Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #12: Algorithmic Styles & Conclusion

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Understand *Shortest Path and Shortest Weighted Path*

• Understand *Minimum Spanning Trees*

• Familiarize with *Backtracking Algorithmic Style*

• Familiarize with *Dynamic Programming Algorithmic Style*

• Familiarize with *Exponential and NP Algorithms*

• Familiarize with *Garbage Collection*
Graph Algorithms
Shortest Path Finding
Weighted Shortest Path

• Now, let’s consider a more general form of the same problem. Attach to each edge a weight indicating the cost of traversing that edge.

• Find a path between designated start and finish nodes that minimizes the sum of the weights of the edges in the path.

• Example: What’s the cheapest airline route way to get from D.C. to N.Y?

• To deal with this problem, we adapt the unweighted shortest path algorithm.
  • Keep nodes in a collection (queue).
  • From the collection, we extracted a node closest to the start.
  • From that node we considered the smallest possible step, updating the distances of the adjacent nodes accordingly.
With weighted graphs, we do the same, but the **step size is determined by the weights**.

- We use a **priority queue** to keep track of the nodes closest to the start.
- We will use a **map** (VDistMap) to associate with each vertex the shortest distance known so far from the start to that vertex.
  - This value starts impossibly high, so that any path we find to that vertex will look good by comparison.
- The **CompareVerticesByDistance** structure provides a **comparison operator** based upon this **shortest-distance-known-so-far**.
  - We’ll use that operator to maintain a priority queue of the vertices based upon the distance.
- The **edge weights** are passed to our algorithm as another **map**, this one **keyed on edges** instead of vertices.

```cpp
typedef hash_map<Vertex, unsigned, VertexHash> VDistMap;
struct CompareVerticesByDistance {
    VDistMap& dist;
    CompareVerticesByDistance(VDistMap& dm):dist(dm){}
    bool operator() (Vertex left, Vertex right) const{
        return dist[left] > dist[right];
    }
};

void findWeightedShortestPath(DiGraph& g, list<Vertex>& path, Vertex start, Vertex finish, hash_map<Edge, int, EdgeHash>& weight ){
    // Dijkstra’s Algorithm
    ...
}
```
Dijkstra Example

Q: Find Shortest Path from A to D

Initialize:

Q: A B C D E
   0   ∞   ∞   ∞   ∞

S: {}
Dijkstra Example

From:  --
Dijkstra Example

\[
\begin{array}{c|c|c|c|c|c}
    & A & B & C & D & E \\
\hline
    Q: & 0 & \infty & \infty & \infty & \infty \\
    & 10 & 3 & \infty & \infty & \infty \\
\end{array}
\]

\[
S: \{ A \}
\]

From: --
Dijkstra Example

\[ S: \{ A, C \} \]

From: -- A

\[ Q: \begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty \\
\end{array} \]
Dijkstra Example

\[ Q: \begin{array}{cccccc}
 A & B & C & D & E \\
 0 & \infty & \infty & \infty & \infty \\
 10 & 3 & \infty & \infty & \infty \\
 7 & 11 & 5 & & & \\
\end{array} \]

\[ S: \{ A, C \} \]
Dijkstra Example

From:    --  A  C

Q:      A  B  C  D  E

0  ∞ ∞ ∞ ∞
10 3 ∞ ∞
7  11 5

S: { A, C, E }

Page 12 Fall 2013 CS 361 - Advanced Data Structures and Algorithms
Dijkstra Example

From:  --  A  C

\[ Q: \begin{array}{cccccc}
   & A & B & C & D & E \\
   0 & \infty & \infty & \infty & \infty & \\
   10 & 3 & \infty & \infty & \\
   7 & 11 & \infty & \\
   7 & 11 & \\
\end{array} \]

\[ S: \{ A, C, E \} \]
Dijkstra Example

$Q:\begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty \\
7 & 11 & 5 \\
7 & 11 & \infty
\end{array}$

From: -- C A C

$S:\{ A, C, E, B \}$
Dijkstra Example

Q:
\[
\begin{array}{cccccc}
A & B & C & D & E \\
0 & \infty & \infty & \infty & \infty \\
10 & 3 & \infty & \infty & \infty \\
7 & 7 & 11 & 5 & \\
\end{array}
\]

