Independent Study

Advanced Algorithms

Final Report

Kushal Gevaria
kgg5247@rit.edu

Advisor
Prof. Ivona Bezakova

Table of Contents Page Number

Advanced Algorithms
Articulation Points, Bridges

Problem Statement

Given a graph with n vertices and edges connecting these vertices, get a vertex v such that, removing that particular vertex will disconnect the graph. Such a vertex is called as articulation point in the graph. Since, it disconnects the graph, it also increases the number of connected components in a given graph [1]. An edge in the given graph is said to be a bridge if there is no other path through which two nodes connected by that edge can be found. So, removing such edge disconnects the graph since there is no possible way to reach one node to another, thereby disconnecting them. [1].

Overview

To find the connection between any two given nodes in a graph, we need to perform path traversal which can be done using breadth first search traversal or depth first search traversal and while doing such traversal graph is considered in terms of tree where edges are represented in the given below format [1] –

1. Edge from parent node to child node in the graph is considered as tree edge.
2. Same tree edge is also considered as forward edge in the graph.
3. Back edge is the opposite of forward edge which is from child node to parent node.
4. Cross edge is an edge between two nodes which are on same level in the given tree representation of the graph.

This report focuses on obtaining such articulation points and bridges that disconnects the graph. First the brute force algorithms are provided which can be improved by providing more efficient algorithm called Tarjan's algorithm.

Now, let's first have a look at the brute force algorithm to obtain articulation points in a given graph which is mentioned below.

Approach

Brute Force Algorithm for Articulation points

For each vertex in the given graph:

- Remove the vertex v
- Check if the graph is connected using BFS or DFS traversal by finding number of strongly connected components in the graph. If number of strongly connected components increases, then v is articulation point
- Add the vertex v back to the graph

Since, we are picking each vertex v in the graph and perform BFS/DFS traversal, the complexity of this algorithm is O(V(V+E)) [1].

Brute Force Algorithm for Bridges

For each edge E in the given graph:

- Remove the edge E
- Check if the graph is connected using BFS or DFS traversal by finding number of strongly connected components in the graph. If number of strongly connected components increases, then edge is bridge (It disconnects connected two vertices as there is no path between them via others vertices as well)
- Add the edge E back to the graph

Since, we are picking each edge E in the graph and perform BFS/DFS traversal, the complexity of this algorithm is O(E(V+E)) [1].

Both brute force approach to find articulation points and bridges run in polynomial time. To improve the run time for this problem, there is an algorithm called Tarjan's algorithm that provides linear run time complexity. Before going through this algorithm, there are some data structures required to store information required for this algorithm which are as follows [1] –

1. A visited array of Boolean elements of size n, i.e. number of vertices in the graph that keeps the track of vertices which are visited with true value while unvisited as false.
2. An articulation point array of Boolean elements of size n, i.e. number of vertices in the graph that keeps the track of vertices which represents articulation points with true value while others as false.
3. An integer array of discovery time of size n, i.e. number of vertices in the graph that keeps the track of vertices time of discovery. It is initialized with 0 which means none of them are discovered yet.
4. An integer array of parents of size n, i.e. number of vertices in the graph that keeps the track of parent of the current vertex. It is initialized to Null which mean currently no one has a parent.
5. An integer array of low time of size n, i.e. number of vertices in the graph. It keeps the track of lowest discovery time after starting from a selected root vertex.

Given below is a recursive algorithm to compute low time for each vertex given discovery times and starting vertex v in the given graph –

void getLow(v) {

- Mark v vertex as visited, and update its low time and discovery time to 1.
- For each child of given vertex v:

  If the child is not visited:

  i. Then update its parent to vertex v
  ii. getLow(child)
  iii. low[v] = min(low[v], low[child])

  Else:

  If the child is a parent of the vertex v, then edge between v and child node is a backward edge
  So, low(v) = min(low[v], discovery[child])

}

Tarjan's Algorithm to find Articulation points

Given below is the Tarjan's algorithm to find articulation points in a given graph –

For each vertex in the given graph:

- If it has no parent, and have more than 1 child, then it is an articulation point (Root node)
- Else for each child, if there is a back edge from child node to ancestor of root node, then child node is not an articulation point. This can be found out if low (child node) < discovery (parent node).
- Else, child node is an articulation point. This can be found out if low (child node) >= discovery (parent node).

Complexity of this algorithm is O(V+E), since we are traversing each vertex just once and marking it visited so that it is not picked up again [1]. Also, after picking each vertex we, just check of edges connecting it child nodes to find if there are the possible articulation points.

Tarjan's Algorithm to find Bridges

This is similar to finding articular points, where edge between a vertex v and its child node is a bridge if, low time of child node is greater than discovery time of u. Thus, actually, the child node is not reachable via any nodes (ancestors of parent node) from a given parent node after removing the edge between this child node and parent node [1].

Complexity of this algorithm is O(V+E), since we are traversing each edge just once and checking the low time and discovery time properties of the nodes connected by the selected edge [1]. Thus, Tarjan's algorithm is much better than the normal brute for approach for both finding articulation points as well as bridges in the given graph with the help of some additional storage of information using appropriate data structures.

Conclusion

In this report, we learned Tarjan's algorithm that provides total number of articulation points present and number of bridges present in a given graph in linear run time complexity O(V+E) which is much better than the brute force approach to solve both the problems, i.e. polynomial run time [1]. There are lots of real-world applications finding this articulation points and bridges like in war, where troops would like to know minimum number of links in the network that can break the connection between the enemies operating over that network. Also, it can be used to identify area of network that can cause major issue failure occurs in just that particular area. Also, it can be used to identify area of network that if caused some problem, may result in failure of whole network to work. Another application is clustering problems and matching problems. Similar application is extended to Karger's min cut problem, where we find number of bridges that disconnects the graph into two or more connected components [4].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Monika Alluri" in this advanced algorithm course, a "geeksforgeeks" link with implementation of the Tarjan's Algorithm to find articulation points in a given graph [3], a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation [2]. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Articulation_points.pdf

[2] https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

[3] http://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/

[4] https://www.cs.rit.edu/~algorg/Materials/Randomized%20Algorithm%20-%20Karger's%20Algo.pdf

Advanced Algorithms
Kandane's Algorithm

Problem Statement

Given an array of length n, we need to find a contiguous subarray with the maximum sum of elements in that subarray. The array can contain both negative as well as positive numbers. Also, it should be contiguous in nature which means, consider an array [10, -2, 3, 4, 5]. Then, [10, 4, 5] cannot be considered as a contiguous subarray.

Overview

Let's first provide a definition for a subarray. Consider the array mentioned in problem statement, A = [10, -2, 3, 4, 5].

Now for the above given array there can be different possibilities of subarrays like [10], [-2, 3], [3, 4, 5] etc. But [10, 4, 5] cannot be a subarray in our case, since it is not contiguous in nature. Also, there are some properties of an array which should be mentioned beforehand which are given below –

1. If the given array contains all positive integers, then we can simply add up all the elements in the array to provide maximum sum as the final result. So, basically maximum sum is the sum of n elements in the array of length n.
2. If the given array contains all negative integers, then we simply need to search a number which is smallest, i.e., contains lowest magnitude with negative sign to provide maximum sum as the final result.

For this particular report, we are considering an array that contains both positive as well as negative integers.

Approach

Brute Force

First, let's consider the brute force approach. Brute force is an exhaustive search algorithm which will try all possible combinations and return the best on from all those combinations. In this problem, we need to find the subarray with maximum sum of all the elements in the subarray [2]. So, the brute force approach will find all possible contiguous subarray and compare them with another. The best subarray with the maximum sum will be selected as an outcome to this problem.

Given below is the pseudocode for brute force algorithm to find subarray with the maximum sum in a given array –

```
int maxSoFar = 0;
int startingIndex = 0;
int endingIndex = 0;
for(i from 0 to n-1){
        int sum = 0;
        for(j from i to n-1){
                sum += a[j];
                if(sum > maxSoFar){
                        maxSoFar = sum;
                        startingIndex = i;
                        endingIndex = j;
                }
        }

}
```

Brute Force Algorithm [1][2]

Since, in the brute for algorithm we are clearly computing outcome for each possible subarray in the given array, clearly the running time complexity is $O(n^2)$ where n is the number of elements in the array or the size of a given array and the space complexity is O(1) since we are consuming any extra space to store the subarray combinations. We are just getting the starting index and the ending index of the optimal output which will later help to find the elements of that subarray which yields maximum sum. We can find an efficient way to find the optimal solution by just traversing the array once which is shown in the next algorithm below.

Kandane's Algorithm with space complexity of O(n)

This algorithm uses a dynamic programming technique where we build up the optimal solution using smaller sets of previously obtained optimal solutions. For the given problem, we will require a one-dimensional array to store the results of the maximum sum for the elements in a given array. Given below is the pseudocode for Kandane's algorithm which is explained later on in this report itself –

```
maxSum[0] = A[0]
for(i from 1 to n-1){
        maxSum[i] = max(maxSum[i-1] + A[i], A[i]);
}
```

Kandane with Space complexity – O(n) [1][2]

In the above-mentioned algorithm, we build up the solution for array of length n by looking at what it looked like when array was of length n-1. Thus, base case for this algorithm would be array of just length 1. In that case, maximum sum of number of elements in the array of length 1 would just be that single element. Now, if we have an array of length 2, then the maximum sum will be considered either by rejecting the second new element that was introduced in this array of length 2 and only considering the first element, or by including that second element if after including that it yields a bigger number than the previous result. So, at index n-1, in the given array of maximum sum we will get the optimal solution of maximum sum of subarray elements for the given array of length n. Here, the time complexity is just O(n) since we are just traversing each element in the actual array once. Also, we need a maximum sum array to store the results obtained for each sub-optimal solution. Thus, the space complexity here is also O(n). Now, this can further be improved in the region of space complexity which is shown below in the modified version of Kandane's algorithm.

Kandane's Algorithm with space complexity of O(1)

Here, instead of storing the sub-optimal solutions in a 1-dimensional array, we just keep a track of two variable that helps to move forward and reach the optimal solution of an array of length n. Given below are the newly introduced variables –

1. "maxSoFar" – This is an integer variable which stores the maximum sum found so far for a particular subarray in the given actual array
2. "maxEndingHere" – This is an integer variable which stores the solution of the maximum sum found for the current processing subarray with end index equal to the end index of the current subarray.

Given below is the pseudocode for modified Kandane's algorithm –

```
int maxSoFar = A[0];
int maxEndingHere = A[0];
for(i from 0 to n-1){
        maxEndingHere = max(maxEndingHere + A[i], A[i]);
        maxSoFar = max(maxSoFar, maxEndingHere);
}
```

Kandane with Space complexity – O(1) [1][2]

Clearly, "maxEndingHere" finds sum value for all possible subarrays and "maxSoFar" keeps track of the best optimal solution from those subarrays. So, we do not need to store one-dimensional array for maximum sum values of all the subarray possible. The only problem with the Kandane's algorithm is that we will not be able to track which elements provided the best maximum sum in the given subarray since we do not know about the starting index and the ending index of the optimal subarray that we found using this algorithm.

Proof of correctness for Kandane's algorithm

- Let's say there is an element K at nth position in the given array A of length n. Then, using the given algorithm we know that $(n-1)^{th}$ position in the maximum sum array provides an optimal solution say H (maxSum[n-1]) for n-1 elements of the actual array. And after introducing nth element, optimal solution would be given as follows – maxSum[n] = max(H + A[n], A[n])
- By contradiction, let's say there exist a suboptimal solution T better than H such that $T + A[n] > H + A[n]$ – (a)
- But, we know that T <= H, since T was not considered as a suboptimal solution for $(n-1)^{th}$ position following the rules of Kandane's algorithm.
- Clearly, if T <= H, then $T + A[n] <= H + A[n]$ – (b)
- Equation (a) and (b) creates a contradiction. Thus, proof by contradiction, this algorithm will never fail and will always result in optimal solution in linear time complexity which is O(n), where n is the number of elements in the array.

Conclusion

In this report, we learned about finding a subarray with maximum sum of its elements in a given array in efficient way using Kandane's algorithm which is much better than the traditional brute force approach where it computes solution for each possible subarray and picks the one with the optimal solution. We also learned about improving the space complexity as well by introducing some important variables which keep track of best solution so far rather than storing all the suboptimal solution using the 1-dimensional array of length equal to the length of actual array elements. There are some applications of this Kandane's algorithm which are provided in the presentation slides given by one of the students like Station Travel in Order where you need to find the cost to reach all the station in the given order in just one direction.

Credits

The above report and the terms as well as algorithms used in it are based on the presentation given by one of the student in this advanced algorithm course and a Wikipedia link that explains the Kandane's algorithm in detail. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/KadanesAlgorithm.pdf

[2] https://en.wikipedia.org/wiki/Maximum_subarray_problem

Advanced Algorithms
Aho-Corasick Algorithm

Problem Statement

Given an array of pattern strings and a huge text, we need to find occurrence of those patterns in the given text as efficiently as possible [1]. Pattern can be any substring of the original text starting at any location. For example, given a text = “hersheet”, and an array of patterns [“hers”, “he”, “sheet”, “shop”, “her”, “she”], we know the first three patterns can be found in the original text while there is no “his” pattern in the same text.

Overview

This problem can be solved using simple brute force algorithm with polynomial run time complexity which is explained later in this report. Also, this report covers an algorithm called Aho-Corasick algorithm which is a linear run time algorithm to find all patterns in the given text. Let’s first discuss about the brute force approach mentioned below.

Approach

Brute Force

For each pattern in the array of patterns, check if the pattern exists in the given text. This checking of pattern in the given text can be performed using Knuth-Morris-Pratt pattern matching algorithm which runs in linear time [4]. Thus, run time complexity of Knuth-Morris-Pratt is O(n) where n is the length of the longest pattern in the array of patterns. And this algorithm is performed k times where k is the number of patterns in that given array of patterns. Thus, overall complexity of this algorithm will be O(n*k) which is clearly a polynomial run time complexity. There is a much efficient way to do this pattern matching for array of patterns using an algorithm called Aho-Corasick algorithm which provided a linear run time solution. This algorithm is mentioned below.

