Midterm 2 Practice Questions and Reading Material

Paul Jones

Midterm 2 Practice Questions and Reading Material

Greedy Algorithms: Horn formulas, Set Cover

Chapter 5.2 from DPV: Huffman encoding

- Is there some sort of *variable-length encoding* in which just one bit is used for the most frequent symbol and the least amout of bits are used for the lesser freqeuent symbols?
- One danger of this is that resulting encodings might not be uniquely decipherable.
  - For instance, considering $\{0, 01, 11, 001\}$. Strings like 001 are ambigous.
  - We need the *prefix-free* property.

Prefix free No codeword can be a prefix of another codeword.

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of} i\text{th symbol in tree})$$

- The cost of a tree is the sums of the frequencies of all leaves and internal nodes, except the root.
- We can state constructing the tree *greedily*: find two symbols of smallest frequencies, i and j, and make them children of a new node, which has the frequency $f_i + d_j$.

```
procedure Huffman(f)
Input: an array of frequencies
Output: an encoding tree

let H be a priority queue of integers, ordered by f
for i = 1 to n {
    insert(H,i)
}
for k = n + 1 to 2n - 1 {
    i = deletemin(H);
    j = deletemin(H);
    create a node numbered k with children i, j
    f[k] = f[i] + f[j];
    insert(H, k)
}
```

Chapter 5.3 from DPV: Horn formulas

- To display human intelligence, a computer must do some logical reasoning.
- Most primitive object in Horn formulas is the *Boolean variable*, which can be `true` or `false`. Ex,

$$x \equiv \text{the murder took place in the kitchen}$$

$$y \equiv \text{the butler is innocent}$$

$$z \equiv \text{the colonel was asleep at 8pm}$$

- A *literal* is either a variable x or its negation $\neg x$. In Horn formulas, knowledge about variables is represented in two kinds of *clauses*:

1. *Implications*, whose left-hand is an "and" of any number of positive literals and whose right-hand side is a single positive literal. For instance,

   $$(z \wedge w) \rightarrow u$$

   "If the colonel was asleep at 8 pm and the murder took place at 8pm, then the colonel is innocent."

2. Pure *negative cluases*, consisting of an "or" of any number of negative literals, as in,

   $$(\neg u \vee \neg v \neg y)$$

   "They can't all be innocent."

- Given a set of these two types, the goal is to determine whetheter there is a consistent explanation, an assignment of true/false values to the variables that satisfies all the cluases. *Satisfaction requirement.*

Chapter 5.4 from DPV: Set cover

- Given a set of towns and these two constraints: (1) each school should be in a town, and (2) no school should have to travel more than 30 miles to reach a school.
- This is a *set cover* problem.

  Set cover

  Input: A set of elements B; sets $S_1, \cdots, S_m \in B$

  Output: A selection of the S_i whose union is B.

  Cost: Number of sets picked

- There is a greedy solution:

```
While all elements B are uncovered:
    Pick the set $S_i$ with the largest number of uncovered elements
```

  Claim: Suppose B contains n elements and the algorithm consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.

- The ration between the greedy algorithm's solution and the optimal solution varies from iinput to input but is *way less* than $\ln n$

Chapter 16.2 from CLRS2: Elements of the greedy strategy

- A greedy algorithm obtains an optimal solution to a problem by making choices.
  - Every choice, the choice is the best *at the moment.*
  - This doesn't always work, but sometimes it does.
- The steps:
  1. Determine the optimal substructure of the problem.
  2. Develop a recursive solution.
  3. Prove that at any stage of the recusion, one of the optimal choices is the greedy choice. That is is always safe to make the greedy choice.
  4. Show that all but one of the subproblems induces by having made the greedy choice are empty.
  5. Develop a recursive algorithm that implements the greedy strategy.

6. Convert the recrusive algorithm to an iterative algorithm.

- The steps, more generally:
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is not always safe.
  3. Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that is we combine an optimal solution to the subprobelm with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Greedy-choice property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

- How greedy algorithms differ from dynamic programming:
  - In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. It is "bottom", going from smaller subproblems.
  - In a greedy algorithm, we make a choice that is best at the moment and then solve the subproblem arising after choice is made.
    - Greedy algorithms are "top-down."

Optimal substructure

If an optimal solution to the porblem contains within it optimal solutions subproblems.

- This is key to greedy algorithms and dynamic programming.

Chapter 16.3 from CLRS2: Huffman codes

Prefix codes Codes in which no codeword is also a prefix of some other codeword.

- In the following psuedocode, assume C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$.
  - The algorithm building the tree T corresponding to the optimal code in bottom-up manner.

```
Huffman(C)
n <- |C|
Q <- C
for i <- to n - 1:
    do allocate a new node z
        left[z] <- x <- Extract-Min(Q)
        right[z] <- y <- Extract-Min(Q)
        f[z] <-  f[x] + f[y]
        Insert(Q, z)

return Extract-Min(Q)
```

Practice questions

- *What is the main principle behind greedy algorithms?*

  The principle behind greedy algorithms is that at every point where a given algorithm can make a choice, instead of trying to figure out if there exists a "globally optimal" decision, a greedy algorithm just goes with the cheapest option in that moment and "hopes for the best."

- *Why does the greedy algorithm work for the coin changing problem given the US coin system?*

  The reason that the US coin system allows the greedy choice to solve the coin change problem is that a matroid can be formed from the set of US coins.

