README.md

GSP-Recognition

A c++/boost implementation of the paper "A linear-time certifying algorithm for recognizing series-parallel graphs and outerplanar graphs" by Francis Y.L. Chin, Hing-Fung Ting, Yung H. Tsin, and Yong Zhang.

Navigation

Dependencies

The following dependencies are needed to build the program

- boost
- cmake
- c++17 compiler (gcc 8+ or clang 7+)

The following commands can be used on a fedora based distribution. Other distributions should have a similar command

```
sudo dnf install gcc boost cmake
```

Alternatively the following should be used for installing clang instead of gcc

```
sudo dnf install clang boost cmake
```

Build

The project uses cmake to build the program. Cmake will try and find the dependencies and automatically link them. It creates a makefile which is used to compile and build the program. After installing the dependencies, the following can be used to build the software on a linux system

1. Clone the repo to obtain the project

    ```
    git clone git@github.com:jdao55/GSP-Recognition.git
    ```

2. Enter project directory

    ```
    cd GSP-Recognition
    ```

3. Create build folder

    ```
    mkdir build && cd build
    ```

4. Run Cmake

```
cmake -DCMAKE_BUILD_TYPE=Release ../
```

or using following for debug

```
cmake -DCMAKE_BUILD_TYPE=Debug ../
```

5. Compile project

```
make
```

or use following command to compile with more thread/jobs (faster compiler time)

```
make -j[n]
```

n= number or jobs/threads ex

```
make -j4
```

Note if no new targets are added, cmake does need to be run again, the follow can be used to compile the code, every subsequent time.

```
make -j4
```

Here is the cmake documentation for further reading on how to use cmake https://cmake.org/cmake/help/latest/guide/tutorial/index.html

Running the program

The program can be run by entering the following command in the terminal
`gspgraph input_filename`
ex.
`./gspgraph ../examples/4SP.txt`

Input file format

The input file represents a graph. The format consits of the number of vertices followed by the edegs in the graph

```
number_of_vertices
v1 v2
v2 v3
.
.
.
```

Ex.

```
4
0 2
1 3
2 1
```

Please see example folder for more examples of the input files

Implementaion details

The following section will be used to docement all the files, classes, and fucntion used the the project. The sections are divided by file name.

AdjacencyList.hpp

struct Node

Constructors and destructors (defaults are ommitted)

```
~Node()
```

custom destructor, updates adjacency list on deletion of node This represents a node in a doubly linked list, used in the AdjacencyList

Members types

- `Node* next` pointer to the next node in list
- `Node* next` pointer to the previous node in list
- `int vertexNumber` vertex number which the node represents
- `Node* partner` pointer the the nodes' partner node

struct AdjacencyList

AdjacencyList structure is used to represent the edges in the graph. Implemented as a doubly linked list(see node structure)

Member types

- `Node *head` Pointer to head(start) of list
- `Node *tail` Pointer to end of list

Procedures

```
void add_back(Node* newNode)
```

- **Summary** Adds edge to the back of the list
- **Parameters**
  - `Node *newNode` New node too be added

- **Return** returns void

```
void add_front(Node* newNode)
```

- **Summary** Adds edge to the front of the list
- **Parameter**
  - `Node *newNode` New node too be added
- **Return** returns void

```
void clear()
```

- **Summary** Deletes Adjacency list
- **Return** returns void

```
void isEmpty()
```

- **Summary** Checks if list is empty
- **Return** True if list is empty false if list is not empty

```
void print()
```

- **Summary** Prints list to stdout
- **Return** void

Certificate.hpp

struct Certificate

Certificate for the algorithm. Contains Contruction tree, outer boundars for positive certificates. And 3in1, 1in3, k23 or k4 for negative certificate. The main algorithm returns this struct as the result.

Member types

- `int type` represents type of the cert see static const members
- `static const int VERTEX_IN_THREE_BICONNECTED =1, THREE_VERTEX_IN_BICONNECTED = 2, K4GRAPH = 3, K23GRAPH = 4, SP_AND_OP = 0, THETHA_GRAPH = 5,` defines the type of certificate (Note this should be replace with enum)
- `int single` index of cut_point in 3 components or, component with 3+ cut points
- `list<int> others` used to store cut_point in 3 components or, component with 3+ cut points structure
- `int ignore` used in OP cert
- `Tree_ptr constructionTree` Construction sequence for sp cert
- `std::array<vector<pair<int,int>>, 6> certPaths` used to stored k23 or k3 graph
- `std::array<vector<pair<int,int>>, 6> thetha_paths` used to store thetha4 graph
- `bool SP` set to true is graph is sp
- `bool OP` set to true is graph is op
- `bool GSP` set to true is graph is gsp
- `vector<pair<int,int>> boundary` list of edges representing the outer boundary of the graph

