CS4102: Backtracking, Exhaustive Search

- Read: Section 4.5
  - And slides here
  - You won't be responsible for the Hamilton cycle code in the book
- In class:
  - Look at these slides
  - Work in groups of at most 4 to do the 3 in-class exercises
  - Turn in you work by the end of class

Graph Search vs. Search in General

- DFS and BFS
  - A graph is given as input
  - We traverse nodes (that exist in the graph)… following edges that exist in a graph
- A more general form: State-space search
  - Each node represents one state of the problem
  - Adjacent nodes are generated dynamically
  - They're legal states reachable from the current state
  - The algorithm generates one or more states based on the current one
  - Chooses which state to search next (possibly remembering other choices)
  - Backtrack when stuck

State-space Search Applied

- Many games and puzzles
  - n-queens problem
  - tic-tac-toe
  - chess
- Many other problems in CS
  - Problem 4.13: subset-sum problem
  - Problem 4.14: Find all m-colorings of a graph
  - These may not be efficient solutions!
    - Exhaustively try all possibilities
- Example later in these slides:
  - Hamilton paths and cycles

More on state-space search later…

Exhaustive Search

- Exhaustive search for graphs is just like DFS with one teeny-tiny change

Remember? Recursive DFS visit

```
def dfs_recurs0(graph, curnode, visited):
    visited[curnode] = True
    alist = graph.get_adjlist(curnode)
    for v in alist:
        if v not in visited
            dfs_recurs0(graph, v, visited)
    # about to back up from curnode….
    return
```

- Let’s change it slightly!

Remember? Recursive DFS visit

```
def exh_srch_recurs(graph, curnode, visited):
    visited[curnode] = True
    alist = graph.get_adjlist(curnode)
    for v in alist:
        if v not in visited
            exh_srch_recurs(graph, v, visited)
    # about to back up from curnode…
    visted[curnode] = False
    return
```

- When done with adj. nodes and about to back up, “forget” you’ve been there
    - Using colors? Set it to “white”

Remember? Recursive DFS visit

```
dfs_recurs(adj,start) {
   // reached node “start”; do something?
   visit[start] = true
   trav = adj[start]
   while (trav != null) {
      v = trav.ver
      if (!visit[v])
            dfs_recurs(adj,v)
      trav = trav.next
   }
  // about to leave “start”; do something?
}
```

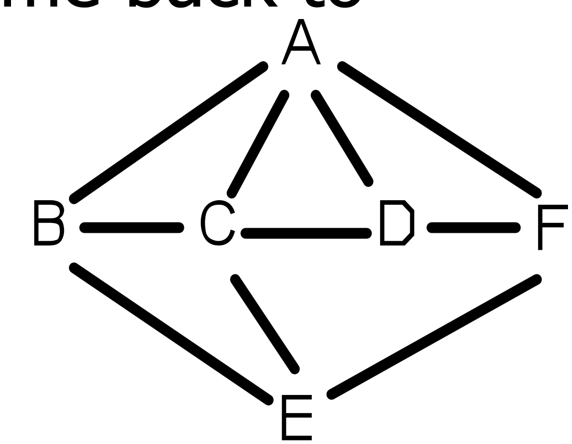
- Let’s change it slightly!

Recursive Exhaustive Search visit

```
exh_search_recurs(adj,start) {
   // reached node “start”; do something?
   visit[start] = true
   trav = adj[start]
   while (trav != null) {
      v = trav.ver
      if (!visit[v])
            exh_search_recurs(adj,v)
      trav = trav.next
   }
  // about to leave “start”; “un-mark” it
  visit[start] = false
}
```

In-class Exercise 1

- Trace exhaustive search on this graph
  - Start at A
- Draw the exhaustive search tree
  - Visit nodes in alphabetic order when there's a choice
  - Note: after you back up from a node, you can visit it again if you come back to it from another path!
  - Your tree will have more than n nodes in it



In-class Exercise 2

- Discuss these questions with your group:
  - What do the set of paths from A to each leaf represent?
  - From the tree, can you identify Hamilton paths?
    - I.e. a simple path that visits all nodes
  - From the tree, can you identify Hamilton cycles?
    - A Hamilton Path that also connects back to start node
- Write down:
  - Describe clearly how you could modify the DFS code to recognize Hamilton paths and Hamilton cycles
  - You can modify the pseudo-code or give me a clear description in words