- *What is the idea in Huffman encoding in order to achieve data compression?*

  The idea of Huffman encoding is that you take the characters which appear most often in your document and you make those the cheapest to encode. You subsequently take the least frequent characters in your document and make them only as expensive as the prefix-free property demands they be.

- *What is the prefix-free property?*

  The prefix-free property is that for a given dictionary, no word in the dictionary has at its beginning any other word.

- *Provide the Huffman encoding algorithm and argue its running time.*

  1. Create a leaf node for each symbol and add it to the priority queue.
  2. While there is more than one node in the queue:
     1. Remove the two nodes of lowest probability.
     2. Crate a new node with the sum of those two probabilities.
     3. Add that node to the queue.
  3. The remaining node is the root node and the tree is complete.

     Since efficient priority que data structures require $O(logn)$ time per insertion, and a tree with n leaves has $2n-1$ nodes, this algoirhtm operates in $O(n \log n)$ time, where n is the number of symbols.

- *Describe the greedy algorithm for logical reasoning with Horn formulas.*

```
Input: a horn formula
Output: a satisfying assignment, if one exists

set all variables to false
while there is an implication that is not satisfied:
    set the right-hand variable of the implication to true

if all pur negative clauses are satisfied: return the assignment
else: return ``formula is not satisfiable``
```

- *What is the property of the greedy algorithm for the set cover example? Prove it.*

Elements of Dynamic Programming, Matrices, Subsequences

Chapter 6.2 from DPV: Longest increasing subsequences

- In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \cdots, a_n$.
  - A *subsequence* is any subset of these number taken in order, of the form $a_{i_1}, a_{i_2}, \cdots, a_{i_k}$ where $1 \geq i_1 < i_2 < ... < i_k \leq n$ and an *increaing* subsequence is one in which the numbers are getting strictly larger.

- For instance, consider the following sequence:

  5 2 8 6 3 6 9 7

  - The longest increasing subsequence is 2, 3, 6, 9.

- You can express this problem as a graph problem where the graph is a directored acyclic one, notice two properties:
  1. It's a DAG
  2. There is a one-to-one correspondance between increasing subsequences and paths in this dag.
- Just find the longest path in the dag!

```
for j = 1, 2, ..., n:
    L[j] = 1 + max{L(i) : (i, j) in E}
return max_j L(j)
```

- This is dynamic programming. We have defined a collection of subproblems with the following property that allows them to solved in a single pass:

  (*) There is an ordering on the subporblems, and a relation that shows how to solve a subproblem given the answer to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

- Each subproblem is solved using the relation:

$$L(j) = 1 + max\{L(i) : (i, j) \in E\}$$

Chapter 6.3 from DPV: Edit distance

- When a spell-checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?
  - Perhaps how much they can be *aligned* or *matched-up*?

```
S - N O W Y   - S N O W - Y
S U N N - Y   S U N - - N Y
```

A dynamic programming solution

- Doing this, the most crucial question is: *What are the subproblems?*
  - As long as they have the propert (*), it is easy to write down the algorithm: iteratively solve one problem after the other in order of increasing size.
- Our goal is to find the distance between two strings. What is a good subproblem?
  - Find the prefix of the first and second strings.

Chapter 15.2 from CLRS2: Matrix-chain multiplication

- An example of *dynamic programming.*

```
MatrixMultiply(A, B) {
    if columns(A) != columns(B)
        then error "incompatible dimensions"
        else for i <- 1 to rows[A]
            do for j <- 1 to columns[B]
                do C[i, j] <- 0
                    for k <- 1 to clumns[A]
                        do C[i, j] <- C[i, j] + A[i, k] * B[k, j]
        return C
}
```

Chapter 15.3 from CLRS2: Elements of Dynamic Programming

- The elements of dynamic programming: optimal substructure and overlapping subproblems.

Optimal substructure If an optimal solution to the poroblem contains within it optimal solutions to subproblems.

- How to find it, some characteristic:
  1. You should that a solution to the problem consists of making choices, where making a choice leaves a subproblem to be solved.
  2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this, you just assume it's been given to you.
  3. Given this choice, you determine which subproblems ensue and how to best characterize the result space of subproblems.
  4. You show that the solutions to the subprobelsm used within the optimal solution to the problem must themselves be optimal. You do so by supposing otherwise and deriving a contradiction.
- Optimal substructure differes in two ways:
  1. How many subproblems are used in an optimal solution to the original problem
  2. How many choices we have in determineing which subporblems to use in an optimal solution.

Overlapping subproblems When a recursive algorithm revists the same problem over and over again, we say that the optimization problem has *overlapping subproblems.*

Chapter 15.4 from CLRS2: Longest common subsequence

Practice questions (3 left)

- *How does the dynamic programming approach decompose the chain-matrix multiplication into subproblems and how are their solutions combined in order to solve a larger problem?*

  Suppose that an optimal parenthesization of $A_iA_{i+1}\cdots A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the "prefix" subchain $A_iA_{i+1}\cdots A_k$ whithin this optimal parenthesization of $A_iA_{i+1}\cdots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\cdots A_k$. Why? If there were a less costly way to parenthesize $A_iA_{i+1}\cdots A_k$ substituting that parenthesization in the optimal parenthesization of $A_iA_{i+1}\cdots A_j$ would produce another parenthesization of $A_iA_{i+1}\cdots A_j$ whose cost was lower than optimum: a contraduction. A similar observation holds for the parenthesization of the subchain $A_{k+1}A_{k+2}\cdots A_j$ in the optimal parenthesization of $A_iA_{i+1}\cdots A_j$: it must be an optimal parenthesization of $A_{k+1}A_{k+2}\cdots A_j$.