Constructors and destructors (defaults(implicit) are omitted)

- `Certificate()` Creates certificate, and initialized default values

Procedures

```
void clear()
```

- **Summary** Clears the type and 'single' variables of the certificate
- **Return** void

```
string print()
```

- **Summary** return string representation of the certificate. The string contains wether the graph is GSP, SP and OP, and Prints the certificate
- **Return** Returns `std::string` representation of graph

```
inline void combinePaths(vector<pair<int,int>> &a, vector<pair<int,int>> &b )
```

- **Summary** Combines paths `a` and `b` into path a, ie a = a + b
- **Parameters**
  - `vector<pair<int,int>> &a` A path in the graph, path a, and path b will be combined into path a
  - `vector<pair<int,int>> &b` A path in the graph
- **Return** void

```
inline bool confirmCert(Graph& full)
```

- **Summary** Certifies the certificate with the corresponding graph
- **Parameters**
  - `Graph& full` The corresponding graph to the certificate
- **Return** True if certificate is valid returns false otherwise

SPandOP.cpp & SPandOP.hpp

functions

```
void run (Graph &g,
          Certificate &cert,
          int source, int sink,
          std::map<int, Tree_ptr>& cut_point_tree_map)
```

- **Sumary** Runs GenCS on a given graph, user can set the value for the source and sink. Overall implementation is the following. The source and sink is setup, GenCS is called with the graph. Then the certificate for the graph is created and returned.
- **Parameters**
  - `Graph &g` The graph(or subgraph/biconnceted component) to run GenCS with
  - `Certificate &cert` The corresponding certificate for the graph g (note this can and probably should be returned instead of being an output parameter, since its best practice not to have output parameters)
  - `int source` sets the source for the SP construction when running GenCS
  - `int sink` sets the sink for the SP construction when running GenCS
  - `std::map<int, Tree_ptr>& cut_point_tree_map)` map holds contruction trees which are attached to the cut points

Sequence.hpp

typedefs

- `using Sequence_ptr = std::shared_ptr<Sequence>` Sequence smart pointer

struct Sequence

This classes implements the sequence used in the SP algorithm see page 11 in paper for futher details and definintions

Constructors and destructors (implicit are ommitted)

```
Sequence()
```

default constructor

```
Sequence(int source,
    int sink,
    string composition,
    Node* sourceNode = nullptr,
    Node* sinkNode = nullptr,
    string direction = "")
```

parameters

- `int source` source vertex for sequence
- `int sink` sink vertex for sequence
- `string composition` type of composition between left and right sub trees
- `Node* sourceNode = nullptr` corresponding source node
- `Node* sinkNode = nullptr` corresponding sink node
- `string direction = ""` identify type for edges (tree edge or back-edge)

member types

- `int s` vertex number of source node
- `int t` vertex number of sink node
- `Node *s_node` corresponding node for s
- `Node *t_node` corresponding node for t
- `Sequence_ptr l` pointer to left subtree
- `Sequence_ptr r` pointer to right subtree
- `string comp` string for composition of the Sequence ie SP, PC orDC (should be change to enum for best practice)
- `string orientation` identifies if edges is either forward edge or back edge (should be change to enum for best practice)

Procedures

```
void clear()
```

- **Sumary** reset sequence to defalut values
- **Return** void

StackNode.hpp

typedefs

- `using Tree_ptr = std::shared_ptr<Tree>;` Tree smart pointer

struct StackNode

node used for the stack used in the SP algorithm see page XX for definition and further details

member types

- `int end` end vertex
- `Tree_ptr sp` sp sequence
- `Tree_ptr tail` tail sequence

Constructors and destructors (implicit are omitted)

```
StackNode()
```

default constructor

Tree.hpp

typedefs

- `using Sequence_ptr = std::shared_ptr<Sequence>` sequence shared smart pointer
- `using Tree_ptr = std::shared_ptr<Tree>` tree shared pointer

struct Tree

Decomposition tree defined on page 11-12 used to represent the SP construction of the graph

member types

- `Sequence_ptr root` pointer to the root of the tree, Sequence type is used to implement the decomposition tree

Constructors and destructors (implicit are omitted)

```
Tree()
```

- default constructor, creates new instance of root

```
Tree(int s, int t,
    std::string comp,
    Node* sourceNode = nullptr,
    Node* sinkNode = nullptr,
    std::string direction = "")
```

- **parameters**
  - `int s` source vertex of the root
  - `int t` sink vertex of the root