Summary of What to Turn In

- Exercise 1:
  - A drawing of the exhaustive search tree for the given graph
- Exercise 2:
  - How to modify **exh_search_recurs()** to find Hamilton paths and cycles

- **Put the names of all group members on the paper and turn it in**

N-Queens Problem

- See the textbook for the explanation
  - Especially Figure 4.5.2 on page 196
- Note:
  - No input graph! Initial state is an empty board
  - Generate new state by placing next queen in next acceptable legal position
  - When impossible to place the next queen, remove it and backtrack to previous state

Comparison to DFS

- How is this like DFS?
  - Follow one path as far as you can.
  - Backtrack as little as possible when stuck
- How not like DFS?
  - No fixed set of edges or nodes to limit how much work you do
  - Less clear what to measure in terms of amount of work.
- Possible measures of work
  - Number of states generated (nodes in the graph)
  - Number of attempts to place a queen (cumulative # of attempts listed by nodes in the graph on p. 196)

In-class Exercise 1

- Problem 3, page 207:
  - Show all solutions to the 4-queens problem
  - Hints:
    - See figure 4.5.2 on page 196 – they've done one solution for you!
    - Do parallel processing in your group
      - Part of the group does the search with the first queen in row 3, while the other part of the group does the search with the first queen in row 4
- Note: please trace the backtracking search to do this so you understand how this works
  - (There are other ways to do figure this out)

State Space Search and Best-First Search

- State-space Search
  - Given a start-state and a goal-state
  - Generate new states that can be "visited" from the current state
  - Choose (somehow) which state to go to next
  - Stop when you reach the goal (or exhaust all possible states)
- Very useful for many problems in Artificial Intelligence
  - Puzzles, games
  - Expert systems
  - Theorem provers
  - Etc.

Heuristic Search

- We could use BFS or DFS on such problems
- Use a a <u>heuristic</u> to evaluate each state
  - Assigns a value f(state) that is some measure of how similar the state is to the goal state
- <u>Best</u>-first Search strategy
  - Like BFS but use a priority queue and visit the state that has the highest heuristic score f(n)
  - Open states: a list of states that could be chosen next (i.e. they're in the PQueue)
  - Closed states: a list of states we've already visited (i.e. they're in the tree)

Best-First Strategy

- The strategy:
  - While there are open states in the PQueue
    - current = PQueue.next();
    - Put current on the closed list.
    - If current is the goal, we're done
    - For each state s that can be generated from current
      - If s is on the closed list, ignore it. Otherwise…
      - Calculate its score f(s)
      - Store (s, f(s)) in the PQueue
    - End for
  - End while

Example: The 8-puzzle

- 8 numbered tiles in a 3x3 frame
- Repeatedly slide a tile into the “blank” position to reach some goal configuration
- Given a current state, generating child-states is what moves are possible
- Heuristic?
  - Count how many tiles (including the blank) are out of position

