COSC1285/2123: Algorithms & Analysis

Brute Force

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 3

Outline

Overview

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 3.

Learning Outcomes:

- Understand the *Brute Force* algorithmic approach.
- Understand and apply:
  - Sorting - selection and bubble sort.
  - Exhaustive search - knapsack.
  - Graph search - DFS, BFS.
- Study a case study of using a brute force approach to solve a problem.

Brute Force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved, and can involve enumerating all solutions and selecting the best one.

Examples:

1. Computing a^n (multiple 'a' n times)
2. Searching for a key of a given value in an unsorted list.

Overview

Sorting

Examples:

- Telephone book – Sorted by surname.
- Height in class – Tallest to shortest.

Sorting

Examples:

- Telephone book – Sorted by surname.
- Height in class – Tallest to shortest.

Why?

- Important to build efficient searching algorithms and data structures, data compression.
- Heavily studied problem in computer science, with several widely celebrated algorithms.

Sorting

Examples:

- Telephone book – Sorted by surname.
- Height in class – Tallest to shortest.

Why?

- Important to build efficient searching algorithms and data structures, data compression.
- Heavily studied problem in computer science, with several widely celebrated algorithms.

Sorting Problem

Given a sequence of n elements $x_1, x_2, \ldots, x_n \in S$, rearrange the elements according to some ordering criteria.

Brute Force Sorting: Selection Sort

Selection sort is a brute force solution to the sorting problem.

Brute Force Sorting: Selection Sort

Selection sort is a brute force solution to the sorting problem.

1. Scan all n elements of the array to find the **smallest** element, and *swap* it with the *first element*.
2. Starting with the second element, scan the remaining $n - 1$ elements to find the **smallest** element and *swap* it with the element in the second position.
3. Generally, on pass $i (0 \leq i \leq n - 2)$, find the **smallest** element in $A[i \ldots n-1]$ and *swap* it with $A[i]$.

