Design of algorithms

Compiled from internet

Thanks to internet community to share

Algorithm design

- **Algorithm Design Paradigms:**
  - General approaches to the construction of efficient solutions to problems.
  - Design patterns or templates
- **Why it is interesting:**
  - They provide templates suited to solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most of today’s high-level languages such as C++, Python, Java.
  - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

Find better alternative

- Programming is an art and there are many possible ways to solve a problem.
- Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.
- So the goal is to find the best one possible

Brute force

- Brute force is a straightforward approach to solve a problem based on the problem’s statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small size instances of a problem.

Brute force examples

- Computing a^n ($a > 0$, n a nonnegative integer) by multiplying a*a*…*a
- Computing n!
- Selection sort
- Bubble sort
- Sequential search
- Exhaustive search: Traveling Salesman Problem, Knapsack problem

Travelling Salesman Problem (TSP)

- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Greedy Algorithms

- Greedy Algorithms "take what you can get **now**" strategy
- The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far.
- At each step the choice must be locally optimal – this is the central point of this technique.
- So you may not get global optimal

Greedy Algorithm examples

- Minimal spanning tree
- Shortest distance in graphs
- Greedy algorithm for the Knapsack problem
- The coin exchange problem
- Huffman trees for optimal encoding

Greedy Algorithms

- Greedy techniques are mainly used to solve optimization problems. They do not always give the best solution.

- Example:
- Consider the knapsack problem with a knapsack of capacity 10 and 4 items given by the < weight :value> pairs: <5:6>, <4:3>, <3: 5>, <3: 4>. The greedy algorithm will choose item1 <5:6> and then item3 <3:5> resulting in total value <8:11>, while the optimal solution is to choose items 2, 3, and 4 thus obtaining total value <10:12>.

- It has been proven that greedy algorithms for the minimal spanning tree, the shortest paths, and Huffman codes always give the optimal solution.

Knapsack problems:

0-1 Knapsack – A thief robbing a store finds n items worth v_1, v_2, .., v_n dollars and weight w_1, w_2, …, w_n pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if he wants to maximize value.

Fractional knapsack problem – Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20% of item i for a weight of $0.2w_i$ and a value of $0.2v_i$.

Divide-and-Conquer, Decrease-and-Conquer

- These are methods of designing algorithms that (informally) proceed as follows:
  - Given an instance of the problem to be solved
  - Split this into several smaller sub-instances (of the same problem)
  - Independently solve each of the sub-instances and then
  - Combine the sub-instance solutions so as to yield a solution for the original instance.

Divide-and-Conquer vs. Decrease-and-Conquer

- With the divide-and-conquer method the size of the problem instance is reduced by a factor (e.g. half the input size)
- With the decrease-and-conquer method the size is reduced by a constant.

Divide-and-Conquer examples

- Computing a^n ($a > 0$, n a nonnegative integer) by recursion
- Binary search in a sorted array (recursion)
- Mergesort algorithm, Quicksort algorithm (recursion)
- The algorithm for solving the fake coin problem (recursion)
    - with n coins, one of which is fake.
    - Fake coin assumed to be lighter than real one.
    - Having scales to compare coins.

Decrease-and-Conquer examples

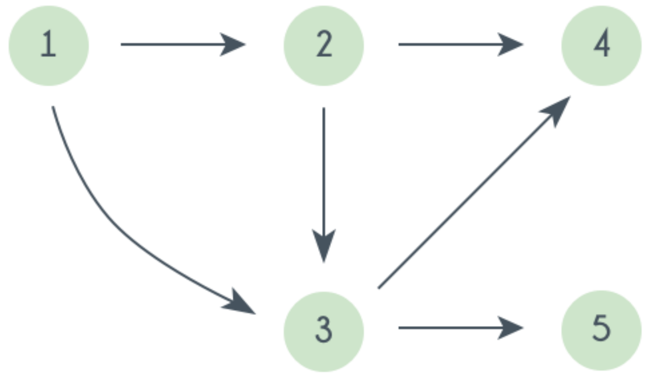
- Insertion sort
- Topological sorting (see next slide for reference)
- Binary Tree traversals: inorder, preorder and postorder (recursion)
- Computing the length of the longest path in a binary tree (recursion)
- Computing Fibonacci numbers (recursion)
- Reversing a queue (recursion)
- Warshall’s algorithm (recursion) (all pair shortest path problem)

Topological sort

- Topological sorting problem: given digraph G = (V, E) ,
- find a linear ordering of vertices such that:
- for any edge (v, w) in E, v precedes w in the ordering
- For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another

Topological sort (cont.)

- Consider the following directed graph.
- Let vertex 1 represent task1, vertex 2 represent task2, etc.
- Possible solutions of the following: 1,2,3,4,5 or 1,2,3,5,4



Divide-and-Conquer, Decrease-and-Conquer

- In general two questions:
    1. How to solve the sub-instance
    2. How to combine the obtained solutions
- The answer to the second question depends on the nature of the problem.
- In most cases the answer to the first question is: using the same method.
- When to stop decreasing the problem instance, i.e. what is the minimal instance of the given problem and how to solve it.
- When we use recursion, the solution of the minimal instance is called “terminating condition”

Dynamic Programming

- One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly since identical sub-instances may arise.

- The idea behind dynamic programming is to avoid this pathology by obviating the requirement to calculate the same quantity twice.

- The method usually accomplishes this by maintaining a table of sub-instance results. Then use table look up to find known results computed earlier. In Python we use dict.

Dynamic Programming vs. Divide and Conquer

- Dynamic Programming is a Bottom-Up Technique in which the smallest sub-instances are explicitly solved first and the results of these can then be used to construct solutions to progressively larger sub-instances.
- Divide-and-Conquer is a Top-Down Technique which logically progresses from the initial instance down to the smallest sub-instance via intermediate sub-instances.