Aho-Corasick Algorithm

This algorithm uses a special data structure called Trie that helps to perform pattern matching in the efficient way. Trie is a special data structure which is similar to normal tree, where edges represent a path to particular pattern. This tree is build using those possible patterns asked in the problem. Given below is an example of Trie structure for the pattern array [“hers”, “he”, “sheet”, “shop”, “her”, “she”] –

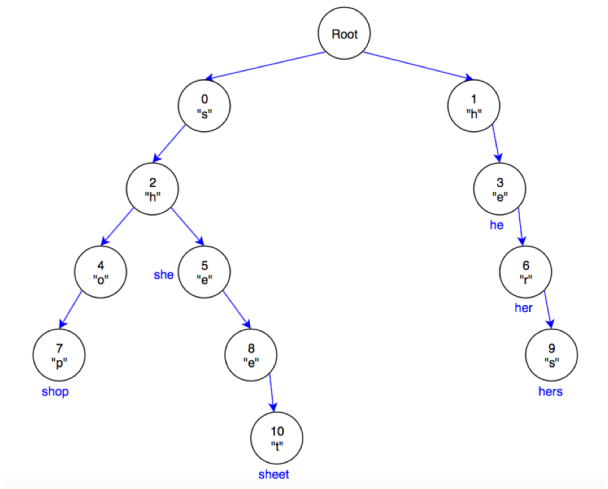

Figure 1. Trie Data Structure

As you can see the figure 1 above, pattern “he” is created by starting from root node and traversing node 1 and node 3. Similarly, “he” pattern can be extended to “her” or “hers” by adding nodes 6 and 9 respectively to the Trie data structure. Thus, this tree is created dynamically by picking each pattern and extending the tree or by creating a new path from the root node if pattern starts with completely unique character with respect to already present child character nodes on the root node. Building this Trie data structure takes O(m) time where m is the total length of all the characters present in all the pattern strings [1].

After creating a Trie data structure, next step is to add fail edges to each and every node in the tree. Let’s say if the given text to search pattern in is “hersheet”, then after node 9 which gives character “s”, there is no character “e”, since we will be traversing right part of the tree to find pattern “hers”. After character “s” it need to switch to the character “s” of node 0 to obtain pattern “sheet”. Thus, we need those fail edges that points directly to character in the child nodes of the closest ancestors that matches to the current processing character to start finding new upcoming patterns in the given text [1]. So, we create those fail edges for each node and they follow given below property [1] –

1. Child of root nodes will have fail edges pointing to the root node itself.
2. Root node will fail to itself.
3. Other nodes will have fail edges that are child nodes of the closest ancestors with character matching to the current processing character.

Given below is an example of Trie data structure after adding the fail edges for each node in the structure –

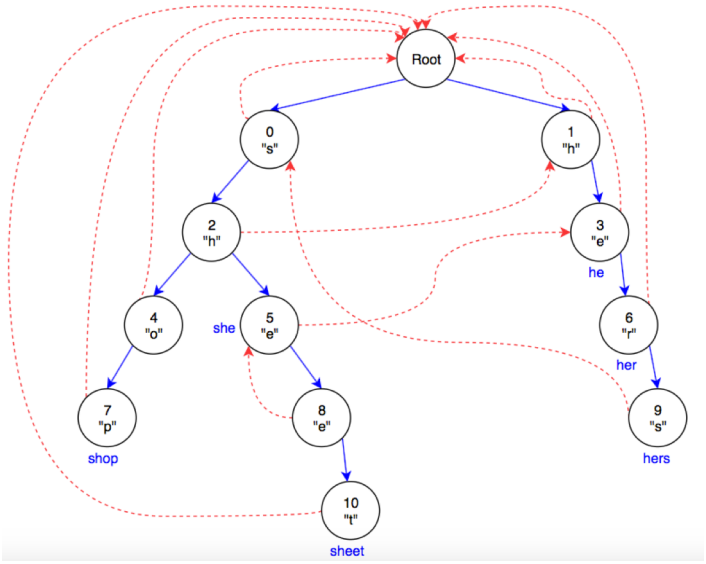

Figure 2. Trie Data Structure with fail edges (Red)

As shown in the figure 2, node 0 and node 1 have fail edges to root node according to the 1st property. Root node will fail to itself, so no need to have a fail edge over there according to 2nd property. For other nodes, a node 9 with character “s” will have a fail edge to node 0 with same character “s” and similarly for the remaining ones. If there is no matching character in the child node of the closest ancestor, then that node will fail to the root node. Adding these fail edges is O(m) where m is the total length of all the characters present in all the pattern strings [1].

Now, while performing pattern matching with given array of patterns [“hers”, “he”, “sheet”, “shop”, “her”, “she”] in the text “hersheet”, we first start with root node and look for first character in the given text which is “h”. So, next node visited is node 1 and next character we get is “e” which is node 3 in the given Trie structure. As soon as we reach node 3, there is a pattern stored at node 3 which is “he”. This pattern is one of the outcome of the array of patterns we need to search. Similarly, we find patterns “her” and “hers” while going through next characters in the text. As soon as we reach

character “s” there is no child node with character “e”, so it fails to child node with character “s” of the closest ancestor which is node 0. From node 0, following the similar process, we can find patterns “she” and “sheet” traversing path from node 0 to node 10. Thus, we found five patterns “hers”, “her”, “he”, “sheet” and “she” asked for in the array of patterns and no pattern of “shop” in the given text “hersheet”. Thus, overall complexity of this algorithm will be O(n) where n is the path that we traverse throughout the Trie data structure to obtain each possible pattern in the given array of patterns [1].

Conclusion

In this report, we learned about performing pattern matching where given a text and an array of strings, we need to find those strings as a pattern in the given text. The brute force approach took O(n*k) where n is the length of the given text and k is the length of string array which uses Knuth-Morris-Pratt algorithm to find the string in given text in linear time [4]. The algorithm focused on in this report is Aho-Corasick pattern matching algorithm that solve the problem in linear time of finding all possible array of patterns in a given text with run time complexity of O(n+m) where m is the length of all the characters of all the patterns combined [2]. This algorithm is much better and efficient than the brute force approach. There is also another algorithm called Rabin-Karp algorithm which is similar to the Aho-Corasick pattern matching algorithm [5]. There are lot of application of this string matching in the area of detecting plagiarism, performing semantic analysis or in bio informatics [1].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by students “Rynah Rodrigues” and “Rinkesh Shah” in this advanced algorithm course, a “geeksforgeeks” link with implementation of the Aho-Corasick Algorithm to perform pattern matching [3], a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation [2]. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Pattern%20Matching.pdf

[2] https://en.wikipedia.org/wiki/Aho–Corasick_algorithm

[3] http://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/

[4] https://www.cs.rit.edu/~algorg/Materials/KMP%20Algorithm.pdf

[5] https://en.wikipedia.org/wiki/Rabin–Karp_algorithm

Advanced Algorithms
Knuth Morris Pratt Algorithm

Problem Statement

Given a string of length n and a pattern of length m which is less than n, we need to find the similar pattern in the given string if it exists. Thus, if a given string is "Hello World" and we want to find a pattern say "lo" then we know there exists such pattern in the given string which starts at index 3(if indexing starts from 0) and pattern ends at index 4. Thus, we will go through one algorithm named Knuth Morris Pratt Algorithm which helps us achieve this goal of finding a pattern in the given string in an efficient way.

Overview

Most simple way to do pattern matching is to check letter by letter in the given string and move forward if we find a match. At the end, if all the letters in the pattern are matched consecutively with the letters in the given string then we have found that pattern. This is not the effective way to perform pattern matching where we need to deal with huge strings and respective huge pattern to find. Let's first look at the brute force approach to perform pattern matching and then, later on, we will have a look at the efficient and much better pattern matching algorithm which is the main focus of this report called Knuth Morris Pratt algorithm [2].

Approach

Brute Force

First, let's consider the brute force approach. Brute force is an exhaustive search algorithm where it searches the pattern letter by letter in the original string and if return true if we found the pattern else it repeats the whole procedure from new index position of the original string which is incremented by one. This is done until we exhaust the original string while searching or we actually end up finding the pattern in the same string.

Given below is the pseudocode for brute force algorithm to find the pattern say P in a given original string say S–

```
int pLength = length of pattern;
int sLength = length of original string;
for(i from 0 to sLength - pLength){
        for(j from 0 to sLength){
                if(P[j] == S[i+j]){
                        j++;
                }
        }
        if(j == pLength){
                // pattern found at index i in the original string
                return true;
        }
}
return false;
```

Brute Force Algorithm [4]

Since, in the brute for algorithm we are clearly performing pattern matching in the original string starting from each index in the original string, we are doing extra computations and checks each time which can be avoided. Thus, here the time complexity to perform pattern matching is O(mn) where let say m is the length of pattern we are searching and n is the length of the original string. There is an efficient way which is provided by an algorithm called Knuth Morris Pratt

algorithm. This algorithm was invented by three people namely Donald Knuth, James H. Morris and Vaughan Pratt and the process of the same algorithm is provided in the next section below.

Knuth Morris Pratt

This algorithm is divided into two stages of computation which are as follows –

1. Prefix Function
2. String Matching Function

Prefix Function –

This function is used to find the matches of the prefix or the beginning of the pattern with any letter that comes up later in the same pattern that is the suffix. Thus, by finding a match of suffix with a prefix will later help us to skip extra checks and avoid multiple shifts of letters to perform pattern matching. This helps to avoid performing matching that was already happened previously and only searches that part of the pattern which is still new with respect to the current position in the original string.

Given below is the pseudocode for the prefix function computation–

```
pLength = length of pattern;
prefix[0] = 0;
int i = 0;
for(j from 1 to pLength){
        while(i > 0 and prefix[i+1] != prefix[j]){
                i = prefix[i]
        }
        if(prefix[i+1] == prefix[j]){
                i += 1
        }
        prefix[j] = i
}
```

Prefix Function [3]

Consider the given below example which shows the prefix array created for a given pattern string –

Index	0	1	2	3	4	5	6
P	A	B	C	A	B	A	C
prefix	0	0	0	1	2	1	0

As you can see, the character A is repeated twice at index 3 and 5 (suffix in this case) for a matched prefix at index 0. Also, we find a pattern “AB” repeated at index 3 and 4 for prefix index 0 and 1.

This prefix computation will help to perform pattern matching much faster by skipping unnecessary checks. Now let us have look at the second stage which is actual pattern matching function. The complexity of this prefix function is O(m) where m is the length of the pattern string.

Pattern Matching Function –

After creating the prefix array for a given pattern of the same length, pattern matching is performed based on this prefix array which is pretty fast as compared to the brute force as it skips some part of the search which was already matched, thereby reducing the time complexity as well.

Given below is the pseudocode for the pattern matching function of Knuth Morris Pratt algorithm –

```
int i = 0; // counter for original string
int j = 0; // counter for pattern string
int k = 0; // index where we get the pattern matched
int n = length of original string;
int m = length of pattern;
String[] S; // original string
String[] P; // pattern
while(n-k >= m){
        // check the entire original string
        while(j <= m and S[i] == P[j]){
                // move forward if characters match
                i += 1
                j += 1
        }
        if(j > m){
                // found the match at index k in the original string
                print k;
        }
        if(j > 0 and prefix[j-1] > 0){
                k = i - prefix[j-1];
        }else{
                if(i == k){
                        i += 1
                }
                k = i; // this skips the string which is already matched
        }
        if(j > 0){
                j = prefix[j-1] + 1
        }
}
```

Main Function [3]

Consider the given below example which shows the string matching for a given pattern string with starting k index of the match –

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	A	A	B	C	A	B	A	C	A	B	C	A	B	A	C	B
k		k = 1	This part is Skipped as already matched in first iteration						k = 8							

So, two matches are found in the string “AABCABACABCABACB” for the given string pattern “ABCABAC” at position 1 and 8 in the original string. This shows that we found a match at index 1 in the string and we won’t find a pattern in the next m characters at least, where m is the length of the pattern string. So, we can skip directly to start searching from $(k + m)^{th}$ character onwards. This can be achieved with the help of prefix array that was created for the pattern string. The complexity of this pattern matching function is O(n) where n is the length of the original string. Thus, the overall complexity of this Knuth Morris Pratt algorithm is O(n + m) which is linear time as compared to polynomial time brute force algorithm.

Conclusion

In this report, we learned about performing pattern matching in the given string where we studied first, the brute force approach which has O(nm) time complexity. This will take a lot amount of time for real life huge string matches and so we studied about a linear time running algorithm called Knuth Morris Pratt algorithm which has complexity O(n + m)

which is subdivided into two parts namely prefix function and actual pattern matching function. This prefix function helps to speed up the pattern matching was previously time-consuming in the brute force approach. There are lots of real-world applications of string matching like in bioinformatics which has sequential data like DNA to perform matching for a particular part of it. String matching plays important role in information retrieval and digital forensics as well [1][2].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Jidnyasa Patil" in this advanced algorithm course, a Wikipedia link that explains the Knuth Morris Pratt algorithm in detail, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/KMP Algorithm.pdf

[2] https://en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm

[3] https://www.youtube.com/watch?v=2ogqPWJSftE

[4] http://www.stoimen.com/blog/2012/03/27/computer-algorithms-brute-force-string-matching/

Advanced Algorithms
Fast Fourier Transform Algorithm

Problem Statement

Given some equations which are polynomial in nature, we need to find an efficient way to perform evaluation or multiplication on these equations to obtain the desired output. To handle this problem especially in the mathematical regions, there is an algorithm called Fast Fourier Transform algorithm that takes these equations as input and gives the desired output in polynomial time much better than traditional brute force approach.

Overview

Given any two polynomial equations to perform evaluation and multiplication on, there are two way to represent these equations in different format which are mentioned below –

Co-efficient way of representation –

If there are two polynomial equations say L and M which are given as follows [1] –

$L(x) = l_0 + l_1x + l_2x^2$

$M(x) = m_0 + m_1x + m_2x^2$

Then, the co-efficient way representation of the above two equations are performing addition would something like this –

$L(x) + M(x) = (l_0 + m_0) + (l_1 + m_1)x + (l_2 + m_2)x^2$

Here, the complexity of performing addition operation and doing evaluation is O(n), while performing multiplication operation is $O(n^2)$ using this brute force approach with co-efficient way of representation [1].

Point-Value way of representation –

If there are two polynomial equations say L and M which are given in the point-value representation as follows [1] –

$L(x) = (x_0,y_0), (x_1,y_1), (x_2,y_2)$

$M(x) = (x_0,z_0), (x_1,z_1), (x_2,z_2)$

Then, the point-value way representation of the above two equations are performing multiplication would something like this –

$L(x) * M(x) = (x_0, y_0* z_0), (x_1, y_1* z_1), (x_2, y_2* z_2)$

Here, the complexity of performing addition operation and doing multiplication is O(n), while performing evaluation is $O(n^2)$ using this brute force approach with point-value way of representation [1].

As we can see above, both the way of representation has certain tradeoffs thus we need an algorithm that can do all the three operations in an efficient way without any tradeoffs. To get rid of these tradeoffs we can convert from one way of representation to another but doing so is still computationally inefficient.

Here, Fast Fourier Transform algorithm comes into picture which provide efficient way to do all the above three operations.

Fast Fourier Transform Algorithm

This algorithm is a type of divide and conquer technique which has complexity of O(nlogn) and thus is better than traditional brute force approach which is $O(n^2)$ [1][2]. In this divide and conquer technique the given polynomial equation is divided into two sub parts called as lower equation and higher equation. Lower equation contains all the terms with lower half of all the powers in the actual equation, while higher equation contains all the upper half of all the powers in the actual equation. For example, consider the given below equation –

$L(x) = l_0 + l_1x + l_2x^2 + l_3x^3 + l_4x^4 + l_5x^5$

Therefore, $L_{low}(x) = l_0 + l_1x + l_2x^2$

And, $L_{high}(x) = l_4 + l_5x + l_6x^2$

So, after combining low and high equations to get the original equation it looks like this –

$L(x) = l_0 + l_1x + l_2x^2 + x^3(l_4 + l_5x + l_6x^2)$

Complexity of converting the polynomial equation into subparts is O(nlogn). Instead of using the brute force approach, we can use this algorithm to perform addition, multiplication and evaluation in a little more efficient way [1].

Conclusion

In this report, we learned about using the given number of equation to evaluate them or multiply the equations to obtain the desired result which is asked for using efficient algorithms. Fast Fourier Transform algorithm is used to perform this task of taking these equations and doing evaluation or multiplication based on the desired output in the efficient way. There are some real-world applications of this problem mostly in the mathematical regions, as well as in other places like n the field of image processing, medical diagnosis like MRI etc [1][2].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student “Geeta Madhav Gali” in this advanced algorithm course, a Wikipedia link that explains the Fast Fourier Transformation algorithm in detail. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/FastFourierTrasform.pdf

[2] https://en.wikipedia.org/wiki/Fast_Fourier_transform

Advanced Algorithms
Top Trading Cycle Algorithm

Problem Statement

Given n students living in the student dorms, we need to find best preference assignment of the dorms with respect to each student. Now if the students are already assigned to the dorms which might not be their first preferred assignment, then that type of matching is not considered as stable matching. So, to solve this problem, we need to find a way of mutual exchange which can happen between two students such that both of them, after performing the dorm exchange, are happy with their preference value getting upgraded. This way both students will get what they need first and this new matching taking place will be a stable matching. This type of exchange is called as mutual-beneficial exchange in the presentation proposed for this report [1].

Overview

There are some properties which provides matching of student with respect to the dorms and are as follows [1] –

1. Stable Matching
   This kind of matching takes place just on the basis of one sided preference. So, in case of marriage, matching can take place based on men's preference list where a woman might not get their first preferred man or the other way around, i.e., matching taking place based on women's preference list where a man might not get their first preferred woman.

2. Core
   It states that dorms assignment to each respective student is considered to be a core assignment if there does not exist a solution of new assignments such that this new assignment is advantageous to each student as compared to the previous assignment. Which means each student get a better preferred dorm as compared to their previous assignment.

3. Strong Core
   It states that dorms assignment to each respective student is considered to be a strong core assignment if there does not exist a solution of new assignments such that this new assignment is advantageous to at least one student in the list as compared to the previous assignment for that particular student. Which means either one or more than one students get a better preferred dorm as compared to their previous assignment.

4. Individually Rational
   This property states that each and every student will get the dorm which is as close as possible to their first preference, but it cannot guarantee that they will get the best dorm they are asking for.

5. Pareto Efficiency
   It states that an outcome of assignment is considered as Pareto efficiency outcome if there are no other assignments which benefits any students with their assignment. Thus, if get an outcome which helps one student with better assignment than the one provided in the Pareto Efficiency outcome, then that would be at the cost of hurting another student. So, we cannot achieve the assignment that might benefit all the students.

6. Strategy-Proof
   It states that we cannot manipulate the outcome by changing the preference list which is not the case of obtaining the optimal assignment. Preference list should not be manipulated and so should the outcome obtained. Such an outcome will be strategic proof.

7. Nash Equilibrium
   This is used in game theory where if a player has chosen a particular strategy based on the choice provided by the

opponent, then it does not get a chance to choose any other strategy and just stick to the chosen one. So, this problem will provide an optimal outcome where the player won't receive any benefits if it changes the strategy based on the choice provided by the opponent which is assumed to be constant in this case.

Approach

Gale – Shapley Algorithm

This algorithm is a simple stable matching algorithm which will work only for one sided preferences as mentioned above in the first property. This algorithm will always provide a stable matching but there are some cons in using this algorithm which are as follows –

a. It always prefers matching based on the proposing side.
b. From given above properties, it does not satisfy the Pareto efficient property.
c. Also, the strategy proof property is satisfied for the proposing side.

The algorithm discussed in detail in the given presentation is Top Trading Cycle Algorithm which is mentioned below.

Top Trading Cycle – Algorithm

This is another algorithm which provide a better matching by using the cycle created in the graph such that the cycle provides better exchange or trade-offs between two students so that the newly formed pair after the exchange is better for both the students as compared to their previous matching.

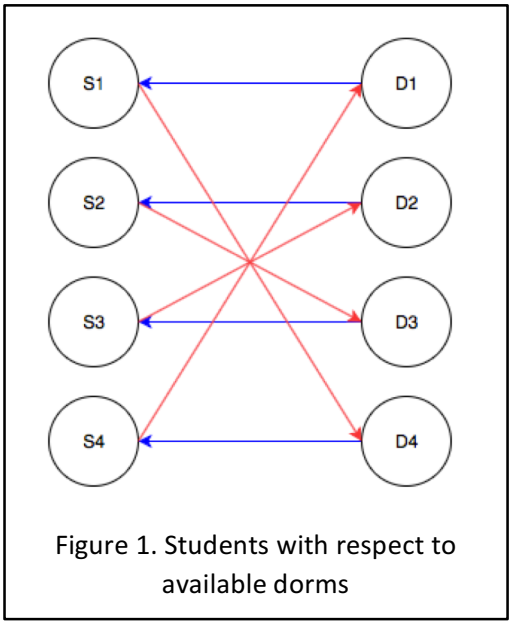

Figure 1. Students with respect to available dorms

Let us consider the given graph shown in figure 1 [1]. The preference list for each of those student in the decreasing order of the preference is as follows –

S1 – D4, D3, D2, D1

S2 – D3, D2, D1, D4

S3 – D2, D1, D3, D4

S4 – D1, D3, D4, D2

The blue edges show the first preference of dorms for each student in the given graph. The red edges show the currently assigned dorms to respective students. So, as per the algorithm let's find first cycle starting from one student and ending at the same student traversing through those alternative blue and red edges. Picking student 1, the cycle that we achieve is as follows –

S1 – D4 – S4 – D1 – S1

Using this cycle, we replace the original matching for S1 and S4 with the ones we found in this cycle, i.e. the red edges. Here, the red edges are S1 – D4 and S4 – D1. So, the new pairing that we found here is (S1, D4) and (S4, D1) which is much better than the previous matching looking at the preference list of those students. After the first iteration, we remove those students and dorms which are already matched with their best preference, since they are not available in the second iteration. So, the graph that we are left with after removing some edges is given in figure 2 [1].

In the second iteration, picking student 2, the cycle that we achieve is as follows –

S2 – D3 – S3 – D2 – S2

Using this cycle, we replace the original matching for S2 and S3 with the ones we found in this cycle, i.e. the red edges. Here, the red edges are S2 – D3 and S3 – D2. So, the new pairing that we found here is (S2, D3) and (S3, D2) which is much better than the previous matching looking at the preference list of those students. After the second iteration, we remove those students and dorms which are already matched with their best preference, since they are not available in the third iteration.

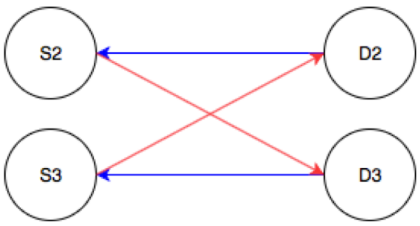

Figure 2. Remaining students with respect to available dorms

Now, we stop since we found all the matching possible for given number of students which are mentioned as follows –

(S1, D4), (S4, D1), (S1, D4), (S4, D1).

Here, we get the best possible matching (optimal solution) for each student. But this might not be always the case depending on the preference list that each student gives.

So, looking at the properties mentioned above, this algorithm provides us a core assignment of matchings and there cannot be any better solution after further exchange of dorms between students. Top – trading cycle algorithm also give us a strategy proof assignment where changing or lying about the preference list would help in any further improvement of the matchings. This algorithm also provides a Pareto efficient assignment which means that there would not be a case such that student benefits from it with better dorm without hurting another student, since there is always a tradeoff which should always benefit both the parties.

Conclusion

In this report, we learned about using Top – Trading Cycle algorithm to perform stable matching between students and the dorms they are assigned to. This algorithm satisfies all the properties required to achieve stable and optimal matching and also covers the cons which were faced during Gale – Shapley stable matching algorithm and is done based on one-sided preference. There are many real-world applications of this Top – Trading Cycle algorithm beside dorm assignment to the students. It also helps in assignment for the students who are new and are without dorms, and looking for students to fill the empty dorms available, besides already matched students with their dorms. This algorithm is also used to assign student to the respective schools based on the school preference list provided by the students and availability of seats in the given list of schools. It is also used in hospitals where a patient who is in need for the kidney is provided with a compatible kidney donor by going through this cycle based on the preference list of most compatible donor list. Similarly, there are many such application where this algorithm would work.

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student “Sanjay Varma Rudraraju” in this advanced algorithm course, a Wikipedia link that explains the Top – Trading Cycle algorithm in detail. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/TTC.pdf

[2] https://en.wikipedia.org/wiki/Top_trading_cycle

Advanced Algorithms

Randomized Algorithm – Karger's Min Cut Problem

Problem Statement

Given a graph which is undirected, with no weights on edges and is connected, i.e., starting given specific node, we can reach to any node following a certain path; then we need to find a minimum cut such that it will disconnect the graph completely. Here, minimum cut represents the total number of minimum edges required to disconnect the given graph. Thus, we need to get rid of this minimum number of edges to disconnect the given graph [1].

Overview

Looking at the problem statement, we can say that a graph can be disconnected if we just try to disconnect one single node from the entire graph. Now, we need minimum number of edges to remove in disconnecting the graph to achieve the min-cut goal. So, all we need to find is the node with minimum degree, where degree stands for total number of edges incoming and outgoing that particular node. Once we find a node with minimum degree, we just need to eliminate all the edges connecting that particular node to other nodes in the given graph.

In the graph given below in figure 1, dotted edges represent the minimum degree for nodes A, F and E. Removing any of those pairs of edges will disconnect any of those three vertices from the remaining graph.

This will disconnect the graph completely. There are two approaches to solve this problem which are mentioned below.

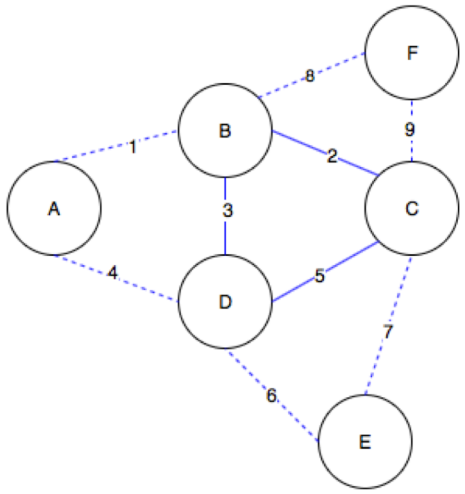

Figure 1. Graph G with selected dotted edges as minimum degree for nodes A, F and E respectively

Approach

Min S-T Cut using Ford Fulkerson Max-Flow Algorithm

This type of algorithm is used to find the maximum flow that can be generated from a given source S to reach the destination T through given set of paths. Here, the edges of the path carry some weight signifying the flow value. So, this algorithm will provide a min-cut solution such that flow passing through such edges described by the min-cut intersection will be maximum. Thus, Ford Fulkerson algorithm tries to find the minimum number of edges the could provide maximum flow. Clearly, it uses the same application of our desired problem statement.

The same Ford Fulkerson Max Flow algorithm can be used on a graph with unweighted edges by replacing all the unweighted edges with weighted edges containing equal amount of weight. So, here, the min-cut will give us the number of edges let say 'n', which in return will provide the max flow of value equal to the number of edges, i.e., 'n', if the weight of all those edges is 1.