Selection Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

COMPARES $\boxed{0}$

Selection Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

COMPARES $\boxed{0}$ +11

Selection Sort Example

1		21	15	25	12	6	8	3	5	19	10	18

COMPARES $\boxed{11}$ +10

Selection Sort Example

1	3		15	25	12	6	8	21	5	19	10	18

COMPARES $\boxed{21}$ +9

Selection Sort Example

1	3	5

25	12	6	8	21	15	19	10	18

COMPARES $\boxed{30}$ +8

Selection Sort Example

1	3	5	6	12	25	8	21	15	19	10	18

COMPARES $\boxed{38}$ +7

Selection Sort Example

1	3	5	6	8		25	12	21	15	19	10	18

COMPARES $\boxed{45}$ +6

Selection Sort Example

1	3	5	6	8	10	12	21	15	19	25	18

COMPARES $\boxed{51}$ +5

Selection Sort Example

1	3	5	6	8	10	12

21	15	19	25	18

COMPARES $\boxed{56}$ +4

Selection Sort Example

1	3	5	6	8	10	12	15		21	19	25	18

COMPARES $\boxed{60}$ +3

Selection Sort Example

1	3	5	6	8	10	12	15	18	19	25	21

COMPARES $\boxed{63}$ +2

Selection Sort Example

1	3	5	6	8	10	12	15	18	19	25	21

COMPARES $\boxed{65}$ +1

Selection Sort Example

1	3	5	6	8	10	12	15	18	19	21	25

COMPARES $\boxed{66}$

Selection Sort

ALGORITHM **SelectionSort** ($A[0 \ldots n-1]$)
/* Order an array using a brute-force selection sort. */
/* INPUT : An array $A[0 \ldots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \ldots n-1]$ 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] then
            min = j
        end if
    end for
    swap A[i] and A[min]
end for
```

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) =$$

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 =$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) =$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

- Needs around $n^2/2$ comparisons and at most $n-1$ exchanges.
- The running time is insensitive to the input, so the best, average, and worst case are essentially the same (Why?).

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

- Needs around $n^2/2$ comparisons and at most $n-1$ exchanges.
- The running time is insensitive to the input, so the best, average, and worst case are essentially the same (Why?).

Why use it?

- Selection sort only makes $\mathcal{O}(n)$ writes but $\mathcal{O}(n^2)$ reads.
- When writes (to array) are much more expensive than reads, selection sort may have an advantage, e.g., flash memory.
- Also, for small arrays, selection sort is relatively efficient and simple to implement.

Stable Sorting

- **Definition**: A sorting method is said to be *stable* if it preserves the relative order of duplicate keys in the file.

Before Sorting	After Sorting
1 Adams	1 Adams
2 Black	1 Smith
4 Brown	2 Black
2 Jackson	2 Jackson
4 Jones	2 Washington
1 Smith	3 White
4 Thompson	3 Wilson
2 Washington	4 Brown
3 White	4 Jones
3 Wilson	4 Thompson

Not all sorting methods are stable.

Is Selection Sort Stable?

Question: Is selection sort stable?

Consider the following examples and apply selection sort on them:

5,5,3,2

`https://goo.gl/forms/gPms042iSsOFWnDv2`

Brute Force Sorting: Bubble Sort

A **bubble sort** iteratively **compares** *adjacent* items in a list and **exchanges** them if they are out of order.

Brute Force Sorting: Bubble Sort

A **bubble sort** iteratively **compares** *adjacent* items in a list and **exchanges** them if they are out of order.

Motivation:

- One of the classic (and elementary) sorting algorithms, originally designed and efficient for tape disks, but with random access memory, it doesn't have much use these days.
- But insightful to study it and to understand why other sorting algorithms are superior in one or more aspects.
- It is simple to code.

Bubble Sort

1. First iteration, compare each adjacent pair of elements and swap them if they are out of order. Eventually **largest** element gets propagated to the end.
2. Second iteration, repeat the process, but only from first to 2nd last element (last element is in its correct position). Eventually **second largest** element is at the 2nd last element.
3. Repeat until all elements are sorted.

Bubble Sort Example – Round 1

15	21	1	25	12	6	8	3	5	19	10	18

COMPARES $\boxed{0}$

Bubble Sort Example – Round 1

15	21	1	25	12	6	8	3	5	19	10	18

COMPARES	1

Bubble Sort Example – Round 1

15	21	1	25	12	6	8	3	5	19	10	18

COMPARES $\boxed{2}$

Bubble Sort Example – Round 1

15	1	21	25	12	6	8	3	5	19	10	18

COMPARES $\boxed{3}$

Bubble Sort Example – Round 1

15	1	21	25	12	6	8	3	5	19	10	18

COMPARES	4

Bubble Sort Example – Round 1

15	1	21	12	25	6	8	3	5	19	10	18

COMPARES $\boxed{5}$

Bubble Sort Example – Round 1

15	1	21	12	6	25	8	3	5	19	10	18

COMPARES $\boxed{6}$

Bubble Sort Example – Round 1

15	1	21	12	6	8	25	3	5	19	10	18

COMPARES $\boxed{7}$

15	1	21	12	6	8	3	25	5	19	10	18

COMPARES $\boxed{8}$

Bubble Sort Example – Round 1

15	1	21	12	6	8	3	5	25	19	10	18

COMPARES $\boxed{9}$

Bubble Sort Example – Round 1

15	1	21	12	6	8	3	5	19	25	10	18

COMPARES $\boxed{10}$

Bubble Sort Example – Round 1

15	1	21	12	6	8	3	5	19	10		25	18

COMPARES $\boxed{11}$

Bubble Sort Example – Round 1

15	1	21	12	6	8	3	5	19	10	18	25

COMPARES $\boxed{11}$

Bubble Sort Example – Round 2

1	15	12	6	8	3	5	19	10	18	21	25

COMPARES $\boxed{21}$

Bubble Sort Example – Round 3

1	12	6	8	3	5	15	10	18	19	21	25

COMPARES $\boxed{30}$

Bubble Sort Example – Round 4

1	6	8	3	5	12	10	15	18	19	21	25

COMPARES $\boxed{38}$

Bubble Sort Example – Round 5

1	6	3	5	8	10	12	15	18	19	21	25

COMPARES $\boxed{45}$

Bubble Sort Example – Round 6

1	3	5	6	8	10	12	15	18	19	21	25

COMPARES 51

Bubble Sort Example – Sorted?

1	3	5	6	8	10	12	15	18	19	21	25

COMPARES $\boxed{51}$

Bubble Sort

ALGORITHM **BubbleSort** ($A[0 \ldots n-1]$)
/* Order an array using a bubble sort. */
/* INPUT : An array $A[0 \ldots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \ldots n-1]$ sorted in ascending order. */