- `std::string comp,` set comp for root
- `Node* sourceNode = nullptr` source node of the root
- `Node* sinkNode = nullptr` sink node of the root
- `std::string direction = ""` direction of the root sequence

procedures

```
void print_tree()
void print_tree(Sequence_ptr leaf)
```

- **Summary** prints the tree to stdout recursively, call function with no parameters to print entire tree
- **parameters**
  - `Sequence_ptr leaf` prints tree with leaf as the root
- **return** void git rebase --continue

```
string get_tree()
string get_tree(Sequence_ptr leaf)
```

- **Summary** returns string representation of the tree recursively, call function with no parameters for entire tree
- **parameters**
  - `Sequence_ptr leaf` returns string of tree with leaf as the root
- __return__git rebase --continue String representation of the tree

```
inline Tree_ptr SC(Tree_ptr left, Tree_ptr right)
```

- **Summary** creates the series composition between left, and right trees
- **parameters**
  - `Tree_ptr left` this will become the left subtree in the composition
  - `Tree_ptr right` this will become the right subtree in the composition
- **return**
  - Tree_ptr which is the series composition between the left and right trees

```
inline Tree_ptr PC(Tree_ptr left, Tree_ptr right)
```

- **Summary** creates the parallel composition between left, and right trees
- **parameters**
  - `Tree_ptr left` this will become the left subtree in the composition
  - `Tree_ptr right` this will become the right subtree in the composition
- **return** Tree_ptr which is the parallel composition between the left and right trees

```
inline Tree_ptr DC(Tree_ptr left, Tree_ptr right)
```

- **Summary** creates the dangling composition between left, and right trees
- **parameters**
  - `Tree_ptr left` this will become the left subtree in the composition
  - `Tree_ptr right` this will become the right subtree in the composition
- **return** Tree_ptr which is the dangling composition between the left and right trees

Vertex.hpp

typedefs

- `using Tree_ptr = std::shared_ptr<Tree>` tree shared pointer

Struct Vertex

Used to represent a vertex in the custom Graph data structure

Member types

- `int dfsNumber` number assign to vertex, is in order with the dfs visits the vertex
- `int parent` parent of the vertex
- `int numberOfChildren` number of children of the vertex in the DFS
- `std::list<StackNode> stk` stack of parallel ears, to be reduced when dfs returns to the current vertex
- `Tree_ptr ear` ear of the parent edge that the vertex belongs to ie Pear(parent->this)
- `bool parentEdge` set to false after initial backtrack to parent vertex
- `bool balert` boundary error alert
- `Tree_ptr b` used to memorize the B path when test K23
- `int k4visit` used to validate negative cercificate
- `int k23visit` used to validate negative cercificate
- `AdjacencyList adjacencyList` the list of adjacent vertices

Constructors and destructors (implicit are ommitted)

```
Vertex()
```

- default constructor

Graph.hpp

Struct Graph

Customs graph structure use for implementing algorithm, implemented as adjacency list. Class also contains GenCS implementation

Member types

- `int numV` number of vertices in graph
- `std::vector<Vertex> vertice` list of vertices in graph, this is also the adjacency list implementation
- `int dfsCount` current dfs count, used for GenCS
- `int r = -1` index of root vertex
- `Node* r_node` root vertex node
- `Tree_ptr b` used to find k23 graph see page 15 for further info
- `map<int, int> relationToMain, returnToMain` legacy variables (can be removed?)
- `unordered_map<int, bool>isCutPoint` maps from (vertex index -> bool). maps to true is vertex is a cutpoint
- `int forcedSource = 1, forcedSink =0;` indices of forced source and sink
- `bool addedForcedEdge = false;` used to indentofy if we force a certain source and sink well calling genCS
- `vector<pair<int,int>> certPaths[6]` used to store k4 or k23 graphs

- `int endpoints[12]` endpoints of the paths in the k4 or k23 graph (p1,p1, p2,p2 p3,p3 ... p6,p6);

Procedures

```
void addEdge(int w, int v)
```

- **Summary** adds edge(w,v) to the graph
- **Parameters**
  - `int w` source vertex
  - `int v` sink vertex
- **Return** void

```
bool earCompare(Tree_ptr l, Tree_ptr r)
```

- **Summary** Implements < for ears, the definition of this is found on page 8
- **Parameters**
  - `Tree_ptr l` left hand side of the < operator
  - `Tree_ptr r` right hand side of the < operator
- **Return** true if l < r, false otherwise

```
void GenCS(Node* w_node,
    Node* v_node,
    Tree_ptr &seq,
    vector<pair<int,int>>& boundary,
    bool& k23found,
    std::map<int, Tree_ptr>& cut_point_tree_map);
```