- See following slides.
- Note: There’s also a 15-puzzle with a 4x4 frame

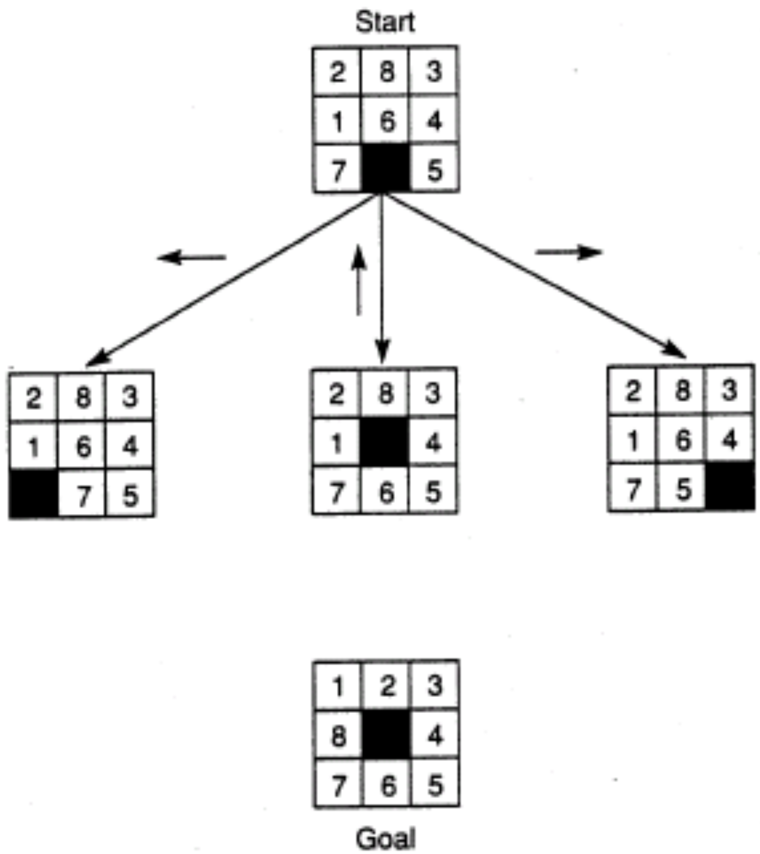


Figure 5.6 The start state, first set of moves, and goal state for an 8-puzzle instance.