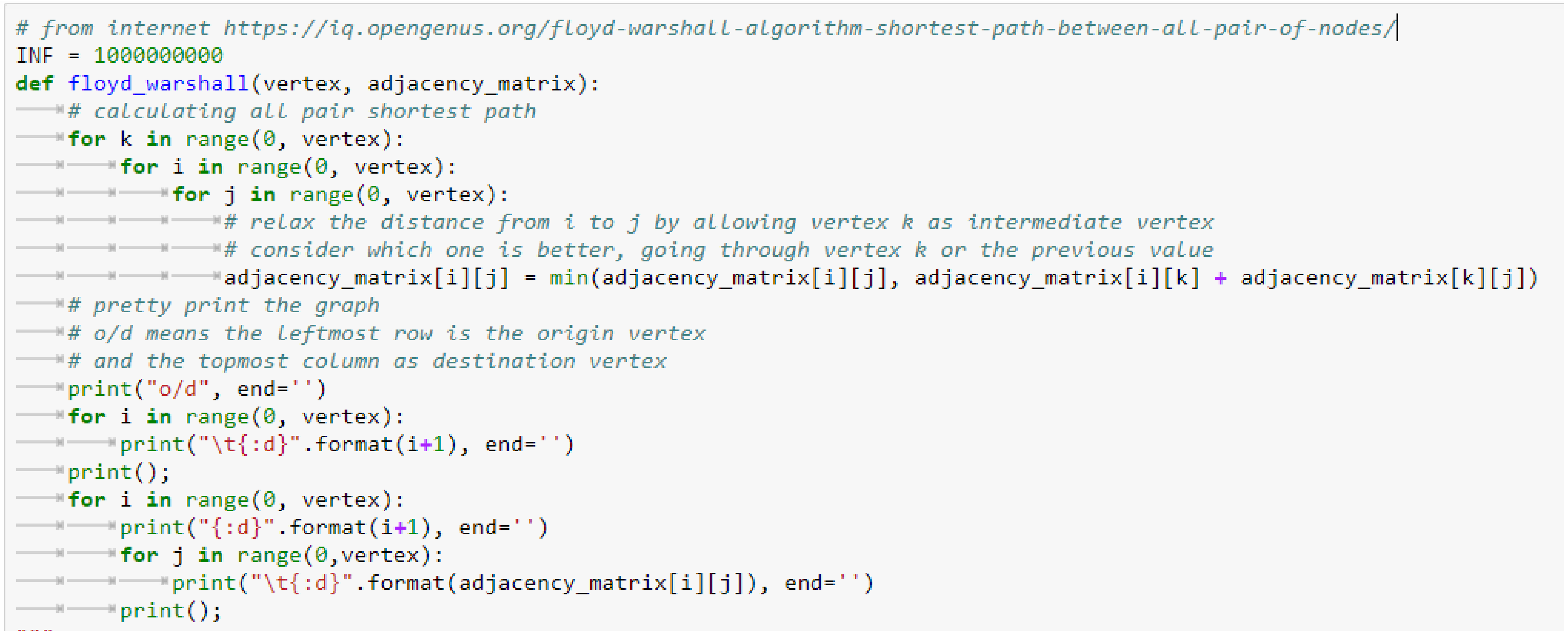
Dynamic Programming examples

- Fibonacci numbers computed by iteration.
- Warshall’s algorithm implemented by iterations

Floyd-Warshall (all pair shortest path algorithm)

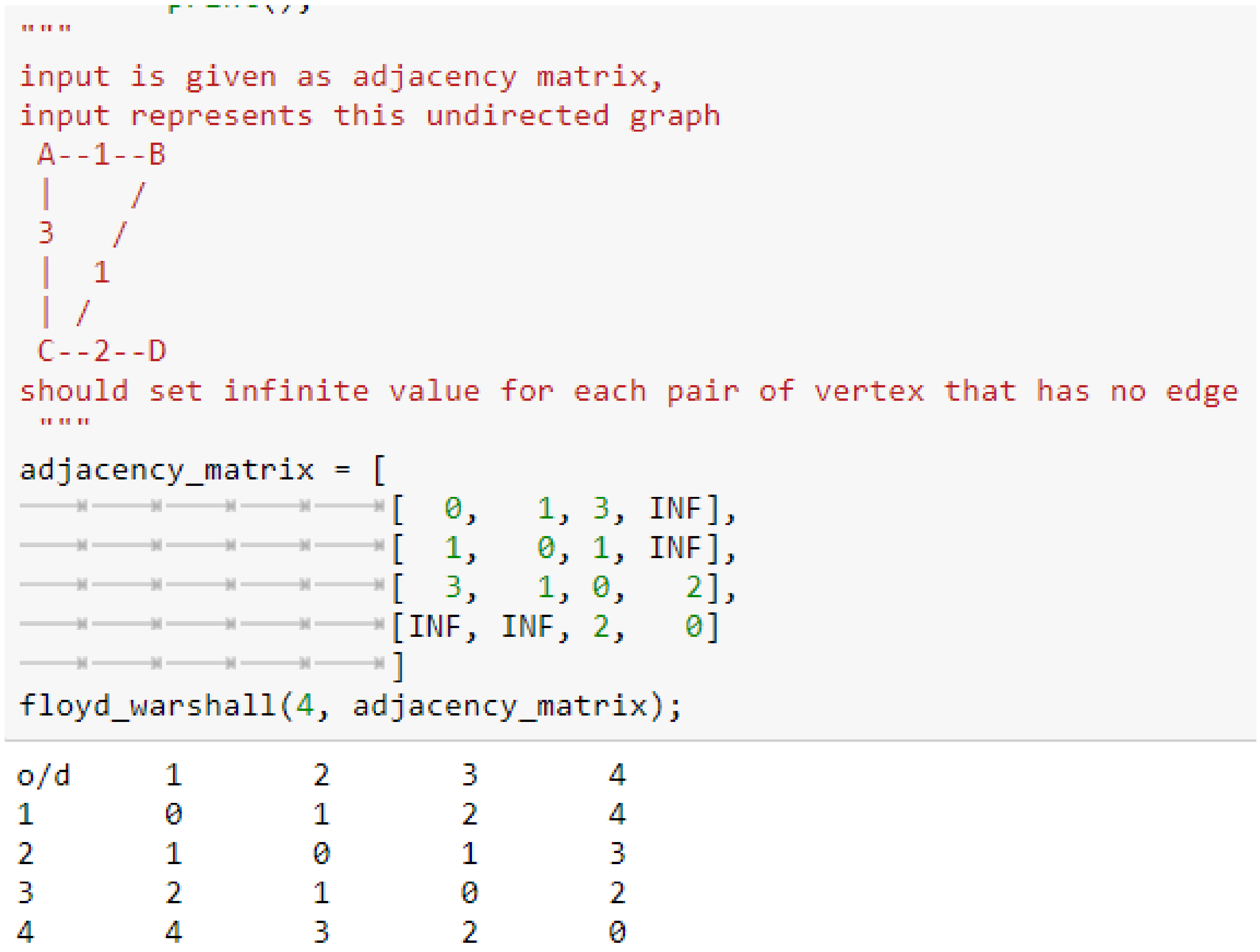
```
• Floyd-Warshal()
•       d[v][u] = inf for each pair (v,u)
•       d[v][v] = 0 for each vertex v
•       for k = 1 to n
•           for i = 1 to n
•               for j = 1 to n
•                   d[i][j] = min(d[i][j], d[i][k] + d[k][j])
```