- **Summary** Implementation of GenCS found on page 14, find the SP compositions of the graph, also returns k4 and k23 for negative certificates.
- **Parameters**
  - `Node* w_node` w vertex node
  - `Node* v_node` v vertex node
  - `Tree_ptr &seq` sequence seq, the current SP composition the graph, the creates construction tree here when function terminates
  - `vector<pair<int,int>>& boundary` the OP boundary of the graph, after terminated the op boundary is returned here
  - `bool& k23found` k23 flag, set to true if a k23 graph has been found
  - `std::map<int, Tree_ptr>& cut_point_tree_map)` map holds contruction trees attach to the cut points
- **Return** void

```
void update_seq(int w, Tree_ptr &seq)
```

- **Summary** update seq by merging all subgraphs on stack(w) with seq. See page 15 for definition. Note k4 graph generating is also done here
- **Parameters**
  - `int w` the vertex stack to be merge with seq
  - `Tree_ptr &seq` sequence seq

- **Return** void

```
void update_ear_of_parent(Tree_ptr f,
    Tree_ptr &seq_u,
    int w, int v,
    Tree_ptr &seq,
    int &s_,
    bool& k23found)
```

- **Summary** updates ear of the parent vertex of w, vertex v. see page 15 for definition. Note k4 graph generating is also done here
- **Parameters**
  - `Tree_ptr f` backedge or tree edge u->v
  - `Tree_ptr &seq_u` sequence for node u
  - `int w` vertex w
  - `int v` vertex v
  - `int &s_` int s_w see definition on page 14
  - `bool& k23found` k23 flag, set to true if a k23 graph has been found
- **Return** void

```
bool isValidCertificate(Tree_ptr cert);
bool isValidCertificateFull(Tree_ptr cert);
```

- **Summary** Checks if construction tree is valid. Full version is to be used on graph without a DFS performed(GenCS has not been called).
- **Parameters**
  - `Tree_ptr cert` construction tree to be checked
- **Return** true if valid false otherwise

```
void postOrderReduction(Sequence_ptr seq);
void postOrderReductionFull(Sequence_ptr seq);
```

- **Summary** Recursive function used by isValidCertificateFull to verify the tree.
- **Parameters**
  - `Sequence_ptr seq` sequence to be checked
- **Return** true if valid, false otherwise

```
bool is_ancestor(const int child, const int ans)
```

- **Summary** Check if ans is an ancestor of child
- **Parameters**
  - `const int child,` child vertex
  - `const int ans` ancestor vertex
- **Return** true if ans is an ancestor of child, false otherwise

```
bool compare_backedge(Sequence_ptr lhs, Sequence_ptr rhs );
```

- **Summary** compares 2 backedges using < ie lhs<rhs? see page 7 for definition
- **Parameters**

    - `Sequence_ptr lhs` left hand side of operator
    - `Sequence_ptr rhs` right hand side of operator
- **Return** true if lhs<rhs, false otherwise

```
void K23Test(int e, int v, bool& balert, bool& k23found) see page 15 for def
```

- **Summary** test if a k23 graph was found
- **Parameters**
    - `int e` e vertex
    - `int v` v vertex
    - `bool& balert` b.alert
    - `bool& k23found` functions sets this bool to true if a k23 is found
- **Return** void

```
bool Check_Path(int s, int t, int path, int *endpoints,int flag)
bool Check_Path_reverse(int s, int t, int path, int *endpoints,int flag)
```

- **Summary** checks if tree path from s to t exists, and adds the path to certPaths[path], reverse checks in reverse direction. only used in finding paths for k4 and k23 graphs
- **Parameters**
    - `int s` start vertex
    - `int t` end vertex
    - `int path` add path point to certPaths[path],
    - `int *endpoints` endpoints for the k4/k23 subgraph
    - `int flag` specify the type of graph the part is being used for either 4=k4 or 23=k23
- **Return** true path exist false otherwise

```
 bool Check_Ear(Sequence_ptr a, int path, int *endpoints,int flag)
```

- **Summary** Check path of ear, added path to certPaths[path]. only used in finding paths for k4 and k23 graphs
- **Parameters**
    - `Sequence_ptr a` the ear to be checked
    - `int path` specify which certpath to add the ear too
    - `int *endpoints` endpoints for the k4/k23 subgraph
    - `int flag` specify the type of graph the part is being used for either 4=k4 or 23=k23
- **Return** true ear is valid false otherwise

```
bool Check_Edge(int s, int t, int path,int *endpoints,int flag)
```

- **Summary** checks if edge (s,t) exists, and adds the path to certPaths[path], only used in finding paths for k4 and k23 graphs
- **Parameters**
    - `int s` start vertex
    - `int t` end vertex
    - `int path` add path point to certPaths[path],
    - `int *endpoints` endpoints for the k4/k23 subgraph
    - `int flag` specify the type of graph the part is being used for either 4=k4 or 23=k23
- **Return** true path exist false otherwise

```
    bool Check_Edge_remove(int s, int t);
    bool Verify_Edge(int s, int t);
```

