EVERY PAIR SHORTEST PATH USING BRANCH AND BOUND

Abhishek Vishwakarma
IIT2017091

Anuj Raj
IIT2017092

Group No.
43

***Abstract*—This paper introduces an algorithm to find the shortest path for every pair of vertices in a graph using branch and bound approach. Two algorithms have been proposed, using branch and bound , Brute force and dijkstra's agorithm. The proposed Algorithm to find shortest path for all source destination pair using branch and bound has factorial complexity.The proposed Dijkstra's algorithm ASAP improves the complexity of the branch and bound approach providing polynomial time solution with complexity $O(V^3)$.**

***Keywords*—** graph, shortest path, single-source shortest-path, SSSP, all-pairs shortest-path, ASAP, recursion, adjacency matrix, distance matrix, branch and bound strategy, cofactor, optimisation problem, visited list, vertices, edges, nodes, Dijkstra,

I. INTRODUCTION

The shortest-path problem is one of the well-studied topics in computer science, specically in graph theory.This is a problem of finding the shortest path or route from a starting point to a final destination. Generally, in order to represent the shortest path problem we use graphs. The majority of shortest-path algorithms fall into two broad categories. The rst category is single- source shortest-path (SSSP), where the objective is to nd the shortest-paths from a single-source vertex to all other vertices. The second category is all-pairs shortest-path (APSP), where the objective is to nd the shortest-paths between all pairs of vertices in a graph. This paper aims to solve the latter category.

II. LITERATURE SURVEY

Solving optimization problems to optimality is often an immense job requiring very efficient algorithms, and the Branch and bound paradigm is one of the main tools in construction of these. A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. It allows the algorithm to find parts of the solution space completely.

The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branching is to divide the search space to smaller sub problems. Then a branch is searched while maintaining a bound which is updated with increasing depth. There are two bounds that are maintained. The upper bound and the lower bound. The upper bound is the value that is to be optimized. Each time if a branch after searched till last that is upto the required the depth of search space. If during the traversal along a branch if the solution that is the lower bound exceeds the upper bound, that branch is terminated.

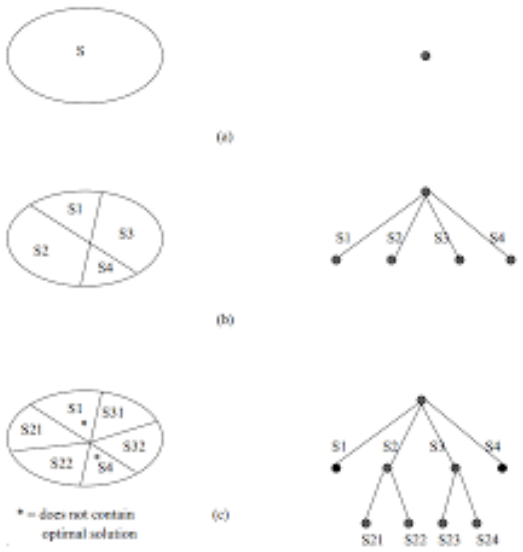

Fig. 1: Branch and Bound Approach

III. ALGORITHM DESIGN AND EXPLANATION

The graph $G(V, E)$ is represented by adjecency matrix. Where a_{ij} contains the weight or cost or distance between node i and j. Another matrix $Dist$ stores the final distance between any two vertices of the graph. Initially d_{ij} is set to INF value.

A. Branch and Bound Brute Force ASAP

There is two layer in this algorithm. First shortest path between source and destination is found. Then this is repeated for each pair of node in the graph. First goes in the following manner:
To find shortest path between source and destination pair we check every possible path from source to destination. Since it is known that cycle does not exist in the shortest path. Thus while travelling through a path there visited node is not visited again.
Travelling through a path the current distance till the current node from source is maintained. There are two sets of vertices visited and non visited. starting with the source node, it is set to visited with current distance set to zero. Each adjacent

node to the current node which is not visited is considered as a separate branch i.e. a separate path. for a particular branch with ith node as the next node, the current distance is updated as:

$currentDist+ = weight(currentNode, ithnode)$

If this currDistance is not less than the last shortest distance bound between source and destination then the branch will not give the shortest path. Finally if a branch has visited the destination and the currDistance is less than last best known shortest distance of the source destination pair then last solution is updated with current solution. when every branch is exhaustively searched the final shortest solution is attained. This algorithm is repeated for all source destination combination.

B. Dijkstra's Algorithm ASAP

Dijkstras algorithm is be used to find the shortest path from one node in a graph to every other node within the same graph data structure, provided that the nodes are reachable from the starting node.

This algorithm will run until all the vertices in the graph have been visited.This means shortest path between two nodes can be saved and looked up after.