S: \{ A, C, E, B \}

From: -- C A C
Dijkstra Example

\[
Q: \begin{array}{cccccc}
    & A & B & C & D & E \\
A & 0 & \infty & \infty & \infty & \infty \\
B & 10 & 3 & \infty & \infty & \infty \\
C & 7 & 11 & 11 & 5 & \\
D & 7 & & & & \\
E & 7 & & & & \\
\end{array}
\]

\[
S: \{ A, C, E, B, D \}
\]

From: -- C A B C
**Dijkstra’s Algorithm**

- The algorithm we will develop is called Dijkstra’s algorithm.
- We begin by initializing distance map and priority queue.
- Repeatedly remove from the priority queue the vertex, \( v \), closest to the start.
- We look in the distance map to get that vertex’s minimum distance from the start, \( d \).
- Then we look at each outgoing edge of \( v \) with adjacent vertex \( w \).
- We know we can reach \( w \) with distance \( d + \text{weight}[*e] \).

```cpp
void findWeightedShortestPath (DiGraph& g, list<Vertex>& path, Vertex start, Vertex finish, map_hash<Edge, int, EdgeHash>& weight{

    Map_hash<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = INT_MAX;
    }

    dist[start] = 0;
    priority_queue <Vertex, VDistMap, CompareVerticesByDistance> pq(g.vbegin(), g.vend(), CompareVerticesByDistance (dist));
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
            Vertex w = (*e).dest();
            if (dist[w] > d + weight [*e]) {
                dist[w] = d + weight [*e];
                pq.adjust(w);
                cameFrom[w] = v ;
            }
        }
    }
    Vertex v = finish; // Extract path
    if (dist[v] != INT_MAX){
        while (!((v == start)) { path.push_front(v); v = cameFrom[v]; }
    }
}
```
Dijkstra’s Algorithm

• It’s possible that exist a shorter way to get to $v$. So we compare the value $d + \text{weight}[*e]$ to the shortest known distance already recorded for $w$.

• If we already know a shorter way to get to $w$, we leave it alone.

• If, however, this new distance is shorter, we update the shortest known distance for $w$
  • adjust its position in the priority queue
  • record that the shortest known way to reach $w$ is via $v$

• This continues until we have emptied out the priority queue.

```c++
void findWeightedShortestPath (DiGraph& g, list<Vertex>& path, Vertex start, Vertex finish, map_hash<Edge, int, EdgeHash>& weight{
    Map_hash<Vertex , Vertex , VertexHash> cameFrom;
    VDistMap dist;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue <Vertex, VDistMap, CompareVerticesByDistance> pq(g.vbegin(), g.vend(), CompareVerticesByDistance (dist));
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e ){
            Vertex w = (*e).dest();
            if (dist[w] > d+weight[*e]) {
                dist[w] = d + weight[*e];
                pq.adjust(w);
                cameFrom[w] = v ;
            }
        }
    }
    Vertex v = finish ; // Extract path
    if (dist[v] != INT_MAX){
        while (!(v == start)) { path.push_front(v); v = cameFrom[v]; } 
        path.push_front (start);
    }
}
```
void findWeightedShortestPath (DiGraph& g, list<Vertex>& path, Vertex start, Vertex finish, map_hash<Edge, int, EdgeHash>& weight{ 
    Map_hash<Vertex , Vertex , VertexHash> cameFrom; 
    VDistMap dist; 
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        // O(|V|)
        dist[*vi] = INT_MAX; // O(1)
    }
    dist[start] = 0;
    priority_queue <Vertex, VDistMap, CompareVerticesByDistance> pq(g.vbegin(), g.vend(), CompareVerticesByDistance(dist));
    while (!pq.empty()) {
        // O(|V|)
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop(); // O(1)
        for (AllOutgoingEdges e = g.outbegin(v), e != g.outend(v), ++e) {
            // O(1)
            Vertex w = (*e).dest();
            if (dist[w] > d+weight[*e]) {
                // O(1)
                dist[w] = d + weight[*e];
                // O(1)
                pq.adjust(w);
                cameFrom[w] = v;
            }
        }
    }
    Vertex v = finish; // Ext race path
    if (dist[v] != INT_MAX){
        // O(1) // O(|V|)
        while (!v == start) { path.push_front(v); v = cameFrom[v]; } 
        path.push_front (start);
    }
};

• Dijkstra’s algorithm is similar to the unweighted shortest path algorithm
• The introduction of the priority queue makes some of the steps more expensive than in the unweighted case.
• The first and final loops are obvious of $O(|V|)$.
• The declaration an initialization of the priority queue is $O(|V|)$
• Adjust call on the priority queue are of $O(\log |V|)$.

