Brute-Force Algorithms

CSC 411

Richard Kelley

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

- 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.

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

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
```

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
```

Efficiency: O(mn)

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 $x = x_0$

Brute-force algorithm

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
        for j ← 1 to i do        //compute x^i
            power ← power * x0
            p ← p + a[i] * power
```

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 $x = x_0$

<u>Brute-force algorithm</u>

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 $x = x_0$

Brute-force algorithm

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
        for j ← 1 to i do        //compute x^i
            power ← power * x0
            p ← p + a[i] * power
```

Efficiency: $O(n^2)$

Polynomial Evaluation: Improvement

Can we do better?

$p \leftarrow a[0]$
$power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    $power \leftarrow power * x_0$
    $p \leftarrow p + a[i] * power$
return p

Polynomial Evaluation: Improvement

Can we do better?

We can do better by evaluating from right to left:

$p(x) = a_0 + a_1x^1 + \ldots + a_{n-1}x^{n-1} + a_nx^n$

Better brute-force algorithm

$p \leftarrow a[0]$
$power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    $power \leftarrow power * x_0$
    $p \leftarrow p + a[i] * power$
return p

Efficiency:

Polynomial Evaluation: Improvement

Can we do better?

We can do better by evaluating from right to left:

$p(x) = a_0 + a_1x^1 + \ldots + a_{n-1}x^{n-1} + a_nx^n$

Better brute-force algorithm

$p \leftarrow a[0]$
$power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    $power \leftarrow power * x_0$
    $p \leftarrow p + a[i] * power$
return p

Can we do any better?

Efficiency: O(n)

Closest-Pair Problem

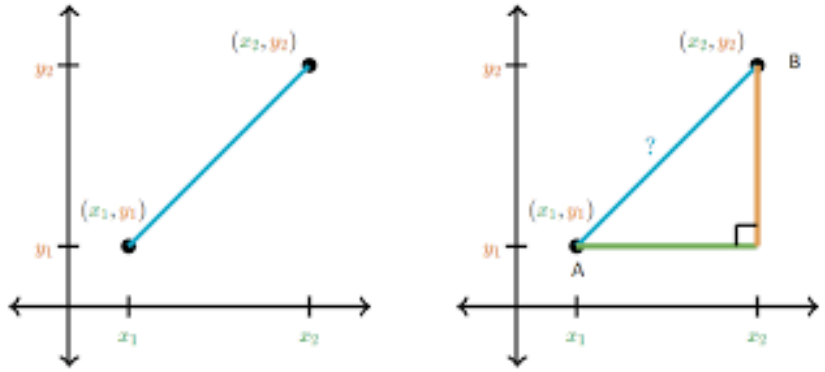
- Find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

