ECS 20 **Chapter 8, Graph Theory**

1. Introduction, Data Structures
   1.1. The atom of computer memory is a “byte.” Each byte is capable of holding 256 different values, 0-255. Each byte has its own address. The post office boxes in a post office are a good analogy. Each post office box has its own number, and the box number is separate from the actual contents of the post office box (letters).
   1.2. A computer variable is an object stored in memory. Different types of objects require different number of bytes for their storage in memory. A letter of the alphabet requires one byte, but an integer typically requires four bytes. Each computer variable will have a unique address, which is the address of its first (lowest numbered) byte.
   1.3. “Data structure” refers to a collection of computer variables that are connected in some specific manner.
   1.4. “Abstract Data Type” refers to a data structure along with a set of operations on that data structure.
   1.5. There are two primitive data structures in most computer languages.
      1.5.1. Arrays are tables consisting of identically sized objects.
         1.5.1.1. Objects are stored in consecutive bytes of memory. Space in memory is set aside for the whole array all at once.
         1.5.1.2. Objects are associated with consecutive index numbers within the table, usually starting with zero or one.
         1.5.1.3. Programs can find any object in an array quickly, no matter the size of the array, because the location can be determined using a simple formula.
            1.5.1.3.1. Address of an array object = Address of first object + index of sought object * size of each object.
         1.5.1.4. Two-dimensional arrays typically use “row major” storage, which stores the objects of the first row consecutively, then the objects of the second row, and so on.
      1.5.2. Pointers and Linked Lists
         1.5.2.1. A “pointer variable”, or “pointer” for short, holds the address of another object that is located somewhere else in memory. In terms of the post office analogy, a pointer variable is a post office box that holds a post office box number. Pointers are typically four bytes in size, which allows them to hold values 0 to 4 billion. Since pointers are variables, they too have addresses. So you can have a pointer variable that holds the address of another pointer variable--the infamous “double pointers” of ECS 30.
         1.5.2.2. A “linked list node” is composed of data object variable(s) and pointer variables that hold the addresses of other linked list nodes.
         1.5.2.3. A “linked list” is composed of a pointer variable that hold the address of the first linked list node in the linked list. In “singly” linked lists, each list node contains only one pointer variable, and it holds the address of the next linked list node in the list. In “doubly” linked lists, each list node contains two pointer variables with the additional pointer variable holding the address of the previous list node.
   1.6. Stacks
      1.6.1. Provide <u>only</u> last-in first-out (LIFO) access to data that is stored in a linear list which is typically either an array or a linked list.
      1.6.2. The “top” of the stack is that object that would next be removed or read from the stack.
      1.6.3. “Push” operation places an object on the top of the stack, and all previous objects are thought to be one position further from the top of the stack.
      1.6.4. “Pop” operation removes the object that is currently on the stack, and thus all other objects are one position closer to being the top of the stack.
      1.6.5. “Top” operation copies the object on the top of the stack, but does not remove it from the stack.
      1.6.6. All three operations of a stack have a time complexity of $\Theta(1)$.
   1.7. Queues
      1.7.1. Provide <u>only</u> first-in first-out (FIFO) access to data that is stored in a linear list which is typically either an array or a linked list.