Floyd-Warshall all-Pair Shortest Path in Python



```python
# from internet https://iq.opengenus.org/floyd-warshall-algorithm-shortest-path-between-all-pair-of-nodes/
INF = 1000000000
def floyd_warshall(vertex, adjacency_matrix):
    # calculating all pair shortest path
    for k in range(0, vertex):
        for i in range(0, vertex):
            for j in range(0, vertex):
                # relax the distance from i to j by allowing vertex k as intermediate vertex
                # consider which one is better, going through vertex k or the previous value
                adjacency_matrix[i][j] = min(adjacency_matrix[i][j], adjacency_matrix[i][k] + adjacency_matrix[k][j])
    # pretty print the graph
    # o/d means the leftmost row is the origin vertex
    # and the topmost column as destination vertex
    print("o/d", end='')
    for i in range(0, vertex):
        print("\t{:d}".format(i+1), end='')
    print();
    for i in range(0, vertex):
        print("{:d}".format(i+1), end='')
        for j in range(0,vertex):
            print("\t{:d}".format(adjacency_matrix[i][j]), end='')
        print();
```

Test Run



```
"""
input is given as adjacency matrix,
input represents this undirected graph
 A--1--B
 |    /
 3   /
 |  1
 | /
 C--2--D
should set infinite value for each pair of vertex that has no edge
 """
adjacency_matrix = [
                    [  0,   1, 3, INF],
                    [  1,   0, 1, INF],
                    [  3,   1, 0,   2],
                    [INF, INF, 2,   0]
                    ]
floyd_warshall(4, adjacency_matrix);
```

```
o/d     1       2       3       4
1       0       1       2       4
2       1       0       1       3
3       2       1       0       2
4       4       3       2       0
```

Transform-and-Conquer

- These methods work as two-stage procedures.
- First, the problem is modified to be more amenable to solution.
- In the second stage the problem is solved.

Transform-and-Conquer

Types of problem modifications:

1. Problem simplification
2. Change in the representation
3. Problem reduction

Transform-and-Conquer: Problem simplification

Example: consider the problem of finding the two closest numbers in an array of numbers.

a) Brute force solution: $O(n^2)$

b) Transform and conquer solution: *O(nlogn)*

Presort the array – *O(nlogn)*

Scan the array comparing the differences - *O(n)*

Transform-and-Conquer: Change the representation

- Example:
- Transform the original binary search tree to **AVL** trees (add balance factor)
- It guarantees *O(nlogn)* search time because AVL tree is a balanced tree.

Transform-and-Conquer : Problem reduction

For example:

Least common multiple

$$\text{lcm}(m,n) = (m*n)/\ \text{gcd}(m,n)$$

Backtracking and branch-and-bound: generate and test methods

- The method is used for state-space search problems.
- State-space search problems are problems, where the problem representation consists of:
  - initial state
  - goal state(s)
  - a set of intermediate states
  - a set of operators that transform one state into another. Each operator has preconditions and postconditions.
  - a cost function – evaluates the cost of the operations (optional)
  - a utility function – evaluates how close is a given state to the goal state (optional)

State-space search

- The solving process solution is based on the construction of a state-space tree
- Nodes represent states, the root represents the initial state, and one or more leaves are goal states.
- Each edge is labeled with some operator.
  - If a node ***b*** is obtained from a node ***a*** as a result of applying the operator ***O***, then ***b*** is a child of ***a*** and the edge from ***a*** to ***b*** is labeled with ***O***.
- The solution is obtained by searching the tree until a goal state is found.

Backtracking uses depth-first search

- Usually without cost function.
- The main algorithm is as follows:

```
Store the initial state in a stack
While the stack is not empty, do:
    Read a node from the stack.
    While there are available operators do:
        Apply an operator to generate a child
        If the child is a goal state – stop
        If it is a new state, push the child into the stack
```

Backtracking uses depth-first search

- The utility function is used to tell how close is a given state to the goal state and whether a given state may be considered a goal state.

- If no children can be generated from a given node, then we ***backtrack*** – read the next node from the stack.

State-Space Search examples

1. A farmer has to move a goat, a cabbage and a wolf from one side of a river to the other side using a small boat. The boat can carry only the farmer and one more object (either the goat, or the cabbage, or the wolf). If the farmer leaves the goat with the wolf alone, the wolf would kill the goat. If the goat is alone with the cabbage, it will eat the cabbage. How can the farmer move all his property safely to the other side of the river?"
2. You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

The approaches to solve the problem

- Representation of the problem state, initial and final states
- Representation of the actions available in the problem, in terms of how they change the problem state.