- <u>Brute-force algorithm</u>

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

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

euclidean vs
manhattan distance



Closest-Pair Brute-Force Algorithm

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$

//Output: Indices *index*1 and *index*2 of the closest pair of points

$dmin \leftarrow \infty$

for $i \leftarrow 1$ **to** $n - 1$ **do**

  for $j \leftarrow i + 1$ **to** n **do**

    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function

    if $d < dmin$

      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$

return *index*1, *index*2

Efficiency: $O(n^2)$

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i)$$

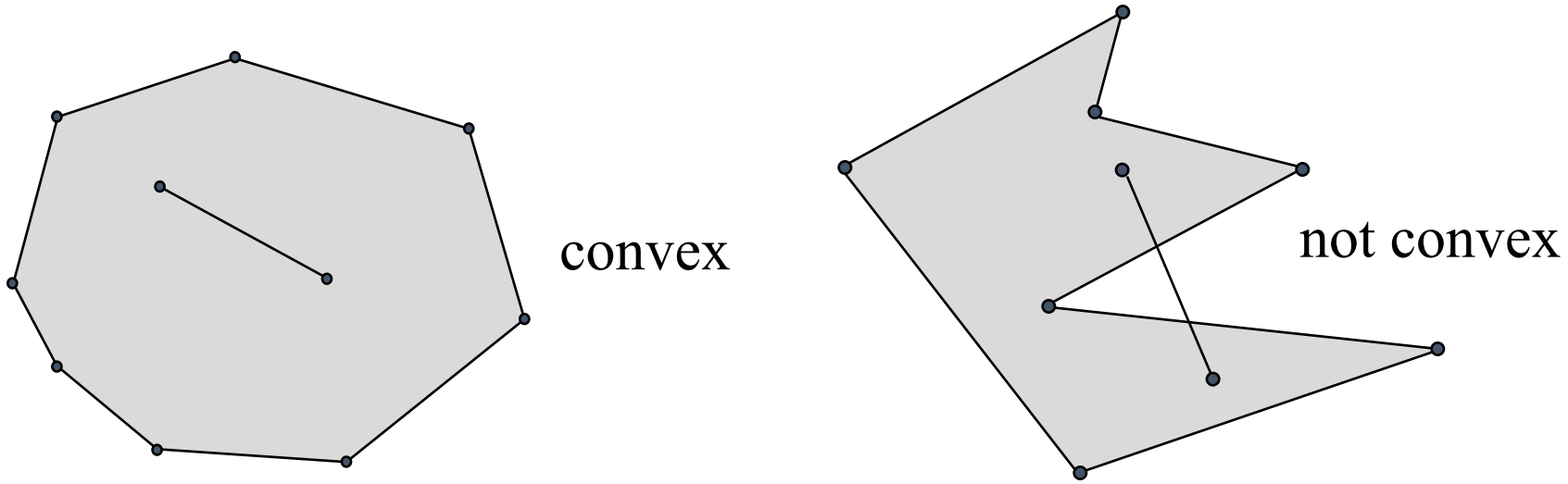
$$= 2[(n - 1) + (n - 2) + \cdots + 1] = (n - 1)n$$

Can you make it faster?

We will find out later…

Convex-Hull Problem

- Definition
  - A set of points (finite or infinite) in the plane is called convex if for any two points *P* and *Q* in the set, the entire line segment with the endpoints at *P* and *Q* belongs to the set