1.7.2. The “front” of a queue can mean either the first position on list, or the object in that first position that would next be removed or read from the queue.
1.7.3. The “back” of a queue can mean either the last position on the list, or the last object that was placed in the queue. In a queue that has only one object in it, the “front” and “back” are the same object.
1.7.4. “Front” operation copies the object at the front of the list, but does not remove it from the queue.
1.7.5. “Pop” or “Dequeue” operation removes the object at the front of the list, and thus all other objects are one position closer to being the front of the queue.
1.7.6. “Push” or “Enqueue” operation appends the object to the back of the list.
1.7.7. “Back” operation copies the object at the back of the list, but does not remove it from the queue.
1.7.8. All four operations of a queue have a time complexity of $\Theta(1)$.
1.8. Priority Queues
1.8.1. Organize its objects so that that only the object with the highest priority is available for immediate access, while the rest of the objects are unavailable for access. Priority queues are implemented in arrays.
1.8.2. “Insert” operation places a new object inside the priority queue based on its priority.
1.8.3. “DeleteMin” or “DeleteMax” removes the object with the highest priority from the priority queue.
1.8.4. “FindMin” or “FindMax” copies the object with the highest priority, but does not remove it from the queue.
1.8.5. Insertion and deletion of objects are $O(\log_2 n)$, and FindMin or FindMax are $\Theta(1)$.
2. Graphs and Multigraphs
2.1. A graph G consists of two sets, where V is its set of vertices, and E is its set of edges.
2.1.1. |V| is the number of vertices, and |E| is the number of edges. If |E| is much larger than |V| the graph is said to be “dense.” If |E| is not much larger than |V|, e.g. |E| = 10*|V| with large |V|, then the graph is said to be “sparse”
2.2. An “edge” is a pair of distinct vertices that are directly connected to each other in the graph with no other intervening vertex. If the pair is unordered, then the graph is “undirected.” If the pair is ordered, then the graph is “directed.”
2.3. A “simple graph” has a maximum of one edge for each possible pair of vertices. A “multigraph” has more than one edge for some pair(s) of vertices.
2.3.1. For a simple undirected graph $|E| < |V|^2$. For a simple directed graph $|E| < 2|V|^2$.
2.4. Vertices u and v are said to be “adjacent” if and only if $(u, v) \in E$.
2.4.1. The “degree” of a vertex, written deg(v) is the number edges that have v in them.
2.4.2. The sum of the degrees of all the vertices of G equals twice the number of edges of the graph. Therefore, the total degree of a graph is always even.
3. Subgraphs, Isomorphic and Homeomorphic Graphs (skipped)
4. Paths, Connectivity
4.1. A “path” in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_n$ such that (w_i, w_{i+1}) is an element of E for $1 \leq i < n$. The “length” of such a path is the number of edges on the path = n - 1. We allow path from a vertex to itself; if this path contains no edges then the path length is 0.
4.2. A “simple path” is a path such that all vertices are distinct except that the first and last could be the same.
4.3. A “cycle” in a graph is a path of length at least three, such that the first vertex is also the last vertex. Such a path is called “closed.”
4.4. A graph is “connected” if there is a path between any two of its vertices.
4.5. Let G be a connected graph, then a vertex v is call a ”cutpoint” if G – v is not connected.
5. Traversable and Eulerian Graphs, Bridges of Konigsberg
5.1. A graph is “traversable” if there is a path which includes all vertices and uses each edge exactly once.
5.1.1. A finite connected graph that has a closed traversable path is called a Eulerian graph, and can exist if and only if each vertex has even degree. Note that vertex may be visited more than once.
5.1.2. Any finite connected graph with exactly two odd vertices is traversable with the traversable trail beginning at one of the odd vertices, and ending at the other odd vertex.
5.2. A Hamiltonian circuit is a closed path that visits every vertex exactly once. Note that a Hamiltonian circuit may repeat edges.

5.2.1. A graph is Hamiltonian if $|V| \geq 3$, and $|V| \leq \deg(v)$ for every $v \in V$.

6. Labeled and Weighted Graphs
   6.1. A "weighted" graph has every edge assigned a non-negative number, called its weight. The "weight" of a path is the sum of the weights of the edges in the path.
   6.2. The Shortest Path Problem is given as input a weighted graph, G = (V,E), and a distinguished vertex, s, and must find the shortest weighted path from s to every other vertex in G.
   6.3. Dijkstra's algorithm to solve the Shortest Path Problem
      6.3.1. Table of three values for each vertex,
         6.3.1.1. "known" indicates if the vertex is a member of the partial MST, initially false. A vertex that has its known variable set to false is said to be "unknown".
         6.3.1.2. "dv" is the distance to distinguished vertex (initially infinity).
         6.3.1.3. "pv" is the previous vertex which indicates the last vertex to cause a change to dv (initially set to ?).

```
1. Initialize entries in table as noted above.
2. Set the dv of vertex $s$ to 0.
3. Select a new vertex, $v$, that has the smallest dv among all unknown vertices.
   Change known of $v$ to true.
   For each unknown vertex $u$ adjacent to $v$, if dv of $u$ > [(dv of $v$) + weight($v$,$u$)] then set dv of $u$ to [(dv of $v$) + weight($v$,$u$)], set pv of $u$ to $v$.
4. If there are unknown vertices go to 3.
```

      6.3.2. The final pv's specify the short path(s) to *s* from each non-*s* vertex.
      6.3.3. O($|V|^2$) without priorities queues, which is optimal for dense graphs, and O($|E| \log_2 |E|$) using a priority queue to store vertices based on their dv.

7. Compete, Regular, and Bipartite Graphs
   7.1. A "complete" graph has every vertex is adjacent to every other vertex.
   7.2. Regular Graphs (skipped)
   7.3. A graph is bipartite if its vertices V can be partitioned into two subsets *M* and *N* such that each edge of G connects a vertex of *M* to a vertex of *N*.
      7.3.1. An example of bipartite graph is a job matching problem.
   7.4. Testing to see if a graph is bipartite: If a graph is connected, its bipartition can be defined by the parity of the distances from any arbitrarily chosen vertex *v*: one subset consists of the vertices at even distance to *v* and the other subset consists of the vertices at odd distance to *v*. Thus, one may efficiently test whether a graph is bipartite by using this parity technique to assign vertices to the two subsets *U* and *V*, separately within each connected component of the graph, and then examine each edge to verify that it has endpoints assigned to different subsets.

