Final Review

Disclaimer

- Topics covered in this review may not appear in the exam.
- Topics not covered in this review may appear in the exam.

- See midterm 1 and 2 review for topics covered before midterm 2

Graph traversal

Textbook Ch 22.2/3/5

Unweighted path length

Problem: in an unweighted graph, find the distances from one vertex v to all the other vertices

- Distance: the length of the shortest path between two vertices

Method:

- Use a breadth-first traversal
- Vertices are added in *layers*
- The starting vertex v is defined to be in the zeroth layer, L_0
- While the k^{th} layer is not empty, all unvisited vertices adjacent to vertices in L_k are added to the $(k + 1)^{\text{st}}$ layer

The distance from v to vertices in L_k is k

Any unvisited vertices are said to have an infinite distance from v

Identifying bipartite graphs

Use a breadth-first traversal for a connected graph:

- Choose a vertex, mark it belonging to V_1 and push it onto a queue
- While the queue is not empty, pop the front vertex v and
  - Any adjacent vertices that are already marked must belong to the set not containing v, otherwise, the graph is not bipartite (we are done);
  - Any unmarked adjacent vertices are marked as belonging to the other set and they are pushed onto the queue
- If the queue is empty, the graph is bipartite

Topological sort

Textbook Ch 22.4

Topological sorting

A topological sorting of the vertices in a DAG is an ordering

$$v_1, v_2, v_3, \ldots, v_{|V|}$$

such that v_j appears before v_k if there is a path from v_j to v_k

Theorem:

A graph is a DAG if and only if it has a topological sorting

Implementation

To implement a topological sort:

- Allocate memory for and initialize an array of in-degrees
- Create a queue and initialize it with all vertices that have in-degree zero

While the queue is not empty:

- Pop a vertex from the queue
- Decrement the in-degree of each neighbor
- Those neighbors whose in-degree was decremented to zero are pushed onto the queue

Critical path

We have a DAG of tasks, each with a performance time

The *critical time* of a task is the earliest time that it could be completed after the start of execution

The *critical path* is the sequence of tasks determining the minimum time needed to complete the project

- If a task on the critical path is delayed, the entire project will be delayed

Finding the critical path

To find the critical time/path, we run topological sorting and require the following additional information:

- We will have to record the critical time for each task
  - Initialize these to zero
- We will need to know the previous task with the longest critical time to determine the critical path
  - Set these to null

Each time we pop a vertex v, in addition to what we already do:

- For v, add the task time onto the critical time for that vertex:
  - That is the critical time for v
- For each adjacent vertex w:
  - If the critical time for v is greater than the currently stored critical time for w
    - Update the critical time with the critical time for v
    - Set the previous pointer to the vertex v

Shortest path

Textbook Ch 24, 25

Dijkstra’s algorithm

Dijkstra’s algorithm solves the single-source shortest path problem

- Assumption: all the weights are positive

Strategy

At each step of the algorithm

- We know the shortest distance to some of the vertices (marked as visited)
- We also know the shortest distance to each unvisited vertex **through visited vertices** (call this the “known distance”)

Consider the unvisited vertex ***v*** that has the shortest known distance

- We are guaranteed that the known distance to ***v*** is the shortest distance from the start node to it

Dijkstra's algorithm

We will iterate $|V|$ times:

- Find the unvisited vertex ***v*** that has a minimum distance to it
- Mark it as visited
- Consider its every adjacent vertex ***w*** that is unvisited:
  - Is the distance to ***v*** plus the weight of the edge (***v***, ***w***) less than our currently known shortest distance to ***w*** ?
  - If so, update the shortest distance to ***w*** and record ***v*** as the previous pointer

Continue iterating until all vertices are visited or all remaining vertices have a distance of infinity