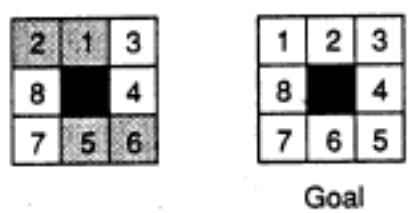


Figure 5.7 An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

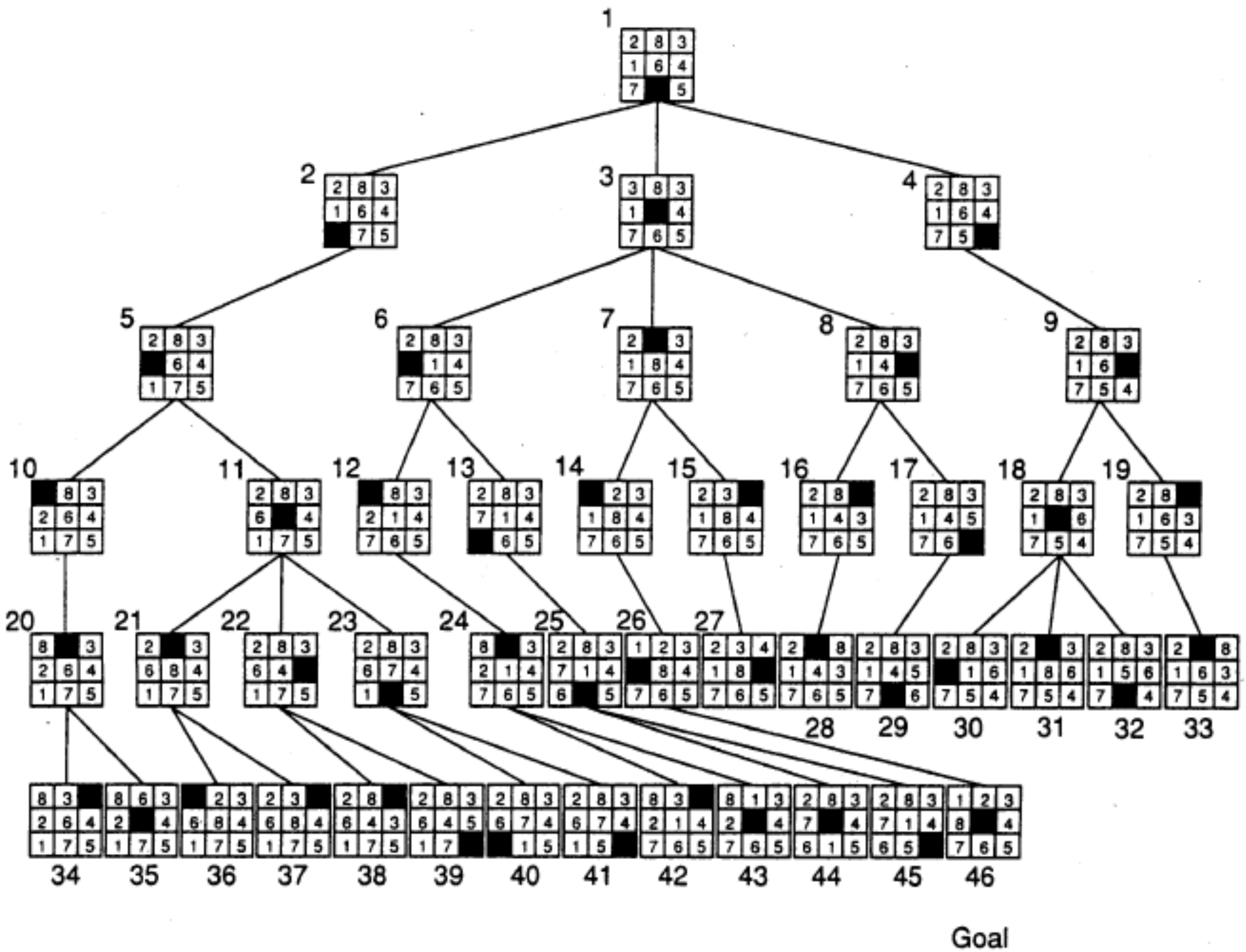


Figure 3.15 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

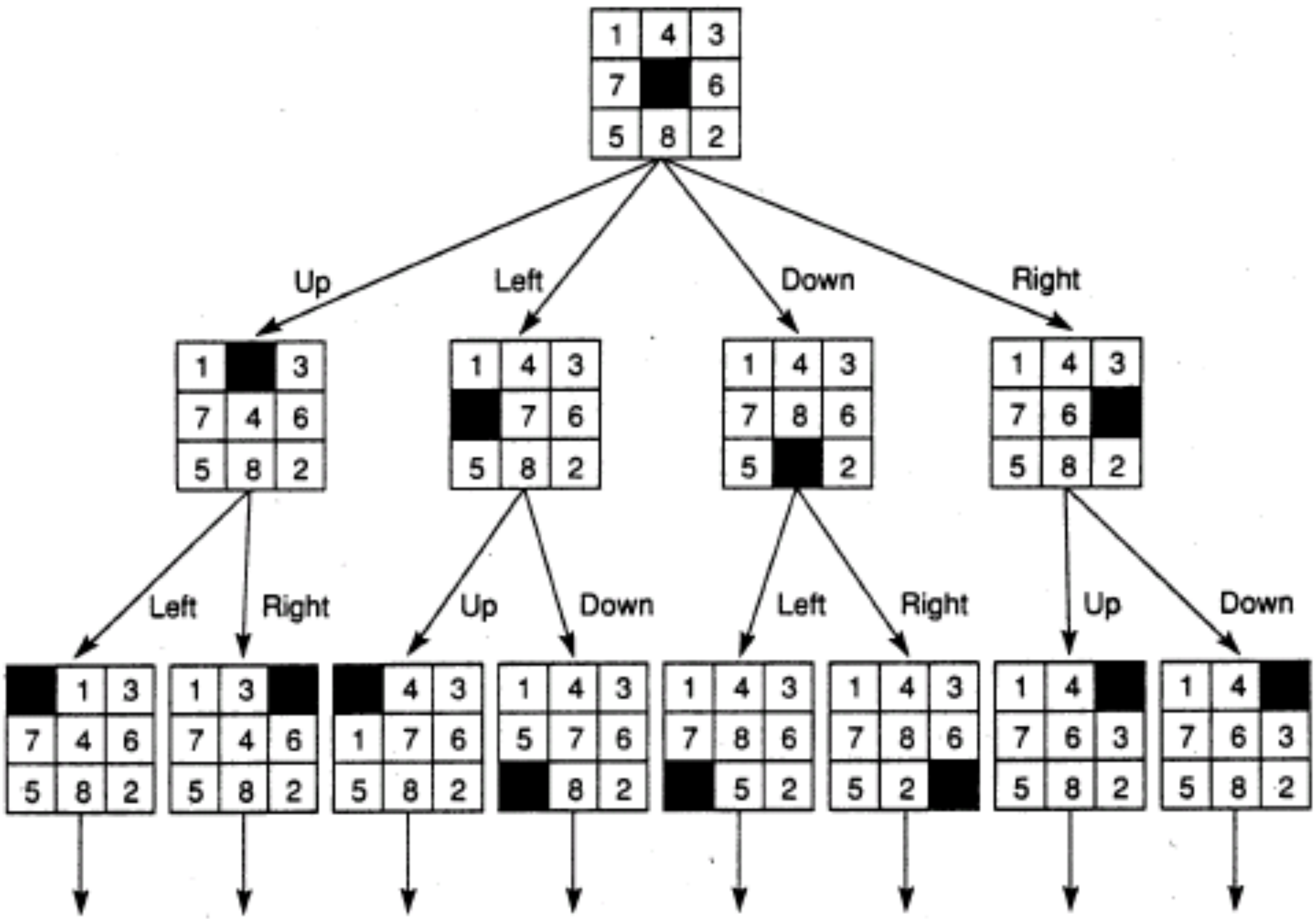


Figure 3.6 State space of the 8-puzzle generated by "move blank" operations.