Dijkstra's Algorithm - Analysis

- $O(1)$
- $O(1)$
- $O(1)$
- $O(1)$
- $O(1)$
- $O(1)$
- $O(|V|)$
- $O(|V|)$
- $O(|V|)$
- $O(\log |V|)$
- $O(1)$
- $O(|V|)$
- $O(1)$
- $O(1)$
Dijkstra’s Algorithm - Analysis

• The **inner loop** executes a total of $|E|$ times.

• The **outer loop** executes $|V|$ times.

• So the total cost of adjusting is $O(|E| \log |V|)$, and the cost of the rest of the two loops is $O(|E|+|V|)$.

• The total is $O(|E|+|V|+|E| \log |V|)$.

• But $|E| \log |V| > |E|

  • $\Rightarrow$ this simplifies to $O(|V|+|E| \log |V|)$.

• The total cost of Dijkstra’s algorithm is $O(|V|+|E| \log |V|)$.
Minimum Spanning Trees
Minimum Spanning Tree

• We want to find the subgraph of the original graph that:
  • is **connected**
  • **spans** (i.e., has every vertex and some edges from) the original graph
  • Involves the edges leading the *minimum possible sum of weights*

• There will only be one path from any vertex to any other vertex
  • if there were more than one → a redundant connection somewhere.
  • Any connected graph with that property is a *tree* known as the “**minimum spanning tree**”

![Network of Hubs](image)

![Minimum spanning tree](image)

Minimum amount of cable = 241
Prim’s Algorithm

- One solution, *Prim’s algorithm*, turns out to be almost identical to Dijkstra’s.
- The main difference is that, *instead of* keeping track of the total *distance to each vertex from the start*, we base our priority queue on the *minimum distance to the vertex from any vertex* that has already been processed.
- Collect edges in `cameFrom` instead of vertices.
- The “*distance*” associated with each vertex is simply the *smallest edge weight* seen so far.

```cpp
void findMinSpanTree (DiGraph& g, set<Edge>& spanTree, 
                     Vertex start, hash_map<Edge, int, EdgeHash>& weight) {

    VDistMap dist;
    hash_map<Vertex, Edge, VertexHash> cameFrom;
    for (AllVertices vi = g.vbegin; vi != g.vend; ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue<Vertex, VDistMap,
                    CompareVerticesByDistance>
                    pq (g.vbegin, g.vend(), CompareVerticesByDistance(dist));

    // Collect edges of span tree
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
            Vertex w = (*e).dest();
            if (dist[w] > weight[*e]) {
                dist[w] = weight[*e];
                pq.adjust (w);
                cameFrom[w] = *e;
            }
        }
    }

    for (hash_map<Vertex, Edge, VertexHash>::iterator p = 
         cameFrom.begin(); p != cameFrom.end(); ++p)
        spanTree.insert ((*p).second);
}
```
Prim’s Algorithm - Example

- At each stage, the algorithm adds a new vertex and an edge that connects the new vertex to the ones in the tree.
- The algorithm starts with any initial vertex.

(a) Spanning tree with vertices A, B
minSpanTreeSize = 2, minTreeWeight = 2

(b) Spanning tree with vertices A, B, D
minSpanTreeSize = 3, minTreeWeight = 7

(c) Spanning tree with vertices A, B, D, C
minSpanTreeSize = 4, minTreeWeight = 14

Q: A B C D
-- 2 12 5

Q: A B C D
-- 2 12 5
-- 2 12 5

Q: A B C D
-- 2 12 5
-- 2 7 5

Q: A B C D
-- 2 12 5
-- 2 7 5
Prim’s Algorithm - Analysis

• The analysis of Prim’s algorithm is **identical to Dijkstra’s**.