In Dijkstra's algorithm, a starting node is given.A distance matrix is maintained to store the distance of nodes from the source vertex in each subsequent round and a visited array is used to check a node is visited or not. Initially, distance matrix is set to infinity and visited array to false.

Every time we decide to move to a new node,we will choose the node with the shortest known distance/cost to visit first.

Once we've moved to the node,we're going to visit,we check each of its neighbouring nodes.

For each neighbouring node, we calculate distance/cost for neighbouring nodes by summing the cost of the edges that lead to the node we're checking from the starting node.

Finally,if the distance/cost to a node is less than a known distance,we update the shortest distance that we have for that vertex.

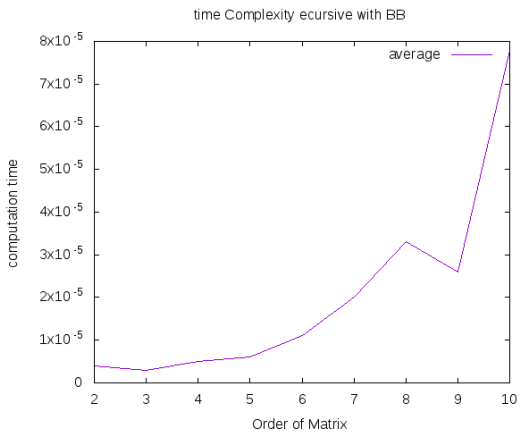

Fig. 2: Average case Branch and Bound Brute force ASAP

Algorithm 1 Branch and Bound(Recursive Algorithm)

procedure FINDDISTANCE($currPath, level, currDistance, src, des, size$)
  $bound \leftarrow dis[src][des]$
  if $currentNode == des$ **then**
    if $currDistance < bound$ **then**
      $dis[src][des] \leftarrow currDistance$
    $return$
  else
    for *each adj non visited node* **do**
      $currDistance \leftarrow currDistance + weight[currentNode][adjNode]$
      if $currDistance < bound$ **then**
        $vis[adjNode] \leftarrow true$
        $currPath[level-1] \leftarrow adjNode$
        $findDistance(currPath, level + 1, currDistance, src, des, size)$
  Prune the node by resetting the currDistance and visited node list\
procedure MAIN
  Store weights in adjacency matrix
  Apply findDistance to all pair of source and destination\

Algorithm 2 Dijkstra Algorithm

procedure DIJKSTRA($v, matrix[], alldist[], src$)
  $dist[] \leftarrow min\ dist\ of\ all\ vertices\ from\ source$
  $set[] \leftarrow visited\ array$
  $dist[] \leftarrow INT\_MAX$
  $set[] \leftarrow false$
  $dist[src] \leftarrow 0$
  for *count* **from** $0 \rightarrow v-1$ **do**
    $u \leftarrow index\ which\ is\ min\ and\ unvisited$
    $set[u] \leftarrow true$
    for I **from** $1 \rightarrow v$ **do**
      if $!set[I]\ and\ dist[u] + matrix[u][i] < dist[I]\ and\ matrix[u][I]\ and\ dist[u] != INT\_MAX$ **then**
        $dist[I] \leftarrow dist[u] + matrix[u][I]$
  for I **from** $1 \rightarrow v$ **do**
    $alldist[src][I] \leftarrow dist[I]$
procedure MAIN
  Store weights in adjacency matrix
  Apply dijkstra() to all vertices as source

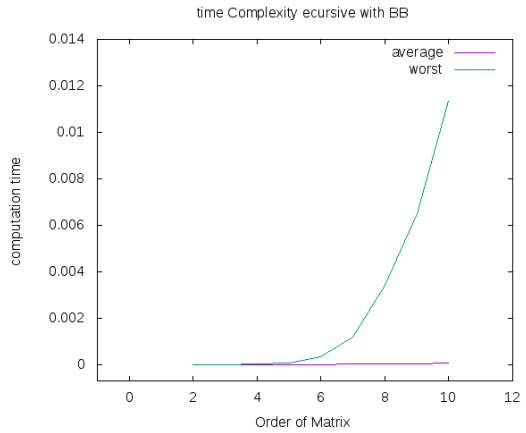

Fig. 3: comparison plot of Branch and Bound Brute force ASAP worst case and average case

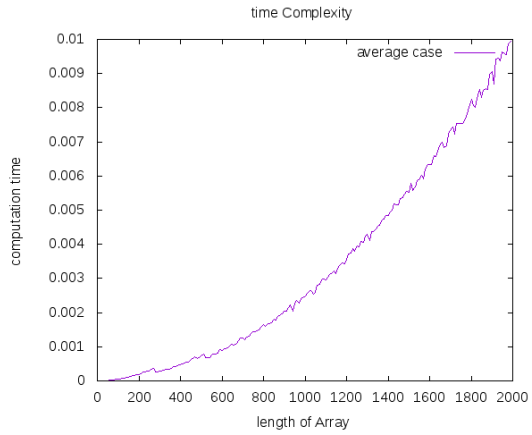

Fig. 4: comparison plot of single source shortest path (Dijkstra's algo)