- **Summary** check if edge exits, Check_Edge_remove will as remove edge if it exist
- **Parameters**
  - `int s` source vertex
  - `int t` sink vertex
- **Return** true if vertex was found, false otherwise

```
map<int ,int> Verify_K4_Structure(int  *endpoints);
map<int ,int> Verify_K23_Structure(int endpoints [12])
```

- **Summary** checks if k4/k23 structure is valid
- **Parameters**
  - `int *endpoints` the endpoints of the subgraph
- **Return** map<int,int> map form (endpoint -> number of paths that terminate here) will throw error is it is not valid(should change this to return bool or std::variant)

```
void setupSourceAndSink(int source, int sink)
```

- **Summary** sets up source and sink for GenCS
- **Parameters**
  - `int s` source vertex
  - `int t` sink vertex
- **Return** void

```
bool checkTwoVertexStructure()
```

- **Summary** checks if this graph consists of only 2 vertexes and one edge - in other words it is a bridge
- **Return** true if graph is a bridge, false otherwise

```
template <typename T>
bool Check_Ear(Sequence_ptr a, int path, int *endpoints, int flag, T &container);
```

- **Summary** similar to other check_ear, but the path is also added to T &container)
- **Parameters**
  - `Sequence_ptr a` the ear to be checked
  - `int path` specify which certpath to add the ear too
  - `int *endpoints` endpoints for hte k4/k23 subgraph
  - `int flag` specify the type of graph the part is being used for either 4=k4 or 23=k23
  - `T &container` generic container which the path is added to, must implement push_back. Currently the only Type used is std::vector
- **Return** void

gsp.hpp & gsp.cpp

Type defs

```
typedef boost::adjacency_list<
    boost::vecS, boost::vecS, boost::undirectedS, boost::no_property,
    boost::property<boost::edge_component_t, std::size_t>>
    graph_t;
typedef boost::graph_traits<graph_t>::vertex_descriptor vertex_t;
typedef boost::graph_traits<graph_t>::adjacency_iterator AdjacencyIterator;
using Tree_ptr = std::shared_ptr<Tree>;
```

Fucntions

```
template <typename T, size_t size>
bool contains_pair(const std::array<std::vector<std::pair<T,T>>, size> &vec, const T s, const T t)
```

- **Summary** utility fucntion checks if pair (s, t) or pair (t,s) is one of the vectors in the array of vectors
- **Parameters**
  - `const std::array<std::vector<std::pair<T,T>>, size>` the array of vectors which the function searchs for the pair
  - `int s` one element in query pair
  - `int t` one element in query pair
- **Return** true if pair exists, false otherwise

```
template <typename T, size_t size>
void remove_pair(std::array<std::vector<std::pair<T,T>>, size> &vec, const T s, const T t)
```

- **Summary** utility fucntion remove instances of pair (s, t) or pair (t,s) in all vectors in the array of vectors
- **Parameters**
  - `const std::array<std::vector<std::pair<T,T>>, size>` the array of vectors which the function searchs for the pair
  - `int s` one element in query pair
  - `int t` one element in query pair
- **Return** void

```
void update_graphs_found(const Certificate &cert, bool & k4, bool & k23) {
```

- **Summary** updates bools k4 and k23 if those graphs were found in the certificate
- **Parameters**
  - `const Certificate &cert` certifciate which is use to update the bool values
  - `bool & k4` set to true if k4 is found
  - `bool & k23` set to true if k23 is found
- **Return** void

```
auto sort_compoentents( map<int, list<int>> &containsCuts,
                        map<int, list<int>> &cutPoints,
                        const int first,
                        const int last,
                        const size_t num_comps)
```

- **Summary** sorts the biconnected componets and corresponding cut vertices, such that comp n and n+1 share a cut vertex. Note the sorting returns the correct order of the indices of the components, rather than doing an inplace sort ie {third,first, second} returns indices {1,2,0}
- **Parameters**
  - `map<int, list<int>> &containsCuts` map from (biconnected comp -> list of cut vertices). maps comp to the list of cutpoints

inside it
    - `map<int, list<int>> &cutPoints` map from (vertex ->list of biconnected comps) maps cut vertex, to list of comps that contain it
    - `const int first` index of first comp
    - `const int last` index of last comp
    - `const size_t num_comps` number of components
- **Return** `std::pair<vector<int>, vector<int>>` where
    - first = sorted indices of components
    - second = sorted indices of cut points

```
Certificate GSP_verify(graph_t &g2, const size_t size)
```

- **Summary** Implements GSP portion of algorithm on page 32. First section finds the biconnected comps, and does the general setup for alogirthm. It then follows the pseudo code from paper. See comments for more detailed implementation details
- **Parameters**
    - `graph_t &g2` Boost graph object, it verifies if graph g2 is SP, OP and GSP
    - `size_t size` number of vertices in graph g2
- **Return** Certificate which verifies if the output of alogirthm

Program Design and Flow

This summary provides the rational for features in the implementation that are not obvious from looking at the algorithm. Comments in the source code provide descriptions of the function and usage of the more obvious implementation details.