• The total cost of Prim’s algorithm is $O(|V|+|E| \log |V|)$.

```cpp
void findMinSpanTree (DiGraph& g, set<Edge>& spanTree, Vertex start, hash_map<Edge, int, EdgeHash>& weight) {
    VDistMap dist;
    hash_map<Vertex, Edge, VertexHash> cameFrom;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
        pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist));
    // Collect edges of span tree
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
            Vertex w = (*e).dest();
            if (dist[w] > weight[*e]) {
                dist[w] = weight[*e];
                pq.adjust (w);
                cameFrom[w] = *e;
            }
        }
    }
    for (hash_map<Vertex, Edge, VertexHash>::iterator p =
        cameFrom.begin(); p != cameFrom.end(); ++p)
        spanTree.insert ((*p).second);
}
```
Algorithmic Styles
Recursion vs. Iteration

• **Recursion** and **iteration** (looping) are equally powerful.

• Any recursive algorithm can be rewritten to use loops instead.
  • The opposite is also true. Any iterative algorithm can be written in terms of recursion only.

• The decision of which to use comes down to issues of:
  • **Expressiveness:** some algorithms are just naturally simpler or easier to write with loops. Some are simpler with recursion.
  • **Performance:** Iteration is usually (though not always) faster than an equivalent recursion.

• Certain distinctive **styles of algorithms** have arisen over the years.
Algorithmic Styles

- Among both iterative and recursive algorithms, we recognize some algorithms by an underlying “idea” of how they work:
  - Simple recursive
  - Divide and conquer
  - Generate and test
  - Backtracking
  - Convergent
  - Dynamic programming
  - Randomized

- Now let’s look at some common forms of both iterative and recursive algorithms.
Divide and Conquer

• The most *widely recognized* algorithmic.

• A problem is *broken into two or more smaller sub-problems*, the sub-problems are solved, and the *resulting sub-problem solutions are recombined* into a whole solution.

• Divide and conquer algorithms frequently are *easiest to express recursively*, though iterative forms are not unknown.

• *Examples* of divide and conquer that we have seen include the binary search, merge sort, and quick sort.
Generate and Test

• Uses some relatively *quick process* to produce a series of “*guesses*” as to the appropriate solution

• Then *tests each guess* in turn to see if it is, in fact, a solution.

• **Example**: use “generate-and-test” to produce random permutations.
  • The random numbers are used to generate a possible value for the \( i \)th number in the permutation.
  • Then a search is used to test and see if that number has already been used.

• Generate-and-test is often used as a *fall-back* when we can’t come up with a better algorithm.

```c
// Generate a random permutation of the integers from 0..n-1, storing the results in array a.

void permute1 (int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        a[i] = rnd(n);
        while ( unorderedSearch(a,i,a[i]) >= 0 )
            a[i] = rnd(n);
    }
}
```
Backtracking

• A variation of generate-and-test is backtracking.

• Backtracking is a technique that can be applied to problems where you have a large, but finite number of variables, each of which may take on a number of discrete values.

• There is some overall test to decide if the entire set of assignments represents an acceptable solution.

• **Example:** Given a maze, find a path from start to finish
  • At each intersection, you have to decide between three or fewer choices: Go straight, Go left, or Go right
  • You don’t have enough information to choose correctly
  • Each choice leads to another set of choices
  • One or more sequences of choices may (or may not) lead to a solution
Backtracking

start

? ?

? ?

dead end

dead end

dead end

dead end

dead end

success!
Backtracking

A tree is composed of

There are three kinds of nodes:

- The (one) root node
- Internal nodes
- Leaf nodes

Backtracking can be thought of as searching a tree for a particular “goal” leaf node.
Backtracking – Eight Queens Problem

• Find an arrangement of 8 queens on a single chess board such that no two queens are attacking one another.

• In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).
  
  ° Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.
The backtracking strategy is as follows:

1) Place a queen on the first available square in row 1.

2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).

3) Continue in this fashion until either:
   a) you have solved the problem, or
   b) you get stuck.

- When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.
Backtracking – Eight Queens Problem

• The neat thing about coding up backtracking, is that it can be done recursively, without having to do all the bookkeeping at once.
  
  • Instead, the stack or recursive calls does most of the bookkeeping

  • (ie, keeping track of which queens we've placed, and which combinations we've tried so far, etc.)
Backtracking – Eight Queens Problem

`perm[]` - stores a valid permutation of queens from index 0 to location-1.