The overall complexity of finding the min-cut using Ford Fulkerson Max Flow algorithm is $O(V^5)$ where V is the total number of vertices in the given graph. We know that the Ford Fulkerson Max Flow algorithm has run time complexity of $O(V^3)$ and it will run for each edge in the given graph, thus giving us the overall complexity of $O(V^5)$ [1,4]. Another approach that can be used to solve the given problem in this report is mentioned below which is much better that the Min S-T Cut using Ford Fulkerson Max Flow algorithm and it is called by the name Karger's Min-Cut algorithm.

Karger's Min-Cut algorithm

This is type of randomized algorithm which gives a solution that might not be the correct one. Thus, it gives a probabilistic answer to the correct solution. Randomized algorithm is an algorithm which gives solution based on random decision made throughout the process [2].

There are two types of randomized algorithms [1] –

1. One which always gives right solutions.
2. One which might not always give right solutions. It has a probability value to achieving the correct solution.

Given below is the pseudocode for Karger's Min-Cut Algorithm –

- Create a duplicate of the original graph G, say DG, to perform updates during the process.
- While DG contains more than 2 vertices {
  Randomly select an edge from DG connecting nodes a and b.
  Remove the edges between a and b, also merge the selected two vertices into one.
  Remove any edges that creates a self-loop on newly created merged vertex.
  }
- Return the number of edges connecting those two remaining vertices in DG at the end of the while loop.

Karger's Min-Cut Algorithm [4]

Let's take an example to run through the above algorithm and find out how random selection might result in a wrong solution or might result in a correct solution. Given below is the graph we need to disconnect using Karger's Min-Cut Algorithm –

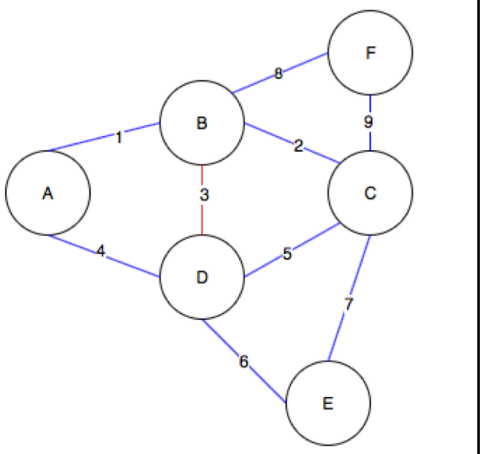

Figure 2. Randomly select edge 3 connecting nodes B and D

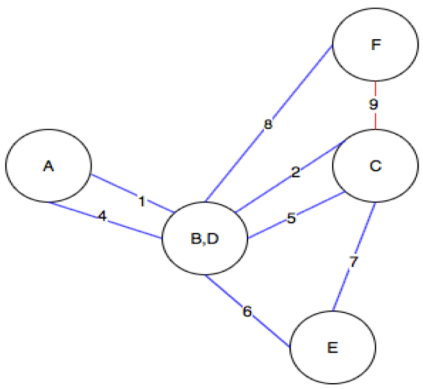

Figure 3. Merge nodes B and D after removing edge between them, then repeat the process by randomly selecting edge 9

First, we will randomly select an edge, let say edge 3 connecting nodes B and D as shown in figure 2. Then we will remove this selected edge and merge nodes B and D and also redirect other edges to this newly created vertex as shown in the figure 3. Then we select another random edge say edge 9 and repeat the same process again until we are left with just two vertices as shown in figure 6 (Wrong solution) and figure 8 (Correct solution).

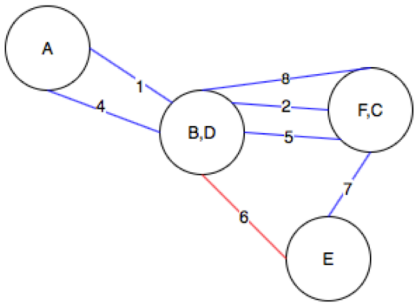

Figure 4. Merge nodes F and C after removing edge between them, then repeat the process by randomly selecting edge 6

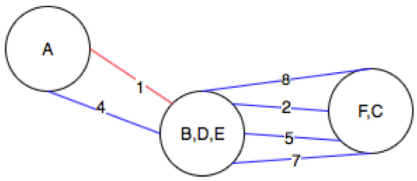

Figure 5. Merge nodes (B, D) and E after removing edge between them, then repeat the process by randomly selecting edge 1 (WRONG random selection)

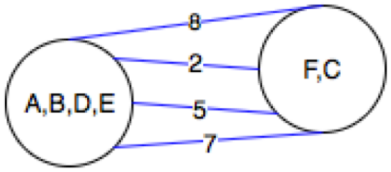

Figure 6. Merge nodes (B, D, E) and A, two vertices left, so min-cut value is 4 which is WRONG SOLUTION

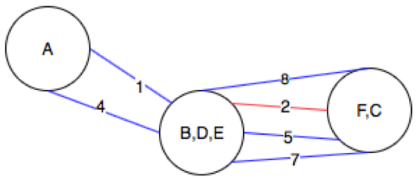

Figure 7. Merge nodes (B, D) and E after removing edge between them, then repeat the process by randomly selecting edge 2 (RIGHT random selection)

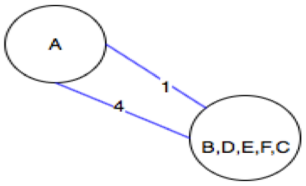

Figure 8. Merge nodes (B, D, E) and (F, C), two vertices left, so min-cut value is 2 which is CORRECT SOLUTION

As we can see, till figure 4 the algorithm went perfectly well. After removal of edge 6 we get a graph shown figure 5 and figure 7. Now, since we are randomly selecting edges in each iteration, figure 5 shows bad selection of random edge which result in larger min-cut value (4) obtained in figure 6, while figure 7 shows right selection of random edge which result in smallest min-cut value (2) obtained in figure 8. Hence proved, this algorithm will not always give us the right

solution to the asked problem due to random edge selection throughout the algorithm. The complexity of this algorithm is $O(V^2 \log V)$, where V^2 is to iterate through a pair of vertices and merging them, and log V is to get the randomly selected edge between them as well as removing it. Since this algorithm will give a correct solution based on some probabilistic value, the probability of actually getting the correct solution in term of vertices in the given graph is $1/V^2$ where V is the number of vertices in the given graph. Evaluation of this probability value is given in detail in the presentation [1]. So, the overall complexity considering the Karger's Min-Cut algorithm and the suggested problem statement of disconnecting graph is $O(V^4 \log V)$ which is better than Ford Fulkerson algorithm.

Another algorithm which is mentioned in the presentation is Karger – Stein algorithm. It is an extension of Karger's Min-Cut algorithm where the time complexity remains the same but the probability of finding correct solution is increased and is given as 1/log V [1].

Conclusion

In this report, we learned about performing Ford Fulkerson min-cut algorithm to find the minimum number of edges that can disconnect the graph. Also, we learned about randomized algorithm called Karger's Min-Cut algorithm which is better than the Ford Fulkerson algorithm but gives correct solution based on the probability value. Another algorithm which is mentioned in the presentation is Karger – Stein algorithm which improves in giving correct solution with higher probability value (1/log V) than previous Karger's Min-Cut algorithm ($1/V^2$). There are lots of real-world applications of this Karger's min-cut algorithm like in war, where troops would like to know minimum number of links in the network that can break the connection between the enemies operating over that network. Also, it can be used to identify area of network that if caused some problem, may result in failure of whole network to work. Another application is clustering problems and matching problems.

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Rushik Vartak" in this advanced algorithm course, a "geeksforgeeks" link with implementation of the Karger's Min Cut algorithm, a Wikipedia link that explains the algorithm in detail, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Randomized Algorithm - Karger's Algo.pdf

[2] https://en.wikipedia.org/wiki/Karger%27s_algorithm

[3] http://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/

[4] https://en.wikipedia.org/wiki/Ford–Fulkerson_algorithm

Advanced Algorithms
Hungarian Algorithm

Problem Statement

There are n tasks to be completed by n people and each person can do certain task with provided cost value. Then the problem statement is to find a solution such that n people do their respective tasks with total cost of completing all the tasks to be minimum. Thus, we need to find combination of person and task such that that particular combination yields lowest total cost value. This type of problem is called as an assignment problem and we need to find the optimal assignment of person to one particular task.

Overview

Looking at the problem statement, we can say that there would be a matrix representation of data where each cell would represent the cost of doing that particular task by one person. There are two approaches to solve this problem. First is the naïve/brute force approach which is efficiently not feasible to use. Then this report mainly focuses on the Hungarian algorithm which is an assignment problem based algorithm that provides solution efficiently as compared to the traditional brute force approach. Let's first discuss the brute force algorithm which is mentioned below.

Approach

Brute Force Algorithm

To solve this assignment problem, brute force approach would be to select each possible pair of person and task, and get the total cost for all the possible assignments. Then, from all those assignments select the one which has the lowest total cost value.

For example, consider there are n people to perform n tasks. Then there would be total n! possible combination of selecting those pairs of person and task groupings. Thus, evaluating this many combinations to get the one with lowest cost value is a tedious work and computationally inefficient as well. The complexity of this brute force algorithm would be O(n!) [1]. This is clearly not a polynomial time solution. This algorithm leads to redundant checks of unnecessary combinations which will not give the lowest cost value. Another algorithm is used to solve this problem in polynomial time which is known as Hungarian algorithm. It reduces the complexity up to great extent.

Hungarian Algorithm

Before going through this algorithm there is a theorem which needs to be followed which is as follows –

Given is a matrix representation of the data containing cost values in each cell for a person to do a particular task. Let's say we subtract minimum value selected from each row to all the values in the same row. Also, we do similar subtraction column wise where minimum value selected from each column is subtracted to all the values in the same column. Then the new matrix obtained will have bunch of zeros in it. So, the theorem states that after doing above subtractions, i.e., reducing the matrix, the solution obtained to solve the asked problem in this report will give the same optimal solution that was previously achieved on the original matrix [1][2]. Thus, reducing the matrix won't affect the outcome of the optimal solution.

Since we are going to do matrix reduction in the Hungarian algorithm, this theorem mentioned above should be taken into consideration. Also, there are set of assumptions that are needed to be taken care of which are as follows [1] –

1. Total number of tasks available and total number of people to handle those tasks should be equal. Thus, here the matrix representation of the data should be a balanced one.
2. In this given problem, one task cannot be done by group of people, or one person cannot handle more than one task. Thus, each person can only have choice to perform on one particular task.
3. If there are n tasks available, then a person has knowledge to do all those tasks without any hesitation.

4. Given matrix representation will have either all cost values or all profit values and solution should either be to reduce the cost or to increase the profit.

After going through the theorem and four set of assumptions let's have a look at the Hungarian algorithm. There are total 7 steps to be followed with certain amount of recursion which are given below [1] –

1. First do row reduction. Pick the minimum value from each row and subtract this value to all the values in the same row respectively. This is the first step of the matrix reduction.
2. Then do the column reduction. Pick the minimum value from each column and subtract this value to all the values in the same column respectively. This is the second step of the matrix reduction.
3. Now, perform the assignment on top of this reduced matrix. According to the theorem mentioned above, this assignment will work on reduced matrix providing optimal solution. Steps for assignment are as follows –
    a. First, for each row check if there exist a zero in it. If the zero is present in a particular cell of that row, then assign the task to that particular person represented by the cell with zero value in the given matrix and mark that cell as box. Similarly, look column wise and assign task to the person wherever we see a zero which is not marked as box.
    b. Now, again go first row wise and cross out all the zeros that are in the same row where the task was assigned by marking zero as box. This way those other zeros in the same row won't interfere, saying that it is already assigned to another person and no more available for another person. Similar process is repeated column wise by crossing of zeros that lie in the column of task which is already assigned to another person.
    c. If while assigning a task to a person, i.e., marking zero as box, if there are more than one zero in the same row or in the same column, then this represents possibility of more than one optimal solution. More than one zero in a row means that particular person can do more than one task with lowest cost. And more than one zero in a given column means that particular task can be done by more than one person with lowest cost. So we have more than one optimal solution.
    d. This process of steps from a-b is repeated until no more zeros are left in the entire matrix. Some of them are either boxed and some of them are crossed.
4. After all the zeros are either crossed or marked box, if the number of zeros marked as box is equal to the number of tasks that need to be assigned then, we have found the optimal solution and the algorithm ends here. But if this is not the case, then move ahead with step 5 mentioned below.
5. In this step, we will draw a bunch or lines across the matrix which will either be horizontal or vertical following the given below steps –
    a. Draw horizontal lines across rows that do not have boxed zeros, i.e. assignments
    b. Draw vertical lines across columns that have zeros in the marked rows that are represented by drawn horizontal lines done in the step 'a'.
    c. Similarly, if there is a zero in a particular column that is boxed (assigned), then draw vertical lines across those rows have the same boxed zero, if the line is not already drawn.
    d. Recursively call steps 5.a – 5.b until no more rows or columns are marked during the process.
    e. Now, pick those unmarked rows and columns that do not have vertical or horizontal line drawn and draw a line through them.
6. Step 5 will give a small reduced matrix, from which we pick the smallest element with lowest value and subtract that value throughout the reduced matrix. This lowest value is then added to the cell values that have intersection of those horizontal line and vertical line drawn in step 5.
7. Step 6 will give a new representation of reduced matrix upon which whole algorithm is repeated from step 3 to obtain the optimal solution.

The overall complexity of this Hungarian algorithm is $O(n^2)$ which is much better than the traditional brute force algorithm. This algorithm works for balanced matrix data, but what if we have less number of people than the given tasks or vice versa. In that case, the matrix data available for the problem is unbalanced. To solve this problem, we can introduce a dummy row or dummy column with all zero-value based upon the requirement. This, will solve the problem of unbalanced data by assigning all the tasks to given number of people and the dummy task remains as it is with an extra person doing no task if the number of people is one more than the number of assigned tasks.