Get 2 gallon water problem : Problem state

Pair of numbers (X,Y):

- X - water in jar 1 called A,
- Y - water in jar 2, called B.
- Initial state: (0,0),
- Final state: (2,_) here "_"   means "any quantity"

Get 2 gallon water problem: Available actions (operators)

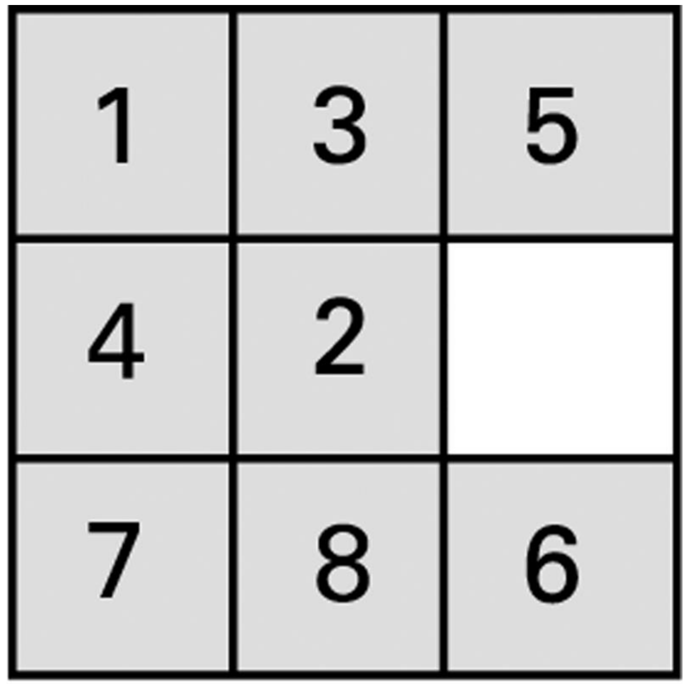
Description	Pre-conditions on (X,Y)	Action (Post-conditions)
O1. Fill A	X < 4	(4, Y)
O2. Fill B	Y < 3	(X, 3)
O3. Empty A	X > 0	(0, Y)
O4. Empty B	Y > 0	(X, 0)
O5. Pour A into B	a. X > 3 - Y b. X ≤ 3 - Y	(X + Y - 3 , 3) (0, X + Y)
O6. Pour B into A	a. Y > 4 - X b. Y ≤ 4 - X	(4, X + Y - 4) (X + Y, 0)

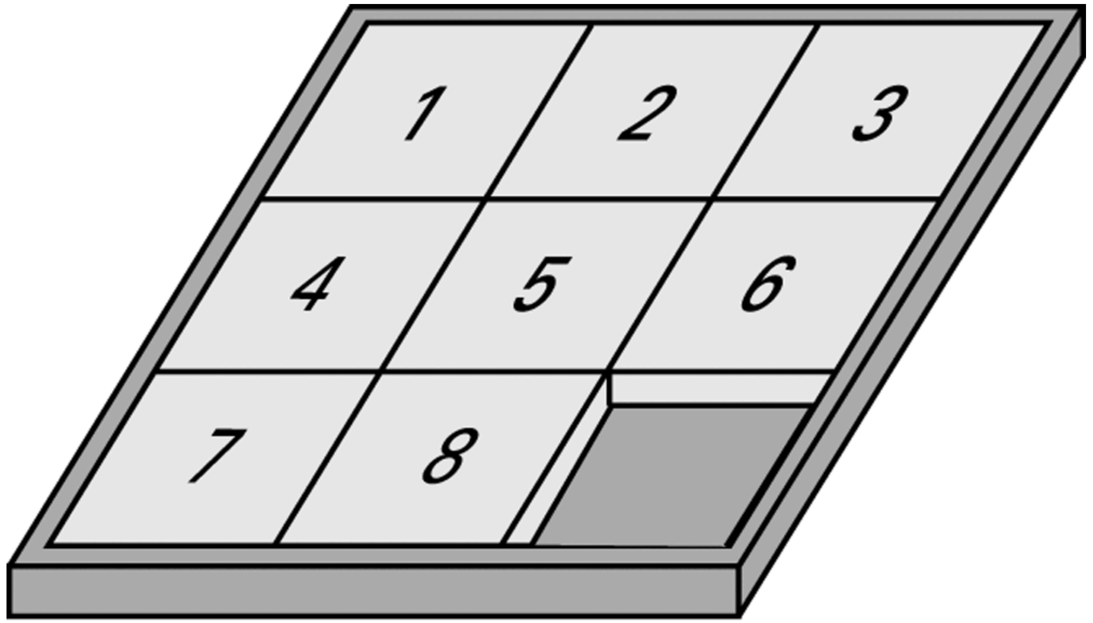
Possible solution: **(0,0) ->O1 (4,0)-> O5 (1,3) ->O4 (1,0)-> O5(0,1) -> O1(4,1) ->O5 (2,3)**