`location` – the column we are placing the next queen

`usedList[]` – keeps track of the rows in which the queens have already been placed.

```c
void solveItRec(int perm[], int location, struct onesquare usedList[]) {
    if (location == SIZE) {
        printSol(perm);
    }

    for (int i=0; i<SIZE; i++) {
        if (usedList[i] == false) {
            if (!conflict(perm, location, i)) {
                perm[location] = i;
                usedList[i] = true;
                solveItRec(perm, location+1, usedList);
                usedList[i] = false;
            }
        }
    }
}
```

- Found a solution to the problem, so print it!
- Loop through possible rows to place this queen.
- Only try this row if it hasn’t been used
- Check if this position conflicts with any previous queens on the diagonal
  1) mark the queen in this row
  2) mark the row as used
  3) solve the next column location recursively
  4) un-mark the row as used, so we can get ALL possible valid solutions.
Eight Queens Problem - Analysis

• The basic **brute-force** algorithm generates all possible permutations of the numbers 1 through 8 (of which there are \(8! = 40,320\)),
  ◦ Uses elements of each permutation as indices to place a queen on each row.
  ◦ Then it rejects those boards with diagonal attacking positions.

• The overall algorithm is \(O(\text{Size}^{\text{Size}}) = O(N^N)\). We say that this algorithm is **exponential** in Size.
  • Such exponential growth is worse, for sufficiently large inputs, than any polynomial big-O().

• The backtracking algorithm, is a slight improvement on the basic method
  ◦ constructs the search tree by considering one row of the board at a time, **eliminating most non-solution board positions at a very early stage** in their construction.
  ◦ Because it rejects row and diagonal attacks even on incomplete boards, it examines only \(15,720\) possible queen placements.
Dynamic Programming
Dynamic Programming

• **Dynamic Programming** is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances (subproblems)

• **Main idea:**
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table
Dynamic Programming - Example

- Recall definition of Fibonacci numbers:

\[ F(n) = F(n-1) + F(n-2) \]
\[ F(0) = 0 \]
\[ F(1) = 1 \]

- Computing the \( n^{th} \) Fibonacci number recursively (top-down):

\[ F(n) \]
\[ F(n-1) + F(n-2) \]
\[ F(n-2) + F(n-3) + F(n-3) + F(n-4) \]
\[ \ldots \]
Dynamic Programming - Example

• Since *subproblems overlap*, we do *not* use recursion.

• Instead, we construct *optimal subproblems “bottom-up.”*

• Dynamic Programming applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  
  • *Subproblem optimality:* the global optimum value can be defined in terms of optimal subproblems

  • *Subproblem overlap:* the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).
Dynamic Programming - Example

- Computing the \( n^{th} \) Fibonacci number using bottom-up iteration and recording results:
  
  \[
  F(0) = 0 \\
  F(1) = 1 \\
  F(2) = 1 + 0 = 1 \\
  \ldots \\
  F(n-2) = \\
  F(n-1) = \\
  F(n) = F(n-1) + F(n-2)
  \]

  \[
  \begin{array}{cccc}
  0 & 1 & 1 & \ldots \ \\
  F(n-2) & F(n-1) & F(n) \\
  \end{array}
  \]

  \textit{What if we solve it recursively?}

- Efficiency:
  - time = \( O(n) \)
  - space = \( O(n) \)
Exponential and NP Algorithms
Hard Problems

• There are some very practical problems for which *no known efficient solution exists*, and for which there is considerable reason to doubt that an efficient solution is possible.

• **Example: Coloring Graph**
  • Color a map such that two regions with a common border are assigned different colors.

• A map can be represented by a graph:
  • Each region (state) of the map is represented by a vertex;
  • Edges connect two vertices if the regions represented by these vertices have a common border
Graph Coloring

• **Definition:** A graph *has been colored* if a color has been assigned to each vertex in such a way that *adjacent vertices have different colors*.

• **Definition:** The *chromatic number* of a graph is the *smallest* number of colors with which it can be colored.
  - In the example above, the chromatic number is 4.

• Graph coloring problems arise in a number of scheduling and resource allocation situations.
  - Ex.: Scheduling a series of meetings so that people attending the meetings are never scheduled to be in two places at once,
  - Ex.: Assigning classrooms to courses, etc.