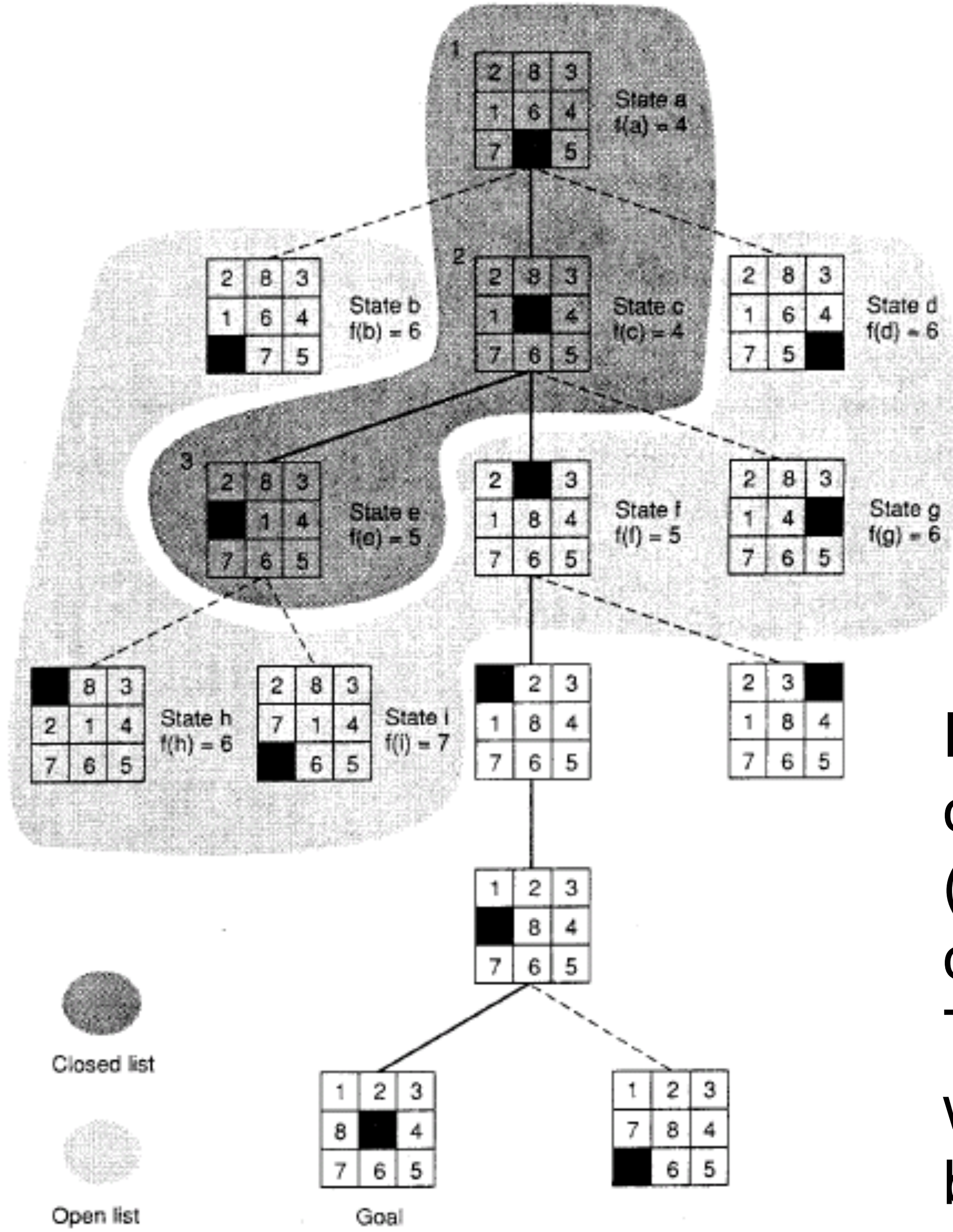


Figure 5.11 **open** and **closed** as they appear after the third iteration of heuristic search.