Branch-and-bound

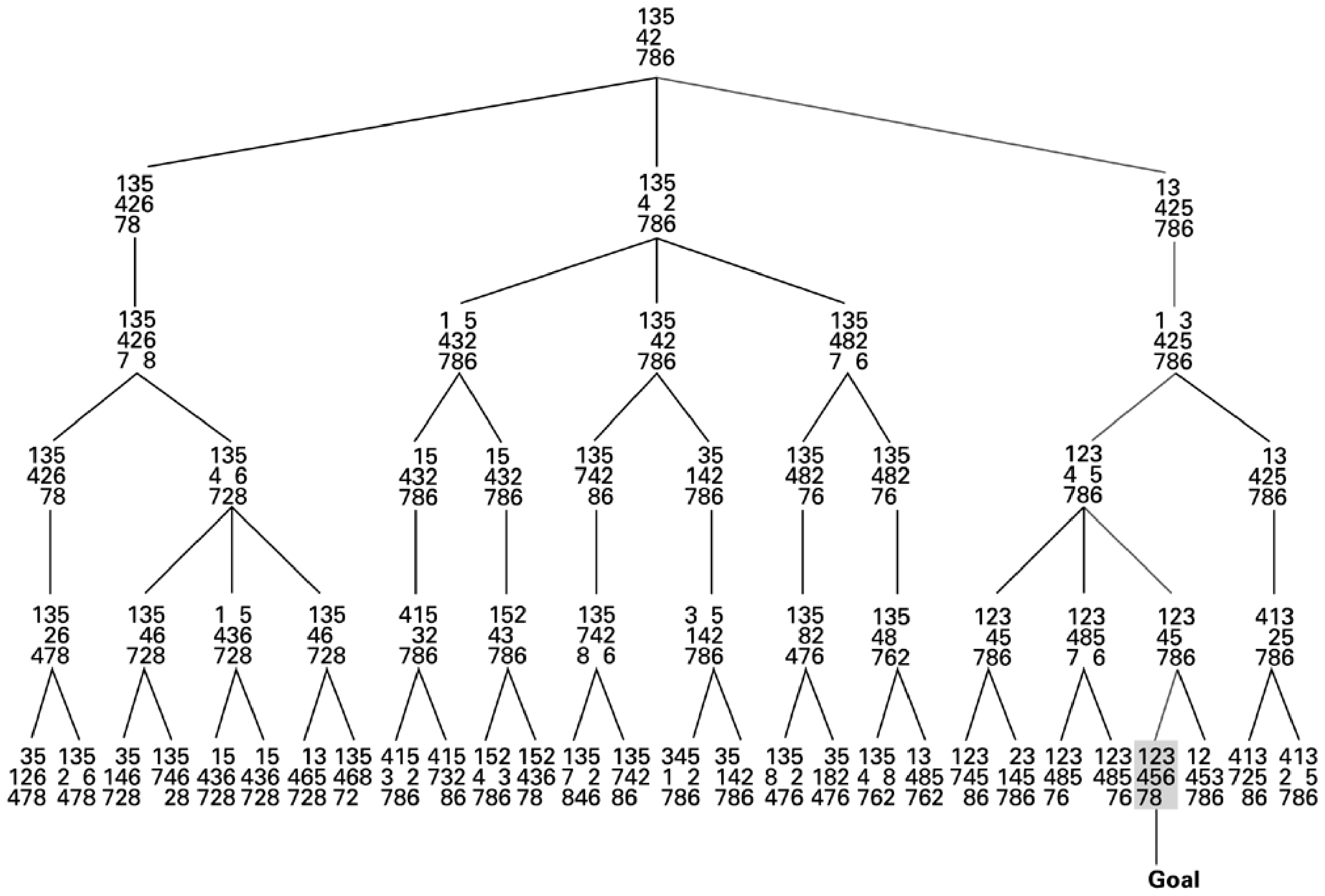
- Branch and bound is used when we can evaluate each node using the cost and utility functions.
- At each step we choose the best node to proceed further.
- Branch-and bound algorithms are implemented using a priority **queue**. The state-space tree is built in a **breadth-first** manner.
- **Example:** the 8-puzzle problem. The cost function is the number of moves. The utility function evaluates how close is a given state of the puzzle to the goal state, e.g. counting how many tiles are not in place, or counting at least how many moves to reach goal.

The 8-puzzle problem: init (left), goal (right)





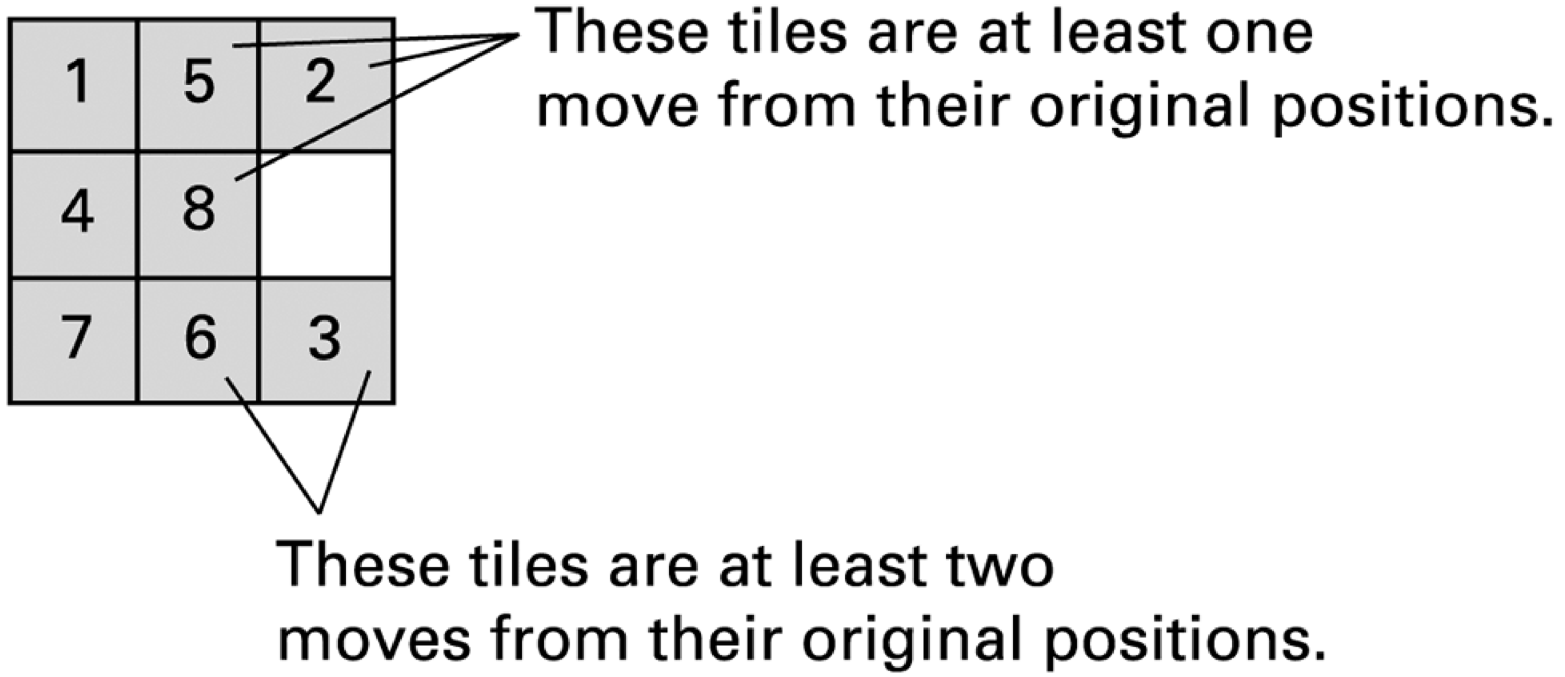
A sample search tree



Heuristic Strategies

- **Heuristic:** A "rule of thumb" for making decisions
- Requirements for good heuristics
  - Must be easier to compute than a complete solution
  - Must provide a reasonable estimate of proximity to a goal