• **Definition:** A graph is *planar* if it can be drawn in a plane without edge-crossings.

• **The four color theorem:** For every planar graph, the chromatic number is ≤ 4.
Graph Coloring - Backtracking

• The most obvious solution to this problem is arrived at through a design referred to as backtracking.
  • We will have one variable for each node in the graph. Each variable will take on any of the available colors.

• Recall that the essence of backtracking is:
  1. Number the solution variables \([v_0, v_1, \ldots, v_{n-1}]\).
  2. Number the possible values for each variable \([c_0, c_1, \ldots, c_{k-1}]\).
  3. Start by assigning \(c_0\) to each \(v_i\).
  4. If we have an acceptable solution, stop.
  5. If the current solution is not acceptable, let \(i = n-1\).
  6. If \(i < 0\), stop and signal that no solution is possible.
  7. Let \(j\) be index such that \(v_i = c_j\). If \(j < k-1\), assign \(c_{j+1}\) to \(v_i\) and go back to step 4.
  8. But if \(j \geq k-1\), assign \(c_0\) to \(v_i\), decrement \(i\), and go back to step 6.

• Backtracking over \(n\) nodes, each of which can take on \(k\) possible colors, is \(O(k^n)\).
NP Problems

- We call the set of programs whose worst case is a polynomial order the class $P$.
  - Suppose that we had an infinite number of computers at our disposal,
  - and could spawn off problems to any number of them in constant time.
  - These computers could then run in parallel with each other.

- We call this machine that allows us to spawn off parallel computations at no cost a nondeterministic machine.

- The set of algorithms that can be run in Polynomial time on a Nondeterministic machine is called the $NP$ algorithms
  - The set of problems solvable by an algorithm from that set are NP problems.
NP Problems

• Since that any problem that can be solved in polynomial time on a single, conventional processor, can certainly be solved in polynomial time on an infinite number of processors $\Rightarrow P \subseteq NP$.

• There are some exponential time algorithms that can’t be solved in polynomial time even on a nondeterministic machine.
  
  ▪ $\Rightarrow$ the NP problems clearly occupy an intermediate niche between these really nasty exponential problems and the polynomial time P problems.

• This leads to one of the more famous unresolved speculations in computer science: Is $P = NP$?
NP-Complete Problems

• The speculation about $P = NP$ is particularly interesting in view of the discovery of a special subset of the NP problems.

• Among the NP problems, there are a few key problems called **NP-complete problems** such that
  • for any NP-complete problem $A$, and
  • for any NP problem $B$
  $\rightarrow B$ can **reduced** (converted) to $A$ in **polynomial time**.

• Consequently, if we could **find a polynomial-time solution** to even one NP-complete problem, we would have a **polynomial-time solution to all the NP problems**.

• Graph coloring is NP-complete.
Garbage Collection
Introduction

- **Objects on the heap that can no longer be reached** (in one or more hops) from any pointers in the activation stack (i.e., in local variables of active functions) or from any pointers in the static storage area (variables declared in C++ as "static") are called **garbage**.

- In this example, if we assume that the airline object is actually a local variable in some function, then **Norfolk and Raleigh appear to be garbage**.
  - Unless there’s some other pointer not shown in this picture, **there’s no way to get to either of them**.
Introduction

- Determining when something on the heap has become garbage is sufficiently difficult that many programming languages take over this job for the programmer.

- The runtime support system for these languages provides automatic garbage collection, a service that determines when an object on the heap has become garbage and automatically scavenges (reclaims the storage of) such objects.

- **Example:**
  - In Java there is no "delete" operator. Java programmers never worry about deleting anything. They just trust in the garbage collector to come along eventually and clean up the mess.
  - C++ does not support automatic garbage collection.
Reference Counting

- **Reference counting** is one of the simplest techniques for implementing garbage collection.

- Keep a hidden *counter* in each object on the heap. The counter will indicate *how many pointers* to that object exist.
  - Each time we reassign a pointer that used to point at this object, we decrement the counter.
  - Each time we reassign a pointer so that it now points at this object, we increment the counter.

- If that *counter ever reaches 0*, scavenge the object.
Reference Counting - Example
Reference Counting - Problem

• Assume that the airline object itself is a local variable in a function and that we are about to return from that function.

• Therefore, that object will be destroyed, and its reference counted pointers to the three hubs will disappear.

• All of these objects are garbage, but none of them have zero reference counts → none of them will be scavenged.

• We’re *leaking memory*, big time!
Mark and Sweep

- **Mark and sweep** is one of the earliest and best-known garbage collection algorithms.
  - It works perfectly well with cycles, but
  - Requires some significant support from the compiler and run-time system.

- The core **assumptions** of mark and sweep are:
  - Each object on the heap has a hidden "mark" bit.
  - Find all pointers outside the heap (i.e., activation stack and static area)
  - For each data object on the heap, find all pointers within that object.
  - We can iterate over all objects on the heap

- The **algorithm works in two stages**.
  - **First (Mark)**, start from every pointer outside the heap and recursively mark each object reachable via that pointer. (i.e., depth-first traversal)
  - **Second (Sweep)**, for each object on the heap:
    - if it’s marked → not garbage
    - If not marked → then it’s garbage and we scavenge it.
Mark and Sweep - Example

- The main algorithm begins the marking phase, looping through the pointers in the activation stack.

- Once the **mark phase** of the main algorithm is complete,
  - We have marked the Boston, N.Y., and Wash DC objects.
  - The Norfolk, Raleigh, Adams, Baker, and Davis objects are unmarked.

- In **the sweep phase**, we visit each object on the heap.
  - The three marked hubs will be kept, but their marks will be cleared for running the algorithm again at some time in the future.
  - All of the other objects will be scavenged.
Mark and Sweep - Issues

- The recursive form of mark-and-sweep requires too much stack space.
  - It can frequently result in recursive calls of the mark() function running thousands deep.
  - Since we call this algorithm precisely because we are running out of space, that’s not a good idea.
  - Practical implementations of mark-and-sweep have countered this problem with an iterative version of the mark function that "reverses" the pointers it is exploring so that they leave a trace behind it of where to return to.

- Even with that improvement, systems that use mark and sweep are often criticized as slow.
  - Tracing every object on the heap can be quite time-consuming. On virtual memory systems, it can result in an extraordinary number of page faults.
  - The net effect is that mark-and-sweep systems often appear to freeze up for seconds to minutes at a time when the garbage collector is running.
Conclusion
Conclusion – Topics Covered

- *Lec 01 - Course Introduction & Object Oriented Design:* Abstraction and Abstract Data Types (ADTs), ADTs and Classes, Implementation of ADTs in C++, Composition, and Overloading.
  - Familiarize with *Course Logistics*
  - Familiarize with *Course Overview*
  - Understand *Introduction to Programming and Data Structures*
  - Familiarize with *Introduction to ADT*
  - Familiarize with *ADT Implementation in C++*
  - Familiarize with *C++ Overview*
Conclusion – Topics Covered


  - Understand *simple sorting and searching techniques* that will be used as examples.
  - Understand *running-time complexity/analysis* of an algorithm using *Big-O notation*
  - Apply *Big-O analysis* on simple sorting/searching algorithms
  - Understand *function template* syntax
  - Understand and use of *recursion*.
Conclusion – Topics Covered

- **Lec 03 - STL and Dynamic Memory:** STL C++ Pointers, Dynamic Memory, Assignment and Initialization, Vectors, and Matrices.
  - Understand *Class Templates*
  - Understand and how to use *The Standard Library (STL)*
  - Familiarize with *C++ STL Vectors*
  - Understand Pointers & Dynamic Memory
  - Familiarize with *The miniVector Class*
  - Familiarize with *The matrixr Class*
Conclusion – Topics Covered

  - Understand *List Container*
  - Understand *Container Iterator*
  - Understand *C++ STL list Container*
  - Understand *Single Linked List Implementation*
  - Understand *Variations of Linked List Implementation*
  - Familiarize with *The miniList Class*
Conclusion – Topics Covered

• **Lec 05 - Stacks & Queues**: Stacks, Recursion, Queues, and Priority Queues.
  
  • Understand *Stacks*
  
  • Understand *C++ STL stack Container*
  
  • Familiarize with *Stack Applications*
  
  • Familiarize with *The miniStack Class*
  
  • Understand *Queues*
  
  • Understand *C++ STL queue Container*
  
  • Familiarize with *The miniQueue Class*
Conclusion – Topics Covered

• *Lec 06 - Stacks & Queues (cont'd):* Stacks, Recursion, Queues, Priority Queues, and Deques.

