CS 440 Theory of Algorithms / CS 468 Algorithms in Bioinformatics

Brute Force

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Brute Force

A straightforward approach usually based on problem statement and definitions

Examples:

1. **Computing a^n ($a > 0$, n a nonnegative integer)**
2. **Computing $n!$**
3. **Multiply two n by n matrices**
4. **Selection sort**
5. **Sequential search**

Brute-Force Sorting Algorithm

Selection Sort **Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n$-2), find the smallest element in $A[i..n\text{-}1]$ and swap it with $A[i]$:**

$A[0] \le \ldots \le A[i\text{-}1] \mid A[i], \ldots, A[min], \ldots, A[n\text{-}1]$

in their final positions

Example: 7 3 2 5

Analysis of Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Time efficiency:

Space efficiency:

Stability:

Brute-Force String Matching

- ***Pattern***: a string of *m* characters to search for
- ***Text***: a (longer) string of *n* characters to search in
- **Problem: find a substring in the text that matches the pattern**

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- All characters are found to match (successful search); or
- A mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Examples of Brute-Force String Matching

1. **Pattern:** 001011
   Text: 10010101101001100101111010

2. **Pattern:** happy
   Text: It is never too late to have a happy childhood.

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Number of comparisons:

Efficiency:

Brute Force Polynomial Evaluation

- **Problem: Find the value of polynomial**
  $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$
  at a point $x = x_0$
- **Algorithm:**

```
p := 0.0
for i := n down to 0 do
        power := 1
        for j := 1 to i do
                power := power * x
        p := p + a[i] * power
return p
```

- **Efficiency:**

Polynomial Evaluation: Improvement

- We can do better by evaluating from right to left:
- Algorithm:

```
p := a[0]
power := 1
for i := 1 to n do
        power := power * x
        p := p + a[i] * power
return p
```

- Efficiency:
- Discussion: why is this algorithm more efficient than the previous one?

Closest-Pair Problem

Find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

Brute-force algorithm

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

```
//Input: A list P of n (n ≥ 2) points P_1 = (x_1, y_1), ..., P_n = (x_n, y_n)
//Output: Indices index1 and index2 of the closest pair of points
dmin ← ∞
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        d ← sqrt((x_i − x_j)^2 + (y_i − y_j)^2) //sqrt is the square root function
        if d < dmin
            dmin ← d; index1 ← i; index2 ← j
return index1, index2
```

Efficiency:

How to make it faster?

Convex Hull Problem

- **Convex hull**
  - **Problem: Find smallest convex polygon enclosing *n* points on the plane**
  - **Algorithm: For each pair of points p_1 and p_2 determine whether all other points lie to the same side of the straight line through p_1 and p_2**
  - **Efficiency:**

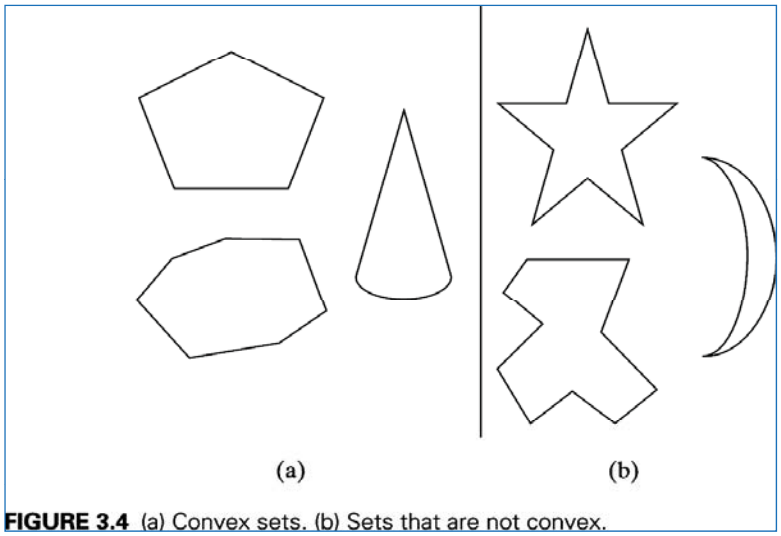


FIGURE 3.4 (a) Convex sets. (b) Sets that are not convex.