```
for i = 0 to n − 2 do
    for j = 0 to n − 2 − i do
        if A[j + 1] < A[j] then
            swap A[j] and A[j + 1]
        end if
    end for
end for
```

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) =$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \frac{(n-1)n}{2}$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

- **Best case**: if original file is already sorted, about $n^2/2$ comparisons & 0 exchanges – $\mathcal{O}(n^2)$.
- **Worst case**: if original file is sorted in reverse order, about $n^2/2$ comparisons & $n^2/2$ exchanges – $\mathcal{O}(n^2)$.
- **Average case**: if original file is in random order, about $n^2/2$ comparisons & less than $n^2/2$ exchanges – $\mathcal{O}(n^2)$.

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i)-0+1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

- **Best case**: if original file is already sorted, about $n^2/2$ comparisons & 0 exchanges – $\mathcal{O}(n^2)$.
- **Worst case**: if original file is sorted in reverse order, about $n^2/2$ comparisons & $n^2/2$ exchanges – $\mathcal{O}(n^2)$.
- **Average case**: if original file is in random order, about $n^2/2$ comparisons & less than $n^2/2$ exchanges – $\mathcal{O}(n^2)$.

Is bubble sort stable?

Early-Termination Bubble Sort

- This modification attempts to reduce redundant iterations, by checking if any exchanges takes place in each pass. If there were no exchanges in the current iteration, the sorting is stopped after the current iteration.

Early-Termination Bubble Sort

- This modification attempts to reduce redundant iterations, by checking if any exchanges takes place in each pass. If there were no exchanges in the current iteration, the sorting is stopped after the current iteration.
- Why does this work?

Early-Termination Bubble Sort

- This modification attempts to reduce redundant iterations, by checking if any exchanges takes place in each pass. If there were no exchanges in the current iteration, the sorting is stopped after the current iteration.
- Why does this work?
- **Best case** - when the original file is already sorted, only one pass is needed, $n-1$ comparisons, 0 exchanges – $\mathcal{O}(n)$.
- **Worst case** - No improvement over the original implementation – $\mathcal{O}(n^2)$.
- **Average case** - Depending on the data set, few iterations can be eliminated at the end of the sort. Therefore, the number of passes is less than $n-1$, and hence cost is lower than the original implementation. The complexity is still likely to be $\mathcal{O}(n^2)$.

Overview

Exhaustive Search

A brute force approach involving *enumerating/generating* **all** possible solutions, then *selecting* the “best” one.

Exhaustive Search

