Important Algorithms for IOI

Nebhrajani A.V.

December 3, 2020

Contents

1 Introduction

Most problems can be reduced to either one of or a combination of the algorithms presented in this notebook. While not meant to be as comprehensive as a set of notes on CLRS, it includes the more challenging and important programs. Sorters and searchers (with the exception of binary search, which I struggled with early on) are omitted since they're basic and C++'s STL implements them well enough. This notebook focuses majorly on graph algorithms, dynamic programming, greedy programs, and trees. It assumes familiarity with C/C++ and basic programming techniques such as arrays/vectors, loops, lists, iteration, and recursion.
Note that it is bad practice to use the header `bits/stdc++.h`: it is **not** a standard header, and increases compile time by including *every* C++ library, not only the required ones. However, it's okay in competitive programming since the programs are small and runtime matters more than compile time. Obviously, if compilation takes too long, the bottleneck is probably this header.

2 Searchers

2.1 Binary Search

```cpp
// To search a sorted array for a value.
#include <bits/stdc++.h>

int binary_search(std::vector<int>& a, int k, int l, int r)
{
  if (r >= l) {
    int mid = (l+r)/2;
    if (k == a[mid]) {
      return mid;
    }
    if (a[mid] < k) {
      return binary_search(a, k, mid+1, r);
    }
    if (a[mid] > k) {
      return binary_search(a, k, l, mid-1);
    }
  }
  return -1;
}

int main()
{
  std::vector<int> arr = {34,54,23,35,13,35,46,678,34,576,46,34,646};
  std::sort(std::begin(arr), std::end(arr));
  int find = binary_search(arr, 46, 0, arr.size());
  std::cout << arr[find] << "\n";
  return 0;
}
```

Complexity: $O(\log_2 n)$ A basic algorithm, with a lot of similarity to QuickSort. r and l are the indexes of the subarray to search recursively. If $r < l$ directly return -1, the element wasn't found. If not, see if the current *mid* is the element we're looking for. If so, return *mid*. Otherwise, recurse on either the LHS or RHS subarray.

3 Graphs

Graphs can be represented as adjacency matrices and adjacency lists. Most algorithms use adjacency lists. Always write a function returning `void` called `add_edge` that adds an edge. It can be modified depending on whether the graph is directed and weighted.

3.1 Graph Searchers

3.1.1 Breath First Search (BFS)

```cpp
// BFS to check if two nodes are connected, and can calculate shortest path
// where edge weights are 1, using the 'parent' vector and backtracking how we
// got there.
#include <bits/stdc++.h>

bool bfs(std::vector<std::vector<int>>& g, int src, int dest)
{
  std::vector<int> visited (g.size());
  // std::vector<int> parent (g.size());
  std::deque<int> q;

  visited[src] = 1;
  q.push_back(src);

  while (q.size() != 0) {
    int j = q[0];
    for (int i = 0; i < g[j].size(); ++i) {
      if (visited[g[j][i]] == 0) {
        visited[g[j][i]] = 1;
        // parent[g[j][i]] = j;
        q.push_back(g[j][i]);
      }
    }
    q.pop_front();
  }
  // for (auto x: parent) {
  //       std::cout << x << "\n";
  // }


  if (visited[dest] == 1) {
    return true;
  }
  else {
    return false;
  }
}

void add_edge(std::vector<std::vector<int>>& g, int a, int b)
{
  g[a].push_back(b);
  g[b].push_back(a);
}

```

```cpp
int main()
{
  int V = 10;
  std::vector<std::vector<int>> g (V);
  add_edge(g,0,1);
  add_edge(g,0,2);
  add_edge(g,1,2);
  add_edge(g,0,3);
  add_edge(g,3,4);
  add_edge(g,3,7);
  add_edge(g,4,5);
  add_edge(g,5,7);
  add_edge(g,5,6);
  add_edge(g,5,8);
  add_edge(g,7,8);
  add_edge(g,8,9);

  if (bfs(g,0,9)) {
    std::cout << "True" << "\n";
  }
  else {
    std::cout << "False" << "\n";
  }

  return 0;
}
```

Complexity: $O(m + n)$ Adjacency list representation. Undirected and unweighted in this case. Usual declaration using `std::vector<std::vector<int>>` over `std::vector<std::list<int>>` since vectors are faster with end appends. Add the edges.
The procedure `bfs` maintains a `deque q`. Start at the `src` node. Visit all of its neighbours who are unvisited. (For `src`, this is by definition *all* its neighbours.) Push them into `q` to visit later. Pop the `q` and push the unvisited neighbours of the node we popped. Keep doing this. At some point, all possible nodes visitable from `src` will have been visited, and `q` will be empty. We keep track of visited nodes using a standard vector `visited`.
Note that if we maintain a `parent` vector, that keeps track of *from whom* we visited the current node, we can backtrace the vector to get the path from `src` to `dest`. This is by definition the shortest path, since BFS doesn't go out of its way to explore longer paths: it follows the most direct (least edges) route from `src` to `dest`. This means BFS is a shortest path finder for unweighted graphs (all edge weights = 1).