8. Tree Graphs
   8.1. A graph is a "tree" if it is connected and has no cycles.
      8.1.1. If we add any edge to a tree, then it will no longer be a tree because it will now contain a cycle.
      8.1.2. If we delete any edge from a tree, then it will no longer be a tree because it is no longer connected.
      8.1.3. A tree has |V| - 1 edges.
   8.2. A "spanning tree" is a tree that connects all of the vertices of G.
   8.3. A "minimum spanning tree", MST, connects all the vertices of G at lowest total cost. A minimum spanning tree exists if and only if G is connected.
   8.4. There are two primary algorithms to find minimum spanning trees in undirected connected graphs with |V| vertices and |E| edges.
   8.5. Prim's algorithm
      8.5.1. Table of three values for each vertex. Same as for Dijkstras for Shortest Path Problem
         8.5.1.1. "known" indicates if the vertex is a member of the partial MST, initially false. A vertex that has its known variable set to false is said to be "unknown".
         8.5.1.2. "dv" is the weight of the lightest edge of the vertex that has its other vertex in the partial MST (initially infinity).
         8.5.1.3. "pv" is the previous vertex which indicates the last vertex to cause a change to dv (initially set to ?).

```
1. Initialize entries in table as noted above.
2. Arbitrarily choose a vertex with which to start the tree. Set its dv to 0.
3. Select a new vertex, $v$, to add to the tree by finding that vertex has the smallest dv among all unknown vertices.
```

Change known of *v* to true.
For each unknown vertex *u* adjacent to *v*, if dv of *u* > weight(*v*,*u*), then set dv = weight(*v*,*u*), and set pv of *u* =*v*.
4. If there are unknown vertices go to 3.
8.5.2. The pv's of the vertices specify the MST.
8.5.3. O($|V|^2$) without priorities queues, which is optimal for dense graphs, and O($|E| \log_2 |E|$) using a priority queue to store vertices based on their dv.
8.6. Kruskal's Algorithm
1. Sort all edges from smallest to largest weight.
2. Select the edges in order of smallest weight and accept an edge if it does not cause a cycle.
3. Terminate when one tree.
8.6.1. Sorting edges takes O($|E| \log_2|E|$). Determining if an edge causes a cycle takes O($\log_2|V|$)
9. Planar Graphs
9.1. A graph that can be drawn in one plane so that its edges do not cross is said to be "planar." Trees are all planar. Useful in laying out circuit boards.
9.2. A particular planar representation of a finite planar graph is called a "map." The map is "connected" if the underlying is connected.
9.3. A given map divides the plane into various regions. The number of regions in a map is |R|. The "degree" of a region *r*, written deg(*r*) is the length of the cycle which borders *r*.
9.4. The sum of the degrees of the regions of a map is equal to twice the number of edges.
9.5. Euler's Formula: For any connected planar graph $|V| - |E| + |R| = 2$.
9.6. For every connected planar graph with $|V| \geq 3$, $|E| \leq 3|V| - 6$. Note that Theorem 8.9 has a typo.
9.7. Kuratowski's Theorem (skipped)
10. Graph Colorings
10.1. A "vertex coloring" is an assignment of colors to the vertices of a graph such that adjacent vertices have different colors.
10.2. The minimum number of colors to paint a graph is its "chromatic number." There is no maximum to the number of colors needed. A complete graph will require |V| colors, since all vertices are adjacent to each other.
10.3. Applications
10.3.1. Vertex coloring models to a number of scheduling problems. In the cleanest form, a given set of jobs need to be assigned to time slots, each job requires one such slot. Jobs can be scheduled in any order, but pairs of jobs may be in conflict in the sense that they may not be assigned to the same time slot, for example because they both rely on a shared resource. The corresponding graph contains a vertex for every job and an edge for every conflicting pair of jobs. The chromatic number of the graph is exactly the optimal time to finish all jobs without conflicts.
10.3.2. A compiler is a computer program that translates one computer language into another. To improve the execution time of the resulting code, one of the techniques of compiler optimization is register allocation, where the most frequently used values of the compiled program are kept in the fast processor registers. Ideally, values are assigned to registers so that they can all reside in the registers when they are used. The textbook approach to this problem is to model it as a graph coloring problem. The compiler constructs an interference graph, where vertices are symbolic registers and an edge connects two nodes if they are needed at the same time. If the graph can be colored with *k* colors then the variables can be stored in *k* registers.
10.3.3. The recreational puzzle Sudoku can be seen as completing a 9-coloring on given specific graph with 81 vertices.
10.4. Welch and Powell Coloring Algorithm
1. Order the vertices of G according to decreasing degrees.
2. Assign the first color C_1 to the first vertex, and then, in sequential order, assign C_1 to each vertex which is not adjacent to a previous vertex which was assigned $C_{1.}$
3. Repeat Step 2 with a second color C_2, and the subsequence of non-colored vertices.
4. Repeat Step 3 with different colors until all vertices are colored.
10.5. Any planar graph is 4-colorable.