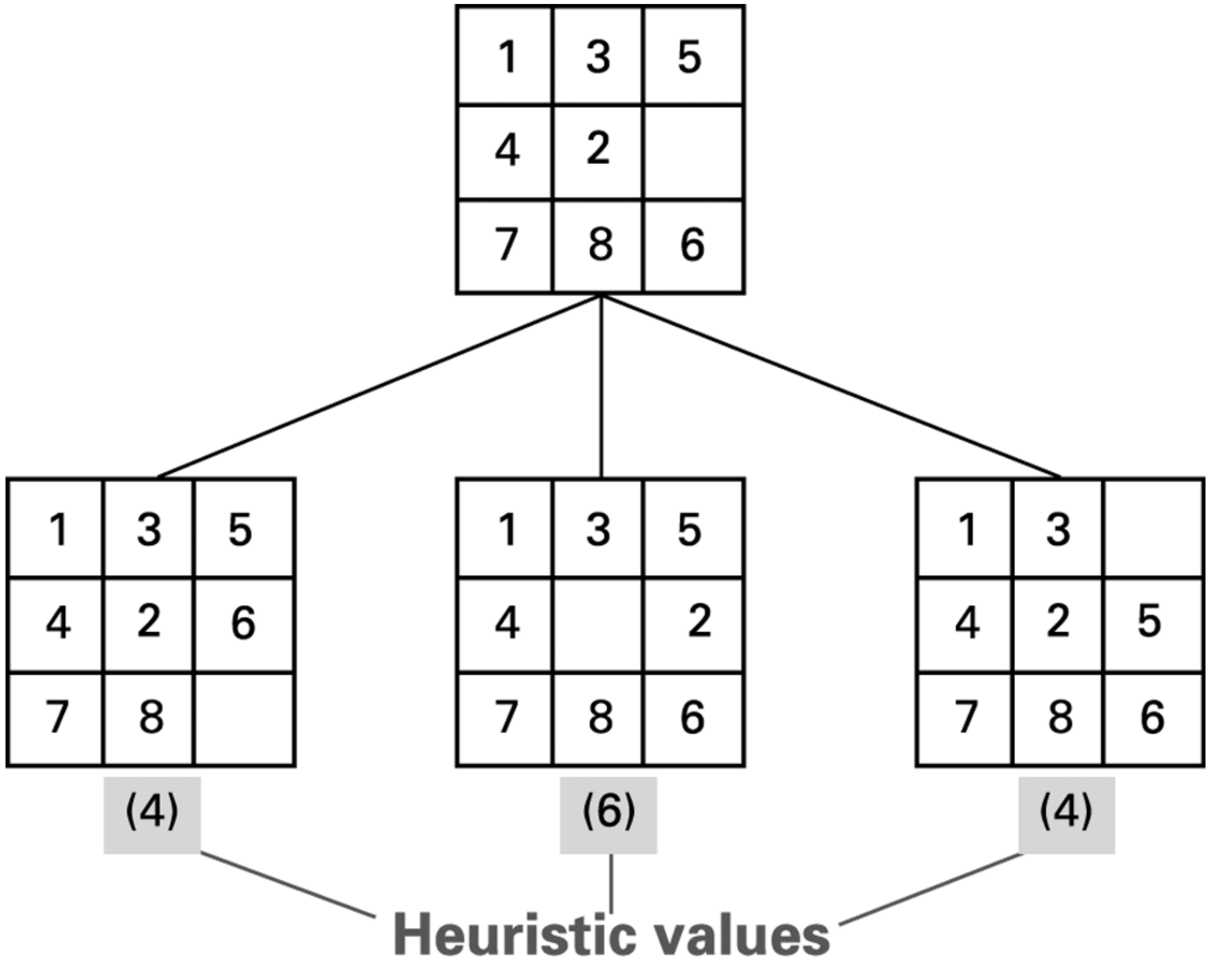
An unsolved eight-puzzle (3*1+2*2 = 7 as heuristic value)


1
5
2
4
8
7
6
3
These tiles are at least one move from their original positions.
These tiles are at least two moves from their original positions.

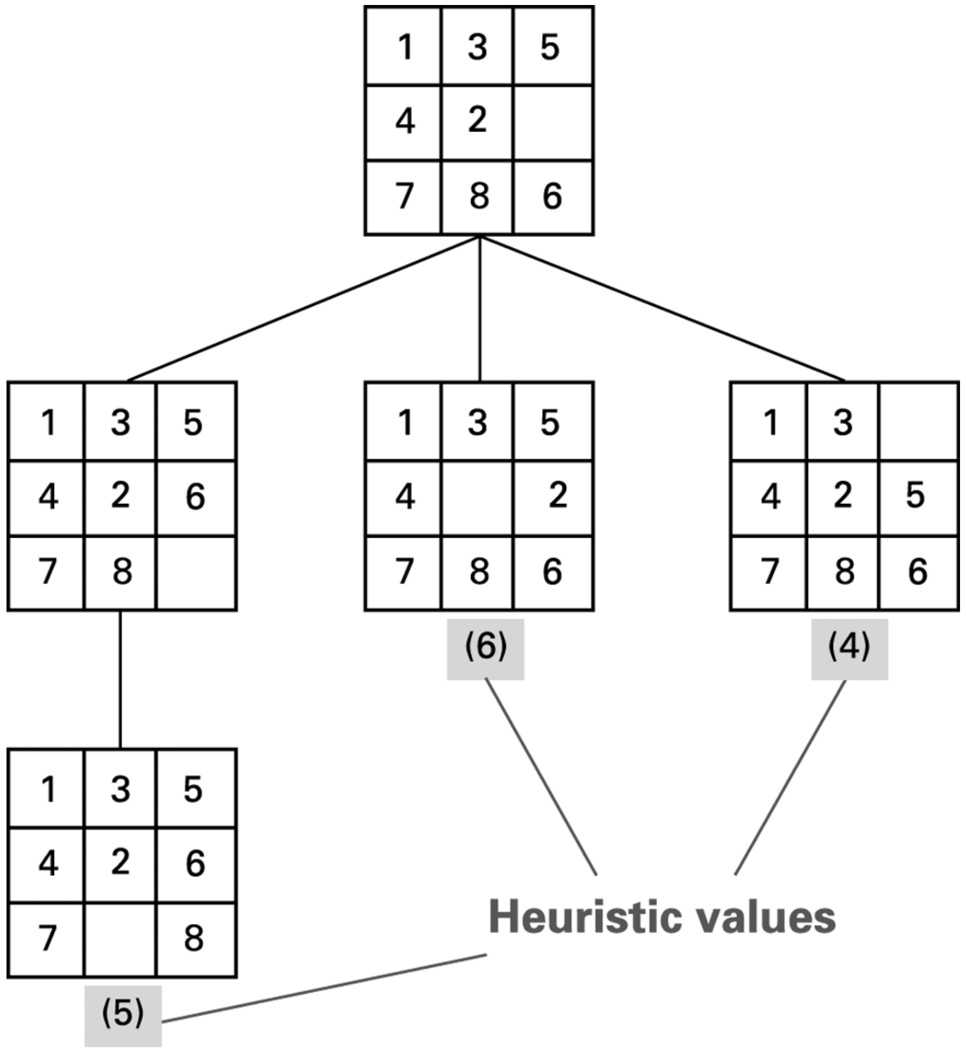
Pseudo code for 8-puzzle problem

```
• Establish the start node of the state graph as the root of the search tree and record its heuristic value.
• while (the goal node has not been reached):
    • Select the leftmost leaf node with the smallest heuristic value of all leaf nodes.
    • To this selected node attach as children those nodes that can be reached by a single production.
    • Record the heuristic of each of these new nodes next to the node in the search tree.
• Traverse the search tree from the goal node up to the root, pushing the production associated with each arc traversed onto a stack.
• Solve the original problem by executing the productions as they are popped off the stack.
```