3.1.2 Depth First Search (DFS)

```cpp
// DFS: the 'stack' is maintained as recursive function calls implicitly.

#include <bits/stdc++.h>

void dfs(std::vector<std::vector<int>>& g, std::vector<int>& visited,
↪  std::vector<int>& parent, int src)
{
  visited[src] = 1;
  for (int i = 0; i < g[src].size(); ++i) {
    if (visited[g[src][i]] == 0) {
      parent[g[src][i]] = src;
      dfs(g, visited, parent, g[src][i]);
    }
  }
}

void add_edge(std::vector<std::vector<int>>& g, int a, int b)
{
  g[a].push_back(b);
  g[b].push_back(a);
}

int main()
{
  int V = 10;
  std::vector<std::vector<int>> g (V);
  std::vector<int> visited (V);
  std::vector<int> parent (V);

  add_edge(g,0,1);
  add_edge(g,0,2);
  add_edge(g,1,2);
  add_edge(g,0,3);
  add_edge(g,3,4);
  add_edge(g,3,7);
  add_edge(g,4,5);
  add_edge(g,5,7);
  add_edge(g,5,6);
  add_edge(g,5,8);
  add_edge(g,7,8);
  add_edge(g,8,9);

  dfs(g,visited,parent,0);
  std::cout << visited[9] << "\n";
  return 0;
}
```

Complexity: $O(m + n)$ DFS maintains a stack. It goes as far along a random path as possible, pushing the node it left from into a stack. As soon as it reaches the end of a path (i.e., it cannot visit an unvisited node), it pops out and tries the same from the previous node it left. This stack can be maintained implicitly by recursive calls to DFS, as shown in the program.
Note that DFS **does not** cover shortest paths, but it can be used for node numbering: by recording the 'time' DFS entered a node (started recursively calling its neighbours) and left it (exited out of its last remaining neighbour and by extension, the node in question).

3.1.3 Applications of Graph Searchers

1. Connectivity: Some parts of a graph may not be connected to other parts of the graph, creating sets of connected nodes. To figure out what exactly these sets *are*, run either search from a `src` (say 0). Observe the resultant `visited` array. The nodes that have been visited are from the first set. Note them down if you need to. Now run a search from an unvisited node. The additional nodes visited are the second set. Repeat until all nodes are visited, and you'll know the connectivity sets of the graph.

2. **TODO** Catching cycles: Using DFS Numbering More advanced concept, catches cycles in graphs using the numbers DFS assigns as it visits nodes.

3.2 Shortest Paths

3.2.1 Single Source: Djikstra's Algorithm

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

struct node
{
  int dest;
  int weight;
};

void add_edge(std::vector<std::vector<node>>& g, int src, int dest, int
↪  weight)
{
  g[src].push_back({dest, weight});
  g[dest].push_back({src, weight});
}

void djikstra(std::vector<std::vector<node>>& g, std::vector<int>& distance,
↪  int src)
{
  std::vector<bool> visited (g.size());
  distance[src] = 0;

  for (int i = 0; i < g.size(); ++i) {
    int min = std::numeric_limits<int>::max();
    int u;
    for (int j = 0; j < visited.size(); ++j) {
      if (!visited[j]) {
        if (distance[j] < min) {
          min = distance[j];
          u = j;
        }
      }
    }
    visited[u] = true;
    for (int k = 0; k < g[u].size(); ++k) {
      if (!visited[g[u][k].dest] && distance[g[u][k].dest] > distance[u] +
      ↪  g[u][k].weight) {
        distance[g[u][k].dest] = distance[u] + g[u][k].weight;
      }
    }
  }
}

int main()
{
  int V = 7;
  std::vector<std::vector<node>> g (V);
  add_edge(g,0,2,80);
  add_edge(g,0,1,10);
  add_edge(g,1,2,6);
  add_edge(g,2,3,70);
  add_edge(g,1,4,20);
  add_edge(g,4,6,10);
  add_edge(g,4,5,50);
  add_edge(g,5,6,5);
  std::vector<int> distance (V, std::numeric_limits<int>::max());

  djikstra(g, distance, 0);
  for (int x: distance) {
    std::cout << x << "\n";
  }

  return 0;
}
```