- *What is the dynamic programming algorithm for chain-matrix multiplication? What is its running time?*
  1. Take the sequence of matrices and separate it into two subsequences.

2. Find the minimum cost of multiplying out each subsequence.
3. Add these costs together, and add in the cost of multiplying the two result matrices.
4. Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

- *You may be asked to compute a product of matrices using the dynamic programming approach and report the number of operations required given the dimensionality of the input matrices.*
- *What type of subproblems does the dynamic programming approach consider for solving the longest increasing subsequence and how does it combine them in order to solve larger ones?*

  The subproblems for the LIS problem as the suffixes for any given input. Find the longest increasing subsequence $A[j..n]$ that includes $A[j]$. Do this for each $j > i$.
- *What is the running time of the dynamic programming solution for the longest increasing subsequence problem?*

  There are n subproblems. For every subproblem i, the worst case is that you need go through all the following items to get the maximum, which is $O(n-i)$

$$\sum_{i=1}^{n} O(i) = O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

- *You may be given an example sequence and asked to compute its longest increasing sub- sequence following a dynamic programming approach. Your solution should provide the intermediate values computed by the algorithm (e.g., length of longest subsequence, previous pointer for each digit).*
- *What are the subproblems defined by the dynamic programming approach for the longest common subsequence problem and how are they combined to solve larger problems?*

  Let $X = \{x_1, x_2, \cdots, x_m\}$ and $Y = \{y_1, y_2, \cdots, y_n\}$, and let $Z = \{z_1, z_2, \cdots, z_k$ be any LCS of X and Y.
  Three cases:

  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
  2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
  3. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X and Y_{n-1}.
- *What is the running time of the dynamic programming solution for the longest common subsequence problem?*

  To build th LCS, you must construct a table with m columns and n rows. Each entry into the table is constant time.

$$O(m \times n)$$

- *You may be given two strings and asked to compute their longest common subsequence following a dynamic programming approach. Your solution should provide the intermediate values computed by the algorithm (e.g., the matrix of values corresponding to the cost of the matching and the parent pointers).*

Elements of Dynamic Programming, Knapsack problem, Graph Representation

Chapter 6.4 from DPV: Knapsack

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His knapsack will hold a total weight of *at most* W pounds. There are N items to pick from, of which $w_1, \cdots, w_n$ and dollar value $v_1, \cdots, v_n$. What's the most valuable combination he can fit in his bag?

Knapsack with repetition

- What are the subproblems? We can look at it two ways:
  1. We can look at smaller knapsack capacities $w \leq W$
  2. We can look at fewer items (for instance, items $1, 2, \cdots, j$, for $J \leq n$)
- The first restriction, roughly,

  $K(w)$ is the maximum achievable value with a knapsack of capacity w.

  - Can we express this in terms of smaller subproblems? Well, if you remove an item i from $K(w)$ In other words, $K(w) = K(w - w_i) + v_i$
- If the optimal solution to $K(w)$ includes the item i, then removing this item from the knapsack leaves an optimal solution to

$$K(w) = \max_{i:w_i \leq w}\{K(ww_i) + v_i\}$$

```
K(0) = 0
for w = 1 to W:
    k(w) = max \{ K(w - w_i) + v_i \}
return K(W)
```

Knapsack without repetition

$K(w, j)$ is the maximum value achievable using a knapsack of capacity w and items $1, \cdots, j$.

- Either item j is needed to achieve optimal value, or it isn't needed:

$$K(w, j) = max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

```
Initmizalize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n:
    for w= 1 to W:
        if w_j > w: K(w, j) = K(w, j - 1)
        else: K(w, j) = max(K(w, j - 1), K(w - w_j, j - 1) + v_j)

return K(W, n)
```

Chapter 3.1 from DPV: Why graphs?

- We can represent a graph with an *adjacency matrix*, if there are $n = |V|$ vertice $v_1, ..., v_n$, this is an $n \times n$ array whose (i, j)th entry is:

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge } v_i \to v_j \\ 0, & \text{otherwise.} \end{cases}$$

  - The biggest convinience of this is that an edge can be checked in constant time.
  - However, it takes $O(n^2)$ space, wasteful if not many edges.
- An alternative representation is an *adjacency list.*

  - It consists of $|V|$ linked lists, one per vertex.
  - Each edge appear in exactly one of the linked lists if the graph is directed or two of the lists if the graph is undirected.
  - The total size of the data structure is $O(|E|)$.
  - Checking an edge is no longer linear, sift through u's adjacency list.

Chapter 22.1 from CLRS2: Representations of Graphs

- A graph is sparse if $|E|$ is much less than $|V|^2$.
- A graph is dense if $|E|$ is close to $|V|^2$.
- Adjency list representations of graphs consist of any array of $|V|$ lists, one for each vertex in V.
  - The adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$.

Practice questions (2 left)

- *What are the subproblems defined by the dynamic programming approach for the knapsack problem with repetition how are they combined to solve larger problems? What is the runtime?*

  Define $K(w)$ as the maximum value achievable with a knapsack of capacity w. The subproblems, then, where v is value of item w at i:

  $$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

  The algorithm falls nicely out:

  ```
  K(0) = 0;
  for w = 1 to W:
      K(w) = max(K(w - w_i) + v_i : w_i <= w)
  return K(W)
  ```

  Each entry can tke up to $O(n)$ time to compute, so the overall running time is

  $$O(nW)$$

- *What are the subproblems defined by the dynamic programming approach for the knapsack problem without repetition how are they combined to solve larger problems? What is the runtime?*

  Define $K(w, j)$ as the "maximum value chieable using a knapsack of capacity w and items $1, \cdots, j$." The answer we seek is $K(W, n)$. The subproblems, therefore, are where j is either needed or not:

  $$K(w, j) = \max\{L(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

  The algorithm consists of filling out a two dimesional table, with $W + 1$ rows and $n + 1$ columns. Each table entry is constant time, even though the table is much larger than in the previous case, the running time is still:

  $$O(nW)$$

- *You may be given the parameters of a knapsack problem and asked to compute the answer following a dynamic programming approach (for either case: with or without repetition). Your solution should provide the intermediate values computed by the algorithm (e.g., the vector or matrix of intermediate payoffs and objects selected by the intermediate solutions).*

- (See handout)
- *What are the advantages and disadvantages of an adjacency matrix representation for graphs?*
  - Advantages:
    - Constant time checking of edges.
  - Disadvantages:
    - Requires $O(n^2)$ space no matter how dense or sparse.
- *What are the advantages and disadvantages of an adjacency list representation for graphs?*
  - Advantages:
    - Less space than matrix, very small for sparse graphs.
  - Disadvantages:
    - For dense graphs, look up for edges will be costly, linear.

Depth-First Search

Chapter 3.2 from DPV

- Depth-first search is a verstile linear-time procedire that reveals a wealth of information about a graph.

  What parts of the graph are reachable from a given vertex?

```
procedure explore(G, v) {
    input:  G=(V, E) is a graph; v in V
    output: visited(u) is set to true for all nodes u reachable from v

    visited(v) = true;
    previsit(v);

    for each edge(v, u) in E:
        if not visited(v): explore(u);

    postvisit(v);
}

procedure dfs(G) {
    for all v in V
        visited(v) = false;

    for all v in V
        if not visisted(v): explore(v);
}
```

- Analysis:
  - To mark all as unvisisted:

$$O(|V|)$$

  - Each edge is exacmied exactly twice,

$$O(|E|)$$

  - Therefore,

$$O(|V| + |E|)$$

Connectivity in undirected graphs

- An undirected graph is connected if there is a path between any pair of vertices.
- In unconnected graphs, there are at least two connected components.
- All it takes to make DFS assign connected component numbers to each on,

```
procedure previsit(v) {
    ccnum[v] = cc
}
```

Chapter 3.3 from DPV

Chapter 3.4 from DPV

Chapter 22.3 from CLRS2

Chapter 22.4 from CLRS2

Chapter 22.5 from CLRS2

- An application for DFS: decomposisiting a graph into strongly connected components.
  - To be a strongly connected component of a directed graph is to have a maxmimal set $C \subset V$ such that for every pair of vertices u and v in C, we have both a path from u to v and vice versa.

Practice questions

- *Provide an algorithm that discovers in linear time the set of nodes in a graph reachable from a specific node. Argue about its correctness.*

```
function get_connected_nodes(node){
    nodes = set();
        foreach(child in node.children){
                nodes.add(child);
                        nodes = nodes.union(get_connected_nodes(child));
                            }
                                return nodes;
                                }
        }
}
```

- *Provide an algorithmic description for depth-first search. What is its running time?*

  Depth first search operates by first marking all vertexes in a graph as unvisisted. Then, it travereses all the vertexes in the graph. If a vertex has not been visisted, that vertex is explored. When a vertex is explored, it is set to have been viststed, and subseqeuntly every vertex that it shares and edge with is explored as well (that is, recursively). Optionally, the explore function can have a previsit and postvisit operation.

There are two important elements to an analysis of depth first search. First, there is the marking of every node as unvisisted, which will take as much time as there are nodes. Then, note that every edge is visisted twice in the explore function. First, when it is found in the first node, and second, when its found in a node that looks at it, that is, in a undirected graph.

$$O(|E| + |V|)$$

- *What is the pre-visit and post-visit order of nodes in depth-first search?*

  For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other. Because $[pre(u), post(u)]$ is essentially the time during which vertex u was on the stack. The last in, first out behavior of the stack.

- *How can depth-first search be used in order to detect the connected components of a graph?*

  If you modify the previsit function of DFS's explore function to number the visist node with a number that begins at 0 and is incremenented every time that explore function is called in the DFS function, you'll get marked nodes with a mark of their component.

- *You may be provided an undirected or directed graph, a start node and asked to provide the search tree arising from a depth-first search. You may also be asked to identify tree edges, forward edges, back edges and cross edges (directed cases).*

- *Show that a directed graph has a cycle if and only if its depth-first search reveals a back edge.*

  If (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

  If the graph *has* as cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$, look at the first node on this cycle to be discovered. Suppose it is v_i. All other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree. In particular, the edge $v_{i-1} \to v_i$ (or $v_k \to v_0$ if $i = 0$) leads from a node to its ancestor and is thus by definition a back edge.

- *What is a directed acyclic graph (DAG)?*

  A cycle in a directed graph is a circular path. A graph without one of these is acyclic. If that acyclic graph's edges have direction, then it is a DAG.

- *What is the topological ordering of nodes in a DAG and why it is useful?*

  To linearize a DAG, perform DFS and perform tasks in decreasing order of their post number.

  Alternatively, find a source, output it, and delete it from the graph.

  It's useful because you can model causalityies, hierarchies, and temporal dependancies.

- *How is it possible to identify a sink node or a source node in a DAG using depth-first search?*

  To indentify a sink node in a DAG, perform DFS and when a node doesn't have any edges, it will be a sink.

  To identify a source node in a DAG using DFS,

- *What is the definition of strongly connected components?*

  If two components in a graph have an edge between u and v going in both directions, then those components are strongly connected.

- *How is it possible to compute the decomposition of a directed graph into its strongly connected components? Provide an algorithm and argue about its running time.*

  The algorithm:

  - Let G be a directed graph and S be an empty stack.
  - While S does not contain all vertices:
    - Choose an arbitrary vertex v not in S. Perform DFS starting at v. Each time that DFS finished expanding a vertex u, push u onto S.
  - Obtain the transpose of G.

  - While S is not empty:
    * Pop the top vertex v from S.
    * Perform DFS starting at v in the transpose graph. The set of visisted vertices will give the strongly connected component containing v.
    * Record this and remove all these vertices from the graph G and the stack S.

The run time: This algorithm performs two traversals of the graph.

$$O(|V| + |E|)$$

Breadth-First Search, Properties of Shortest Paths

Chapter 4.1 from DPV: Distances

- The *distance* between two nodes is the length of the shortest path between them.

Chapter 4.2 from DPV: Breadth-first search

- Each vertex is queued exactly once, when it is first encountered, so there are $2|V|$ queue operations.
  - The rest of the work is done in the algorithm's innermost loop.
  - Over the course of execution, this loop looks at each edge once in directed graphs or twice in undirected graphs, and therefore takes $O(|E|) time.
- Depth first search makes deep uncursion into a graph, retreating only it runs out of new nodes to visit.
  - BFS makes sure to visist vertices in increasing order of their distance from the starting point. This is broader, shallower search, rather like the progpagation of a wave upon water.
  - Same as DFS, but with a queue instead of a stack.
  - Nodes not reachable from s are simply ignored.

Chapter 22.2 from CLRS2: Breadth-first search

- BFS is one of the simplest algorithms for searching a graph.
- Given a graph $G = (V, E)$ and a dinstinguished source vertex s, BFS systematically explores the edges of G to "discover" every vertex that is reachable from s.
- It's named this because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - Discoveres distances at k before $k + 1$.

Practice questions

- *Provide the algorithm that performs breadth-first search on a graph.*

```
procedure bfs(G, s)
input: Graph G=(V, E) directed or undirected; vertex s in V
output: For all vertices u reachable form s, dist(u) is set to distance
        from s to u.

for all u in V:
    dist(u) = inf
dist(s) = 0
Q = [s] (queue just containing s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) in E:
        if dist(v) = inf
            inject(Q, v)
            dist(v) = dist(u) + 1
```

- *What is the inductive argument for the correctness of the approach?*

  What we expect:

  For each $d = 0, 1, 2 \cdots$, there is a moment at which:

  1. All nodes at distance $\leq d$ from s have their distances correctly set;
  2. All other nodes have distances set to ∞; and
  3. The queue contains exactly the nodes at distance d.

  Proof: Assume, for the purpose of contraduction, that some vertex receives a d value not equal to its shortest path distance. Let v be th vertex with minimum $\Delta(s, v)$ and

- *What is the running time of the algorithm?*

$$O(|V| + |E|)$$

  Each vertex is put on the queue exactly once, when it is first encountered, so there are $2|V|$ queue operations. The loop then looks at each edge once (or twice in undirected), and therefore takes $O(|E|)$ time.

- *What is the definition of a single-source shortest path problem?*

  in which we have to find shortest paths from a source vertex v to all other vertices in the graph.

- *What is the definition of a single-destination shortest path problem?*

  in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v.

  Protip: Reverse the direction of SSSP.

- *What is the definition of a single-pair shortest path problem?*

  in which we have to find the shortest path from source s to destination d.

- *What is the definition of all-pairs shortest path problem?*

  in which we have to find shortest paths between every pair of vertices v, v' in the graph.

- *Which of these problems are equivalent?*

  None of them?

- *Do we know faster algorithms for the single-pair shortest path problem than the single-source shortest path problem?*

  Yes, single pair will be faster than single-source, single source has to compute more.

  Djikstra's: $O(|E| + |V| \log |V|)$

  Bellman-Ford: $O(|V| \times |E|)$

- *What is the "optimal substructure" property of shortest paths? Prove its validity.*

  If you split a shortest path in half, the shortest path from the source to halfway will be the first half of the shortest path to the destination.

- *What happens with shortest paths on graphs that contain negative cycles? Can a shortest path contain positive cycles?*

  1. There are going to be infinitely many shortest paths, you can always go around that vertex more than once.
  2. Yes.

Dijkstra's algorithm

Chapter 4.3 from DPV: Lengths on edges

- BFS treats all edges as having the same length. This is rarely true in applications where shortest paths are to be found.
  - Annotating every edge $e \in E$ with length l_e.
  - If $e = (u, v)$, we will sometimes also write $l(u, v)$ or l_{uv}

Chapter 4.4 from DPV: Dijkstra

Chapter 4.5 from DPV: Priority queue implementations

Array

- The simplimest implementation.
  - Key values for all potential elements.
- `insert` and `decreasekey` is fast, because it just involves adjusting the key value.
  - `deletemin` requires a scan of the list.

Binary heap

- `insert`: place a new element at the bottom of the tree and let it "bubble up", where the number of switches is at most $\log n$, height of the tree.
  - `decrease key` is similar.
- `deletemin` requires deleting the root.

Chapter 24.3 from CLRS2: Dijkstra's algorithm

- Solves the single single-source shortest path problem on a weighted directed graph $G = (V, E)$ for the case where the edge-weights are non-negative.

Practice questions

- *Provide Dijkstra's algorithm for computing single-source shortest paths.*

```
procedure dijkstra(G, l, s)
input:   Graph G = (V, E), directed or undirected;
         positive edge length {l_e : e in E}; vertex s in V
output:  For all vertices u reachable from s, dist(u) is set
         to the distance from s to u

for all u in V:
    dist(u) = inf
    prev(u) = nil
dist(s) = 0;

H = makeque(V) (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) in E:
        if (dist(v) > dist(u) + l(u, v))
        dist(v) = dist(u) + l(u, v)
        prev(v) = u
        decreaseKey(H, v)

djkstra(G, w, s) {
    initialize-single-source(G, s)
    s <- empty set
    q = V[G]
    while Q is not empty
        do u = extract-min(Q)
        S = S union {u}
        for each vertex v in Adj[u]
            do relax(u, v, w)
    }
```

- *What is the requirement in order to be able to apply Dijkstra's algorithm?*

  Edge weight must all be positive.

- *You may be provided an example graph and asked to return the search tree that arises from Dijkstra, as well as the state of the priority queue during its iteration of the algorithm.*

  Can do.

- *Prove the correctness of Dijkstra's algorithm on non-negative weight graphs.*

  Cannot do.

- *What is the best running time of Dijkstra's algorithm and for what implementation of the priority queue data structure? What is the running time of Dijkstra's algorithm if the priority queue is implemented as a simple array? What is the running time of Dijkstra's algorithm if the priority queue is implemented as a binary heap?*

Implementation	`deletemin`	`increase/decreasekey`	Total
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert)\log \lvert V\rvert)$

Implementation	`deletemin`	`increase/decreasekey`	Total
d-ary hear	$O(\frac{d \log \lvert V \rvert}{\log d})$	$O(\frac{\log \lvert V \rvert}{\log d})$	$O((\lvert V \rvert \times d + \lvert E \rvert)\frac{\log \lvert V \rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V \rvert)$	$O(1)$	$O(\lvert V \rvert \log \lvert V \rvert + \lvert E \rvert)$

- *When is each of these implementations preferred over the other?*

  It depends whether the the graph is *dense* or *sparse* edges. To be dense is to have many edges, to be sparse is to have fewer edges. For all graphs, $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then cearly the array implementation is faster. On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $||V|^2 \log |V|$.

Bellman-Ford and Floyd-Warshall algorithms

Chapter 4.6 from DPV: Shortest paths in the presence of negative edges

- How can you account for the fact that when you have a negative edge, the shortest path may go through more nodes than any other path?
- What has to change to accomodate this sort of change?
  - Distance estimates under dijkstra are always overestimates or exactly correct. Consider:

    ```
    procedure update((u, v) in E) {
        dist(v) = min(dist(v), dist(u) + l(u, v));
    }
    ```

  - This *update* operation is simply an expression of the fact that the distance to v cannot possibly be more than the distance to u. It has the following properties:
    1. It gives the corect distance to v in the case where it is the second last node in the shortest path to v.
    2. It will never make `dist(v)` too small, making it *safe*.

Negative cycles

- In situations with negative edge weights, it doesn't make sense to even ask about shortest paths.
- The shortest-path problem is ill-posed in graphs with negative cycles.
  - Shortest path slipped in when we used the notion of "existence" (evidently).
- There is a negative cycle if and only if some `dist` value is reduced during the final round.

Chapter 4.7 from DPV: Shortest paths in dags

- There are two subclasses of graph that exclude the possibility of negative cycles:
  1. Graphs without negative edges
  2. Graphs without cycles
- The key source of efficiency:

  In any path of a dag, the vertices appear in increasing linearized order.

- You can find the longest path in a dag by negating all edge lengths.

Chapter 6.1 from DPV: Shortest paths in dags, revisited

- The special feature of dags is that the nodes can be *linearized*, that is, put on a straight line from left to right.
- You can compute all distances with a single pass of the following algorithm:

```
initialize all dist(.) values to inf;
dist(s) = 0;
for each v in V\{s}, in linearized order:
    dist(v) = min_{(u, v) in E} (dist(u) + L(u, v));
```

- Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by indentifying a collection of subproblem and tacking them one by one, smallest first, using the answers to small problems to help figure out larger onces, until the whole lot of them are solved.
  - In dynamic programming, you aren't always given a dag, the dag is implicit.
  - It's nodes are the subproblems we define, and it edges are the dependancies between subproblems:
    - If we solve subproblem B we need to answer subproblem A, making a conceptual edge from A to B.

