $\hat{f}(n_0) = \hat{h}(n_0) \le f(n_0)$
      - n_0 : *n** of the lemma at this stage

9.2.2 Admissibility of A^* (Cont'd)

- Induction step
  - assume the conclusions of the lemma at the time m nodes have been expanded ($m \geq 0$)
  - prove the conclusions true at the time m+1 nodes have been expanded
- Continuing the proof of the theorem
  - A* must terminate
  - A* terminates in an optimal path
- Admissible
  - Algorithm that is guaranteed to find an optimal path to the goal
  - With the 3 conditions of the theorem, A* is admissible
  - Any $\hat{h}$ function not overestimating h is admissible

9.2.2 Admissibility of A* (Cont'd)

- Theorem 9.2
  - If A_2^* is more informed than A_1^*, then at the termination of their searches on any graph having a path from n_0 to a goal node, every node expanded by A_2^* is also expanded by A_1^*
  - A_1^* expands at least as many nodes as does A_2^*
  - A_2^* is more efficient
- Figure 9.6
  - $\hat{h} \equiv 0$: uniform-cost search
  - $\hat{f}(n) = \hat{g}(n) = \mathrm{depth}(n)$: breadth-first search
  - uniform-cost/breadth-first search: admissible

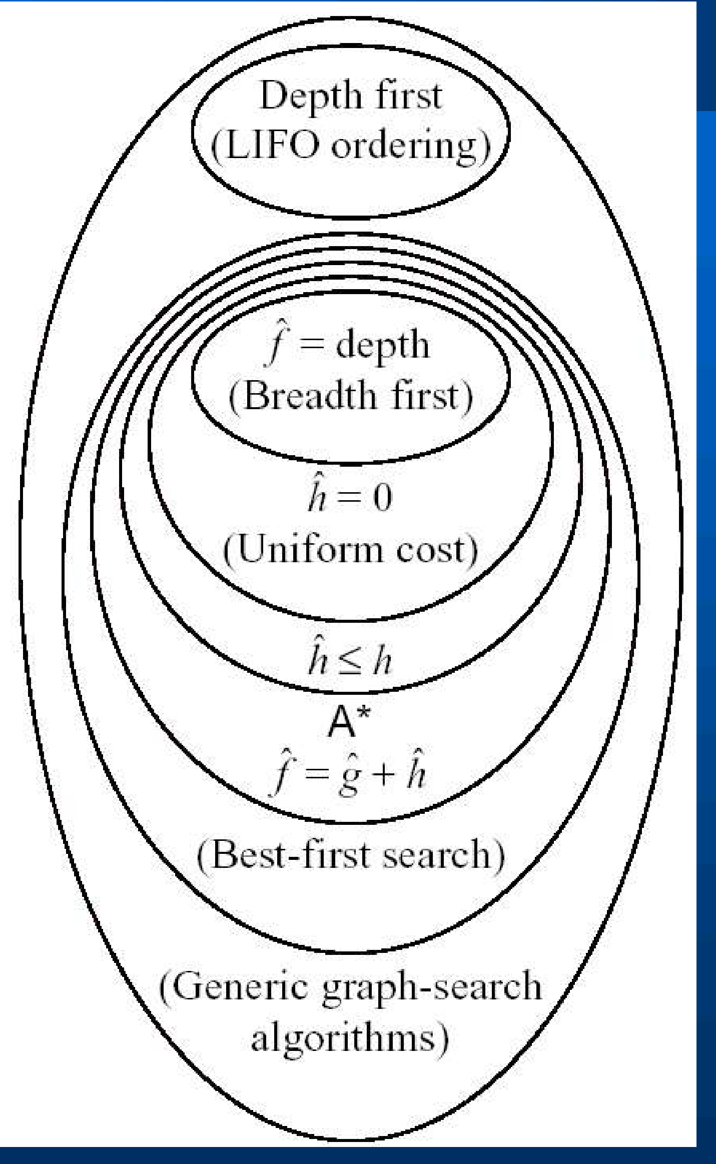

Figure 9.6 Relationships Among Search Algorithm

9.2.3 The Consistency (or Monotone) Condition

- Consistency condition
  - n_j is a successor of n_i
  - $\hat{h}(n_i) - \hat{h}(n_j) \le c(n_i, n_j)$
  - $c(n_i, n_j)$: cost of the arc from n_i to n_j
  - Rewriting

    $\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j)$ $\qquad$ $\hat{h}(n_j) \ge \hat{h}(n_i) - c(n_i, n_j)$
  - A type of triangle inequality