A brute force approach involving *enumerating/generating* **all** possible solutions, then *selecting* the "best" one.

Method:

- Generate a list of all potential solutions to the problem in a systematic manner.
- Evaluate potential solutions one by one, disqualifying infeasible ones, and keeping track of the best one found so far.
- When all items have been evaluated, announce the best solution(s) found.

Exhaustive Search

A brute force approach involving *enumerating/generating* **all** possible solutions, then *selecting* the "best" one.

Method:

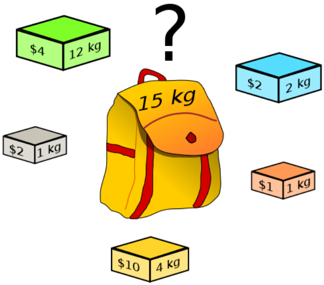
- Generate a list of all potential solutions to the problem in a systematic manner.
- Evaluate potential solutions one by one, disqualifying infeasible ones, and keeping track of the best one found so far.
- When all items have been evaluated, announce the best solution(s) found.

Typically applied to combinatorial problems, and insightful to study brute force solutions to them, as some problems can only be solved optimally by exhaustive search.

Knapsack Problem

Given n items of known weights $w_1, \ldots, w_n$ and the values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.[a]

[a] http://en.wikipedia.org/wiki/File:Knapsack.svg



Applications of Knapsack Problem

Brute Force Knapsack Algorithm

Algorithm:

1. Consider all subsets of the set of n items.

Brute Force Knapsack Algorithm

Algorithm:

1. Consider all subsets of the set of *n* items.
2. Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).

Brute Force Knapsack Algorithm

Algorithm:

1. Consider all subsets of the set of n items.
2. Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).
3. Find the subset of the largest value among them.

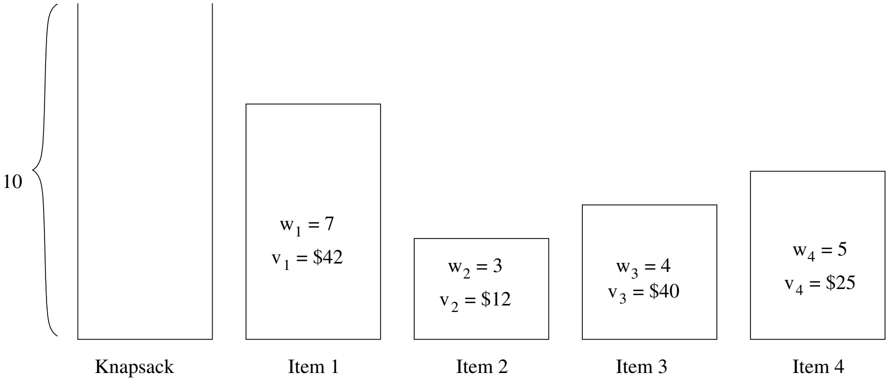
Brute Force Knapsack Algorithm

Algorithm:

1. Consider all subsets of the set of *n* items.
2. Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).
3. Find the subset of the largest value among them.

Complexity: Since the number of subsets of an n-element set is 2^n, an exhaustive search produces an $\mathcal{O}(2^n)$ algorithm.

Knapsack Problem

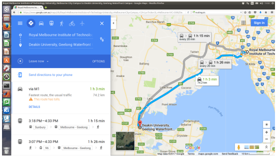


Knapsack Problem

Subset	Total Weight	Total Value
	0	\$0
$\{1\}$	7	\$42
$\{2\}$	3	\$12
$\{3\}$	4	\$40
$\{4\}$	5	\$25
$\{1,2\}$	10	\$36
$\{1,3\}$	11	Not Possible
$\{1,4\}$	12	Not Possible
$\{2,3\}$	7	\$52
$\{2,4\}$	8	\$37
$\{3,4\}$	9	\$65
$\{1,2,3\}$	14	Not Possible
$\{1,2,4\}$	15	Not Possible
$\{1,3,4\}$	16	Not Possible
$\{2,3,4\}$	12	Not Possible
$\{1,2,3,4\}$	19	Not Possible

Overview

Searching in Graphs

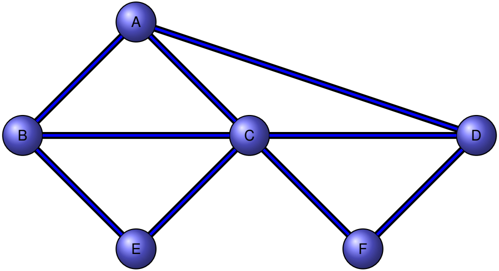


(a) How to find the shortest path in a road network?

(b) How to find a path through a maze?

(c) How to determine if a power network is connected?

Graph example