IV. EXPERIMENTAL STUDY

For the experimental analysis of this algorithm, size of the adjacency matrix varied, that is number of nodes is varied to see how the execution time varies with number of nodes. The inputs are taken from the standard random generator in the range $1 < W < 20$. Where w_{ij} is the weight or cost of the edge between i and j. The graph of order vs execution time. is plotted using .dat file generated from the source code. see Fig 2 and Fig 3.

V. TIME COMPLEXITY AND DISCUSSION

A. Time Complexity

1) Branch and Bound Brute Force ASAP: For every pair of node out node V nodes, there exist

$$n(p) = 1 + \binom{V-2}{1}.1! + \binom{v-2}{2}.2! + ... + \binom{V-2}{V-2}.v - 2!$$

where $\binom{n}{r}$ is combination of r vertices out of n vertices.

total possible paths (if the graph is complete, which is the worst case). Thus total worst complexity is $O(V^2 * \sum_{k=0}^{V-2} \binom{V-2}{k}.k!)$

but since the bound is updated periodically since all possible path branch is not executed average running time is much faster as depicted in 3.

2) Dijkstra's Algorithm ASAP: Dijkstra's approach as discussed in the algorithm design section optimises the branch and bound by applying the greedy approach. The most general case of APSP is a graph with non-negative edge weights. In this case, Dijkstras algorithm can be computed separately for each vertex in the graph. The time complexity of Dijkstra's algorithm for a single source is be $O(V^2)$ This is the worst case complexity when the graph is complete. Best case will be the case when the there is no connected edge . Here V is the number of vertices in the graph. Thus for every source Dijkstra's will have the order $O(V^3)$

VI. CONCLUSION

The proposed Algorithm to find shortest path for all source destination pair using branch and bound has factorial complexity. Average cases experiment with sparse adjacency matrix shows that the branch and bound strategy is much faster than the hypothesis. The proposed Dijkstra's algorithm ASAP improves the complexity of the branch and bound approach providing polynomial time solution with complexity $O(V^3)$.

REFERENCES

[1] https://stackoverflow.com/questions/27624325/difference-between-branch-bound-extended-list-and-dijkstras-algorithm-on
[2] https://www.geeksforgeeks.org/branch-and-bound-algorithm/
[3] https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Appendix

April 22, 2019

Branch and Bound Brute force ASAP

```
#include<bits/stdc++.h>

using namespace std;
#define N 10
#define INF INT_MAX
int arr[N][N];
int dist[N][N];
int vis[N] = {0};
void add_edge(int u,int v, int w){
   arr[u][v] = w;
   arr[v][u] = w;
}

void findDistance(int currPath[], int level, int currWeight, int source,
    int des, int v) {
   if(currPath[level-1] == des) {
      if(currWeight < dist[source][des])
         dist[source][des] = currWeight;
      return;
   }
   else {
      for(int i = 0; i < v; i++) {
         if((currPath[level-1] != i && arr[currPath[level-1]][i] != INF)
             && vis[i] == 0 ) {
            if(currWeight < dist[source][des]) {
               vis[i] = 1;
               currPath[level] = i;
               currWeight += arr[currPath[level-1]][i];
               findDistance(currPath, level+1, currWeight, source, des,
                   v);
            }
            currWeight -= arr[currPath[level-1]][i];
             for(int k = 0;k < v;k++)
               vis[k] = 0;
            for (int j=0; j<=level-1; j++)
             vis[currPath[j]] = 1;
```

```
        }
      }
   }
}

void init(int source) {
   for(int i = 0; i < N; i++) {
      vis[i] = 0;
   }
   vis[source] = 1;
}

int main() {
   srand(time(NULL));
   ofstream file;
   file.open("graphBranchBoundWorst.dat");
   clock_t tic, toc;
   float tim;
   int t;
   cin >> t;
   for(int v = 2; v < t; v++){
      int v1, v2, e, eLimit, w;
      for(int i = 0; i < v; i++) {
         for( int j = 0; j < v; j++) {
            if(i != j) {
               arr[i][j] = INF;
               dist[i][j] = INF;
            }
            else if(i == j) {
               arr[i][j] = 0;
               dist[i][j] = 0;
            }
         }
      }
      e = (v*(v-1))/2;
      for(int i = 0; i < v; i++) {
         for(int j = 0; j < v; j++) {
            if(i != j) {
               w = rand()%20+1;
               add_edge(i, j, w);
            }
         }
      }
      int currPath[N];

      cout << "adjency Matrix : " << "vrtices : " << v << " edges : " <<
          e<< endl;
      for(int i = 0; i < v; i++) {
         for( int j = 0; j < v; j++) {
            if(arr[i][j] == INF) cout << setw(5) << "INF";
```

```
            else cout << setw(5) << arr[i][j];
         }
         cout << endl;
      }
      tic = clock();
      for(int source = 0; source < v; source++) {
         currPath[0] = source;
         for(int des = 0; des < v; des++) {
            init(source);
            findDistance(currPath, 1, 0, source, des, v);
         }
      }
      toc = clock();
      tim = toc - tic;
      tim = tim/CLOCKS_PER_SEC;
      //file << v << '\t' << tim << endl;
      cout << "distance Matrix : " << endl;
      for(int i = 0; i < v; i++) {
         for( int j = 0; j < v; j++) {
            if(arr[i][j] == INF) cout << setw(5) << "INF";
            else cout << setw(5) << arr[i][j];
         }
         cout << endl;
      }
   }
   file.close();
   return 0;
}
```