  • Familiarize with *Stack Applications – Function Call*
  • Understand with *Stack Applications – Expression Evaluation*
  • Understand with *postfix Expression and the conversion from infix*
  • Understand *Queues*
  • Understand *Bounded/Circular Queue*
  • Understand *Priority Queue*
Conclusion – Topics Covered

• **Lec 07 - Trees:** General Trees, Tree Traversing, Binary Search Trees, and Heaps
  • Familiarize with *Trees and Related Terminologies*
  • Understand *different Traversing Schemes*
  • Familiarize with *Binary Trees and Related Facts*
  • Familiarize with *Binary Trees Operations*
  • Familiarize with *Binary Search Trees and its ADT*
  • Understand *BST operations find, insert and delete*
  • Understand *How fast is the BST operations*
  • Familiarize with *Applications of BST*
Conclusion – Topics Covered

• **Lec 08 - Associative Containers:** Sets, Maps, Hashing, and Balanced Trees
  • Familiarize with *Associative Containers*
  • Understand *Set, MultiSet, Map, and MultiMap Containers*
  • Familiarize with *miniSet and miniMap Implementation*
  • Understand *Hashing and Hash Tables*
Conclusion – Topics Covered

- **Lec 09 - Associative Containers/Sorting**: Hashing, Balanced Trees, Insertion Sort, and Shell Sort
  - Understand *Declaration of Associative Container*
  - Familiarize with *The Hash Class*
  - Familiarize with *Balanced Search Trees - AVL*
  - Familiarize with *Balanced Search Trees – B-Tree*
  - Understand *2-3-4 Trees*
  - Understand *Red-Black Trees*
  - Understand *Insertion Sort and Its Worst Case Analysis*
  - Understand *Shell Sort and Its Worst Case Analysis*
Conclusion – Topics Covered

• Lec 10 - Sorting (Cont'd): Merge, Quick, Heap Trees, and Heap Sort.
  • Familiarize with Sorting Speed Limit
  • Understand Merge Sort and its Time/Space Analysis
  • Familiarize with Divide and Conquer
  • Understand Quick Sort and its Best/Average Worst Case Analysis
  • Understand Binary Heap Trees
  • Understand Heap Sort and its Time Analysis
Conclusion – Topics Covered

• **Lec 11 - Graphs:** Graphs

  • Familiarize with *Graph terminologies and applications*
  • Understand *Graph Implementations*
  • Familiarize with *Graph ADT*
  • Understand *Graph Traverse Methods (DFS & BFS)*
  • Understand *Partial Orders and Topological Sort*
  • Understand *Shortest Path and Shortest Weighted Path*
Conclusion – Topics Covered

• **Lec 12 - Recursive Algorithmic Styles:** Recursion vs. Iteration, and Recursive Algorithmic Styles, Garbage Collection, and NP.
  
  • Understand *Shortest Path and Shortest Weighted Path*
  • Understand *Minimum Spanning Trees*
  • Familiarize with *Backtracking Algorithmic Style*
  • Familiarize with *Dynamic Programming Algorithmic Style*
  • Familiarize with *Exponential and NP Algorithms*
  • Familiarize with *Garbage Collection*
Final Exam Topics

• Lec 01 - Course Introduction & Object Oriented Design
• Lec 02 - Algorithms & Complexity
• Lec 03 - STL and Dynamic Memory
• Lec 04 - List & Iterators
• Lec 05 - Stacks & Queues
• Lec 06 - Stacks & Queues (cont'd)
• Lec 07 - Trees
• Lec 08 - Associative Containers
• Lec 09 - Associative Containers/Sorting
• Lec 10 - Sorting (Cont'd)
• Lec 11 - Graphs
• Lec 12 - Recursive Algorithmic Styles
Student Opinion Survey

• Students are notified via email that they can provide feedback on the course.
  • The email to students contains a web link to the survey

• Can also access the Student Opinion Survey from the University's Current Students page:
  • Go to http://www.odu.edu
  • Click Current Students
  • Click Student Opinion Survey (under Academics)

• Feedback is anonymous

• Lowest assignment grade will be dropped for survey participant students
Questions?