Convex Hull Problem

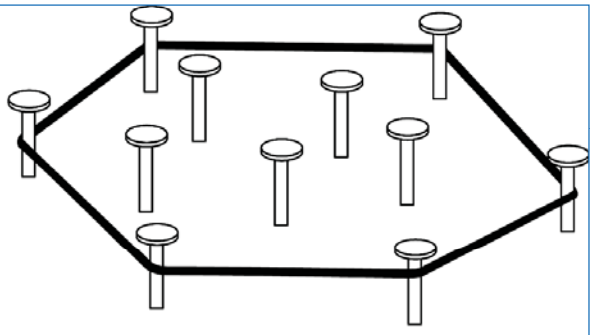


FIGURE 3.5 Rubber-band interpretation of the convex hull

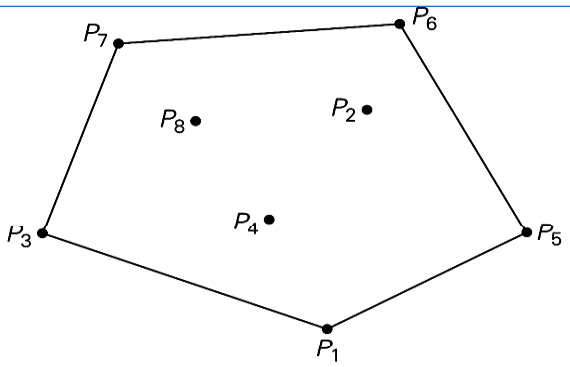


FIGURE 3.6 The convex hull for this set of eight points is the convex polygon with vertices at P_1, P_5, P_6, P_7, and P_3.

Brute-Force Strengths and Weaknesses

- **Strengths**
  - **Wide applicability**
  - **Simplicity**
  - **Yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)**
- **Weaknesses**
  - **Rarely yields efficient algorithms**
  - **Some brute-force algorithms are unacceptably slow**
  - **Not as constructive as some other design techniques**

Exhaustive Search

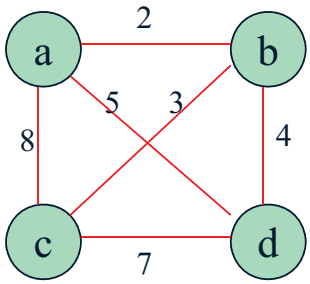
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- **Generate a list of all potential solutions to the problem in a systematic manner**
  - **all solutions are eventually listed**
  - **no solution is repeated**
- **Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far**
- **When search ends, announce the solution(s) found**

Example 1: Traveling Salesperson Problem (TSP)

- **Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city.**
- **Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph.**
- **Example:**



TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

More tours?

Less tours?

Efficiency:

Example 2: Knapsack Problem

Given *n* items:

- **weights:** w_1 w_2 … w_n
- **values:** v_1 v_2 … v_n
- **a knapsack of capacity *W***

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency:

Example 3: The Assignment Problem

There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is C[*i*,*j*]. Find an assignment that minimizes the total cost.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

How many assignments are there?

Pose the problem as the one about a cost matrix:

Assignment Problem by Exhaustive Search

$$C = \begin{matrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{matrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
	etc.

(For this particular instance, the optimal assignment can be found by exploiting the specific features of the number given. It is:)

Final Comments on Exhaustive Search

- **Exhaustive-search algorithms run in a realistic amount of time only on very small instances**
- **In some cases, there are much better alternatives!**
  - **Euler circuits**
  - **Shortest paths**
  - **Minimum spanning tree**
  - **Assignment problem**
- **In many cases, exhaustive search or its variation is the only known way to get exact solution**
- **Are we guaranteed to find the optimal solution?**

Tackling Difficult Combinatorial Problems

There are two principal approaches to tackling difficult combinatorial problems:

- **Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time**
  - **E.g., Back-Tracking and Branch-and-Bound**
- **Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time**

Branch-and-Bound

- **An enhancement of backtracking**
- **Applicable to optimization problems**
- **For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)**
- **Uses the bound for:**
  - **Ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far**
  - **Guiding the search through state-space**

State and State Space of Problem

- **State of a problem**
  - **Specific combination from all possibilities**
- **Two important states in a problem**
  - **Initial State: first state given by the problem**
  - **Goal State: solution state where the problem wants to reach**
- **State Space of a problem**
  - **Space containing all possible states of the problem**
  - **Usually represented by a graph or a tree**
- **Rules are needed for changing from one state to another.**
- **Solving a problem means searching for a path in the state space from a initial state to a goal state.**
  - **Especially useful for solving puzzles, games, etc.**