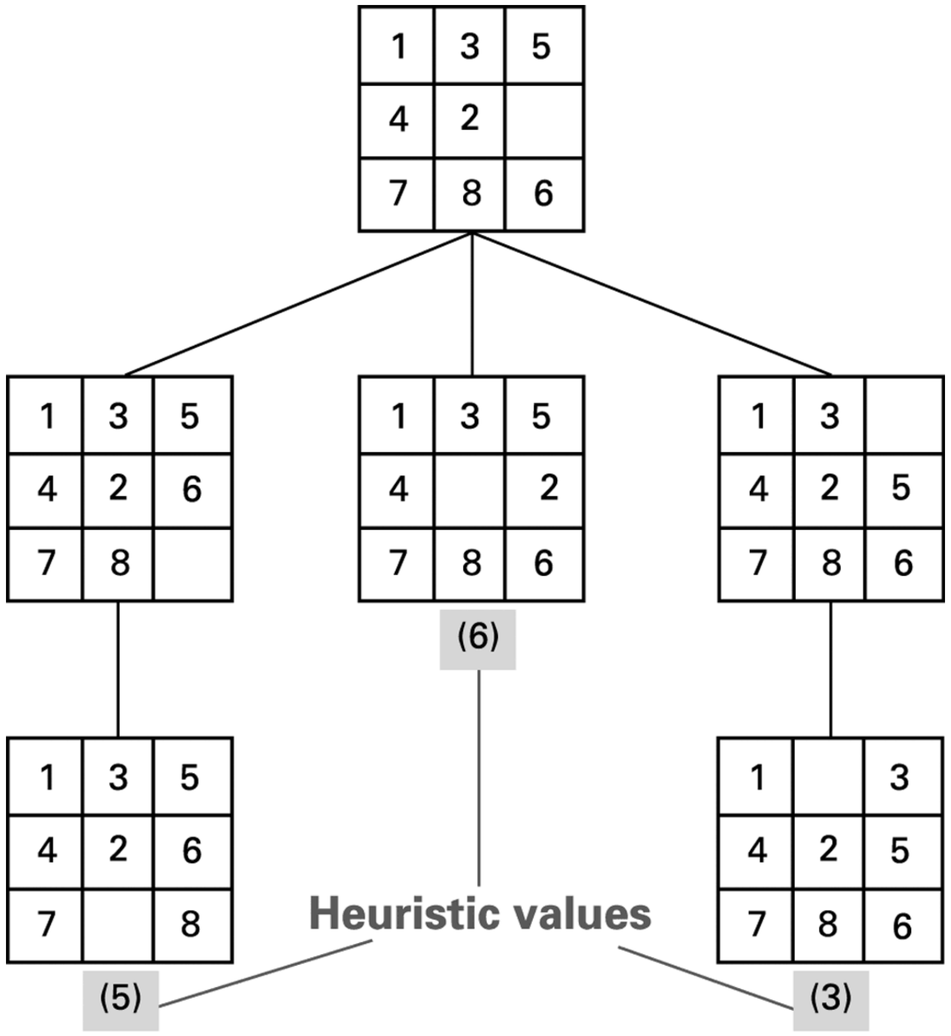
The beginnings of our heuristic search



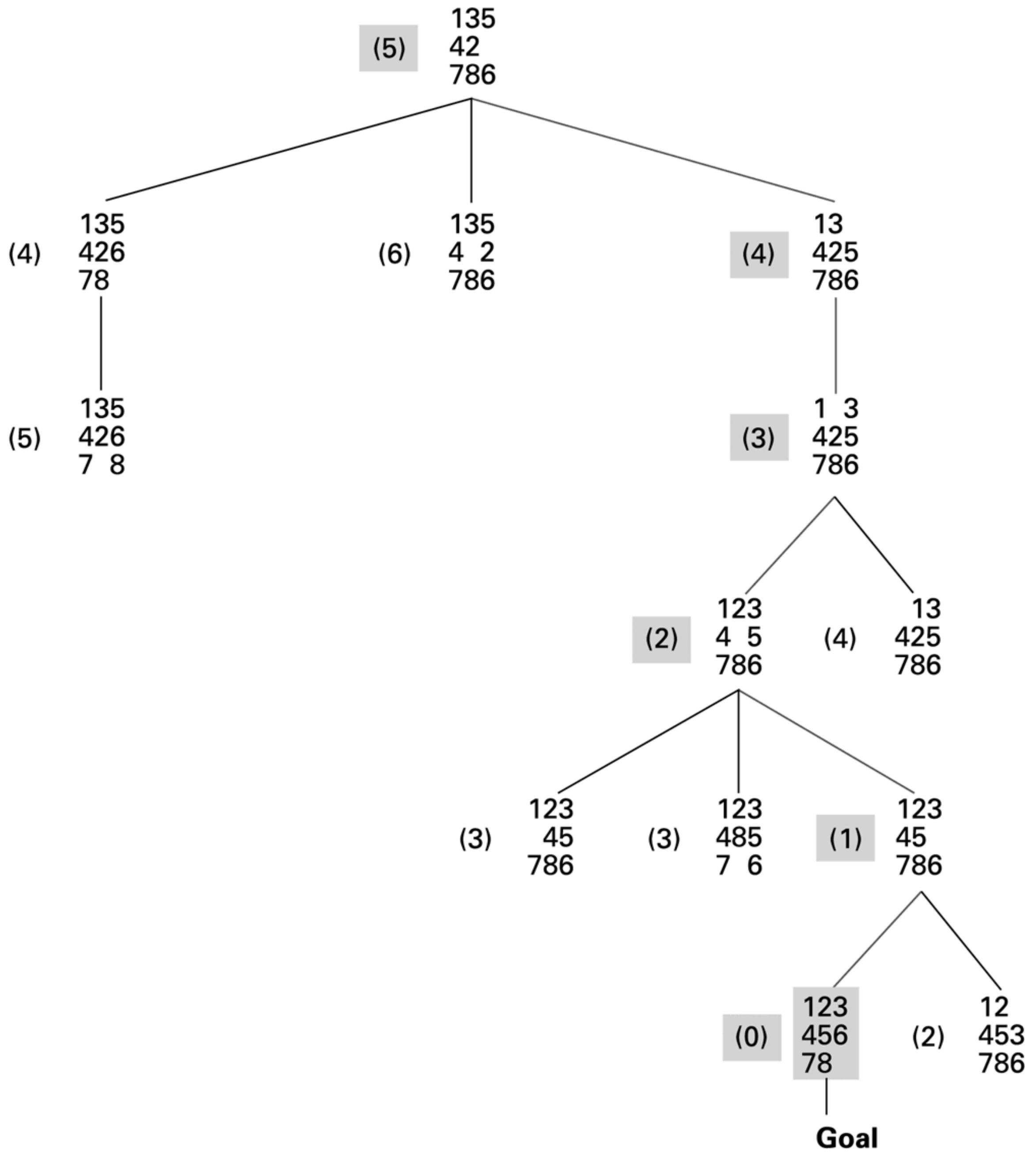
The search tree after two passes



The search tree after three passes



The complete search tree formed by our heuristic system



Genetic algorithms

- Used mainly for optimization problems for which the exact algorithms are of very low efficiency.
- GAs search for good solutions to a problem from among a (large) number of possible solutions. The current set of possible solutions is used to generate a new set of possible solution.
- They start with an initial set of possible solutions, and at each step they do the following:
  - evaluate the current set of solutions (current generation)
  - choose the best of them to serve as “parents” for the new generation, and construct the new generation.

When to stop?

- The loop runs until a specified condition becomes true.
- E.g. a solution is found that satisfies the criteria for a “good” solution

  or
- The number of iterations exceeds a given value

  or
- No improvement has been recorded when evaluation the solutions.

Issues for GAs

- How large to be the **size of a population** so that there is sufficient diversity? Usually the size is determined through trial-and-error experiments.
- How to represent the solutions so that the representations can be manipulated to obtain a new solution? One approach is to represent the solutions as strings of characters (could be binary strings) and to use various types of "crossover" (explained below) to obtain a new set of solutions. The strings are usually called **"chromosomes"**
- How to evaluate a solution?. The function used to evaluate a solution is called **"fitness function"** and it depends on the nature of the problem.

Issues for GAs (cont.)

- How to manipulate the representations in order to construct a new solution? The method that is characteristic of GAs is to combine two or more representations by taking substrings of each of them to construct a new solution. This operation is called **“crossover”**.
- How to choose the individual solutions that will serve as parents for the new generation? The process of choosing parents is called **“selection”**. Various methods have been experimented here. It seems that the choice is dependent on the nature of the problem and the chosen representation. One method is to choose parents with a probability proportional to their fitness. This method is called “roulette wheel selection”
- How to avoid convergence to a set of equal solutions? The approach here is to change randomly the representation of a solution. If the representation is a bit string we can flip bits. This operation is called **“mutation”**

The algorithms to obtain one generation

- Compute the fitness of each chromosome
- Select parents based on the fitness value and a selection strategy
- Perform crossover to obtain new chromosomes
- Perform mutation on the new chromosomes (with fixed or random probability)

GA

```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

Genetic algorithm examples

- The knapsack problem solved with GAs
- The traveling salesman problem

Conclusion remarks

- Usually a given problem can be solved using various approaches
- Some approaches result in much more efficient solutions than others.
- Consider again the Fibonacci numbers computed recursively using the decrease-and-conquer approach, and computed by iterations using dynamic programming. In the first case the complexity is $O(2^n)$, while in the second case the complexity is $O(n)$.
- On the other hand, consider sorting based on decrease-and-conquer (insertion sort) and brute force sorting. For **almost sorted** files insertion sort will give almost linear complexity, while brute force sorting algorithms have quadratic complexity.

Conclusion remarks (cont.)

- The basic question here is:
- How to choose the approach?
  - First, by understanding the problem
  - Second, by knowing various problems and how they are solved using different approaches.