Examples of Convex Sets

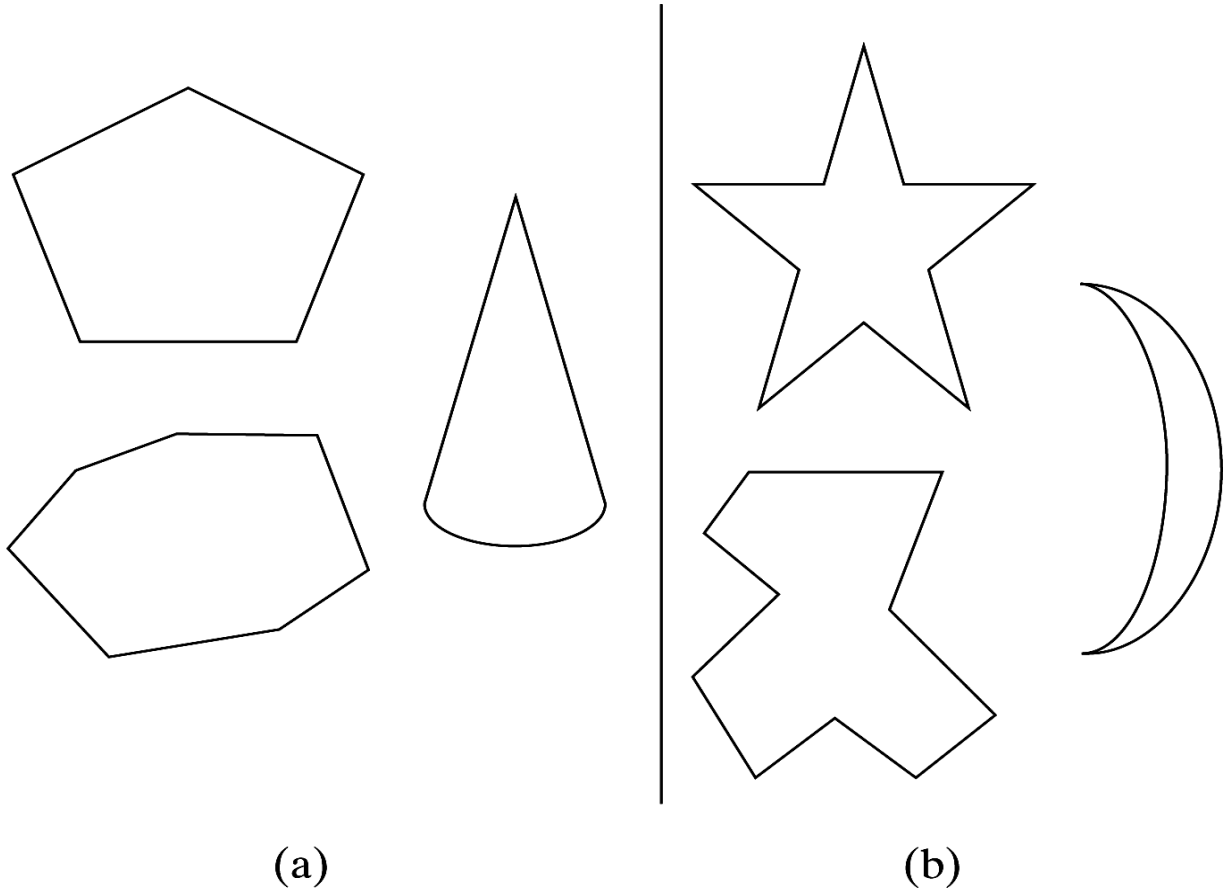
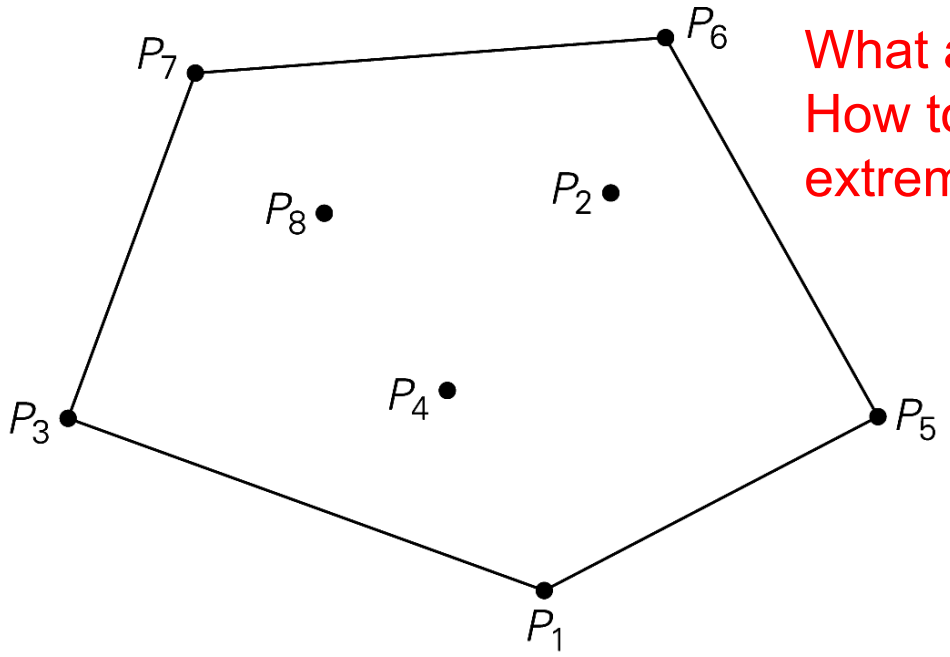


FIGURE 3.4 (a) Convex sets. (b) Sets that are not convex.

Convex-Hull Problem

- Extreme points



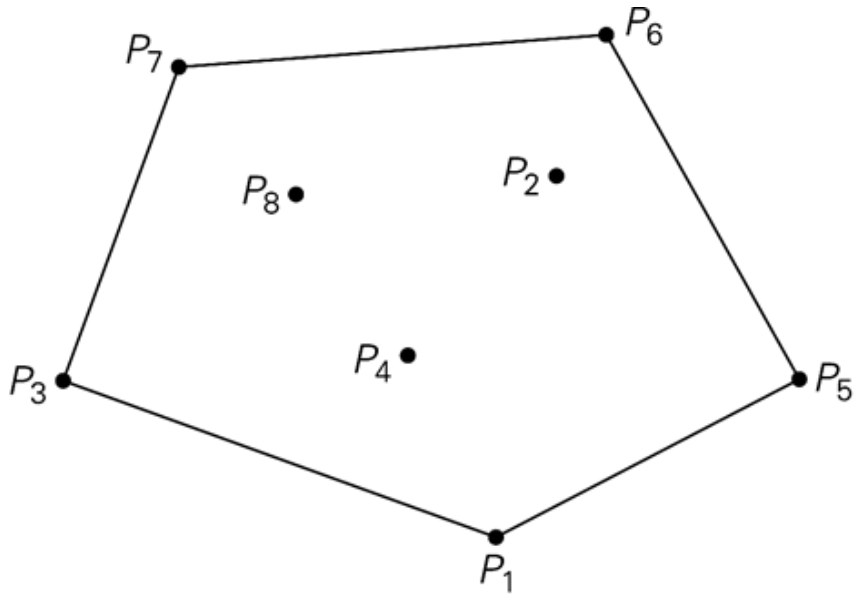
What are extreme points?
How to identify those extreme points?

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.

Convex-Hull Problem

- Brute-Force Convex-Hull Algorithm

- Find the pairs of points (P_i,P_j) from a set of n points
  - The line segment connecting P_i *and* P_j is a part of its convex hull's boundary if and only if the other points of the set lie on the same side of the straight line through P_i *and* P_j



Convex hull brute force algorithm

- The straight line through two points (x1, y1), (x2, y2) in the coordinate plane can be defined by the following equation
  - ax + by = c

    where a = y2 – y1, b = x1 – x2, c = x1*y2 - y1*x2

- Such a line divides the plane into two half-planes: for all the points in one of them: ax + by $\geq$ c, while for all the points in the other, ax + by $\leq$ c.

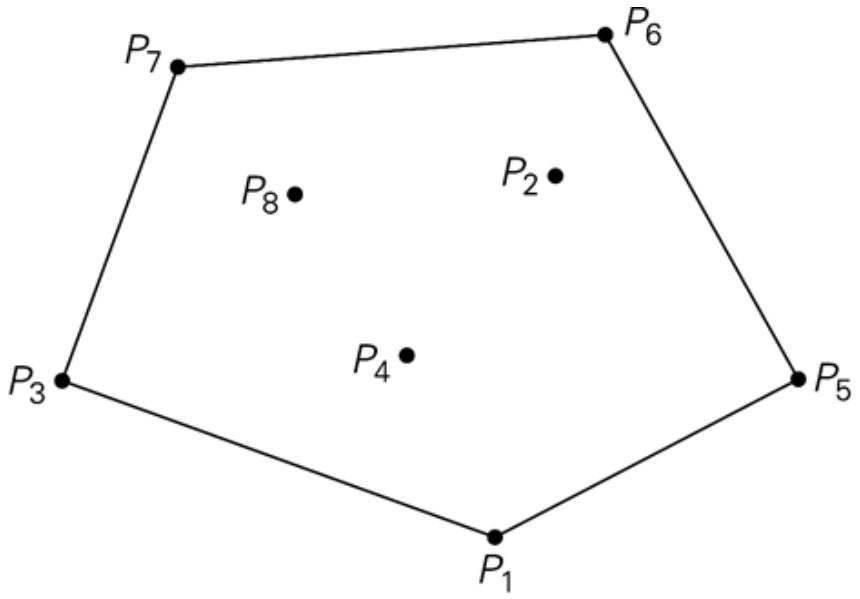
Convex hull brute force algorithm

- Algorithm: For each pair of points p_i and p_j determine whether all other points lie to the same side of the straight line through p_i and p_j, i.e. whether ax+by-c all have the same sign (no 3+ points are co-linear)