River Crossing Problem: Relationship between states

- **Where are the "solutions"?**

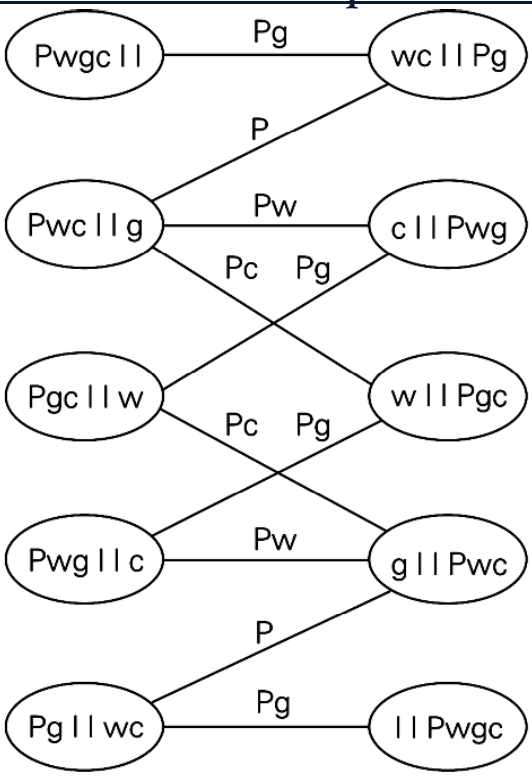
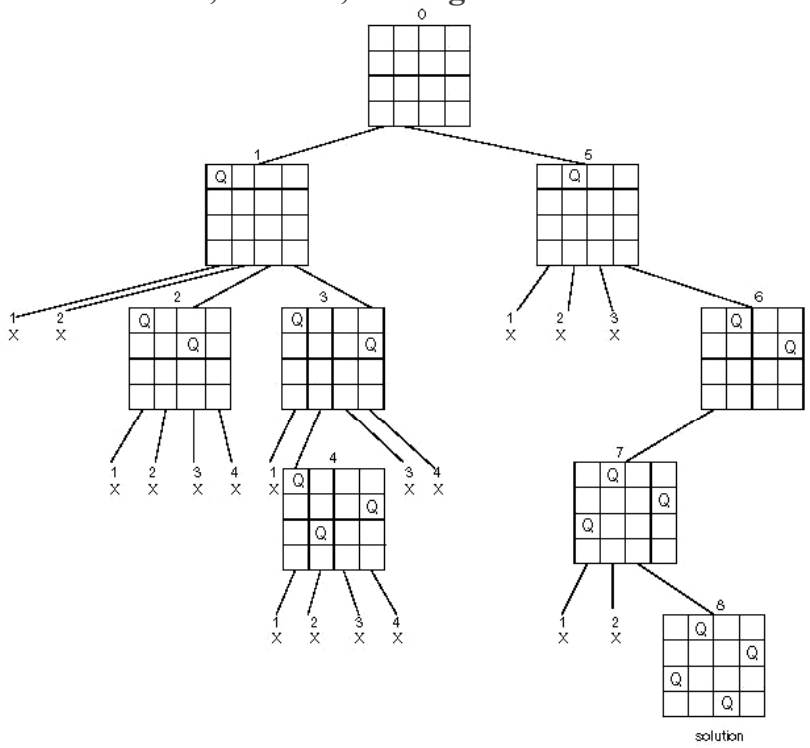


FIGURE 6.18 State-space graph for the peasant, wolf, goat, and cabbage puzzle

State-Space Tree of the 4-Queens Problem

4-Queens Problem: Place 4 queens on an 4-by-*4* chess board so that no two of them are in the same row, column, or diagonal



Example: Assignment Problem

Select one element in each row of the cost matrix *C* so that:
- **No two selected elements are in the same column**
- **The sum is minimized**

Example

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Example: First two levels of the state-space tree

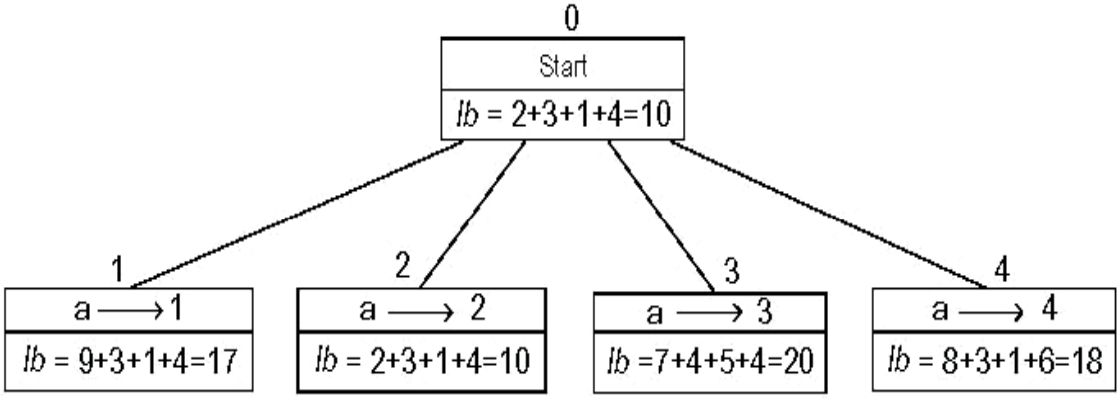


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

Example (cont.)