Chapter 24.1 from CLRS2: The Bellman-Ford Algorithm

- The Bell-Ford Algorithm solves the single-source shortest-path in the general case in which edge weight may be negative.
  - Given a weight, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source.
    - If there is such a cycle, the algorithm indicates that there is no such solution.
  - If there is no cycle, the algorithm produces the shortest paths and their weights.
- The algorithm uses relaxation, progessively decreasing an estimate $d[v]$ on the weight of a shortest path.
  - The algorithm returns true if and only if there are no negative weighted cycles that are reachable form the source.

Chapter 24.2 from CLRS2: Single-source shortest paths in directed acyclic graphs

- By relaxing the edges of a weighted dag according to a toplogical sort of vertices, we can compute shortest paths from a single source in $\Omega(V + E)$ time.
  - Shortest paths are always well-edfined in a dag, since even if there are non-negative edge weights, there are no negative cycles (because there are no cycles).
- The algorithm starts by topologically sorting the dag to impose a linear ordering on the vertices. If there is a path from u to v, it is topologically latter.
  - Only need a single pass.
    dag-shortest-paths(G, w, s) { topologically sort the vertices of G initialize-single-source(G, s); for each vertex U, take in topologically sorted order do for each vertex v in Adj[u] do relax(u, v, w); }
- The running tim eof this is easy.
  - The topological sort is
$$\Omega(V + E)$$
  - The initialize-single-source takes
$$\Omega(V)$$
  - Then to two inner loops examine every V and E.
$$\Omega(V + E)$$

Practice questions (4 left)

- *Provide the Bellman-Ford algorithm for computing single-source shortest paths.*

```
procedure shortest-paths(G, l, s) {
input:  Directed graph G = (V, E);
        edge lengths {l_e : e in E} with no negative cycles;
        vertex s in V
output: For all vertices u reachable form s, dist(u) is set
        to the distance from s to u.

for all u in V:
    dist(u) = inf;
    prev(u) = nil;

dist(s) = 0;
repeat|V| - 1 times:
    for all e in E:
        update(e);
}

procedure BellmanFord(list vertices, list edges, vertex source) {
    // step 1: initialization
    for each vertex v in vertices:
        if v is source then distace[v] := 0;
        else distace[v] := infinity
        precessir[v] := null

    // step 2: relax edges repeatedly
    for i from 1 to size(vertices) - 1:
        for each edge (u, v) with weight w in edges:
            if distace[u] + v < distance[v]:
            distance[v] := distance [u] + w
            predecessor[v] := u

    // step 3: check for negative weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph containts negative weight cycle."
}
```

- *What is the running time of the approach and why?*

  The Bellman-Ford algorithm run in time $O(VE)$ since the initialization takes $O(V)$ time, and each of the $|V| - 1$ passes over the edges takes $\Theta(E)$ time, and the for look takes $O(E)$ time.

- *Why is it correct?*

- *You may be provided a graph and asked to trace the dynamic programming matrix that arises from the operation of Bellman-Ford.*

- *How can you detect the existence of negative cycles using the Bellman-Ford algorithm?*

  There is a look at the end like so,

```
 for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph containts negative weight cycle."
```

- *What is an efficient approach for computing single-source shortest paths on directed acyclic graphs that may contain negative weight edges and what is the running time of this solution?*
- *Why is the Floyd-Warshall algorithm preferred over calling |V| times the Bellman-Ford algorithm on a graph to solve all-pair shortest path problems?*
- *What is the dynamic programming formulation of Floyd-Warshall, i.e., what are the subproblems defined by the algorithm and how are they combined in order to address more complex problems?*

Floyd-Warshall and Johnson's algorithms

Chapter 25.2 from CLRS2: The Floyd-Warshall algorithm

- A dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$.
  - Floyd-warshall runs in $\Omega(V^3)$ time.
  - Negative weight edges may be present, but we assume that there are no negative-weight cycles.
  - Will follow the dynamic-programming process to develop the algorith,

The structure of a shortest path

- The FW algorithm uses a different charaterization of the shortest path than the matrix-multiplcation-based all-pairs algorithms.
  - The algorithm considers the "intermediate" vertices of a shortest path, where intermediate means a node on the shortest path.
- Assume that the vertices of G are $V = \{1, 2, \cdots, n\}$ and consider the ubset $\{1, 2, \cdots, k\}$ of vertices for some k.
  - The Floyd-Warshall algorithm explain a relationship between path p and shortest paths from i to j with all intermediate nodes in $1, 2, \cdots, k-1\}$.