Conclusion

In this report, we learned about performing algorithm based on assignment problem of assigning n number of tasks to n number of people. First, the brute force approach provides optimal solution but is not computationally feasible as the complexity is O(n!). Then we learned about Hungarian algorithm that provides a polynomial time solution of $O(n^2)$ which is much better as compared to the traditional brute force approach. There are many real-world applications of this assignment problem where a company needs to send a set of employees to different states to get certain job done by each employee. Similarly, this Hungarian algorithm can be used to solve problem of finding maximum weight bipartite matching [1].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Harnisha Gevaria" in this advanced algorithm course, a "topcoder" link with the implementation of the Hungarian algorithm, a Wikipedia link that explains the algorithm in detail, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Hungarian.pdf

[2] https://en.wikipedia.org/wiki/Hungarian_algorithm

[3] https://www.topcoder.com/community/data-science/data-science-tutorials/assignment-problem-and-hungarian-algorithm/

Advanced Algorithms
Sweep Line Algorithm

Problem Statement

Given n line segments in a 2-dimensional space with each line represented by two points with respective x and y coordinates, we need to find all the intersecting points that are created by these n line segments. So, there can be multiple cases occurring where all the n segments might be intersecting each other resulting in n^2 intersections, or only some of them might be intersecting creating let say I intersections. Thus, these reports provide algorithms that can give us the number of intersections taking place in the most efficient way possible [1].

Overview

Line segments in a 2-dimensional space can be very spread out or far from each other creating minimum intersections between them or they can be very compact occupying a small space and creating the maximum number of intersections. Given below is an example of how the 3 line segments would look like in a 2-dimensional space in figure 1.

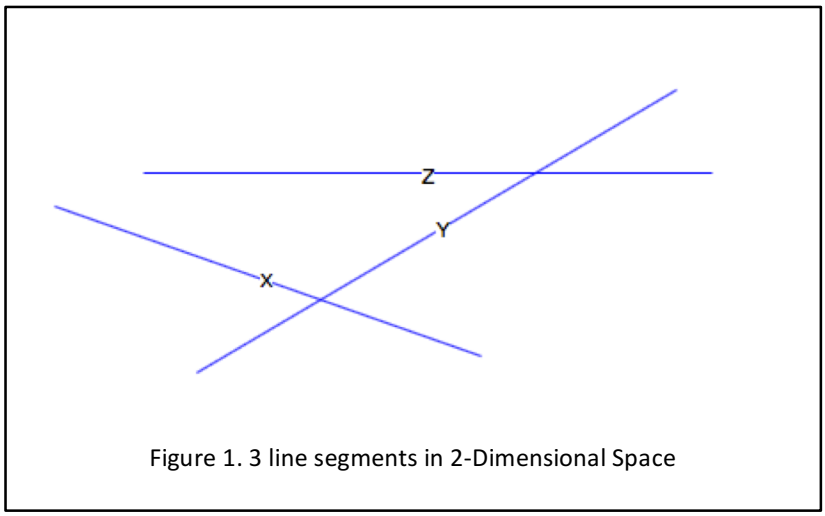

Figure 1. 3 line segments in 2-Dimensional Space

In the above example, there are two intersections taking place between lines X, Y (First intersection), and between lines Y, Z (Second intersection). There are two algorithms to solve this problem. Let's first consider the brute force approach to solving this problem which is mentioned below.

Approach

Brute Force Algorithm

In the brute force approach, we will each possible pair of lines and check whether those two lines intersect each other or not. Checking the intersecting point between two lines will be in constant time, i.e., O(1). But, here we are doing redundant checks between the pair of lines that do not intersect each other at all. So, the overall complexity of this brute force approach is $O(n^2)$ where n is the number of line segments [1]. We can provide with another approach that would reduce the complexity by only checking for the intersecting points that actually exists, instead of checking for all the pair of lines even if they don't intersect at all. This type of approach is mentioned below which is named as sweep line algorithm.

Sweep Line Algorithm

This algorithm can be used in two-dimensional space to solve the problems related to the geometry where we need to locate the where the point lies with respect to other points during that particular event of the sweep line [4]. It uses the concept of a vertical sweep line, which takes into consideration some events that might be helpful to solve this problem. In our case, there are three possible events considering line segments which are as follows –

1. The starting point of a given line segment if sweep line moves from left to right.
2. The ending point of a given line segment if the sweep line moves from left to right.
3. The intersecting point between two line segments.

These three events recorded every time the sweep line passes through any point of given line segment. Also, we need to store these events using a data structure. The best data structure to be considered for this problem is priority queue where we can store the name or label of these line segments.

Three events in the priority queue will behave as follows [1] –

1. During the start point of any line segment, the label of this line is inserted in the priority queue based on its y coordinate in the descending order. During this insertion, we also check if this line creates intersections with the lines having labels in the priority queue adjacent to this newly inserted label.
2. During the end point of any line segment, the label of this line is deleted from the priority queue. This shows there won't be any event occurring related to this line segment which ends here. So, new intersections that might occur after this event would be between line segments which are just adjacent to this removed line segment, i.e., the above line and the below line segment.
3. During the intersection point, we mark the intersection taking place and inverse the labels of the line segments causing this intersection. This inversion is necessary as after this event, the lines will change their direction in terms of events occurring and we need to check intersections between lines with this newly inverted line segments.

Consider the given below example to perform sweep line algorithm and find out intersections between three lines –

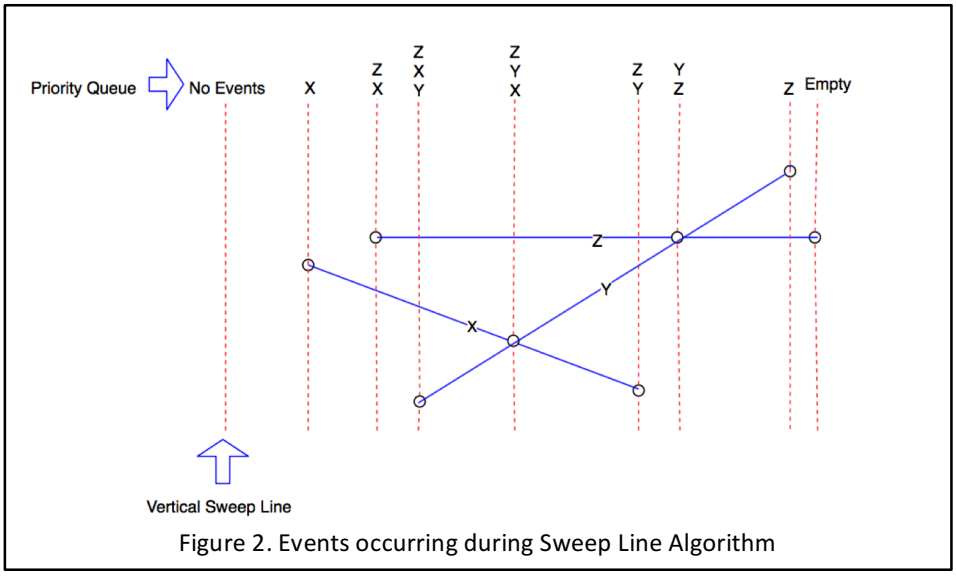

Figure 2. Events occurring during Sweep Line Algorithm

In the above example, the priority queue shows the events occurring during the sweep line algorithm. First, we get the starting point of X line, then Z line starting point appears above X and Y line starting point appears below line X. The 4th event is an intersection between lines X, Y. Thus, we inverse their labels in the priority queue. Now, the end point of X line appears so, we remove label X from the queue since there will be no further events with respect to line X. Next, the intersection happens between line Z and Y, so we inverse their labels in the priority queue. Finally, we remove both these labels when sweep line reached their endpoints. The algorithm stops when we hit the last endpoint after traversing through each point of n line segments.

As you can see, we need to add/remove or inverse the labels during each event. So, we need a data structure that can perform this in an efficient manner. In the presentation, the best and effective data structure mentioned is binary search tree which can do all these operations in log(n) time where n is the number of line segments. Also, the binary tree should be balanced after each removal and insertion of the label. There are some practical data structures that can deal with this problem like the Red black tree or the AVL tree. We can use these tree data structures to solve the problem of a self-balancing binary tree [1]. The presentation provided pseudocode to implement an AVL tree [1].

Finally, the time complexity of this sweep line algorithm is O((n+k) * log(n+k)) where n+k is actually the event occurring given the balanced binary tree data structure to store those events [1].

There are some special cases which are not covered in this report for example if the line is vertical and parallel to the sweep line, or the two line segments have the same starting point or ending point or both. Also, there might be the case where three or more line segments intersect each other at the same point. These special cases should also be considered while implementing the sweep line algorithm.

Conclusion

In this report, we learned about sweep line algorithm which is the much efficient way than the normal brute force algorithm to find out all the intersection point between given n line segments. The complexity of brute force approach is $O(n^2)$ while the complexity of sweep line algorithm is little better which is O((n+k) * log(n+k)). This algorithm can also be used to create a Voronoi diagram given n points in 2-dimensional space [4]. Also, this algorithm can be extended to use in 3-Dimensional space where we use a sweep plane instead of sweep line in the field of computational geometry [4].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Abhishek Jaitley" in this advanced algorithm course, a "geeksforgeeks" link with implementation of the Sweep Line Algorithm for n line segments, a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/SweepLine.pdf

[2] http://jeffe.cs.illinois.edu/teaching/373/notes/x06-sweepline.pdf

[3] http://www.geeksforgeeks.org/given-a-set-of-line-segments-find-if-any-two-segments-intersect/

[4] https://en.wikipedia.org/wiki/Sweep_line_algorithm

Advanced Algorithms

Travelling Salesman Problem Using Approximation Algorithms

Problem Statement

As the report title says itself, given some number of cities to visit say "n", the salesperson wants to make sales by visiting each and every city just once starting from a particular city and then after visiting each city in the path, come back to the city from where the salesperson started his/her journey. Now, a salesperson can visit one city from another either directly or via some other cities in between. This shows that the graph representation of these cities connecting each other is a complete graph. Problem statement is that salesperson wants to minimize the cost of travelling each city and returning back to where the salesperson started. So, edges representing the connection between those cities will always be a positive number. Starting from one city, visiting all the cities in the path and return back to the starting city represents a cycle which is also called as a Hamiltonian cycle. Thus, we need to find the Hamiltonian cycle that has the lowest cost.

Overview

The brute force approach would be to consider all possible Hamiltonian cycle which seems to be exhaustive search, so main focus of this report is an alternative approach using approximation algorithms. Let's first discuss about the brute force approach to solve this problem which is given below.

Approach

Brute Force Algorithm

Given n cities to visit by a travelling salesperson, there can be (n-1)! possible Hamiltonian cycle [1][3]. Thus, brute force approach would be look through the cost of each of those (n-1)! Hamiltonian cycle and pick the one which has lowest travelling cost. Clearly this seems to be an exhaustive search process. Consider if there are around 30 cities to visit, then using brute force approach there would be 29! Possible Hamiltonian cycles. This number is huge to deal with and the algorithm will run for ever. Another approach to deal with this problem is dynamic programming by storing solution to smaller subsets and increasing the size of subset as the algorithm approaches forward. But travelling salesperson problem using dynamic programming is still an exponential time solution [3]. Finally, this travelling salesperson problem seems to have no polynomial time algorithm to provide the optimal solution. If there is a problem that does not have a polynomial running time algorithm, then such problems can be solved using approximation algorithms. In this report, our main focus is to deal with those approximation algorithms that provides a feasible solution close enough to the optimal solution for this travelling salesman problem.

Approximation Algorithms

When a particular problem does not have a polynomial run time algorithm that provides optimal solution, then approximation algorithms are used that provide a solution which is said to be an approximation of the optimal solution. Thus, this approximation algorithms will provide a feasible solution that is as close as possible to the optimal solution [6]. This approximation value to the algorithm is used as α throughout this report [1]. So, if there is solution say L_o which is an optimal solution to any given problem and L is the solution obtained using an approximation algorithm, then the equation that relates optimal solution to the approximated solution is as follows [1]–

$L <= \alpha L_o$

This approximated solution is also called as a heuristic function to the given problem. Thus, using approximation algorithm we need to find a good heuristic function that will provide solution as close as possible to the optimal solution in polynomial run time.

There are three approximation algorithms to solve the travelling salesperson problem which are mentioned below [1] –

1. Nearest Neighbor Method – Greedy Approach
2. 2-Approximation using Minimum Spanning Tree
3. 3/2-Approximation using Minimum Spanning Tree and Perfect Matching

Before, going through the above approximation algorithms, there is one assumption that need to be taken care of. If the salesperson wants to visit from one city to another, then it would always be cheaper to go directly rather than going via a third city or via a path of other cities. This property is called as triangle inequality [1].

Nearest Neighbor Method

This is a greedy approach in which it looks for the city which is closest to current position of the salesperson. Thus, if a salesperson starts from a particular city, then look for the city which has lowest cost from that starting city and move to that closest city. Repeat the same process visiting closest city each time with respect to the current city salesperson is at. Thus, here salesperson looks for the local minimum. Clearly this is a greedy approach and will not always provide the optimal solution. Given below are the steps to follow for the nearest neighbor algorithm –

- Start from any city
- Pick the next city which is closest to the current city. Thus, pick the edge with lowest cost.
- Repeat the above process from newly visited city to next city which is not yet visited.
- After traversing through all the cities, reach back to the starting city from the last visited city in the path.

In this algorithm, the approximation value α comes out to be $(1+ \log_2 n) / 2$ [1]. Thus, here if the problem size increases, i.e., if the number of cities to visit increases, then the α value will also increase and the goodness of the heuristic function will decrease. This means the upper bound to the optimal solution will increase. So, in this approximation algorithm, we don't get a heuristic function that provides a constant upper bound value to the optimal solution. Another approximation algorithm will provide a heuristic function with constant α value which his mentioned below. Complexity of this approximation algorithm is $O(n^2)$ since the data of cost values for travelling from one city to another is given in matrix format and the algorithm traverses through each cell of the matrix [1].

2-Approximation using Minimum Spanning Tree

In this algorithm, a minimum spanning tree is created from the given graph representation of the cities connected to each other using Prim's algorithm or Krushal's algorithm. Starting from any given city or node in the minimum spanning tree, we will do a normal depth first search traversal walk through to visit each city. This DFS walk through will visit each city twice in the given minimum spanning tree which will violate the rule of travelling salesperson problem of visiting each city just once. So, as soon as we seen a visited city again in the path of DFS walk through, we just bypass from the city visited before this city to the city that will be visited after. For example, if the DFS walkthrough for 3 cities is 1-2-1-3-1. In this case, city 1 is visited again to reach city 3. So, we just bypass from city 2 to directly city 3 instead of going through 1, thereby creating a Hamiltonian cycle 1-2-3-1. This is the feasible solution obtained using 2-Approximation algorithm. Also, bypassing will always be cheaper than going via third city based on the triangle inequality property. The approximation equation in this algorithm is as follows [1]–

$L <= 2 * L_0$

The proof of correctness for this algorithm is mentioned in detail in the slides [1]. Thus, here the approximation value α is a constant. So, if the optimal solution of minimum cost to travel all the cities is say 40, then this approximation algorithm will give a solution that is bounded between 40 and 80. So, this algorithm might even give optimal solution but that is not always the case. There is still an improvement in terms of getting lower approximation value α better than 2 which is shown in the next algorithm. Complexity of this approximation algorithm is $O(n^2)$ since the data of cost values for travelling from one city to another is given in matrix format and the algorithm traverses through each cell of the matrix [1].

1.5-Approximation using Minimum Spanning Tree and Perfect Matching

In this algorithm, a minimum spanning tree is created from the given graph representation of the cities connected to each other using Prim's algorithm or Krushal's algorithm [7]. After creating the minimum spanning tree, cities with odd number of edges are taken into consideration and a perfect matching algorithm is run on these set of odd degree nodes. After performing the perfect matching, the new edges that we get are intersected with the edges that are already present in the minimum spanning tree. This way we created even number of edges for each node in the minimum spanning tree. Also, the perfect matching is performed to obtain the edge connection with lowest cost value between those odd degree nodes. So, the newly introduced edges will carry low cost. Finally, we repeat the process mentioned in 2-Approximation of creating a DFS walk through on this newly formed subgraph from the edges of the minimum spanning tree and the edges of the minimum weight perfect matching. And the same procedure is repeated again to obtain the Hamiltonian cycle out of this subgraph representing the feasible solution close to the optimal solution. The approximation equation in this algorithm is as follows [1]–

$L <= 1.5 * L_0$

The proof of correctness for this algorithm is mentioned in detail in the slides [1]. Thus, here the approximation value α is a constant. So, if the optimal solution of minimum cost to travel all the cities is say 40, then this approximation algorithm will give a solution that is bounded between 40 and 60 which is much better upper bound than the 2-Approximation algorithm. So, this algorithm might even give optimal solution but that is not always the case. Complexity of this approximation algorithm is $O(n^3)$ which shows a tradeoff to get better approximation value after using additional perfect matching algorithm along with the concept of minimum spanning tree [1].

Conclusion

In this report, we learned about solving the Travelling Salesperson Problem which is clearly an NP-Hard problem since there is no polynomial time algorithm available that provides optimal solution. Brute force approach and dynamic programming are both exponential run time algorithms. So, we learned about approximation algorithms that are used when a particular algorithm does not provide optimal solution in polynomial run time. We learned about three approximation algorithms in this report namely Nearest neighbor method, 2-Approximation and 1.5 Approximation providing different heuristic function solutions to the given travelling salesperson problem. There are lots of real world application of this problem like Interview Scheduling, Vehicle routing problem which need implementation of travelling salesperson problem to work in polynomial time [1].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by myself in this advanced algorithm course, a "geeksforgeeks" link with implementation of the Travelling Salesman Problem using naïve approach and dynamic programming as well as another link with the implementation of 2-approximation using minimum spanning tree [3][4], a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/TSP.pdf

[2] https://en.wikipedia.org/wiki/Travelling_salesman_problem

[3] http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

[4] http://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/

[5] https://en.wikipedia.org/wiki/Christofides_algorithm

[6] https://en.wikipedia.org/wiki/Approximation_algorithm

[7] https://en.wikipedia.org/wiki/Prim%27s_algorithm

Advanced Algorithms
Strassen's Algorithm

Problem Statement

Given two matrices, perform matrix multiplication in the efficient way. In this report, we will learn about approaching problems that need matrix multiplication a lot, especially in the mathematical area where matrix multiplication should be done as fast as possible. We will first get to know the naïve/brute force approach to perform matrix multiplication and then a little better with smaller run time complexity called Strassen's algorithm.

Overview

Matrix multiplication takes place a lot in linear algebra and in mathematical regions. It is also used a lot in computer graphics, where image reflection is manipulated using matrices. A simple naïve approach to perform matrix multiplication is not that tedious if the numbers in both matrices are smaller to deal with. This will make multiplication between the numbers happen as fast as possible. But what if the numbers in each cell of the two matrices are huge, represented in terms of long numbers. Then, multiplication between each two numbers will take a long time. Thus, we need a way to come up with an algorithm that reduces the number of multiplications taking place while performing matrix multiplication. There are three algorithms mentioned in this report to solve the problem of matrix multiplication which are given below –

1. Naïve/Brute Force Approach
2. Divide and Conquer
3. Strassen's Algorithm

Let's first discuss the brute force approach to perform the matrix multiplication.

Brute Force Algorithm

Brute force approach is traversing through each element in 1st matrix row wise and multiplying them with corresponding elements in 2nd matrix column wise. Given below is the representation of brute force approach of performing matrix multiplication –

A	B
C	D

Matrix 1

E	F
G	H

Matrix 2

Resulting matrix 3 =

AE+BG	AF+BH
CE+DG	CF+DH

Here, 8 multiplications are performed along with 4 addition operations. In this algorithm, two loops are to traverse through each element in both the matrices and third loop is to do computation and storing in new resulting matrix. Thus, here the complexity is $O(n^3)$ where n is the size of the matrix [1].

Divide and Conquer Algorithm

Another approach to perform matrix multiplication is divide and conquer, where give matrix is divided into smaller set of matrices recursively until we have only one element representation of each matrices. Even after this recursive approach the number of multiplication and additions remains the same as normal brute force approach. This recursion for 2X2 matrix multiplication is provided using relation given below [1] –

$T(n) = 8T(n/2) + O(n^2)$

In the above equation, 8 is the number of time multiplication is performed for the n/2 matrix size obtained after dividing the original matrix into smaller matrices. This algorithm needs one assumption to follow that the size of matrix should be 2X2. The above equation can be obtained using Master's theorem. Since the number of multiplication remains the same, the time complexity of this algorithm is $O(n^3)$ which is same as the brute force approach [1]. So, there is no scope of improvement in using this algorithm to perform huge matrix multiplication problems. Another algorithm mentioned next is Strassen's algorithm that provides a little improvement in terms of number of multiplications performed.

Strassen's Algorithm

To provide improvement in matrix multiplication a German mathematician namely Volker Strassen came up with an algorithm named after himself called Strassen's algorithm [2]. This algorithm provides a little improvement in terms of number of multiplications performed by introducing certain formulas including addition and multiplication operations. It uses the same divide and conquer approach.

There are 7 formulas (multiplications) carried out which are mentioned below to get the answer to matrix multiplication [1] –

P1 = A * (F-H)

P2 = (A+B) * H

P3 = (C+D) * E

P4 = D * (G-E)

P5 = (A+D) (E+H)

P6 = (B-D) (G+H)

P7 = (A-C) (E+F)

Terms used in these 7 formulas are with respect to the matrices 1 and 2 presented in the brute force section. After calculating the above 7 terms, the resultant matrix will look something like this –

Resulting matrix 3 =

P4 + P5 + P6 – P2	P1 + P2
P3 + P4	P1 + P5 + P7 – P3

Here, 7 multiplications are performed. In this algorithm, two loops are to traverse through each element in both the matrices similar to the brute force approach and third loop is to do computation and storing in new resulting matrix. This recursion for 2X2 matrix multiplication is provided using relation given below [1] –

$T(n) = 7T(n/2) + O(n^2)$

In the above equation, 7 is the number of time multiplication is performed for the n/2 matrix size obtained after dividing the original matrix into smaller matrices. This algorithm needs one assumption to follow that the size of matrix should be 2X2. The above equation can be obtained using Master's theorem. Thus, here the complexity is $O(n^{\log 7})$, i.e., $O(n^{2.8})$ where n is the size of the matrix [1].

Even with little improvement, this algorithm is not used much to perform matrix multiplication. There are other better matrix multiplication algorithms than Strassen's algorithm. The fastest algorithm to perform matrix multiplication is Coppersmith-Winograd algorithm with time complexity of $O(n^{2.37})$ [4].

Conclusion

In this report, we learned about efficient way to perform matrix multiplication using Strassen's algorithm. Even if Strassen's algorithm provides a little better solution to matrix multiplication, the difference in the time complexity between naïve approach and this algorithm is not much. So, at the end naïve approach is preferred to perform simple matrix multiplication. This has lot of application in the mathematical area as well as programming where there is a need to perform multiplication between huge matrices containing big numbers where it is better if the multiplication between numbers is reduced as much as possible providing same exact solution that the naïve approach would give. There is another algorithm which is fastest to perform matrix multiplication and it is called as Coppersmith-Winograd algorithm [4].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Prajakta Made" in this advanced algorithm course, a "geeksforgeeks" link with implementation of the Strassen's Algorithm, a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Strassen's.pdf

[2] https://en.wikipedia.org/wiki/Strassen_algorithm

[3] http://www.geeksforgeeks.org/strassens-matrix-multiplication/

[4] https://en.wikipedia.org/wiki/Coppersmith–Winograd_algorithm

Advanced Algorithms
Online Algorithms

Problem Statement

Given a huge set of known data to a particular algorithm, we can achieve the optimal solution to the asked problem. But what if we need to solve a problem and the data available is not clear. So, the data is dynamically achieved during the run time of the algorithm. In this case we have no idea about the pattern of the data arriving as an input to this algorithm. So, this type of algorithm, processing input that is arriving dynamically is called as online algorithms. Thus, online algorithms should be such that it provides solution as close as possible to the optimal solution as fast as possible even if the data is dynamically arriving and algorithm have no idea about the pattern of it.

Overview

Online algorithms should provide a solution which should be as close as possible to the optimal solution by making sequence of smart decisions in the phase of uncertainty [1]. Such decision making might not guarantee the exact optimal solution. This comparison can be provided by creating a competitive analysis.

Let's say the optimal solution to a certain problem is costOP and the algorithm A is an online algorithm which using a sequence of smart decisions gets a solution costA, then the r-competitiveness can be found out using given below equation –

CostA(d) <= r * CostOP(d), where d are the sequence of decision taken to reach the solution.

This type of approach for using online algorithm have lots of real-world applications, some of which are mentioned below like Ski-Rental, Searching Pizza Unit etc.

Ski-Rental Problem

Let's say a bunch of people plan to go for skiing and need to rent ski shoes. What if they are not sure on the amount of days they are going to spent doing skiing so that they could decide on either renting the ski shoes or buying them. Shop employee gives them a choice of renting shoes for 1$ per day or buying them for some B$.

Now, for the optimal algorithm, there will be a decided number of days say N. So, if N > B, then optimal algorithm will suggest to buy the shoes for B$, but if N < B, then it is optimal to rent the shoes. Such decision cannot be made using online algorithms directly as it has no idea about the amount of day they will spent skiing. So, the online algorithm will start with renting shoes by paying 1$ each day, and as soon as they hit the mark of B$, they will buy the shoes instead of renting it for yet another day. Thus, online algorithm in this problem is said to be 2-competative looking at the given below equations [1] –

1. If N <= B, then CostOP = N$ while CostA = N$
2. If N > B, then CostOP = B$ while CostA = B$(renting first B days) + B$(Buying on B+1th day)

Pizza Searching Problem

Let's say there are 2n rooms evenly spaced on both the side of the person standing in between and there is a treat of free pizza in one of those rooms. In case on optimal solution, the person knows all the room information and also knows where the free pizza is being given. So, optimal solution would be going directly to that particular room where the free pizza is given. In case of online algorithm, the person starts moving in zig-zag manner, by first moving towards right one step and then peeking through the first room on the right. Then, if there is no pizza in that room, the person moves back two steps to reach first room on the left and peek through that room to check for free pizza. Similarly, the person check through all the rooms in zig-zag manner to reach the destination. This online algorithm in this problem is said to be 9-competative looking at the given below equations [1][3] –

CostA <= CostOP + 2(4CostOP)

So, CostA <= 9 * CostOP

Paging Problem

Paging is also an example of online algorithm, where after receiving a new page with cache size already full, randomly a page is evicted based on the decision made using online algorithm. Let's say at most a pager can store only K pages in the cache. Fetching a particular page from this cache will happen in constant time.

Now, if a new page comes up, then the replacement of this page with one of the already stored page in the cache is done using certain given below approaches [1][3] –

1. Least Recently Used (LRU)
   The most recently used page is evicted from the cache and replaced with new page
2. Least Frequently Used (LFU)
   The page which is not used frequently is evicted from the cache and replaces with new page
3. First In First Out(FIFO)
   The page that was first entered in the cache is the first to be evicted when the cache size becomes full and a new page arrives to be entered in this cache.

All the above page removal techniques are K competitive in nature.

Also, the page eviction technique can be random, where page is randomly selected from the cache to be evicted to store newly arrived page in the cache. This randomness can cause three adversaries which are given below [1] –

1. Adaptive
2. Omniscient
3. Oblivious

These adversaries are explained in detail in the presentation slides and the competitive analysis of choosing randomly is O(log K) [1].

Conclusion

In this report, we learned about importance of online algorithms in the real-world problems where we don't have any information of the future data that is received in chunks rather than having all the data at once and already knowing about it. Thus, it explains how such algorithms can be used to dynamically make certain decisions which should favor the problem with solution as close as possible to the optimal one. Dynamic decision making should be precise while using these online algorithms. Such algorithms are used in sensor networks, network traffic applications [1]. It is also used in huge database transactions happening all around the world, where the amount of transaction happening is unknown to the system, and it should smartly allocate resources to store as well as perform such transactions.

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Vishal Kole" in this advanced algorithm course, Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/online.pdf

[2] https://en.wikipedia.org/wiki/Online_algorithm

[3] http://people.seas.harvard.edu/~minilek/cs224/fall14/lec/lec8.pdf

Advanced Algorithms
Triangulation Algorithm

Problem Statement

Given a polygon of any shape, divide it into set of triangles by connecting two points creating a set of diagonal lines inside the polygon. This process is known as triangulation. So, if there is an art gallery which needs to be protected using some cameras such that those cameras cover entire polygon structure of the art gallery. The problem statement is to find a solution which helps to use lowest number of cameras required to cover entire art gallery. So, finding the placement of those cameras inside the given polygon is what we need to find in this problem to achieve the optimal solution. This can be done using triangulation algorithm.

Overview

If the given polygon structure is more complex, then the number of cameras required will also increase. Thus, more number of vertices might lead to more cameras. But even with same number of vertices say n, the structure of polygon will provide a final call on the number of cameras required to deal with that polygon. The simplest polygon structure to deal with is convex polygon, in which only one camera is required to cover the entire convex polygon. For a given polygon with n vertices, there can be n-2 possible triangles that can be formed by creating diagonal line by connecting two vertices [1]. Given below is an example of a complex polygon with 10 vertices, and so there are 8 triangles formed as show in the second figure –

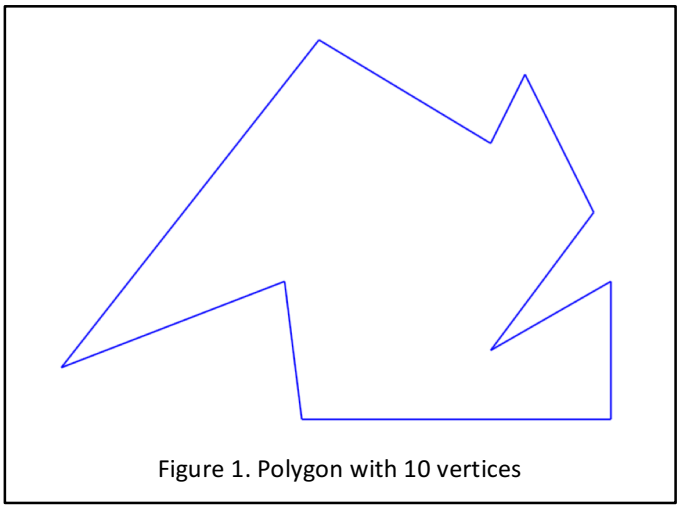

Figure 1. Polygon with 10 vertices

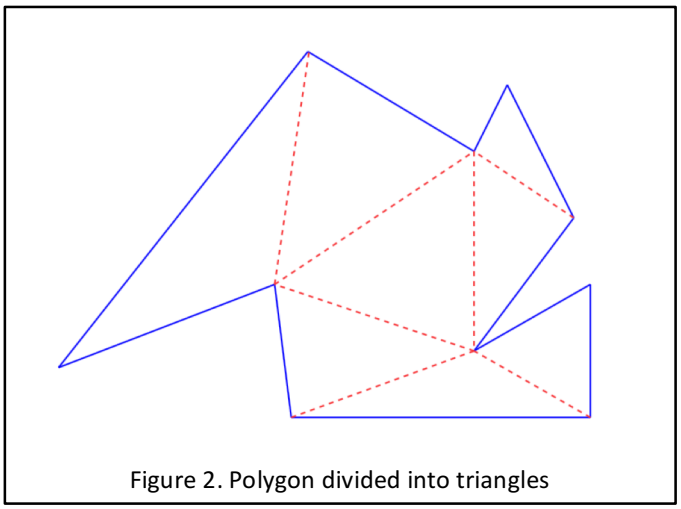

Figure 2. Polygon divided into triangles

Now, after dividing polygon with n vertices into n-2 triangles, the placement of the cameras can be done in each of those n-2 triangles. Thus, we need to have n-2 cameras. But still here, n-2 cameras are too much to cover entire polygon. Another approach is to place the cameras on those red diagonal lines instead inside each triangle. This way the camera will cover both the triangles sharing that diagonal line. Thus, total cameras now needed will be n/2. But there is still some room for improvement where if we pick a vertex that shares lots of triangles, then placing camera at that vertex will cover all those triangles.

3-coloring triangulation polygon

In this method vertices of the polygon are colored using three coloring technique with color “white”, “grey” and “black”. Here, two vertices connected by an edge does not share same color. This 3-coloring can be achieved using normal depth first search traversal. Each triangle in the given polygon will have three vertices with colors “white”, “grey” and “black”. Given below is an example of triangulated polygon with 3 coloring –

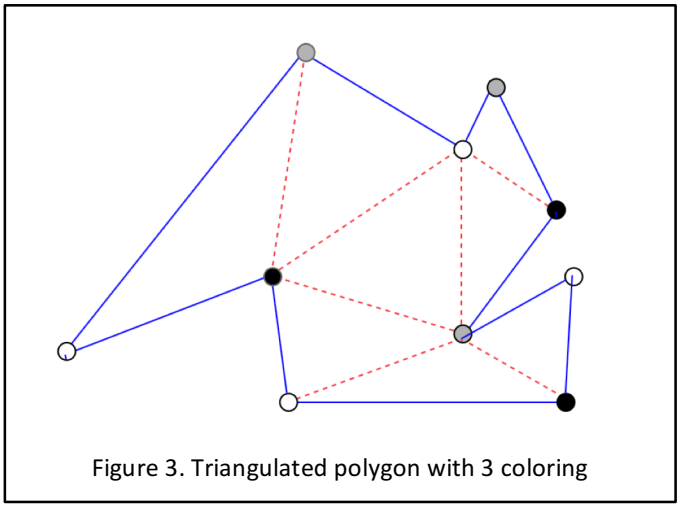

Figure 3. Triangulated polygon with 3 coloring

After performing 3 coloring, the cameras can be placed on vertices of the polygon that have lowest same colored vertices. In figure 3, cameras can be placed on “black” colored vertices or “grey” colored vertices, since both of them have lowest same color vertices, i.e., 3. Thus, here total number of cameras required are n/3 which is much better than the previous solution. This is the worst-case scenario, where there are n/3 peeks representing triangle or cone shape and the vertex on the leftmost cone is connected directly to the vertex on the rightmost cone. Thus, in this case there is a need of n/3 cameras. But, in other cases, there is still improvement with cameras less than n/3, required to fill the entire polygon. Now, since we know n/3 is the best solution of number of cameras required, we need to find the efficient algorithm to achieve this solution. In a given polygon of n vertices, n-2 triangles are formed and n-3 diagonals are required to divide the given polygon into those triangles [1][2]. Given below is the brute force approach to perform triangulation.

Brute Force Algorithm

1. For a selected edge e1, pick edge e2 that shares one vertex with edge e1 and draw a diagonal line using these two vertices on the other side of the common vertex of these two edges.
2. If the diagonal drawn intersect any edge of the polygon, then it is not a diagonal
3. If the diagonal is complete with no intersection happening in between, then it is a correct diagonal that divides given polygon into two parts.
4. Recursively perform steps 1 to 3 for n-3 diagonals.

This brute force approach has a time complexity of $O(n^4)$ where n is the total number of vertices in the given graph. If the given polygon is convex in nature, then finding diagonal lines is very easy by picking a particular vertex and connecting it with other vertices not adjacent to the current vertex. This can be done in linear time, i.e. O(n). Thus, another algorithm that is better than brute force approach converts a polygon into small monotone pieces of polygons which are convex in nature. This algorithm is mentioned below.

Triangulation of Monotonous Polygons

It uses the concept of horizontal sweep line that starts from the top and adds vertices to the stack data structure when they interact with the sweep line. At start two vertices encountered are entered in the stack as it is. Now, as soon as the third vertex is encountered, check if this vertex lies on the right side of the line segment formed by first two vertices. If it lies on the right side, then connect this vertex to the topmost vertex in the stack. After doing so, remove this topmost vertex from the stack.

If the vertex encountered lies on the left side of the line segment formed by first two vertices, then just push this new vertex into the stack. Repeat this procedure until there are no more vertices encountered by the sweep line that is horizontally moving downwards. This algorithm of creating monotonous pieces takes linear time since it visits each vertices of the polygon just once. Thus, after creating monotonous polygons, triangulation can be easily performed with linear time complexity since each monotonous polygon represent convex nature. Thus, here the overall complexity of performing this algorithm is $O(n^2)$ [1]. There are some terms related to types of vertices while dealing with constructing monotonous polygons like start vertex, end vertex, merge vertex, split vertex and regular vertex. These types are identified when sweep lines passes through them and looks at the nature of the edges using these vertices. Detailed explanation about this type of vertices is explained in the slides [1].

Conclusion

In this report, we learned about performing triangulation of a polygon which has a very important real-world application to deal with, for example placing of cameras in a polygon representing a complex structure. Thus, here dividing a polygon into triangles helps to figure out how many cameras are required to cover each of those triangles. After doing triangulation, we learned that best possible solution of number of cameras needed is n/3 where n is the number of vertices in the polygon using the 3-coloriung technique. To perform triangulation, brute force technique gives $O(n^4)$ time complexity for processing the polygon directly. But, if the given polygon is divided into monotonous pieces that represents a set of convex polygons, then triangulation can be performed efficiently in linear time per monotonous polygon. Thus, here the overall complexity of performing triangulation on monotonous polygons is $O(n^2)$ [1].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student “Shashank Rudroju” in this advanced algorithm course, a “github” link with implementation of the Triangulation Algorithm, a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Triangulation.pdf

[2] https://en.wikipedia.org/wiki/Delaunay_triangulation

[3] https://github.com/themadcreator/delaunay/blob/master/src/org/delaunay/algorithm/Triangulation.java

Advanced Algorithms
Red-Black Tree

Problem Statement

Binary search tree provides operations like insertion, deletion and searching a node in O(log n) run time if the tree is well balanced. But what if while inserting new nodes in the tree, the binary search tree becomes unbalanced or it becomes a skewed binary tree. In that case, the same three operations would take O(n) run time to perform. To avoid this situation while insertion as well as while deletion, in this report we are going to focus on new implementation of tree called Red-black tree which is similar to binary search tree but it self-balances after each insertion and deletion operation. Thus, here the Red-black tree provides O(log n) as worst case time complexity for insertion, deletion and search operations.

Overview

There are certain properties which needs to be followed in order to construct a correct red-black tree which are as follows –

1. Each and every node in the tree are either colored black or red.
2. Root node in the tree is always a black node.
3. Leaf nodes are represented by NILL which are always black in color.
4. If a node is red, then it cannot have child nodes with red color, or a parent with red color.
5. A path from any node to the leaf node containing NILL will always contain same number of black nodes.

Also, while performing the insertion and deletion, the algorithm uses two terminologies which are mentioned below –

1. Grand Parent – It is the parent of the parent node of the current inserted or deleted node.
2. Uncle – It is the sibling of the parent node of the current inserted or deleted node.

Given below is an example of how the Red-black tree looks like –

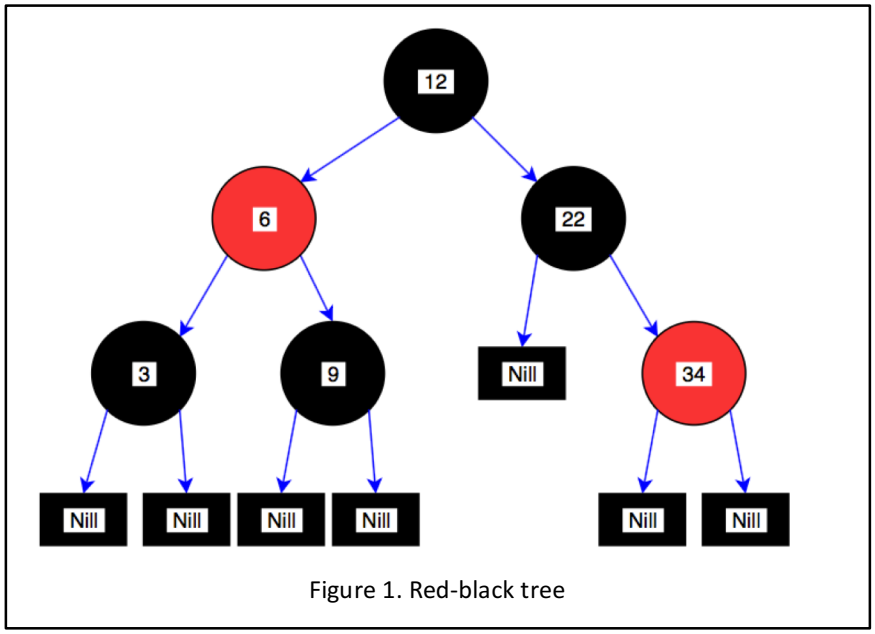

Figure 1. Red-black tree

Now, let's first look at the insertion process in the Red-black tree.