Dijkstra's Algorithm ASAP

```
#include<stdio.h>
#include<stdbool.h>
#include<limits.h>


void addMatrix(int v,int matrix[][v+1],int start,int end,int val)
{
   matrix[start][end]=val;
   matrix[end][start]=val;
}

int minDistance(int distance[],bool set[],int v)
{
   int min=INT_MAX,minIndex;

   for(int i=1;i<=v;i++)
   {
      if(set[i]==false && distance[i]<min)
      {
         minIndex=i;
         min=distance[i];
      }
   }

   return minIndex;
}

void dijsktra(int v,int matrix[][v+1],int alldist[][v+1],int src)
{
   int distance[v+1];
   bool set[v+1];

   for(int i=1;i<=v;i++)
   {
      distance[i]=INT_MAX;
      set[i]=false;
   }

   distance[src]=0;

   for(int count=0;count<v-1;count++)
   {
      int u=minDistance(distance,set,v);

      set[u]=true;

      for(int i=1;i<=v;i++)
      {
```

```
        if(!set[i] && matrix[u][i] && distance[u]!=INT_MAX &&
             distance[u]+matrix[u][i]<distance[i])
           distance[i]=distance[u]+matrix[u][i];
      }
   }

   for(int i=1;i<=v;i++)
      alldist[src][i]=distance[i];
}

int main()
{
   int v,e,start,end,val,t,a,b;
   scanf("%d%d",&v,&e);

   int matrix[v+1][v+1];
   int alldist[v+1][v+1];

   for(int i=1;i<=v;i++)
   {
      for(int j=1;j<=v;j++)
         matrix[i][j]=0;
   }

   while(e--)
   {
      scanf("%d%d%d",&start,&end,&val);
      addMatrix(v,matrix,start,end,val);
   }

   for(int i=1;i<=v;i++)
      dijsktra(v,matrix,alldist,i);

   for(int i=1;i<=v;i++){
      for(int j=1;j<=v;j++){
         printf("%d ",alldist[i][j]);
      }

      printf("\n");
   }

}
```

```
anuj@Jarvis ~ $ g++ graph.cpp
anuj@Jarvis ~ $ ./a.out
8
adjency Matrix : vertices : 2 edges : 1
    0    6
    6    0
distance Matrix :
    0    6
    6    0
adjency Matrix : vertices : 3 edges : 3
    0    2   14
    2    0    3
   14    3    0
distance Matrix :
    0    2   14
    2    0    3
   14    3    0
adjency Matrix : vertices : 4 edges : 6
    0    9    3    2
    9    0    3   14
    3    3    0   19
    2   14   19    0
distance Matrix :
    0    9    3    2
```

input size limit of vertices : 8

```
distance Matrix :
    0    8   19   19   11   18
    8    0    7    4    4   17
   19    7    0   19    7    9
   19    4   19    0    7    2
   11    4    7    7    0    1
   18   17    9    2    1    0
adjency Matrix : vertices : 7 edges : 21
    0    5    4   13   14    3    6
    5    0    9    1    6   19   13
    4    9    0    2   17    4   12
   13    1    2    0    2   18    6
   14    6   17    2    0   16   10
    3   19    4   18   16    0   13
    6   13   12    6   10   13    0
distance Matrix :
    0    5    4   13   14    3    6
    5    0    9    1    6   19   13
    4    9    0    2   17    4   12
   13    1    2    0    2   18    6
   14    6   17    2    0   16   10
    3   19    4   18   16    0   13
    6   13   12    6   10   13    0
anuj@Jarvis ~ $
```

second screenshot