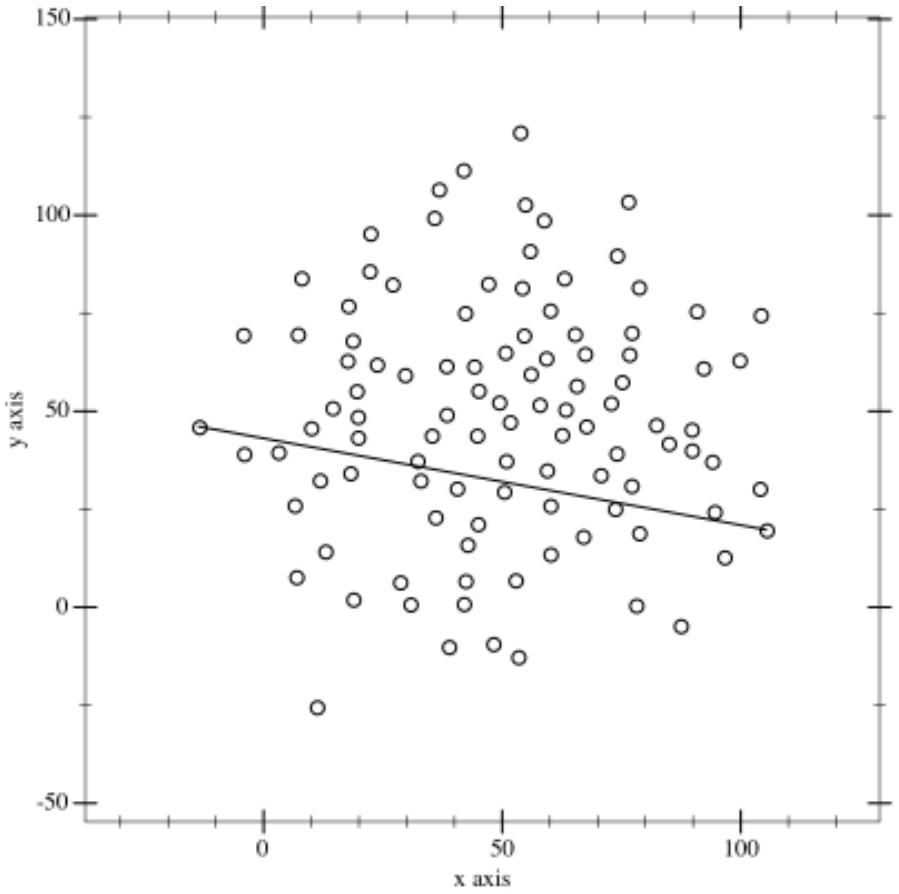
- Efficiency: $\Theta(n^3)$

- Can we do better?
  - Divide-and-conquer

  (see quickhull algorithm)



Quickhull



Worst-case quadratic.

For input precision that grows as a function of the number of points *n*, there was a conjectured worst case that was *n* log *h*, where *h* is the size of the convex hull. This was just disproved in October 2024!

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4989035

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

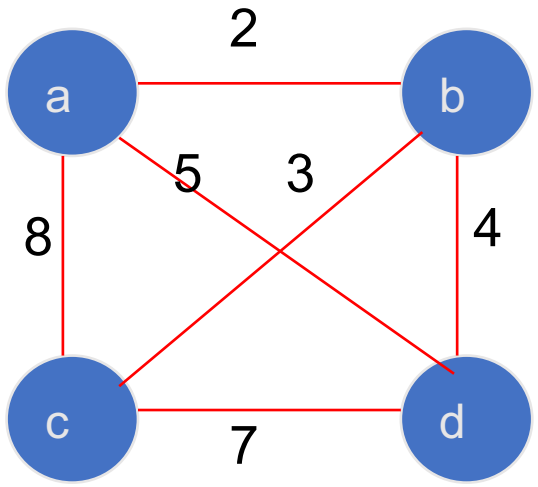
Exhaustive Search

- 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 (see algorithms in Sec. 5.4)
  - 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: Traveling Salesman Problem

- 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

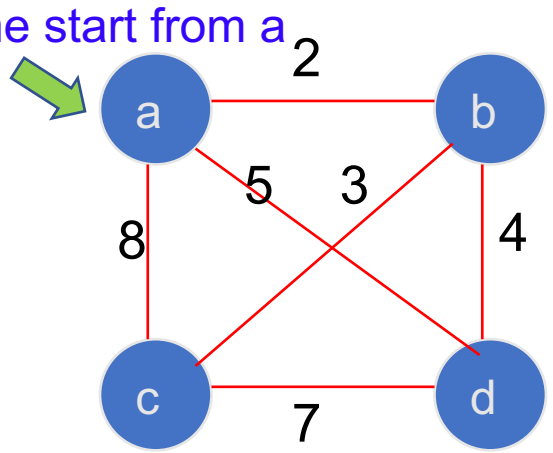
Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

Which one is the best?

More tours?

Less tours? **No other tours.**

Efficiency: **O(n!)**



Traveling Salesman Problem (TSP)

- Theoretical interest
  - Basic NP-complete problem (➔ not easy)
- TSP is easy enough to explain, but it is very difficult to solve.

- Practical (real world) application for TSP
  - Vehicle routing problem
  - Genome sequencing
  - Manufacturing
  - Plane routing
  - Telephone routing
  - Job sequencing

Variants (just FYI)

- Euclidean Traveling Salesman Selection Problem
- Asymmetric Traveling Salesman Problem
- Symmetric Wandering Salesman Problem
- Selective Traveling Salesman Problem
- TSP with distances 1 and 2, TSP(1,2)
- K-template Traveling Salesman Problem
- Circulant Traveling Salesman Problem
- On-line Traveling Salesman Problem
- Time-dependent TSP
- The Angular-Metric Traveling Salesman Problem
- Maximum Latency TSP
- Minimum Latency Problem
- Max TSP
- Traveling Preacher Problem
- Bipartite TSP
- Remote TSP
- Precedence-Constrained TSP
- Exact TSP
- The Tour Cover problem
- ...

Example: Knapsack Problem

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

Input:

Output:

- 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

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

Other combinations?

Efficiency:

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

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

Other combinations?

Efficiency: $O(2^n)$

What is Brute Force?

- Attempting to guess the correct solution by trying all, or a chosen subset of all possible options

- Runs through the entire available space
  - Combination
  - Permutation

- One of the oldest and easiest types of design approach
- Generally viewed as the easy way in, but rarely yields good time efficiency.

Example: 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 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	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

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

cost matrix

one person per job

How many possible ways of assignment?

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.	etc.

(The optimal assignment can be found in **cubic time** via the "*Hungarian algorithm*".)

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

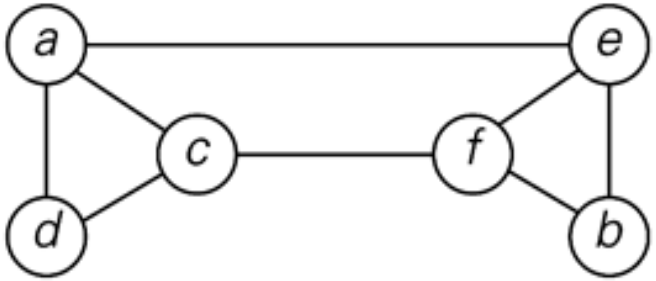
- In many cases, exhaustive search or its variation is the only known way to get exact solution

Graph Traversal

Many problems require processing all graph vertices (and/or edges) in systematic fashion

Graph traversal algorithms:

- Depth-first search (DFS)
- Breadth-first search (BFS)



Q: how we represent a graph?
Q: what kind of data structure we can use?

Recall in chapter 1:
adjacency matrix
adjacency list

Depth-First Search (DFS)

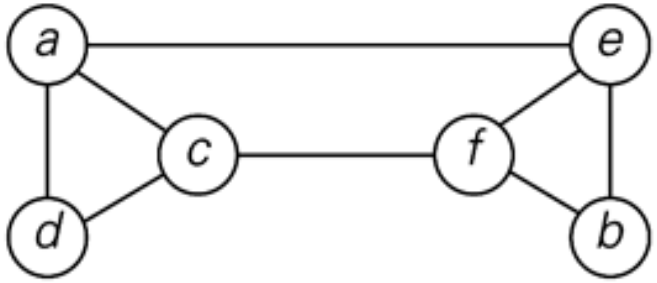
- Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.

- Uses a stack (from CSC 280)
  - a vertex is pushed onto the stack when it's reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex

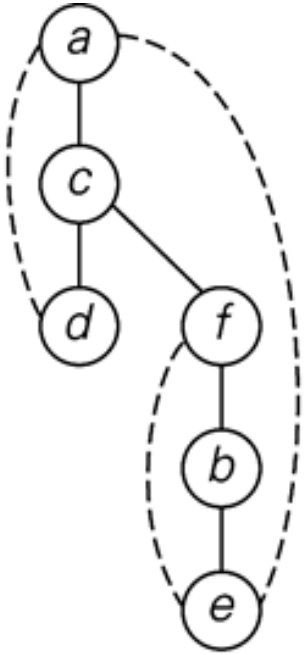
- "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

Example: DFS traversal of undirected graph

Given such graph: G=(V, E)



DFS tree:



DFS traversal stack:

$e_{6,2}$
$b_{5,3}$
$d_{3,1}$ $f_{4,4}$
$c_{2,5}$
$a_{1,6}$

order of visited

order of becoming a dead node

back edges: shown with dashed lines

Pseudocode of DFS

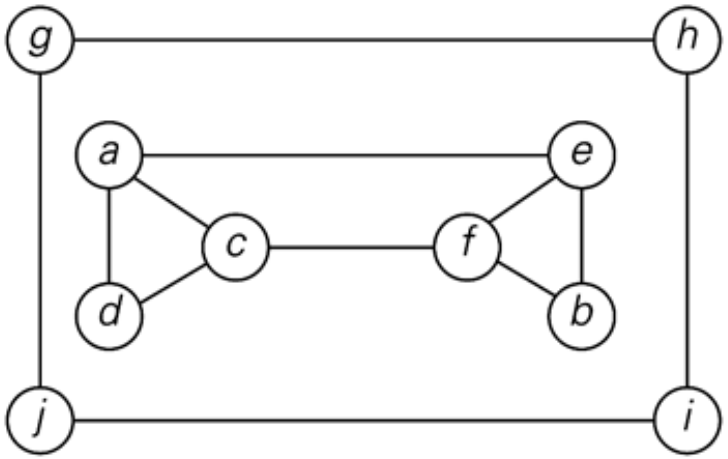
ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

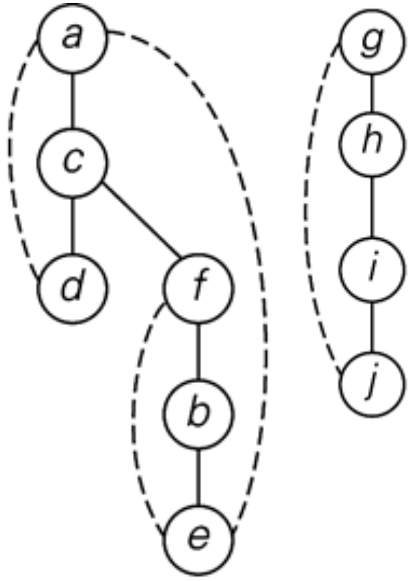
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)

Example: DFS traversal of undirected graph

Given such graph: G=(V, E)



DFS tree:



DFS traversal stack:

$e_{6,2}$
$b_{5,3}$ $j_{10,7}$
$d_{3,1}$ $f_{4,4}$ $i_{9,8}$
$c_{2,5}$ $h_{8,9}$
$a_{1,6}$ $g_{7,10}$

Notes on DFS

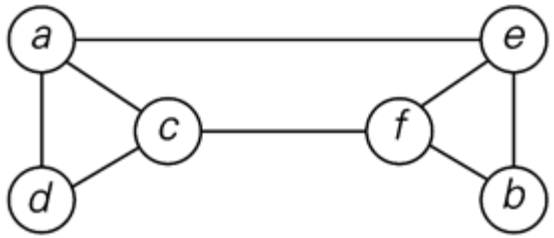
- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which vertices become dead-ends (popped off stack)
- Applications:
  - checking connectivity, finding connected components
  - checking acyclicity
  - finding articulation points and biconnected components
  - searching state-space of problems for solution (AI)

Breadth-first search (BFS)

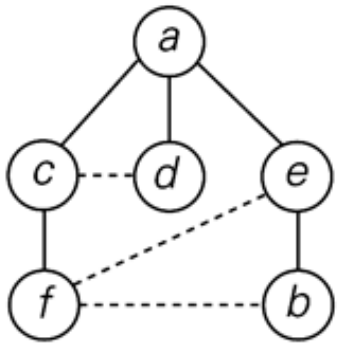
- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Example of BFS traversal of undirected graph

Given such graph: G=(V, E)



BFS tree:



BFS traversal queue:

$a_1\ c_2\ d_3\ e_4\ f_5\ b_6$

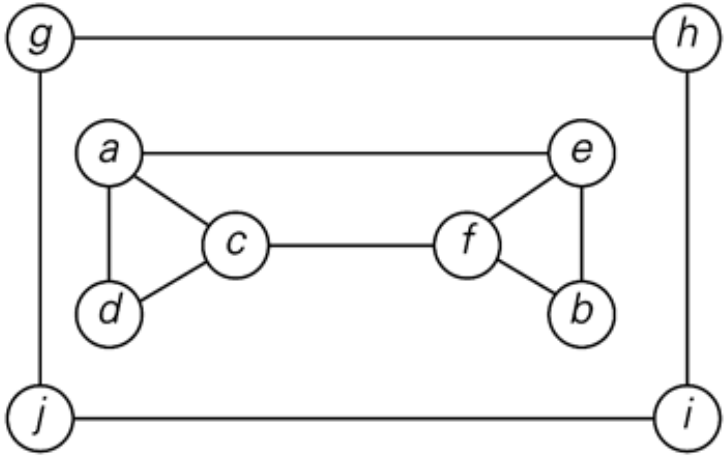
Pseudocode of BFS

ALGORITHM *BFS*(G)

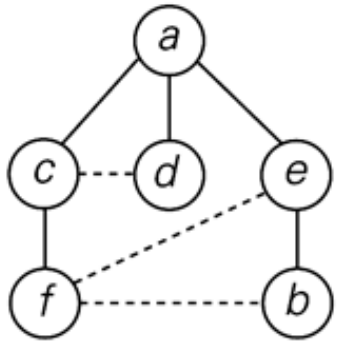
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

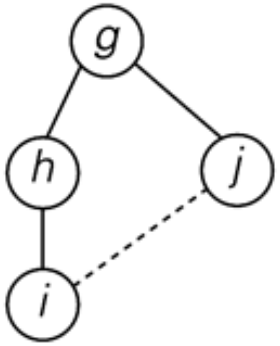
Example of BFS traversal of undirected graph

Given such graph: G=(V, E)



BFS tree:





BFS traversal queue:

$a_1\ c_2\ d_3\ e_4\ f_5\ b_6$

$g_7\ h_8\ j_9\ i_{10}$

Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$

- Yields single ordering of vertices (order added/deleted from queue is the same)

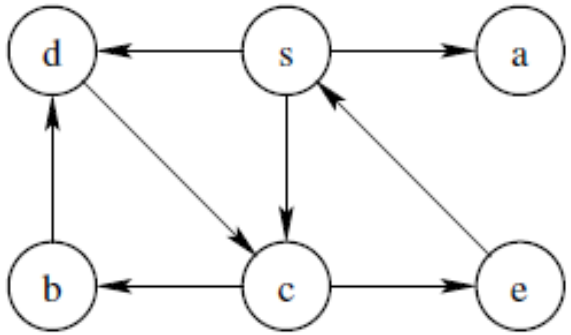
- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges

Example: DFS/BFS

- Find the visited node order for each type of graph search, starting with node s
  - Write the adjacency matrix for the graph
  - Write the adjacency linked list for the graph

- Depth First Search
  - Solution: s a c b d e

- Breadth First Search
  - Solution: s a c d b e



Summary: Brute Force Algorithms

- Brute Force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved
- In many cases, Brute Force does not provide you a very efficient solution
- Brute Force may be enough for small or moderate size problems with current computers