10.6. By connecting one vertex in each region of a map and then connecting the vertices with edges that do not cross, we can see that the regions of a map create a planar graph with as many vertices as there are regions in the map. Therefore, a map can be colored with no more than 4 colors.

11. Representing Graphs in Computer Memory

11.1. Adjacency Matrix is a |V| x |V| two-dimensional table that contains a 1 where two vertices are adjacent, and a 0 otherwise.

11.1.1. Space $O(|V|^2)$.

11.1.2. Time determine if two vertices are adjacent is O(1). Finding all vertices adjacent vertices Θ(|V|)

11.2. Adjacency List is a one-dimensional table of linked lists, one for each vertex of a graph. Each linked list contains a list node storing representing a vertex that is adjacent to the vertex of the table.

11.2.1. Space O(V + 2E)

11.2.2. Time determine if two vertices are adjacent is O(|V|), but more accurately O(number of vertices adjacent). Finding all vertices adjacent vertices O(|V|), but more accurately Θ(number of vertices adjacent).

11.2.3. Adjacency List using an array, has the linked lists replaced by a (possibly dynamically allocated) array and a adjacent count variable. The array stores representations of vertices that are adjacent to the vertex of the table.

12. Graph Algorithms are both O(|V| + 2|E|) for undirected graphs, and O(|V| + |E|) for a directed graphs.

12.1. Both algorithms rely on a way to indicate whether a vertex has been visited. This can be a separate Boolean variable, or any other sentinel method, e.g. having a vertex's number negative vs. positive. If the graph is not connected, then the algorithm will need to be restarted with an unvisited as many times as there are separate connected subgraphs.

12.2. Depth-first search (DFS) relies on a stack to adjacent vertices as the process moves "down" the graph.

12.2.1. DFS Algorithm:

1. Initialize all vertices to Unvisited.
2. Set the starting the starting vertex as Visited, and push it onto the stack.
3. If the stack is empty exit.
4. Pop a vertex, *v*, from the stack.
5. For each Unvisited vertex adjacent to *v*, mark the vertex as Visited, and push it on the stack.
6. Go to 4.

12.2.2. One application is in artificial intelligence solution searches, e.g. chess, using an objective function to assign a value to each vertex. "Branch and bound" backtracks up from searching a particular branch when the selected vertex's function value is too much worse from the best found so far.

12.3. Breadth-first search (DFS) relies on a queue to store adjacent vertices.

12.3.1. DFS Algorithm:

1. Initialize all vertices to Unvisited.
2. Set the starting the starting vertex as Visited, and enqueue it into the queue.
3. If the queue is empty exit.
4. Dequeue a vertex, *v*, from the queue.
5. For each Unvisited vertex adjacent to *v*, mark the vertex as Visited, and enqueue it on the stack.
6. Go to 4.

12.3.2. One application is for the Shortest Path Problem in an unweighted graph.

13. Traveling-Salesman Problem

13.1. Let G be a complete weighted graph, find a Hamiltonian circuit for G of minimum weight. A Hamiltonian circuit is a closed path that visits every vertex exactly once.

13.2. The complete graph with |V| ≥ 3 has (|V| - 1)! / 2 Hamiltonian circuits when we do not distinguish between a circuit and its reverse so O(|V|!).

13.3. The Nearest Neighbor Algorithm does not guarantee the minimum weight, but is faster, $O(|V|^2)$, and often close to the minimum.

1. Mark all vertices a unvisited.

2. Start with any vertex, mark it as visited, and call it the current vertex.
3. Find the unvisited adjacent vertex with minimum weight edge from the current vertex, O(|V|), mark it visited, and make it the current vertex.
4. If the number of vertices visited is less than |V| then go to 3, else continue to step 5.
5. Complete the Hamiltonian circuit by selecting the edge from the current vertex to the start vertex.

13.4. It is an example of an "NP-Complete" problem. NP-Complete problems cannot be solved in polynomial time, but their solutions can be verified in polynomial time.

13.4.1. The first NP-Complete problem was the Boolean Satisfiability Problem which is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE.

13.4.2. Another NP-complete problem is the decision Subset Sum Problem, which is this: given a set of integers, does any non-empty subset of them add up to zero?

13.4.3. The Knapsack Problem is given a set of items, each with a weight and a value, and then must determine the count of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items. The problem often arises in resource allocation with financial constraints.

13.4.4. If a person ever finds a polynomial solution to one of the NP-Complete problems, then that solution can be converted to solution for all of the NP-Complete problems! Nobel Prize here they come!