Insertion Method

The node which is newly inserted in the red-black tree is always marked with red color at first. After inserting the new red node three cases takes place which are mentioned below –

1. The parent node of current added red node has black color. In this case, inserting a red node does not violate the property of the red-black tree, so it can easily be added without any fix.
2. The parent node of current added red node has red color. Also, it's uncle has red color. In this case, both the parent and the uncle node will change the colors from red to black and the grandparent node will be of red color.
3. The parent node of the current added red node has red color, but it's uncle has black color. To solve this situation, there are two cases that can occur which are given below.
    a. If the current added node is the right child of its parent node, then we need two steps to perform. First perform left rotation on the parent node of the currently added red node. After that perform right rotation on the grandparent node of the currently added red node on the new structure obtained in the first step. After performing these two steps, we can fix the coloring using steps 1 or 2 if necessary.
    b. If the current added node is the left child of its parent node, then we need to perform a single step to do right rotation on grandparent node of the currently added red node. After that we can fix the coloring using steps 1 or 2 if necessary.

Time complexity of performing the left or right rotation is constant, i.e., O(1), while the time complexity of fixing the color after performing rotations or directly is O(log n). Thus, the overall time complexity of the insertion method in the worst case will always be O(log n). Pseudo code for implementation of the insertion method by handling those three cases is mentioned in the presentation slides [1].

Deletion Method

Deletion of the node from a given red-black tree is comparatively more complex than the insertion method, since after deleting a particular node, lots of alternative cases can happen which might violate the red-black tree properties, also making the tree unbalanced. There is a possibility of completely changing the structure of the tree after deletion of a particular trivial node from the red-black tree. Before deleting a particular node, we mark that node as double black. After deleting a node, there can be 4 cases that needs to be handled which are mentioned below –

1. Sibling of current selected node is a black node and its children are also black nodes. In this case, double black node is deleted and its parent is marked as double black node. Also, the sibling node is marked as red node. This creates another case which can be solved using one of the given scenarios of deleting node
2. Sibling f current selected node is a red node and its children are black nodes. In this case, if the selected node is a left child of its parent node, then right rotation is performed on the parent node, else if the seletec node is a right child of its parent node, then left rotation is performed on the parent node. After performing rotation, recoloring is done to solve the violation of red-black tree.
3. Sibling of current selected node is a black node and its left child is a red node and its right child is a black node. In this case, we perform right rotation on the current selected double black node around the sibling node and also exchange colors between the old sibling and the newly formed sibling after performing right rotation. After doing so, recoloring is done to solve the violation of red-black tree.
4. Sibling of current selected node is a black node and its left child is a black node and its right child is a red node. In this case, left rotation is performed on the parent node of the currently selected double black node. This rotation will create a new structure with new grandparent node. This grandparent node has the color of parent node of the selected double black node. The parent of the selected double black node is then marked as new double blacked node. Also, the color of the uncle node of the selected double black node is colored as black node.

Time complexity of performing the left or right rotation is constant, i.e., O(1), while the time complexity of fixing the color after performing rotations or directly is O(log n). Thus, the overall time complexity of the deletion method in the worst case will always be O(log n). Pseudo code for implementation of the deletion method by handling those four cases is mentioned in the presentation slides [1].

Conclusion

In this report, we learned about implementation of Red-black tree which is very much similar to the normal binary search tree that provides insertion and deletion operations with worst time complexity of O(log n) by self-balancing the tree after each insertion and deletion [2]. Thus, even search operations result in O(log n) after any number of insertion and deletion operations. Such implementation of tree helps a lot when the insertion inputs are unknown which might result in skewed binary tree. This is easily handled by Red-black tree and tree is never unbalanced. Red-black tree are complex to implement but still are used by C++ and Java to create map structures which are used more often in the programming life. There are similar data structures like AVL tree and B tree, but red-black tree is the best in terms of maintaining the worst time complexity of O(log n) for insertion, deletion and search operations [4].

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student "Ketan Joshi" in this advanced algorithm course, a "geeksforgeeks" link with implementation of the Red-black tree insertion and deletion methods [3], a Wikipedia link that explains the algorithm in detail with other applications for the same, and also some other sources of explanation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/Red-Black%20tree.pdf

[2] https://en.wikipedia.org/wiki/Red–black_tree

[3] http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

[4] https://en.wikipedia.org/wiki/AVL_tree

Advanced Algorithms

Efficient Maximal Network Flow Algorithm

Problem Statement

Given a connected graph with number of nodes in between, one source node from where the flow is originating and one sink node where all the flow is converged back, we need to find all the paths from source node to sink node such that these paths combined carry maximum flow. Thus here, each edge connecting two nodes have some weight to carry the flow. In this report, we are going to focus on the algorithm called push-relabel algorithm which is efficiently much better than Ford-Fulkerson and Edmond-Karp algorithms.

Overview

First let's discuss about the Ford-Fulkerson and Edmond-Karp algorithms a bit. Both algorithms find the source to sink paths using augmentation paths, thus representing the augmentation path finding algorithms. In both the above algorithms, augmenting paths are found from source to sink using the residual graph created after each iteration. This step is repeated again and again until we found all the augmenting paths from the residual graph with maximum flow passing from source through sink [4][5]. The complexity of Ford-Fulkerson algorithm is O(Ef) where E is the number of edges and f is the maximal flow contained by each path. The time complexity of Edmond-Karp algorithm is $O(VE^2)$ where V is the number of vertices in the graph included source and sink vertices while E is the total number of edges in the given graph. There is an algorithm called push-relabel algorithm developed by Andrew Goldberg and Robert Tarjan which is efficiently better than Ford-Fulkerson algorithm and Edmond-Karp algorithm as well and this new algorithm is widely used as compared to the other two mentioned algorithms [1][2]. So, this report is focused towards use and implementation of this push-relabel algorithm which is mentioned below.

Approach

Push-Relabel Algorithm

This algorithm changes the graph step by step using local information to obtain all the paths from the residual graph that is built during those steps which when combined has maximum flow going through those paths. This algorithm is performed on the same graph structure which contains one source node, one sink node and all the other nodes in between. In this algorithm, with respect to a particular node, all the flow is pushed towards this node even though it does not have the capacity as much as the flow pushed towards it. Now, this algorithm uses some data structure and parameters used throughout the algorithm which are mentioned as follows [1] –

1. Preflow – This represents the current amount of flow carried by an edge between two nodes say a and b.
2. Excess flow - The amount of flow which is exceeding then the certain capacity hold by that particular node in the graph
3. Edge capacity is denoted by c(a, b) where a and b are two nodes connected by an edge
4. Residual edge capacity is denoted by $c_f(a, b)$ where a and b are two nodes connected by backward residual edge
5. Height function represents the current height for a particular node. This parameter is hold by each node in the given graph representing its current height.
6. A residual graph representing the backward edges which are used to show the amount of flow that can be sent backwards to the source node if needed

This algorithm works mainly on two methods namely, push method and relabel method. Detail information about these two methods is as follows –

Push Method

This method is used to push the flow from one node to another. It pushes the entire flow if even if the capacity is not matched by the receiving node. Also, the pushing of flow takes place from node which is above the receiving node in terms of height parameter mentioned previously in this report. The node with excess flow is called upon to push this excess flow to the node with height less that the current node selected to push method [1].

Relabel Method

In this method, the height of the current node selected by this method is increase by one and checked if there is a particular node lower than this current node to push its excess flow to [1]. This method is only called for nodes that have excess flow towards it.

Given below is the most generalized push-relabel algorithm –

1. Initialize all the parameters and data structures required by this algorithm.
2. While there is a need to perform push operation or relabel operation
   Perform the asked operation
3. Stop when there are no more nodes with excess flow left in them to transfer to other nodes.

Complexity of this algorithm completely depends on the push and relabel operations and it accounts for $O(V^2E)$ run time, where V is the total number of vertices and E is the total number of edges connecting those vertices in the given graph.

Now, another algorithm which is little better than push-relabel algorithm is called as Relabel-To-Front algorithm which is explained below –

Relabel-To-Front Algorithm –

This algorithm is almost similar to push-relabel algorithm with addition of the concept of admissible edges [1]. An edge is considered as admissible edge if the flow can be transferred from one node with more height to another node with less height. Also, the node with more height should have some excess flow to push to other nodes with less height.

The push method will create some inadmissible edges during the process but the same method will never create admissible edges [1].

The relabel method will create all the edges as inadmissible for a given node say a, if we found that one admissible edge is leaving from node a.

In this algorithm, the push and relabel method are combined into just one method implementation and it is called as Discharge method. The complexity of this algorithm is $O(V^3)$ which is little better than the push-relabel algorithm and much better than the Ford-Fulkerson algorithm and Edmond-Karp algorithm.

Conclusion

In this report, we learned algorithms to find the maximum flow that can be obtain moving from source node to sink node passing through different paths in the given graph structure. First, the two algorithms namely Ford-Fulkerson and Edmond-Karp provided a solution but not efficient enough that word on finding augmenting paths through the process to obtain the maximal flow. Then, we learned about new algorithm called pus-relabel algorithm which is much better than the above mentioned two algorithms with time complexity of $O(V^2E)$ where V is the total number of vertices and E is the total number of edges connecting those vertices in the given graph. Then, we find out another approach called Relabel-To-Front algorithm which provides a little better run time complexity of $O(V^3)$ as compared to push-relabel algorithm.

Credits

The above report and the terms, as well as algorithms used in it, are based on the presentation given by a student “Alex Hoover” in this advanced algorithm course, a Wikipedia link that explains the push-relabel algorithm in detail and a “geeksforgeeks” link with method implementation. Links are mentioned as follows –

[1] https://www.cs.rit.edu/~algorg/Materials/push_relabel.pdf

[2] https://en.wikipedia.org/wiki/Push–relabel_maximum_flow_algorithm

[3] http://www.geeksforgeeks.org/push-relabel-algorithm-set-1-introduction-and-illustration/

[4] https://en.wikipedia.org/wiki/Ford–Fulkerson_algorithm#Integral_example

[5] https://en.wikipedia.org/wiki/Edmonds–Karp_algorithm

Advanced Algorithms
Research Topics

1. Honesty is profitable than Dishonesty. Designing a truth mechanism for allocating indivisible good without money

This topic was covered by Professor Hadi Hosseini in Rochester Institute of Technology where he talked about the how being honest would always be a profitable approach rather than being dishonest. This concept can be applied in different aspect of the real-world problem including the computer science world. For example, consider if a person knows about a particular algorithm that benefits himself/herself rather that other people around. Then that person will use the algorithm for his/her own benefits without being honest. This is a problem which can be faced in different algorithms for example in Gale-Shapely algorithm where if the preference list of men is already know, then it can be easily manipulated to assign a particular preference by a particular woman to get her best choice first. This will not be fair for other women and thus being dishonest will not be profitable at all with respect to all the women combined. So, professor explained in the topic that coming up with a system that can provide an algorithm such that if someone manipulate the algorithm, it will affect the person itself which would result in poor choice of manipulating the algorithm at the first place itself.

2. Does randomness solve all the problems?

This topic was covered by Professor Russell Impagliazo from the University of California San Diego where he talked about how introducing randomness can solve all the problems of different types. We know that there are two approaches to solve the problem which are either deterministic approach to solve a problem or randomized approach to solve a problem. For some set of problems, it is always better to provide certain randomization to the algorithm to get correct results. Also, some randomized algorithms will run much faster and efficiently than the deterministic algorithms. Professor also talked about people using randomized algorithm to give heuristic solutions to a given problem. Also, the algorithm covered in this course called Travelling Salesperson Problem covers this intuition that since this algorithm does not provide a polynomial time optimal solution, there is a need for randomization with certain approximation to provide a heuristic function with fissile solution. Also, professor mentioned that this area has a lot of improvement requires with lots of challenges still to be faced. But, there are lots of opportunity in this area after further improvements.

APPENDIX A

TOTAL HOURS: 153 HOURS

This time includes each presentation attendance, studying about the algorithms, and preparing report for each algorithm

1. Articulation Points, Bridges and Biconnected Graphs
   Presenter: Mounika Alluri
   Date: 6th September 2017
   Time Spent: 9 Hours

2. Kadane's algorithm
   Presenter: Rynah Rodrigues and Rinkesh Shah
   Date: 13th September 2017
   Time Spent: 7 Hours

3. Aho-corasick algorithm
   Presenter: Rynah Rodrigues, Rinkesh Shah
   Date: 20th September 2017
   Time Spent: 9 Hours

4. Knuth-Morris-Pratt algorithm
   Presenter: Jidnyasa Patil
   Date: 27th September 2017
   Time Spent: 7 Hours

5. Fast Fourier Transform algorithm
   Presenter: Geeta Madhav Gali
   Date: 11th October 2017
   Time Spent: 14 Hours

6. Top Trading Cycle
   Presenter: Sanjay Varma Rudraraju
   Date: 18th October 2017
   Time Spent: 7 Hours

7. Randomized Algorithms - Karger's Min Cut Problem
   Presenter: Rushik Vartak
   Date: 23rd October 2017
   Time Spent: 9 Hours

8. Hungarian Algorithm
   Presenter: Harnisha Gevaria
   Date: 25th October 2017
   Time Spent: 8 Hours

9. Sweep Line Algorithm
   Presenter: Abhishek Jaitley
   Date: 1st November 2017
   Time Spent: 8 Hours

10. Travelling Salesman Problem Using Approximation Algorithms
    Presenter: Kushal Gevaria
    Date: 8th November 2017
    Time Spent: 22 Hours

11. Strassen Algorithm
    Presenter: Prajakta Made
    Date: 15th November
    Time Spent: 11 Hours

12. Triangulation algorithm
    Presenter: Shashank Rudroju
    Date: 29th November
    Time Spent: 12 Hours

13. Red-Black Trees
    Presenter: Ketan Joshi
    Date: 4th December 2017
    Time Spent: 15 Hours

14. Online Algorithms
    Presenter: Vishal Kole
    Date: 6th December 2017
    Time Spent: 5 Hours

15. Advanced Network Flow Algorithms
    Presenter: Alexander Hoover
    Date: 11th December 2017
    Time Spent: 10 Hours