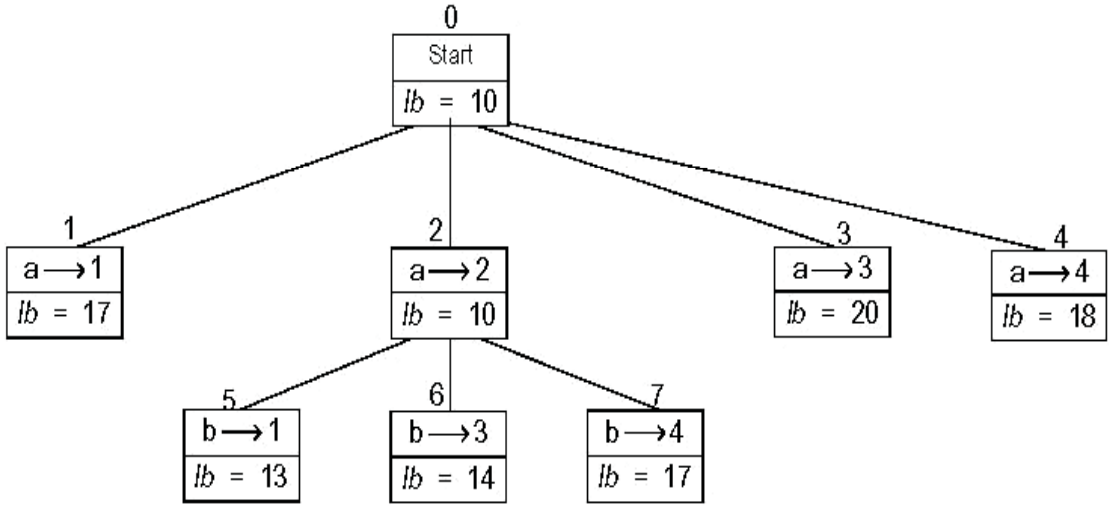


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space tree

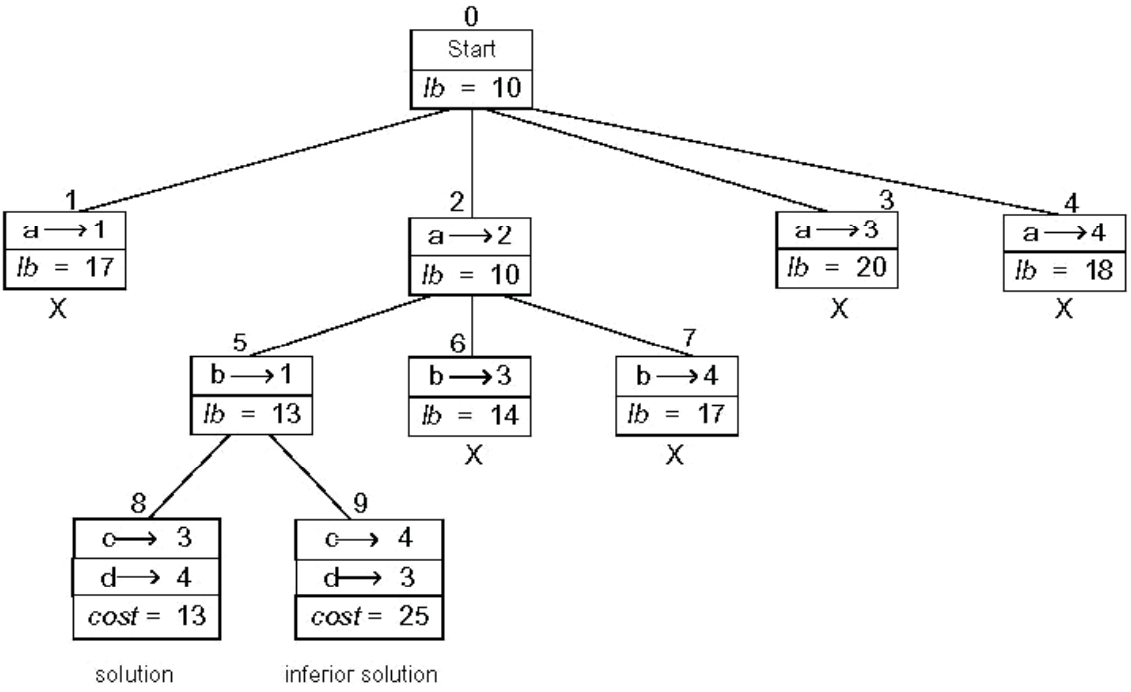


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm