Computer Science 431
Algorithms
The College of Saint Rose
Spring 2015

Topic Notes: Brute-Force Algorithms

Our first category of algorithms are called *brute-force algorithms*.

Levitin defines brute force as a straightforward approach, usually based directly on the problem statement and definitions of the concepts involved.

We have already seen a few examples:

- consecutive integer checking approach for finding a GCD
- matrix-matrix multiplication

Another is the computation of a^n by multiplying by a n times.

Brute-force algorithms are not usually clever or especially efficient, but they are worth considering for several reasons:

- The approach applies to a wide variety of problems.
- Some brute-force algorithms are quite good in practice.
- It may be more trouble than it's worth to design and implement a more clever or efficient algorithm over using a straightforward brute-force approach.

Brute-Force Sorting

One problem we will return to over and over is that of *sorting*. We will first consider some brute-force approaches.

We will usually look at sorting arrays of integer values, but the algorithms can be used for other comparable data types.

Bubble Sort

We begin with a very intuitive sort. We just go through our array, looking at pairs of values and swapping them if they are out of order.

It takes $n - 1$ "bubble-ups", each of which can stop sooner than the last, since we know we bubble up one more value to its correct position in each iteration. Hence the name *bubble sort*.

```
bubble_sort(A[0..n-1]) {
  for (i = 0 to n-2)
    for (j=0 to n-2-i)
      if (A[j+1] < A[j]) swap A[j] and A[j+1]
}
```

The size parameter is n, the size of the input array.

The basic operation is either the comparison or the swap inside the innermost loop. The comparison happens every time, while the swap only happens when necessary to reorder a pair of adjacent elements. Remember that a swap involves three assignments, which would be more expensive than the individual comparisons.

The best, average, and worst case for the number of comparisons is all the same. But for swaps, it differs. Best case, the array is already sorted and we need not make any swaps. Worst case, every comparison requires a swap. For an average case, we would need more information about the likelihood of a swap being needed to do any exact analysis.

So we proceed by counting comparisons, and the summation will also give us a the worst case for the number of swaps.

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 \\ &= \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] \\ &= \sum_{i=0}^{n-2} (n-1-i) \\ &= \frac{(n-1)n}{2} \in \Theta(n^2). \end{aligned}$$

So we do $\Theta(n^2)$ comparisons. We swap, potentially, after each one of these, a worse case behavior of $\Theta(n^2)$ swaps.

Selection Sort

A simple improvement on the bubble sort is based on the observation that one pass of the bubble sort gets us closer to the answer by moving the largest unsorted element into its final position. Other elements are moved "closer" to their final position, but all we can really say for sure after a single pass is that we have positioned one more element.

So why bother with all of those intermediate swaps? We can just search through the unsorted part of the array, remembering the index of (and hence, the value of) the largest element we've seen so far, and when we get to the end, we swap the element in the last position with the largest element we found. This is the *selection sort*.

```
selection_sort(A[0..n-1]) {
  for (i = 0 to n-2)
    min = i
    for (j=i+1 to n-1)
      if (A[j] < A[min]) min = j;
    swap A[i] and A[min]
}
```

The number of comparisons is our basic operation here.

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-2} 1 \\ &= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \\ &= \sum_{i=0}^{n-2} (n-1-i) \\ &= \frac{(n-1)n}{2} \in \Theta(n^2). \end{aligned}$$

Here, we do the same number of comparisons, but at most $n - 1 = \Theta(n)$ swaps.

Brute-Force String Match

The *string matching* problem involves searching for a *pattern* (substring) in a string of *text*.

The basic procedure:

1. Align the pattern at beginning of the text
2. Moving from left to right, compare each character of the pattern to the corresponding character in the text until
   - all characters are found to match (successful search); or
   - a mismatch is detected
3. While pattern is not found and the text is not yet exhausted, realign the pattern one position to the right and repeat Step 2

The result is either the index in the text of the first occurrence of the pattern, or indices of all occurrences. We will look only for the first.

Written in pseudocode, our brute-force string match:

```
brute_force_string_match(T[0..n-1], P[0..m-1]) {
  for (i=0 to n-m)
    j=0
    while (j<m) and P[j] == T[i+j]
      j++
    if j==m return i
  return -1
}
```

The sizes of the input strings, m for the pattern and n for the text, are the problem size parameters and the basic operation is the element to element comparisons.

We have significant differences among the best, average, and worst cases. Best case, we find the pattern at the start of the text and we need only m comparisons: $\Theta(m)$. In the worst case, we encounter "near misses" at each starting location in the text, requiring about n searches of size m, resulting in $\Theta(nm)$. On average, however, most non-matches are likely to be detected quickly: in the first element or two, leading to an average case behavior of $\Theta(n)$.

We will consider improvements for string matching later in the semester.

In class exercise: Exercise 3.1.4, p. 102

Closest Pairs

Computational geometry is a rich area for the study of algorithms. Our next problem, *closest pairs*, comes from that field.

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

Brute-force algorithm: Compute the Euclidean distance between every pair of distinct points and return the indices of the points for which the distance is the smallest.

```
brute_force_closest_points(a set of n points, P) {
  dmin = infinity
  for (i=1 to n-1)
    for (j=i+1 to n)
      d = sqrt(P[i].x - P[j].x)^2 + (P[i].y - P[j].y)^2))
      if (d < dmin)
        dmin = d
        index1 = i
        index2 = j
  return index1, index2
}
```

The problem size is defined by the number of points, n. The basic operation is the computation of the distance between each pair of points. It needs to be computed for each pair of points, so the best, average, and worst cases are the same.

Before we continue, however, note that there is a way to improve the efficiency of this algorithm significantly. Computing square roots is an expensive operation. But if we think about it a bit, it's also completely unnecessary here. Finding the minimum among a collection of square roots is exactly the same as finding the minimum among the original numbers. So it can be removed. This leaves us with the squaring of numbers are our basic operation.

The number of squaring operations can be computed:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 \\ &= 2 \sum_{i=1}^{n-1} (n-i) \\ &= 2[(n-1) + (n-2) + \cdots + 1] = 2 \left[\frac{(n-1)n}{2} \right] \in \Theta(n^2). \end{aligned}$$

Convex Hulls

Staying in the realm of computational geometry we consider the *convex-hull problem*.

The *convex hull* of a set of points in the plane is the smallest convex polygon that contains them all.

So given a set of points, how can we find the convex hull? First, we should find the *extreme points* – those points in our set that lie "on the fringes" which will be the vertices of the polygon formed by the points in the convex hull. Second, we need to know in what order to connect them to form the convex polygon.

Our brute-force approach is anything but obvious. Our solution will depend on the observation that any line segment connecting two adjacent points on the convex hull will have all other points in the set on the same side of the straight line between its endpoints.

Recall from your high school math that the straight line through two points (x_1, y_1) and (x_2, y_2) can be defined by

$$ax + by = c, a = y_2 - y_1, b = x_1 - x_2, c = x_1 y_2 - y_1 x_2.$$

We can tell which side of this line any point is by computing $ax + by$ and seeing if it is greater than or less than c. All points where $ax + by < c$ are on one side of the line, those where $ax + by > c$ are on the other. Points on the line will satisfy $ax + by = c$, of course.

So our algorithm will need to consider each pair of points, and see if all other points lie on the same side. If so, the points are part of the convex hull. If not, they are not.

```
brute_force_convex_hull(a set of n points, P) {
  create empty set of line segments L
  for (each point p1 in P)
    for (each point p2 in P after p1)
      a = p2.y - p1.y; b = p1.x - p2.x;
      c = p1.x*p2.y - p1.y*p2.x
      foundProblem = false
      for (each point p3 in P (not p1 or p2))
        check = a*p3.x + b*p3.y - c
        if (check does not match others)
          foundProblem=true
          break
      if (!foundProblem) add segment p1,p2 to L
  extract and return list of points from L
}
```

See Example:
`/home/cs431/examples/BruteForceConvexHull`

For this problem, the size is determined by the number of input points. Its basic operation is the compute of the check in the innermost loop (probably multiplications).

For best, average, and worst case behavior, the differences would arise in how quickly we could determine that a given segment is not part of the convex polygon for those that are not. Best case, we would quickly find evidence for those which are not. Worst case, we need to look through most of the points before making a determination.

Considering the worst case, we can see that the number of basic operations will be $\Theta(n^3)$.

Exhaustive Search

An *exhaustive search* is 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.

The typical approach involves:

1. Generation of a list of all potential solutions to the problem in a systematic manner (*e.g.*, our permutations from the first problem set – we will see others).
2. Evaluation of the potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far.

3. Announcing the solution when the search ends.

We will look briefly at a few examples of problems that can be solved with exhaustive search.

Traveling Salesman

The *traveling salesman problem (TSP)* asks us, 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.

Solutions to this kind of problem have important practical applications (delivery services, census takers, etc.)

An alternative statement of the problem is to find the shortest Hamiltonian circuit in a weighted, connected graph. A *Hamiltonian circuit* is a cycle that passes through all of the vertices of the graph exactly once.

If we think about it a bit, we can see that it does not matter where our tour begins and ends – it's a cycle – so we can choose any city/vertex as the starting point and consider all permutation of the other $n - 1$ vertices.

So an exhaustive search would allow us to consider all $(n - 1)!$ permutations, compute the cost (total distance traveled) for each, returning the lowest cost circuit found.

The text has a small example.

Knapsack Problem

Another well-known problem is the *knapsack problem*.

Given n items with weights $w_1, w_2, ..., w_n$ and values $v_1, v_2, ..., v_n$, what is the most vaulable subset of items that can fit into a knapsack that can hold a total weight W.

The exhaustive search here involves considering all possible subsets of items and finding the subset whose total weight is no more than W with the highest value.

For n items, this means we have to generate 2^n subsets, giving us that as a lower bound on our cost of $\Omega(2^n)$.

An example of this problem with a knapsack capacity of 16:

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

All possible subsets (not including the empty set, which would be the answer only if all items were too heavy to be in the knapsack even alone):

subset	weight	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	too heavy
{1,2,4}	12	60
{1,3,4}	17	too heavy
{2,3,4}	20	too heavy
{1,2,3,4}	22	too heavy

So the winner is {2,3} with a total weight of 15 and a total value of 80.

Assignment Problem

Our final exhaustive search example is the *assignment problem*. It may be stated as the assignment of n jobs among n people, one job per person, where the cost of assigning person i to job i is given by $C[i, j]$, and we wish to minimize the total cost across all possible assignments.

For example, consider this cost matrix:

	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

Our exhaustive search here involves the generation of all possible assignments, computing the total cost of each, and selection of the lowest cost.

How many assignments as possible? Well, we can assign the first person any of n jobs. Then, there are $n - 1$ to choose for the second person, $n - 2$ for the third, and so on, until there is just one remaining choice for the last person: a total of $n!$.

We can list the solutions by generating all permutations of the numbers 1, 2, ..., n (sound familiar?), and treating the number in each position as the job to assign to the person at that position.

Exhaustive Search Wrapup

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances.

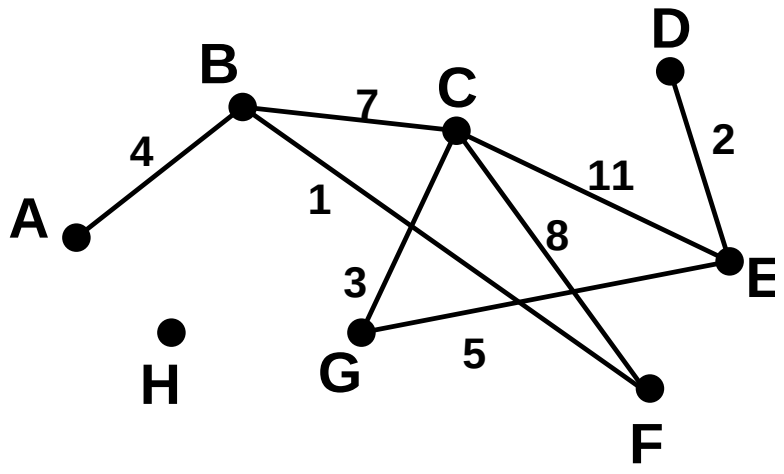
- Sometimes, much better approaches exist (and we will see some of those soon).
- But sometimes, an exhaustive search is the only known way to be guaranteed an exact solution.

Graph Traversals

We return to graph structures for our next group of algorithms. In particular, we consider the *graph traversal* problem, that of visiting all vertices in a graph.

The two main approaches are the *depth-first search (DFS)* and the *breadth-first search (BFS)*.

We will examine the ideas using the example graph we have seen before.



In either case, we will choose a starting vertex, and visit all other vertices in the same connected component of the graph as that starting vertex. Either traversal will visit all vertices in the connected component, but each will visit the vertices in a different order.

Depth-first Traversal

A depth-first traversal proceeds by moving from the last visited vertex to an adjacent unvisited one. If no unvisited adjacent vertex is available, the traversal backtracks to a previously visited vertex which does have an unvisited adjacent vertex. When we have backtracked all the way to the starting vertex and no further adjacent vertices remain unvisited, the algorithm terminates.

For the example graph, the following is a DFT starting at A:

A, B, C, E, D, G, F

Note that we do not visit the disconnected vertex H.

If we wanted to perform a DFT that includes all vertices, we could restart the algorithm with an unvisited vertex.

See the algorithm on p. 124-125.

Note the implicit use of a stack (here, the call stack to manage the recursion).

The following will perform a DFT of the vertices in the same connected component as the starting vertex:

```
dfs(G=(V,E), a starting vertex s)
  create empty stack L
  L.push(s)
  while (L.notEmpty)
    v = L.pop()
    if (v not yet visited)
      visit(v)
      for each vertex w, adjacent to v
         if (w not yet visited)
           L.push(w)
```

The efficiency class of this algorithm depends on which graph representation is used.

For an adjacency matrix, we have $\Theta(|V^2|)$, and for adjacency list, we have $\Theta(|V| + |E|)$.

The intuition behind this is that for each vertex in the graph, we have to visit each incident edge. With an adjancency matrix, this involves looking at each of $|V|$ matrix entries (including those which are null, indicating that such an edge does not exist). For the adjancency list, we have the exact list of edges readily available, so across all vertices, we visit $\Theta(|E|)$ total edges.

Breadth-first Traversal

A breadth-first traversal proceeds by visiting all adjacent vertices of the last visited vertex before proceeding to any subsequent adjacencies.

For the example graph, the following is a BFT starting at A:

A, B, C, F, G, E, D

See the text's algorithm for BFT on p. 126.

Note that this proceeds by visiting the starting vertex, then the immediate neighbors of the starting vertex, then the neighbors of those neighbors, and so on.

We accomplish this with the queue structure. Note that the DFT algorithm above can be converted to a BFT simply by swapping the stack for a queue:

```
bfs(G=(V,E), a starting vertex s)
  create empty queue L
  L.enqueue(s)
  while (L.notEmpty)
    v = L.dequeue()
```

```
if (v not yet visited)
  visit(v)
  for each vertex w, adjacent to v
     if (w not yet visited)
       L.enqueue(w)
```

The efficiency classes of this algorithm are the same as those of DFT, and depend on which graph representation is used.

For an adjacency matrix, we again have $\Theta(|V^2|)$, and for adjacency list, we have $\Theta(|V| + |E|)$.