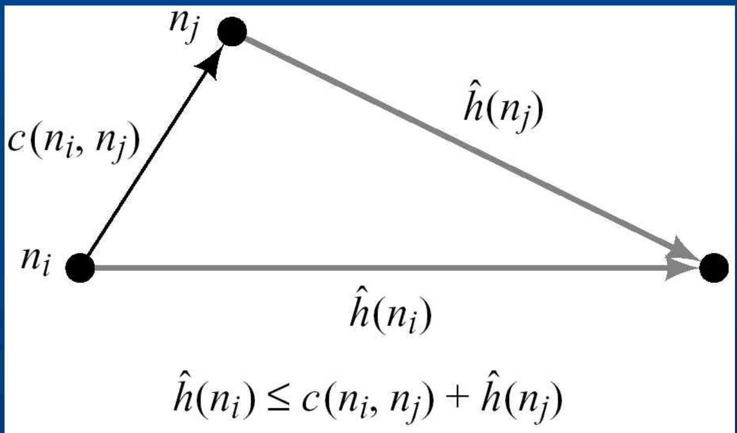

Figure 9.7
The Consistency Condition

9.2.3 The Consistency (or Monotone) Condition (Cont'd)

- Implies that $\hat{f}$ values of the nodes are monotonically nondecreasing as we move away from the start node

$$\hat{h}(n_j) \geq \hat{h}(n_i) - c(n_i, n_j)$$

$$\hat{h}(n_j) + \hat{g}(n_j) \geq \hat{h}(n_i) + \hat{g}(n_j) - c(n_i, n_j)$$

$$\hat{g}(n_j) = \hat{g}(n_i) + c(n_i, n_j)$$

$$\rightarrow \hat{f}(n_j) \geq \hat{f}(n_i)$$

- Consistency condition on $\hat{h}$ is often called the monotone condition on $\hat{f}$
- Theorem 9.3
  - If the consistency condition on $\hat{h}$ is satisfied, then when A* expands a node *n*, it has already found an optimal path to *n*

9.2.3 The Consistency (or Monotone) Condition (Cont'd)

- Argument for the admissibility of A* under the consistency condition
  - Monotonicity of $\hat{f}$: search expands outward along contours of increasing $\hat{f}$ values
  - The first goal node selected will be a goal node having a minimal $\hat{f}$
  - For any goal node, n_g, $\hat{f}(n_g) = \hat{g}(n_g)$
  - The first goal node selected will be one having minimal $\hat{g}$
  - Whenever a goal node, n_g, is selected for expansion, we have found an optimal path to that goal node ($\hat{g}(n_g) = g(n_g)$)
  - The first goal node selected will be one for which the algorithm has found an optimal path

9.2.4 Iterative-Deepening A*

- Breadth-first search
  - Exponentially growing memory requirements
- Iterative deepening search
  - Memory grows linearly with the depth of the goal
  - Parallel implementation of IDA*: further efficiencies gain
- IDA*
  - Cost cut off in the first search: $\hat{f}(n_0) = \hat{g}(n_0) + \hat{h}(n_0) = \hat{h}(n_0)$
  - Depth-first search with backtracking
  - If the search terminates at a goal node: minimal-cost path
  - Otherwise
    - increase the cut-off value and start another search
    - The lowest $\hat{f}$ values of the nodes visited (not expanded) in the previous search is used as the new cut-off value in the next search

9.2.5 Recursive Best-First Search

- RBFS (recursive best-first search)
  - uses slightly more memory than does IDA*
  - generates fewer nodes than does IDA*
- Backing up $\hat{f}$ value
  - When a node *n* is expanded, computes $\hat{f}$ values of successors of *n* and recomputes $\hat{f}$ values of *n* and all of *n*'s ancestors
- Process of backing up
  - Backed-up value, $\hat{f}(m)$, of node *m* with successors m_i

$$\hat{f}(m) = \min_{m_i} \hat{f}(m_i)$$

9.2.5 Recursive Best-First Search (Cont'd)

- Description
  - One of successors of node n has the smallest $\hat{f}$ over all ***OPEN*** nodes, it is expanded in turn, and so on.
  - When other ***OPEN*** node, n', (not a successor of n) has the lowest value of $\hat{f}$
    - backtracks to the lowest common ancestor, node k
    - k_n: successor of node k on the path to n
    - RBFS removes the subtree rooted at k_n, from ***OPEN***
    - k_n becomes an ***OPEN*** node with $\hat{f}$ value (its backed-up value)
    - Search continues below that ***OPEN*** node with the lowest value of $\hat{f}$