Depth-First Search – Overview

Depth-First Search (DFS) - Traversal:

Depth-First Search (DFS) - Traversal:

1. Choose an arbitrary vertex and mark it visited.

Depth-First Search – Overview

Depth-First Search (DFS) - Traversal:

1. Choose an arbitrary vertex and mark it visited.
2. From the current vertex, proceed to an unvisited, adjacent vertex and mark it visited.

Depth-First Search – Overview

Depth-First Search (DFS) - Traversal:

1. Choose an arbitrary vertex and mark it visited.
2. From the current vertex, proceed to an unvisited, adjacent vertex and mark it visited.
3. Repeat 2nd step until a vertex is reached which has no adjacent, unvisited vertices (dead-end).

Depth-First Search – Overview

Depth-First Search (DFS) - Traversal:

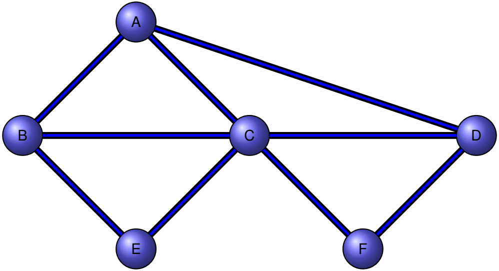
1. Choose an arbitrary vertex and mark it visited.
2. From the current vertex, proceed to an unvisited, adjacent vertex and mark it visited.
3. Repeat 2nd step until a vertex is reached which has no adjacent, unvisited vertices (dead-end).
4. At each dead-end, backtrack to the last visited vertex and proceed down to the next unvisited, adjacent vertex.

Depth-First Search – Overview

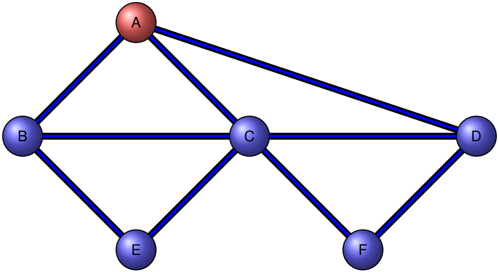
Depth-First Search (DFS) - Traversal:

1. Choose an arbitrary vertex and mark it visited.
2. From the current vertex, proceed to an unvisited, adjacent vertex and mark it visited.
3. Repeat 2nd step until a vertex is reached which has no adjacent, unvisited vertices (dead-end).
4. At each dead-end, backtrack to the last visited vertex and proceed down to the next unvisited, adjacent vertex.
5. The algorithm halts there are no unvisted vertices.

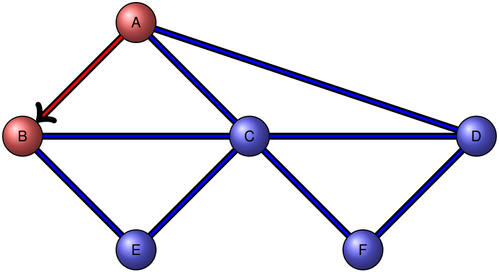
Depth-First Search – Example



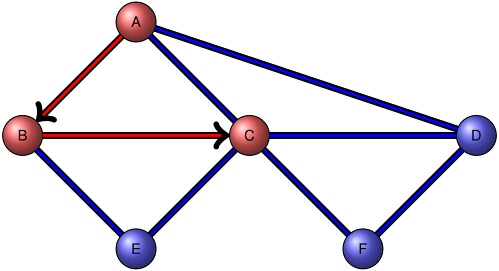
Depth-First Search – Example



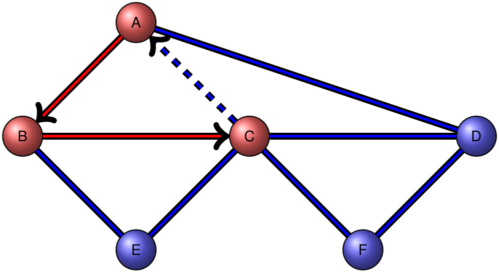
Depth-First Search – Example



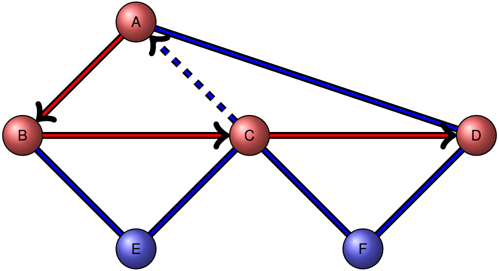
Depth-First Search – Example



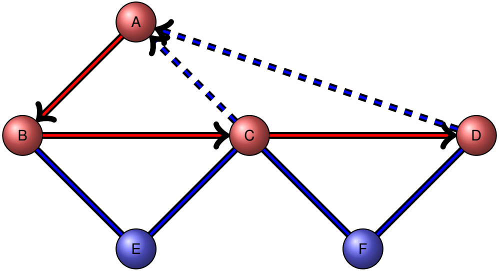
Depth-First Search – Example



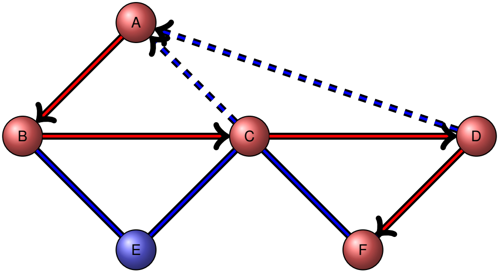
Depth-First Search – Example



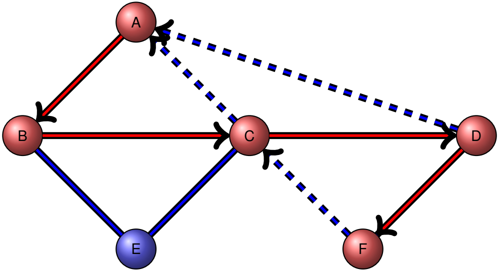
Depth-First Search – Example



Depth-First Search – Example

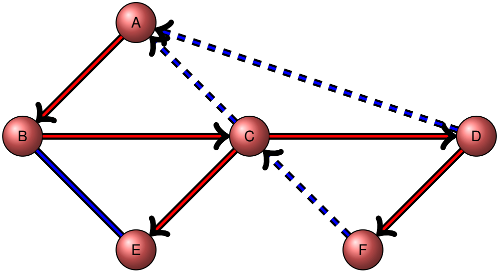


Depth-First Search – Example

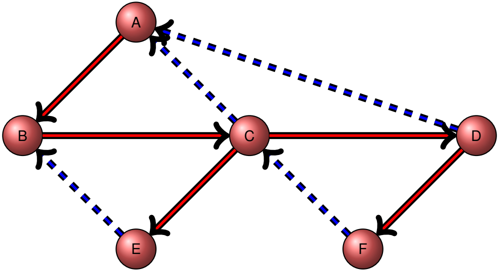


Backtrack.

Depth-First Search – Example

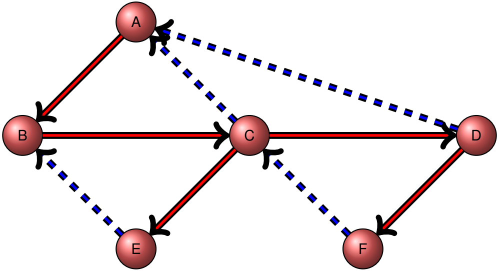


Depth-First Search – Example



Backtrack

Depth-First Search – Example



Depth-First Search – Pseudocode

ALGORITHM **DFS** (G)
/* Implement a Depth First Traversal of a graph. */
/* INPUT : Graph $G = \langle V, E \rangle$ */
/* OUTPUT : Graph G with its vertices marked with consecutive */
/* integers in initial encounter order. */