Here f(n) is a count of how many tiles (incl. the blank) are out of place.
The next state that will be chosen will be State-f with score 5

A Better Use of Heuristics

- If f(n) is the number of tiles out of place, this is really an estimate of how many moves are need to reach the goal.
- Better idea: let f(n) = g(n) + h(n) where
  - g(n) is the cost to the current node (the length of the path here), and
  - h(n) is an estimate of the cost to reach the goal from the current node

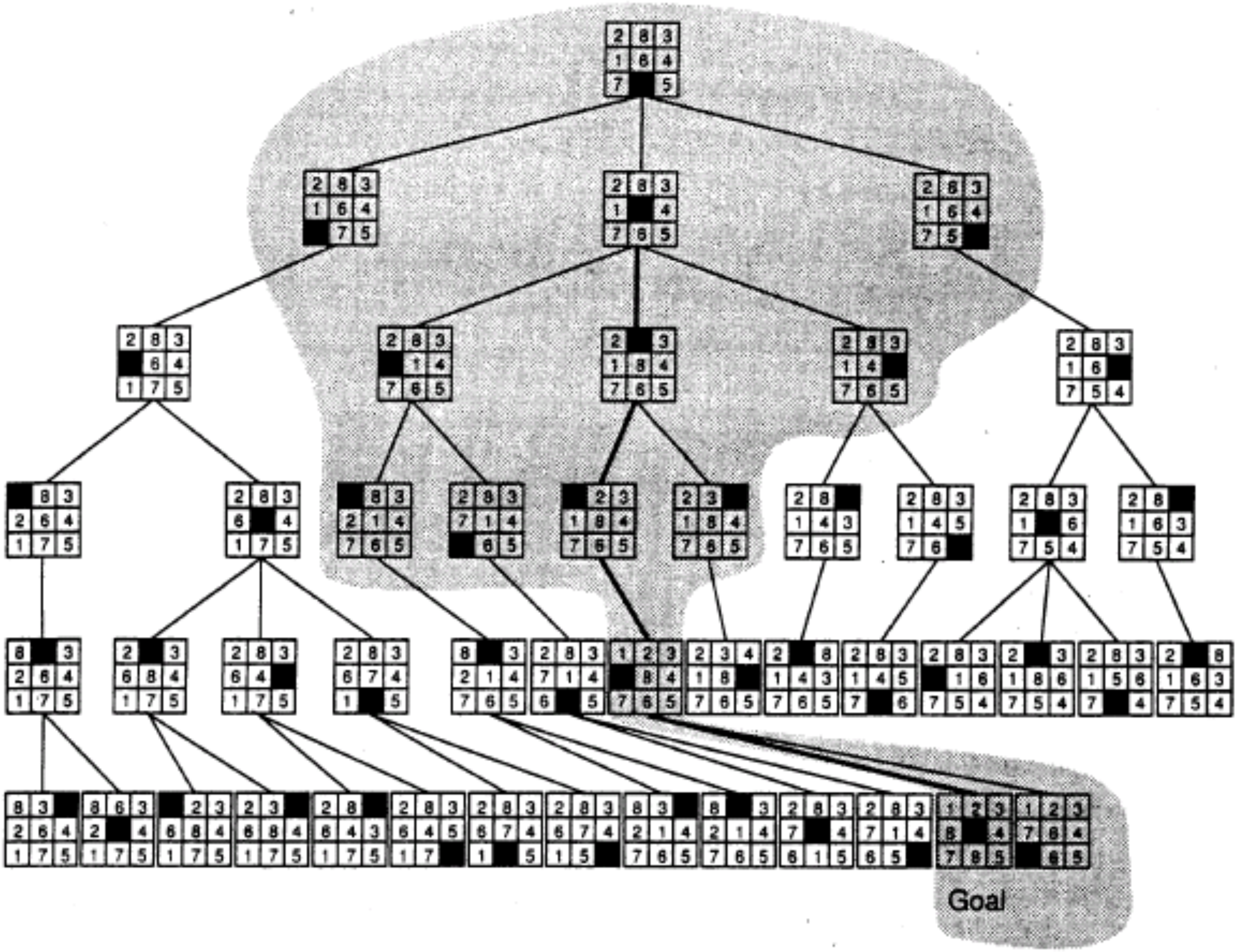


Figure 5.12 Comparison of state space searched using heuristic search with space searched by breadth-first search. The portion of the graph searched heuristically is shaded. The optimal solution path is in bold. Heuristic used is **f(n) = g(n) + h(n)** where **h(n)** is tiles out of place.