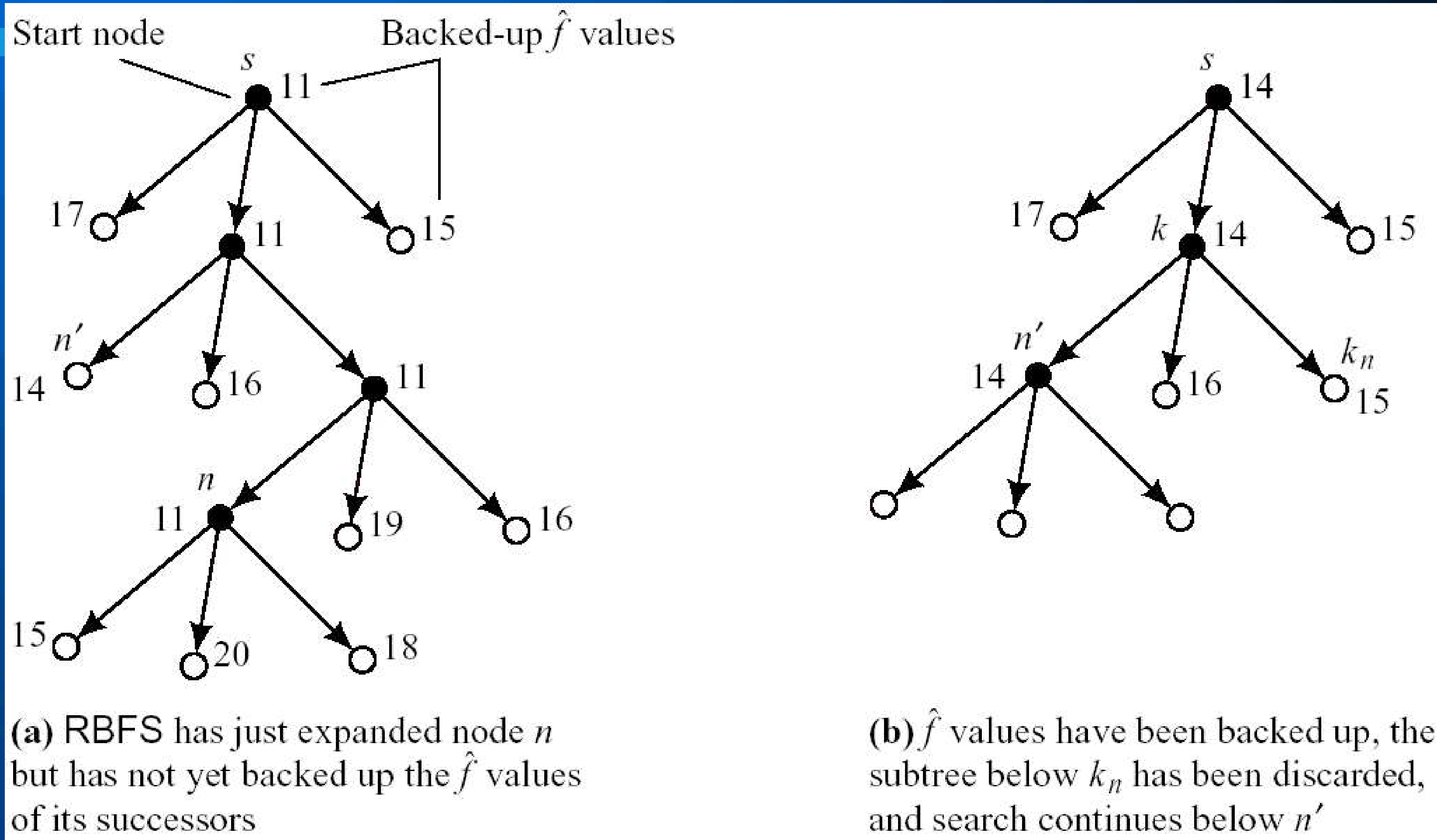

Figure 9.9 Recursive Best-First Search

9.3 Heuristic Functions and Search Efficiency

- Selection of heuristic function
  - Crucial for the efficiency of A*
  - $\hat{h} \equiv 0$
    - assures admissibility
    - Uniform-cost search → inefficient
  - $\hat{h}$ = the highest possible lower bound on h
    - maintains admissibility
    - expands the fewest nodes
- Using relaxed model
  - $\hat{h}$ functions are always admissible