The program takes as input any connected boost graph and determines if it is series-parallel or outerplanar or not. To certify that the program has made the correct determination it generates a construction sequence certificate for the series-parallel construction of the graph and a boundary edge list certificate for evaluation of outerplanar graphs. The program also does the validation of the series-parallel certificate. Validation of the outerplanar certificate is left for future work, as well as fixing the k23 generation.

To accomplish its tasks the program first decomposes the graph into its various biconnected components. It then uses hashmaps and lists to find if there are any vertexes in more then 2 biconnected components, or more then 2 articulation vertexes in any biconnected components. In the event of either of these conditions we return a generated certificate for this case as it can not be SP.

If the graph can still be SP, we use the hashmaps and lists to order(sort) the biconnected components so that one biconnected component leads into the next one.

The program then prepares to use the fundamental area of the algorithm on a biconnected component. If it is not the first or last component, we need to make sure there is an edge beetween the 2 articulation vertexes so that the construction tree ends with this edge. For this to work this edge must be used first. To perform this we order the adjacency lists so that it is first. If we had to add this edge we mark this in the graph - so that the graph knows that this edge was actually added in afterwards.

Finally we perform a recursive depth first search (DFS) of the input graph to produce a spanning tree and then as the recursion unwinds it builds a possible series-parallel construction sequence using all edges of the graph. The construction sequence is the certificate that shows that the graph has a series-parallel construction. The construction sequence is assembled from collections of edges that would form ears, if an ear decomposition of the graph was produced. As the construction sequence is being assembled the program checks the connection of source and sink vertices of the segments being assembled to determine if the segments are being assembled into a forbidden structure. If a k4 subdivision is detected the program stops processing if a k23 subdivision is detected the program sets a flag variable (k23found) and continues. If no k4 subdivision is found then the graph is series-parallel, if in addition no k23 subdivision is found then the graph is outplanar. If the program completes the construction sequence it then validates the construction sequence to prove that determination of a series-parallel graph by the program is indeed correct.

Internally the graph being processed is represented by an array of adjacency lists, one for each vertex in the graph. After confirming the input file name the main function (main.cpp) populates a Boost graph object. from there the we use the boost functions to decompse the graph as described above into our internal graph sturture which uses adjacency lists. An adjacency list is a simple doubly linked list of Node structures, both defined in AdjacencyList.hpp. These vertices are adjacent to each other thus when populating the array of adjacency lists a Node representing each vertex number is added to the adjacency list of the other vertex. Thus Nodes are added in pairs. Each Node contains a field to hold the vertex number it represents and a pointer to the other Node that was created with it at the time of entry. This pointer field is referred to as the "partner" Node. The Node structure has a destructor function that properly removes the Node from the adjacency list in which it is found. The purpose behind this is to ensure that Nodes can be removed from the adjacency lists in O(1) time and that when attempting to remove an edge from the graph that the specific pair of Nodes representing the endpoints of that edge are removed. To remove a Node from an adjacency list it is only necessary to call the "delete" function on the pointer to that Node. (Thus avoiding having to search the list for the Node before attempting to remove it which would be O(n) and to delete the entire array of adjacency lists would be O(n*n).) To remove a specific edge, we can call delete on the pointer to the Node representing one of the vertexes defining the edge and also call delete on the pointer to its partner Node. The array of adjacency lists is used in the creation of the construction sequence and deleting specific edges is done in the validation of the construction sequence.

The partner node idea is just so that we can use the adjacency list to move from one vertex to the other vertex that this node represents its adjacency to.

When we use the run function, run selects a pointer to a Node from the adjacency lists whose vertexNumber is is the root passed into run. It assigns this pointer as the root Node of the graph and assigns it's number as the root vertex number. It passes this Node pointer as an argument to the initial call to the function GenCS(). By doing so the DFS performed by GenCS() will will be rooted at this vertex. This is so we can force a particular construction by specifing the source and sink.

Tree.h defines the Tree structure. The Tree structure is however just a wrapper for a binary tree whose nodes are Sequence structures defined in Sequence.h. The Tree structure maintains a pointer to the root Sequence structure in the binary tree. Operations on a Tree are just operations on the underlying Sequence binary tree. The Sequence binary tree is used to hold the constructions steps performed to series and parallel connect edges of the graph together. The internal nodes of the tree have fields which hold the type of composition used to connect its child nodes (comp) which will be either "SC" for series connect, or "PC" for parallel connect and fields to indicate the source vertex number (s) and the sink vertex number (t) of the resulting composition. The other fields in an internal node are not used by the program and are assigned null. The leaf nodes of the binary tree are the edges being connected. Leaf nodes have comp set to "EDGE" and have nullptr as children. Leaf nodes have the edge's source vertex number for s and sink vertex number for t. It also has the orientation field (orientation) populated with either "BACK" if it is a back edge or "TREE" if it is a tree edge in the DFS tree. It also has source Node (s_node) and sink Node (t_node) fields populated with pointers to Nodes in the adjacency lists that have vertex numbers s and t respectively. The s_node and t_node values are partners of each other so a leaf node has links back to a specific edge placed in the adjacency lists. Tree.h also defines the function to output a representation of the Tree to a text file (print_tree()), and a function to serial connect two Trees and a function to parallel connect two Trees to form larger Trees.

The final construction sequence (seq) returned by the initial call to GenCS() is a Tree and the fact that leaf nodes in the underlying Sequence binary tree are linked to specific edges in the adjacency lists is important to the construction sequence validation function. After the initial call to GenCS() made in the main program returns the isValidCertificate() function is called with seq as its argument. A post-order traversal of the Sequence binary tree in seq is performed and as a leaf node is encountered delete is called on the s_node and t_node Node pointers of the leaf thus removing that specific edge from the adjacency lists in O(1) time. Also as the traversal returns from an "SC" connected internal node, the adjacency list of the vertex which would be the common connection between the child nodes is checked that it is empty. If it is not empty an exception is thrown. If the traversal completes without throwing an exception and if the construction sequence is valid then all nodes in the adjacency lists should have been removed. The function checks that all adjacency lists are empty. If so the program outputs that the certificate is valid.

GenCS() is passed a pair of pointers to Nodes in the adjacency lists as arguments. This differs from the algorithm which passes vertex numbers. The relationship between the two arguments in the program is the same as the relationship between the vertices in the algorithm. The vertexNumber of the second Node is the parent of the vertexNumber of the first Node in the DFS tree. Also the two Nodes are partners of each other. The exception is the initial call to GenCS() in which the first node is the root Node and there is no parent vertex. This does not cause a problem as the function checks that the first Node vertexNunber is not equal to the root (0) before it tries to access the value of the parent Node vertexNumber. The reason for doing this is that GenCS() creates a Tree representing the parent edge before being added to the construction sequence Tree, it needs pointers to the actual pair of partner Nodes that represent the edge as arguments for the constructor for the Tree. In GenCS() the vertex numbers of the arguments are still used, they are obtained from the vertexNumber fields of the Nodes.

GenCS() is always called with a pair of partner Nodes as arguments execpt for the initial call). GenCS() recursively calls itself while processing the Nodes in an adjacency list. It makes the call if the vertexNumber of the Node from the adjacency list has not been visited before in the DFS. This makes the vertexNumber of the Node a child of the vertex number of the adjacency list in the DFS tree. The arguments passed to the call of GenCS() is the pointer to the Node from the adjacency list and the pointer to its partner Node. Since the vertexNumber of the Node from the adjacency list is adjacent to the vertex of the adjacency list. The partner Node will have vertexNumber equal to that of the adjacency list. Thus the relationship that the second argument Node vertexNumber is the parent of the first argument Node vertexNunber in the DFS tree is maintained. Because GenCS() is always called with the pointer to the Node from the adjacency list and its partner which can be obtained from the first argument the second argument is redundant but has been retained because it more closely matches the function signature shown in the algorithm.

The assignment of source vertex (s) and sink vertex (t) in the construction of a Tree which is only an edge conforms to the same usage specified in the algorithm. However the values of s and t in a Tree that has been composed of multiple edges such as the Trees "seq" and "seq_u" in the program correspond to the opposite labels when referenced by the algorithm. Thus when the algorithm asks for "source of seq" or "source of seq_u" the appropriate value to return is the t value of the seq or seq_u Tree constructed in the program. This is a consequence of the labeling of source and sink for a path being opposite to that for an edge.