Chapter 25.3 from CLRS2: Johnson's algorithm for sparse graphs

- Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \log V + VE)$ time.
  - For sparese graphs, it is asymptotically better than either repeated squaring matrices of the Floyd-Warshall algorithm.
  - The algorithm either returns a matrix of short-path weights for all pairs of matrices or reports that the graph contains a negative weight cycle.
- Johnson's algorithm uses the technique of *reweighting*, which is:
  - If all edge weights w in graph $G = (V, E)$ are nonnegative, we can find shortest paths between all pairs of certices by running Dijkstra's algorithm once from each vertex.
  - With the Fibonacci-heap min-priority queue, the running time of this all-pairs will be $O(V^2 \log V + VE)$.
  - If G has negative-weight edges but no negative-weight cycles, we simply compute a new set of nonnegative edge weights that allow us to use the same method.
    - The new set of edge weights must satisfy two important properties:
      1. For all pairs of vertices u, $v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $w*$
      2. For all edges (u, v), the new weight $w*(u, v)$ is nonnegative. ### Chapter 6.6 from DPV2

Practice questions (3 left)

- *Provide the Floyd-Warshall algorithm for solving all-pair shortest path problems. What is the running time of the approach?*

```
floud-warshall() {
    let dist be a |V| x |V| array of minimum distances initialized to inf
    for each vertex v:
        dist[v][v] = 0;

    for each edge (u, v) {
        dist[u][v] := w(u, v); // the weight of the edge
    }

    for k from 1 to |V|
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j]
                    dist[i][j] = dist[i][k] + dist[k][j]
}
```

- *You may be provided a graph and asked to compute the dynamic programming matrix that arises from a few iterations of the Floyd-Warshall algorithm.*

- *What is the transitive closure of a graph and how can it be computed following a dynamic programming approach? What is the relation to the Floyd-Warshall algorithm?*

  The transitive closure of G is defined as the graph $G* = (V, E*)$, where $E* = \{(i, j) : \text{there is a path from vertex} i \text{to vertuex} j \text{in} G\}$.

- *If a graph has non-negative edges, is it preferable to run the Floyd-Warshall algorithm to solve an all-pair shortest path problem or is it preferable to call |V| times Dijkstra's algorithm?*

  The complexity for running Dijkstra on all nodes will be $O(EV + V^2 \log V)$. This complexity is lower than $O(V^3)$ iff $E < V^2$.

- *What is the main idea in Johnson's algorithm and why can it be preferable over the Floyd-Warshall algorithm when solving all-pair shortest path problems?*

  The main idea behind Johnson's algorithm is that it uses Bellman-Ford to remove all negative edge weights from a graph using an artificial vertex, and then performs Djikstra's algorithm to solve the all-pair shortest path problem.

  It will be preferable over Floyd-Warshall because it has a better run time.

- (See handout)

- *Given the above transformation of weights for a graph G(V,E), how should the values h be computed for the vertices of the graph so that all weights end up having non-negative values?*

- *Describe Johnson's algorithmic steps. What is its running time?*

```
Johnson(G) {
    compute G', where V[G'] = V[G] union {s},
        E[G'] = E[G] union {(s, v) : v in V[G]} and
        w(s, v) = 0 for all v in V[G]

    if Bellman-Ford(G', w, s) = false
        then print "the input graph contain a negative-weight cycle"
        else for each vertex v in V[G']
```

```
            do set h(v) to the value of delta(s, v)
                computed by Bellman-Ford

        for each edge (u, v) in E[G']
            do w*(u, v) := w(u, v) + h(u) - h(v);
        for each vertex u in V[G]
            do run Dikstra(G, w*, u) to compute delta *(u, v) for all v in V[G]
                for each vertex v in V[G]
                    do d_{u, v} = delta *(u, v) + h(v) - h(u)

        return D
}
```

- The steps to the algorithm:
  1. A new node q is added to the graph, connected by zero-weight edges to each of the other nodes.
  2. The Bellman-Ford algorithm is used, starting from the new vertex q to find for each vertex v the minimum height $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
  3. The edges of the original graph are reweighted using the value computed by the Bellman-Ford algorithm: an edge from u to v, having length $w(u, v)$, is given the new length $w(u, v) + h(u) - h(v)$.
  4. q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.
- Time complexity:
  * Using Fibonacci heaps in the implementatino of Dijkstra's algorithm is $O(V^2 \log V + VE)$.
    · The algorithm uses $O(VE)$ time for the Bellman-Ford stage of the algorith,
  * $O(V \log V + E)$ for each of the V instantiations of Dijkstra's algorithm.
  * Thus, when the graph is sparse, the total time will be faster than the Floyd-Warhshall algorithm, which solves the same problem in $O(V^3)$.

Related Exercises

Related exercises from DPV

- Chapter 3: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.9, 3.11, 3.12, 3.15, 3.16, 3.18, 3.21, 3.22, 3.26, 3.31
- Chapter 4: 4.1, 4.2, 4.3, 4.5, 4.8, 4.9, 4.10, 4.11, 4.12, 4.17, 4.21
- Chapter 5: 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, 5.26, 5.28, 5.29, 5.30, 5.31, 5.33, 5.34
- Chapter 6: 6.1, 6.2, 6.3, 6.4, 6.7, 6.12, 6.17, 6.18, 6.19, 6.21, 6.22

Related exercises from CLRS2

- Chapter 15: 15.2.1, 15.2-2, 15.2-4, 15.2-5, 15.3-3, 15.4-2- Chapter 16: 16.2-4, 16.2-5, 16.2-7, 16.3-1, 16-3.2, 16.3-3, 16.3-4, 16.3-5, 16.3-6
- Chapter 22: 22.1.3, 22.1.5, 22.2.3, 22.2.6, 22.3.1, 22.4.3, 22.4.5, 22.5.1, 22.5.4, 22.5.6
- Chapter 24: 24.3.4, 24.3.6, 24.3.8
- Chapter 25: 25.2.6, 25.2.8, 25.3.4