```
count = 0                          ▷ number of nodes visited
for i = 0 to v do                  ▷ mark all nodes unvisited
    Marked[i] = 0
end for
for i = 0 to v do                  ▷ visit each unmarked node
    if not Marked[i] then
        DFSR (i)
    end if
end for
```

Depth-First Search – Pseudocode

ALGORITHM **DFSR** (v)
/* Recursively visit all connected vertices. */
/* INPUT : A starting vertex v */
/* OUTPUT : Graph G with its vertices marked with consecutive */
/* integers in initial encounter order. */

```
count = count + 1                       ▷ increment the node visited counter
Marked = count                          ▷ mark node as visited
for v' ∈ V adjacent to v do             ▷ recursively visit all
    if not Marked[v'] then              ▷ unmarked adjacent nodes
        DFSR (v')
    end if
end for
```

Depth-First Search – Summary

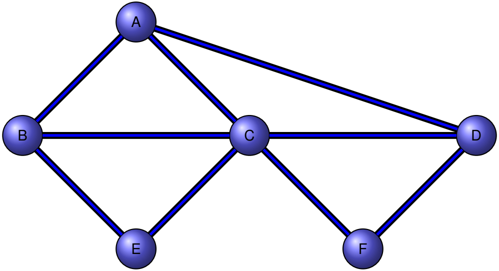
- A DFS search can be implemented with graphs represented as:
  - Adjacency matrices: $\mathcal{O}(|V|^2)$
    - It is a graph traversal, so we need to iterate over all vertices ($|V|$ of these). For each vertex, we need to check the neighbours of it. For the matrix represenation, the only way we can guranatee to find all neighbours of vertex i is to do a linear scan across its row in the matrix, which has $|V|$ elements. So $|V|$ * $|V|$ gives $\mathcal{O}(|V|^2)$ complexity. The traversal also needs to setup visited status, which requires $\mathcal{O}|V|$ complexity, but the quadratic term dominates.
  - Adjacency lists: $\mathcal{O}(|V| + |E|)$
    - Similarly to the matrix representation, we need to iterate over all the vertices ($|V|$ of these). For each vertex, we need to check the neighbours of it. Different for the adjacency list represenation, we only need to scan through the elements in the associated linked list. The total number of elements across all the linked list (and total number of neighbours to consider) is $\mathcal{O}(|E|)$. The traversal also needs to setup visited status, which requires $\mathcal{O}|V|$ complexity. Hence the total is $\mathcal{O}(|V| + |E|)$.

Breadth-First Search – Overview

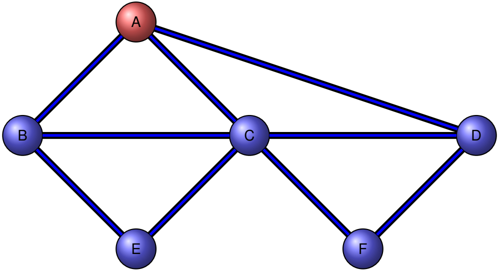
Breadth-First Search (BFS) - Traversal:

1. Choose an arbitrary vertex v and mark it visited.
2. Visit and mark (visited) each of the adjacent (neighbour) vertices of v in turn.
3. Once all neighbours of v have been visited, select the first neighbour that was visited, and visit all its (unmarked) neighbours.
4. Then select the second visited neighbour of v, and visit all its unmarked neighbours.
5. The algorithm halts when we visited all vertices.

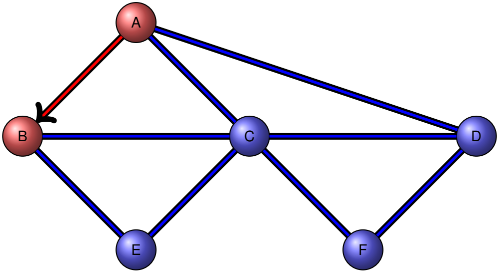
Breadth-First Search – Example