When the graph definition file is read in by the main program it creates an array of Vertex structures, one Vertex for each vertex in the graph definition file. Each Vertex holds information associated with that vertex from the graph while the program operates. One of the fields in the Vertex structure is the adjacency list for that vertex. The complete definition of the Vertex structure is given in Vertex.h. Another field is "ear" which holds a pointer to a Tree that represents a back edge that would be computed if the ear() function defined in the paper accompanying the algorithm was calculated for the parent edge of the corresponding vertex in the DFS tree. That is it is the value of ear(v->w) where w is the vertex associated with that Vertex structure. The ear() function defined in the paper is never computed in the algorithm, instead the value of ear() of some edge is simply assigned a value. Only the ear(v->w) value, where v->w is the parent edge of w in the DFS tree is stored in the Vertex structure for vertex w. The ear() value for other edges, in particular back edges, are not stored in the program. This is done to avoid the complications associated with identifying arbitrary edges in some storage container within the program. The only time when the ear() of some edge other than a parent edge is encountered is when a back edge is passed as an argument to update_ear_of_parent(). There is no need to look up or compute the ear() of a back edge as the value is simply the back edge itself. In update_ear_of_parent() if argument f is a back edge then a variable local to update_ear_of_parent() is used to hold ear(f) and its value is just the input back edge f. If f is a tree edge it is the parent edge of some vertex w, in this case the local variable for ear(f) is assigned the value stored in the ear field in the Vertex structure associated to vertex w.

When generate negative certificate (k23 and k4). The program saves the path to a certificate file. When verifying the negative certificate, an array of int stores all vertex number of the endpoints for each path.

Verify_K4_Structure will take this array and create a map to check that how many path terminated on corresponding vertex as value if the number of distinct endpoints is not 4 or the number of path terminated on one endpont is not exactly 3, our certificate is wrong which means the program has a bug. Otherwise, the map will be returned which will be used to verify the edges in the negative certificate (Verify_K23_Structure does a similar thing)

Check_Edge takes two integers as source and the sink of the an edge. And also takes a int indicates the path# and an array that contains all the endpoints. It simply check path# of the vertex vertices[s] and vertices[t]. If s and t is not endpoints but has already assigned other path# which means that the negative certificate we have is not dis-join, then return false. Also, if s and t are not in the other one's adjacency list which means {s,t} is not in g, false will be returned as well. Otherwise, it will return true. If it is true check edge will also add it to the vector that represents this path. These paths are then used in the certificate.

Check_Ear takes a Seqence , a path# and an array that stores endpoints as input. It will check all the edges in such Sequence whether is legal or not by calling Check_Edge for each Edge. Check_Path has similar functionality which checks all the edges in the path by calling the Check_Edge. There is a version of the check_ear which takes in an vector. As the ear is traversed we add the edges into the path vector. This is so that the spandop file can add an ear into the certificate vector. Finally, there is a function called check_path_reverse - which checks the path in a similar way to the normal check_path - except the edges are added to the certication path in reverse.

Verify_Edge is as same as Check_Edge excepts it only check the adjacency list of the two endpoints of the edge to see whether this edge is in graph g or not

The certificate class holds all of the details for the certicate. It holds the paths to be confirmed when there is a negative certiciate, the details on which vertexes or components fail the checks, the outplanar boundary path and the connection tree.
In addition to information it contains functions to check the certicate against the full original graph. During decomposition a second internal graph object is created and all edges are added to it. This graph is then passed into the certification algorithm which will verify the certificate against this graph.

For the certification process a few additional functions were added. To confirm k4 or k23 graphs, we use the vector of edge pairs to find the endpoints - and fill in an array. We use the k4 or k23 verifier again to check that these paths would work. We then use confirmPaths to confirm that this path does exist in the graph. This uses the check_edge function to check that the edges exist. This works since we are using a new graph - so the previous tracing does not affect the result. To confirm the construction tree we use the isValidCertificateFull function. It does a similar thing to the other isValidCertificate and postReduction functions except it works on a sequence that is not made for this graph. These functions use the vertex numbers and adjacency lists instead. These functions also confirm that in parallel constructions, the tails and sources match, and for series composition the source of one is the tail of the other in order.

When reporting k23 or k4 subgraphs in the csgen function we first use the basic ears to identify several of the endpoints. If needed we then identify another endpoint, which is different then the endpoints already identified. In some cases there may be a seperate set of checks if the found ear is trivial. This is needed since trivial ears do not have a sequence for them. When searching for a seperate ear - we look at each adjacent vertex's ear. If that ear is -1 that means it is a trivial ear. This is because it uses a vetex whose DFS number is higher and thus the ear is uninitilized. Whenever we search for an ear in this way we need to be careful not to enter -1 as an endpoint, we need to use the vertex number as the ear sink instead.

In cases where a k4 graph contains a forced edge - we keep the edge in the cert paths. It is included when testing endpoints, however we pass in the 2 vertexes into the confirm path function so that we can skip the edge that is missing. It is also forgone in a similar way when printing out the details of the certficate.