Run time when using

- A binary heap: $O(|E| \ln(|V|))$
- A Fibonacci heap: $O(|E| + |V| \ln(|V|))$

Floyd-Warshall algorithm

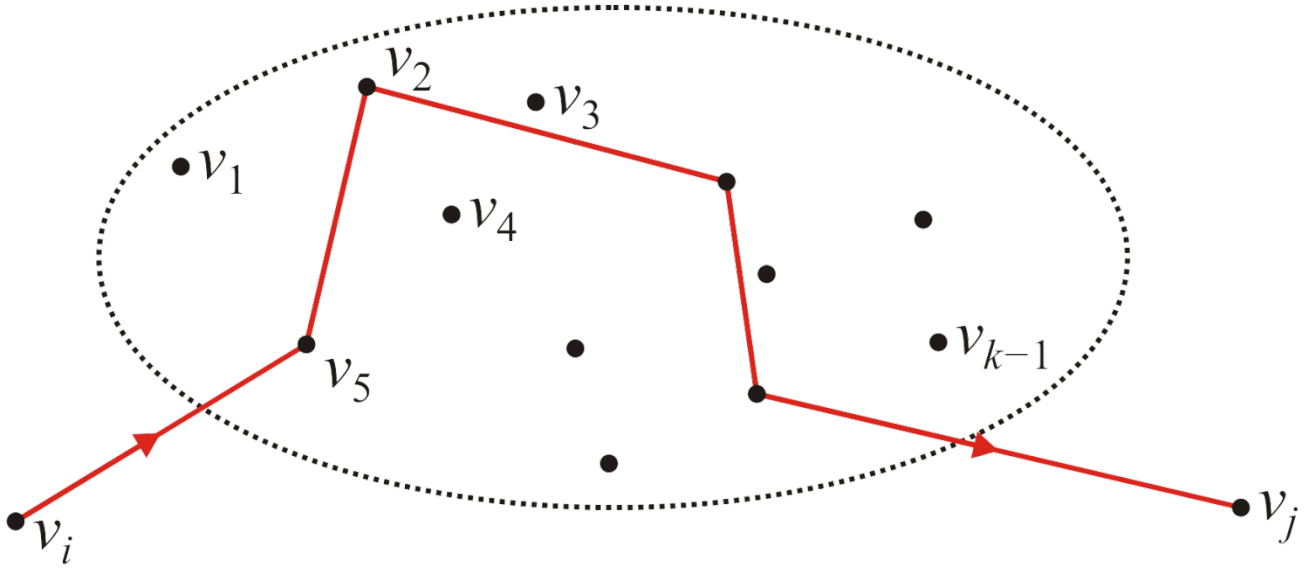
Floyd-Warshall algorithm

- It finds the shortest path between all pairs of nodes
- It works with positive or negative weights with no negative cycle

The General Step

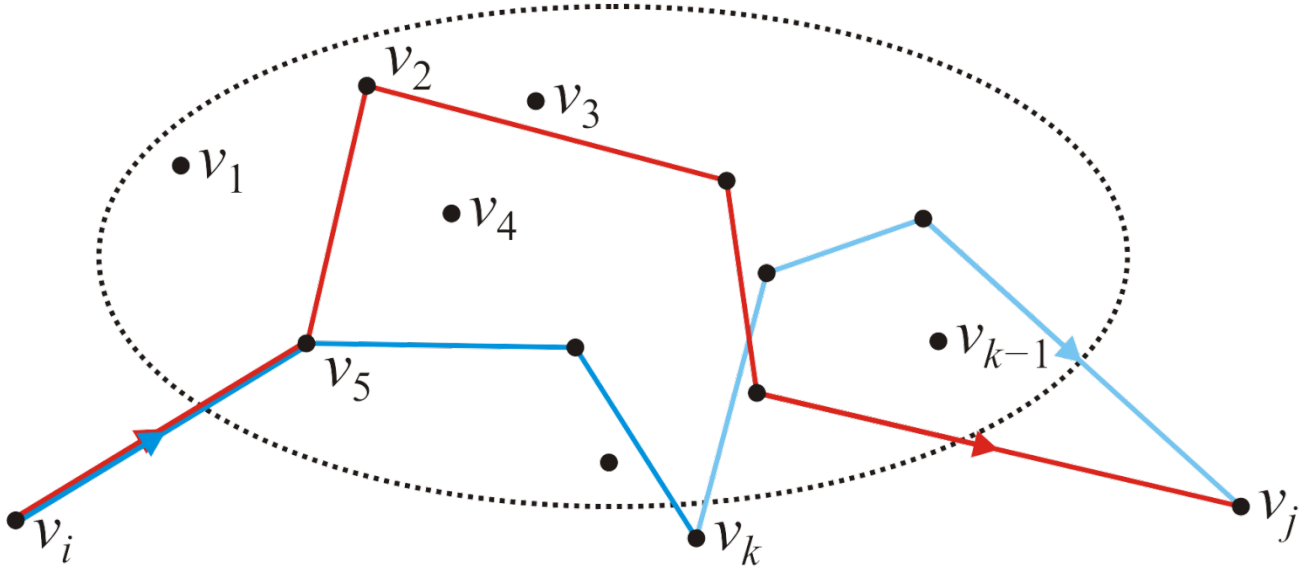
Define $d_{i,j}^{(k-1)}$ as the shortest distance, but only allowing intermediate visits to vertices v_1, v_2, …, v_{k-1}

- Suppose we have an algorithm that has found these values for all pairs



The General Step

Thus, we calculate $d_{i,j}^{(k)} = \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\}$



The Floyd-Warshall Algorithm

```
// Initialize the matrix d
//   ...

for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            d[i][j] = std::min( d[i][j], d[i][k] + d[k][j] );
        }
    }
}
```

Run time: $\Theta\left(|V|^3\right)$

Sorting

Textbook Ch 2, 7, 8

Lower-bound Run-time

The general run time is $\Omega(n \ln(n))$

The proof:

- Any comparison-based sorting algorithm can be represented by a comparison tree
- Worst-case running time cannot be less than the height of the tree
- How many leaves does the tree have?
  - The number of permutations of n objects, which is $n!$
- What's the shallowest tree with $n!$ leaves?
  - A complete tree, whose height is $\lg(n!)$
  - It can be shown that $\lg(n!) = \Theta(n \ln(n))$

Insertion sort

For any unsorted list:

- Treat the first element as a sorted list of size 1

Then, given a sorted list of size $k-1$

- Insert the k^{th} item into the sorted list
- The sorted list is now of size k

Insertion sort

The following table summarizes the run-times of insertion sort

Case	Run Time	Comments
Worst	$\Theta(n^2)$	Reverse sorted
Average	$O(d + n)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	Very few inversions: $d = O(n)$

Bubble sort

Starting with the first item, assume that it is the largest

Compare it with the second item:

- If the first is larger, swap the two,
- Otherwise, assume that the second item is the largest

Continue up the array, either swapping or redefining the largest item

Bubble sort

After one pass, the largest item must be the last in the list

Start at the front again:

- the second pass will bring the second largest element into the second last position

Repeat $n - 1$ times, after which, all entries will be in place

Bubble sort

Some improvements of bubble sort :

- reduce the number of swaps,
- halting if the list is sorted,
- limiting the range on which we must bubble
- alternating between bubbling up and sinking down

Run-Time

The following table summarizes the run-times of our modified bubble sorting algorithms; however, they are all worse than insertion sort in practice

Case	Run Time	Comments
Worst	$\Theta(n^2)$	$\Theta(n^2)$ inversions
Average	$\Theta(n + d)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	$d = O(n)$ inversions

Heap sort

Use a max-heap:

- Place the objects into a max-heap, stored in the input array
- Repeatedly pop the top object and move it to the end of the array, until the heap is empty
- $\Theta(1)$ memory

Heap sort

The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$O(n \ln(n))$	No worst case
Average	$O(n \ln(n))$	
Best	$\Theta(n)$	All or most entries are the same

Merge Sort

The merge sort algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list

This strategy is called *divide-and-conquer*

Merge Sort

The following table summarizes the run-times of merge sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n \ln(n))$	No best case

Quicksort

Splits the array into two sub-lists and sorts them

- Chose an object in the array and partition the remaining objects into two groups relative to the chosen entry
- Choose the median of the first, middle, and last entries in the list

In the best case, we select the median element and the list will be split into two equal sub-lists, so the run time is $\Theta(n \ln(n))$

In the worst case, we select the min/max element, and then the run time is $\Theta(n^2)$

Selecting median-of-three:

- Choose the median of the first, middle, and last entries in the list

Quicksort

Implement quicksort in place

- Swap the pivot to the last slot of the list
- We repeatedly try to find two entries:
  - Staring from the front: an entry larger than the pivot
  - Starting from the back: an entry smaller than the pivot
- Such two entries are out of order, so we swap them
- Repeat until all the entries are in order
- Move the leftmost entry larger than the pivot into the last slot of the list and fill the hole with the pivot

Quicksort

The additional memory

- Function call stack
  - Each recursive function call places its local variables, parameters, *etc*., on a stack
- Average case: the depth of the recursion is $\Theta(\ln(n))$
- Worst case: the depth of the recursion is $\Theta(n)$

Bucket sort

Bucket sort

- Create a bit vector
  - We will term each entry in the bit vector a *bucket*
- Set each bit to 0 (indicating false)
- For each element to be sorted, set the bit indexed by the element to 1 (true)
- Walk through the vector and for each bit which is 1, record that number

We don't have *arbitrary* data

- We have one further constraint: the items being sorted are integers within a small range
- If the size of the range (i.e., number of buckets) is $O(n)$, then we get a $\Theta(n)$ algorithm

Counting sort

Modification: what if there are repetitions in the data

- In this case, a bit vector is insufficient

Two options, each bucket is either:

- a counter, or
- a linked list

The first is better if objects in the bin are the same

Radix Sort

We have the following algorithm:

- Suppose we are sorting decimal numbers
- Create an array of 10 queues
- For each digit, starting with the least significant
  - Place the i-th number into the bin corresponding with the current digit
  - Remove all digits in the order they were placed into the bins in the order of the bins

Radix Sort

The following table summarizes the run-times of radix sort for sorting n numbers on the range $0, \ldots, m - 1$

Case	Run Time	Comments
Worst	$\Theta(n \ln(m))$	No worst case
Average	$\Theta(n \ln(m))$	
Best	$\Theta(n \ln(m))$	No best case

It requires $\Theta(n)$ memory for the queues

- It is only useful to use radix sort over quicksort if $n = \omega(m)$