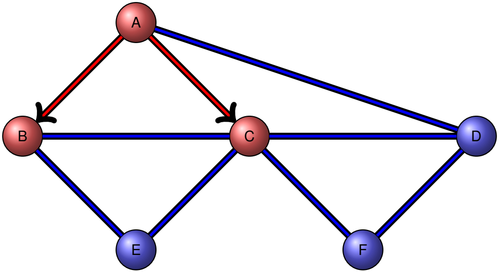
Breadth-First Search – Example



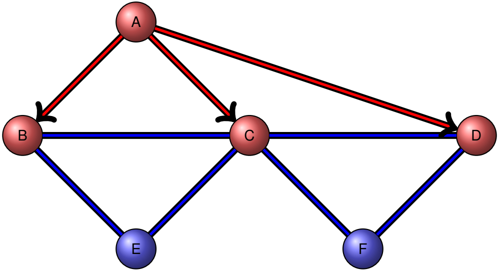
Breadth-First Search – Example



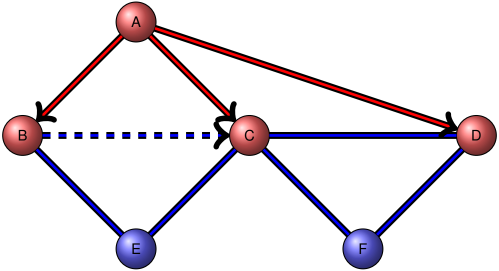
Breadth-First Search – Example



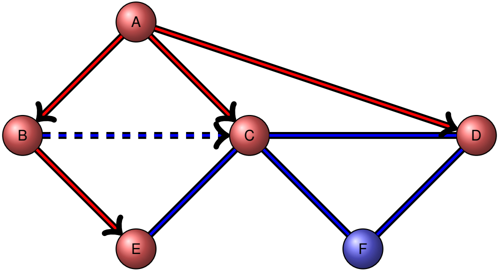
Breadth-First Search – Example



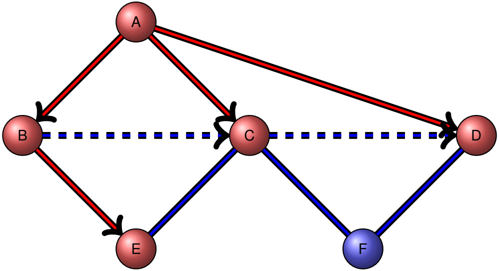
Breadth-First Search – Example



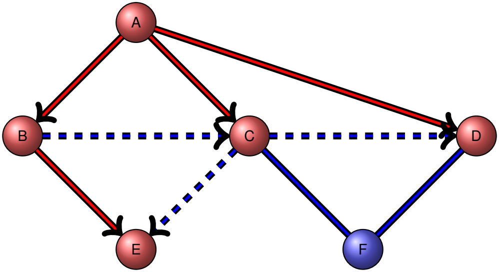
Breadth-First Search – Example



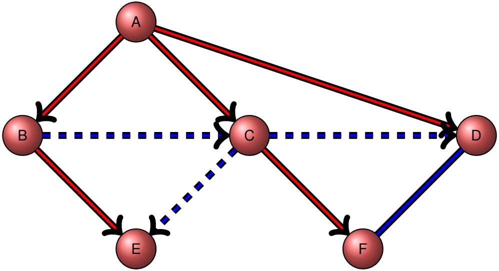
Breadth-First Search – Example



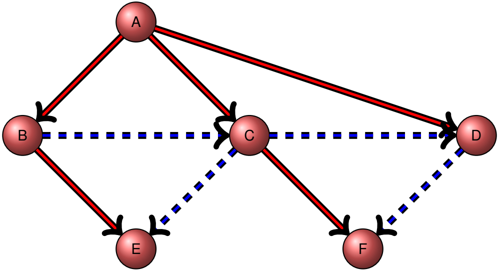
Breadth-First Search – Example



Breadth-First Search – Example



Breadth-First Search – Example



Graph Search – Analysis

	DFS	**BFS**
Applications	connectivity, acyclicity	connectivity, acyclicity, shortest paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Overview

Case Study

From this week onwards, we are going to study a particular problem and how to solve it using one of the algorithms we learn in that week's paradigm.

The structure will be a general problem statement, we then discuss how to map this to a problem we know the algorithm for, then solve it using that algorithm. There maybe more than one algorithm that can solve a problem, then we should evaluate in terms of the problem requirements and characteristics such as time complexity.

Case Study - Problem

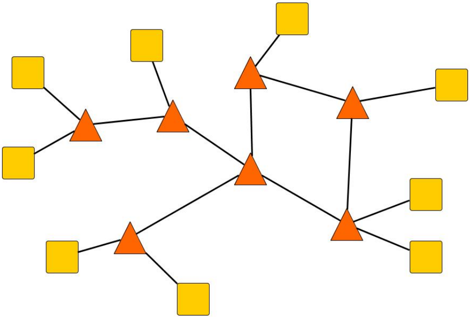
Case Study Problem

ABC Gold Plated Power Company operates a power transmission network and recently had some failures in their lines and shutdown of some substations. They want to determine if all their substations and customers' homes are connected to the network.

They have asked you to help them. How would you approach this problem?

Case Study - Mapping the Problem to a Known Problem

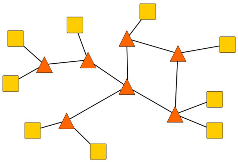
This can be mapped into a graph problem. Each substation and home is a vertex in that graph. Each transmission line between substation and/or home is an edge.

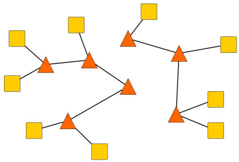


Case Study - Solving the Problem

Finding whether this graph is connected is equivalent to finding if all substations and homes are connected.

We know we can use either DFS and BFS to determine if a graph is connected. A DFS or BFS traversal of this graph is fully connected if it contains all vertices (why is this so?)





(d) Fully connected. (e) Not fully connected.

Overview

Summary

- Introduced the *Brute force* algorithmic approach.
- Sorting - selection and bubble sort.
- Computational Geometry - convex hull
- Exhaustive search (enumeration) - knapsack
- Graph search - DFS, BFS

Next week: Decrease-and-conquer and learn about more algorithms that can be used to solve interesting problems.