9.3 Heuristic Functions and Search Efficiency (Cont'd)

- Selecting $\hat{h}$ function
  - must consider the amount of effort involved in calculating it
  - Less relaxed model: better heuristic function (difficult in calculating)
  - Trade off between the benefits gained by an accurate $\hat{h}$ and the cost of computing it
- Using $\hat{h}$ instead of the lower bound of h
  - increases efficiency at the expense of admissibility
  - $\hat{h}$: easier to compute
- Modifying the relative weights of $\hat{g}$ and $\hat{h}$ in the evaluation function $\hat{f} = \hat{g} + w\hat{h}$
  - Large values of w: overemphasize the heuristic component
  - Very small values of w: give the search a predominantly breadth-first character

9.3 Heuristic Functions and Search Efficiency (Cont'd)

- Simultaneous searches from both the start and a goal node
  - Breadth-first search
    - search frontiers meet between start and goal
    - guaranteed to find an optimal path
  - Heuristic search
    - Two search frontiers might not meet to produce an optimal path
- Effective branching factor
  - describes how sharply a search process is focused toward a goal
  - B = the number of successors of each node in the tree having the following properties
    - Nonleaf node has the same number (B) of successors
    - Leaf nodes are all of depth d
    - Total number of nodes is N

$$B + B^2 + \ldots + B^d = N$$

$$\frac{(B^d - 1)B}{(B-1)} = N$$

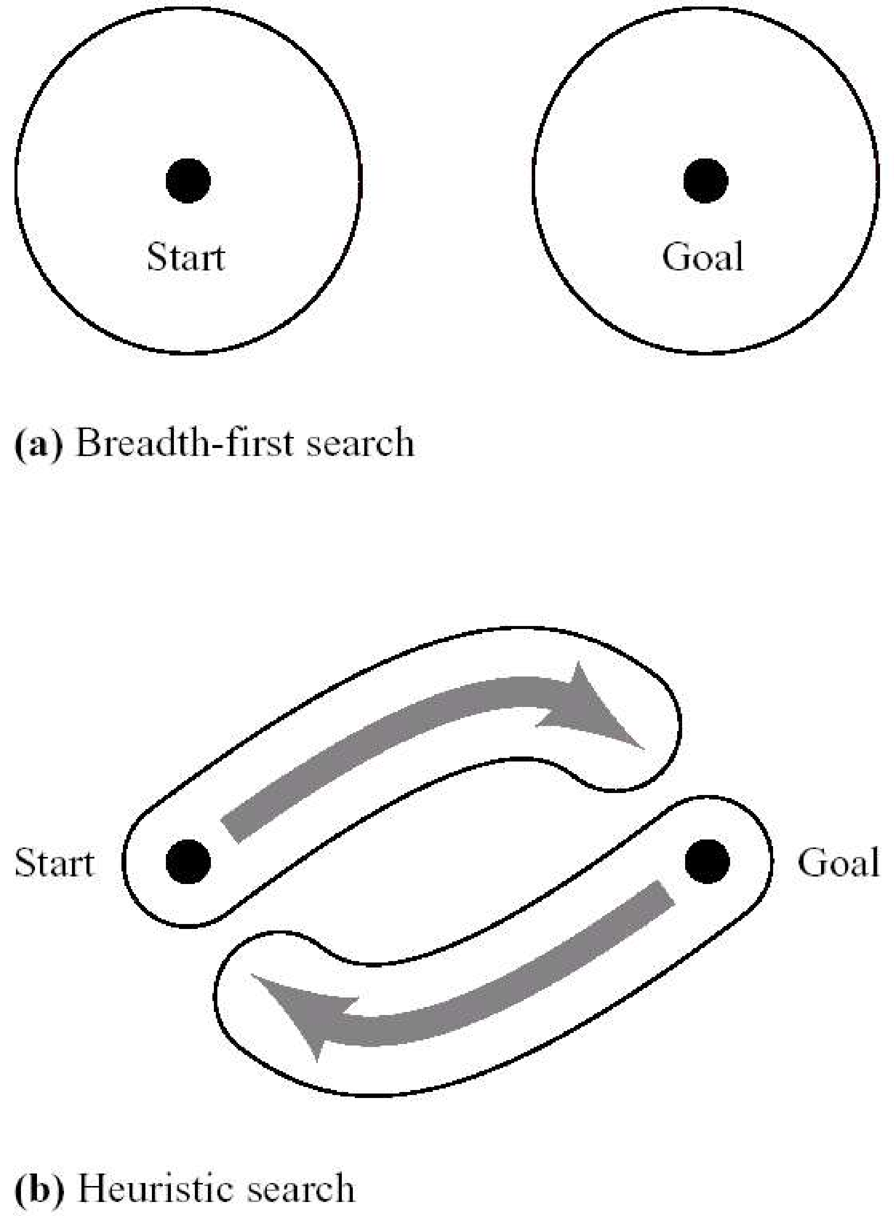

Figure 9.10 Bidirectional Searches

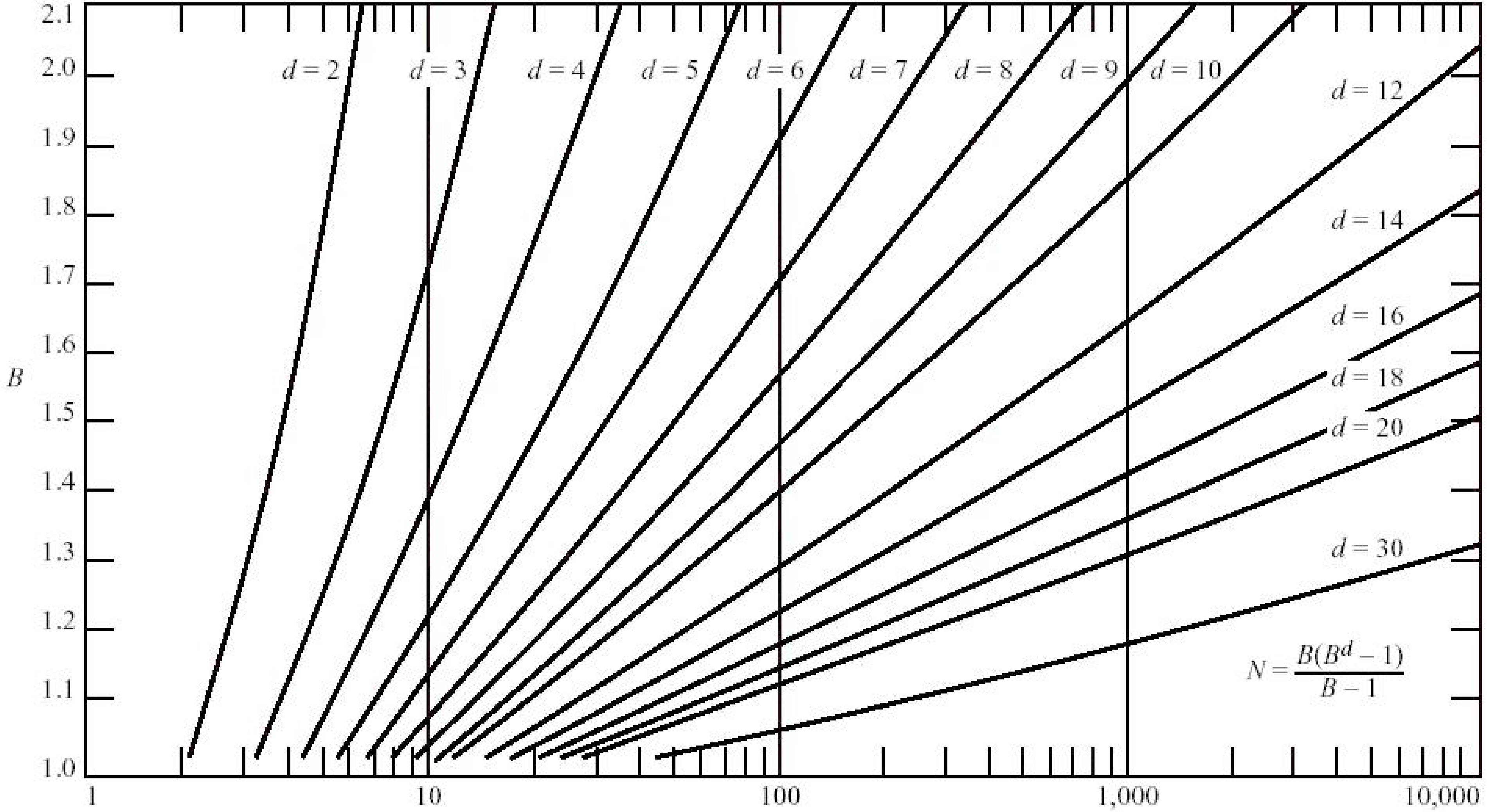

Figure 9.11 *B* Versus *N* for Various Values of *d*

9.3 Heuristic Functions and Search Efficiency (Cont'd)

- 3 important factors influencing the efficiency of algorithm A*
  - The cost (or length) of the path found
  - The number of nodes expanded in finding the path
  - The computational effort required to compute $\hat{h}$
- Time complexity: $O(n)$
  - Breadth-first search: $O(B^d)$
  - Uniform-cost search ($\hat{h} \equiv 0$): $O(B^{C/c})$
    - C: cost of an optimal solution
    - c: cost of the least costly arc

9.4 Additional Readings and Discussion