PROBLEMAS GRAFOS JUEZ

---

09 - Los amigos de mis amigos son mis amigos

Lo hemos resuelto con búsqueda en anchura con éxito pero no se ha podido hacer con búsqueda en profundidad debido a la manera en la que esta trabaja, recorriendo los vértices conectados y saltando cuando acaba a otro no descubierto aún.

Por ello, para recorrer solamente los vértices que están conectados mediante aristas, utilizamos la búsqueda en anchura, breadth_first_search() , y contamos la cantidad de vértices conectados entre sí. También marcamos en un vector los nodos visitados para solo recorrer con la búsqueda los nodos necesarios, ya que si no haría una por nodo.

La implementación de la solución está realizada con el método de herencia de una clase default_bfs_visitor creándonos una clase custom_bfs_visitor que realizará las acciones que deseemos cuando se visite un vértice sobreescribiendo la función void discover_vertex(Vertex u, const Graph & g), en este caso añadir +1 al contador de amigos de ese subgrafo y marcarlo en la lista de vértices visitados vec[ ].

10 - Grafo bipartito

Ha podido resolverse correctamente con esta librería utilizando la función is_bipartite(), ya que esta resuelve el ejercicio directamente.

11 - Arborescencias

Ha podido resolverse correctamente con esta librería utilizando la función breadth_first_search() y transpose_graph().

La implementación se basa en averiguar cuántas posibles raíces tiene el grafo, y si solamente existe una opción, comprobar que desde esa raíz son alcanzables todos los vértices y que no existe más de un camino para viajar a cada vértice.

Utilizamos transpose_graph(), para transponer el grafo y así buscando en la lista de adyacencia, los vértices con valor 0, obtenemos el número de raíces posibles. A continuación, con la búsqueda en anchura, comprobamos si se han visitado todos los vértices con un custom visitor que sobreescribe la función discover_vertex() contando los nodos por los que pasa, y si hay más de un camino por el que alcanzar algún vértice con examine_edge(), comprobando si el vértice destino de las aristas ya había sido visitado con anterioridad, ya que esto estaría marcado en un vector de booleanos donde marcamos si ya hemos visitado un nodo. Si se cumplen todas las condiciones sería arborescencia.

12 - La ronda de la noche

El problema se planteó conectando únicamente los nodos de casillas vacías y de origen y destino. Durante la lectura de datos se crean referencias al nodo E y P para saber dónde empieza la búsqueda y qué casilla es la de destino, además de las posiciones de los sensores guardarlas como pares en un vector. Después había que crear el grafo a partir de la matriz de chars. Para ello creamos la función de colocar sensores, donde se marcan en la matriz de chars las casillas afectadas por un sensor con un char especial.

De esta manera, en la matriz de chars podemos identificar los nodos que se conectarán y los que no. Para conectar los nodos recorremos la matriz y creamos conexiones entre los nodos equivalentes al inferior y derecho (la referencia de esto la tenemos gracias a la matriz), de esta manera se crean las conexiones pertinentes. Una vez creado el grafo, lo que queda sería realizar una búsqueda en anchura desde el nodo que guardamos como origen. Este algoritmo recorre el grafo al completo y permite un visitor especial que nos proporciona boost, mediante el cual se guarda en un array de boost las distancias entre el nodo origen y el resto accesibles desde el mismo. De esta manera, si accedemos a la posición adecuada del mismo, podemos saber la distancia que hay hasta P.

```
record_distances(distances.begin(),boost::on_tree_edge());
```

El problema que tiene esta implementación es que el límite del array de boost es de 100.000 posiciones, y en el problema se especifica que estas podrían llegar a ser 1 millón. Además el visitor no acepta como entrada un vector de la librería std o arrays dinámicos, con lo cual esto es un punto a tener en cuenta.

13 - Ovejas negras

Resuelto utilizando la función breadth_first_search().

Para obtener el número de ovejas, empezamos creando la matriz de chars, de forma similar al ejercicio anterior para las casillas con valor “.” una vez conectadas, se realiza búsqueda en anchura con origen en el vértice (0,0), así todas las casillas conectadas a esta se marcan a true en un array de booleanos, y por tanto los interiores de las ovejas blancas quedarán a false. Esto es posible porque las ovejas nunca se solapan ni están pegadas a los bordes.

Tras esto, se recorre la matriz por filas y columnas comprobando que todas las casillas “.” son true, si por lo contrario se encuentra alguna a false, es que se ha descubierto una oveja blanca, y marcamos mediante búsqueda en anchura todas las casillas de la oveja a true, para que no cuente ovejas de más, y se aumenta el contador de estas.

Para marcar las casillas de la oveja a true, utilizamos un VertexData, para obtener la posición del vértice en el array de booleanos.

14 - Petroleros hundidos

Para resolver este ejercicio utilizamos de nuevo una matriz de chars auxiliar donde guardamos los valores “agua” o “petróleo” representados por ‘ ’ y ‘#’ respectivamente.

En la función firstSearch creamos el grafo y le damos a los vértices los valores definidos por nosotros en VertexData, que incluye dos enteros para saber la posición relativa a la matriz y el char para saber qué representa.

Una vez hecho esto, creamos las aristas del grafo mirando esta vez también los vértices diagonales, porque el problema lo requiere.

Después para todos los vértices, si no han sido marcados a false (son agua), recorremos en profundidad para saber el tamaño de la mancha. Guardamos un contador normal y otro que guarde el máximo descubierto hasta el momento y hacemos esto para todas las manchas.

Una vez hecho esto, realizamos nSearch las veces que se introdujera por consola. Esta función mete una nueva mancha en los sitios necesarios y crea conexiones de aristas, si es posible, con cualquiera de las casillas adyacentes y comienza una bfs desde dicha posición, comprobando así si la nueva mancha que pueda generar es más grande que lo anterior.

15 - Pavimentar Barro City

Ha podido resolverse correctamente con esta librería utilizando la función kruskal_minimum_spanning_tree().

También utiliza breadth_first_search(), ya que debemos comprobar mediante el custom_bfs_visitor (comentado en amigos de mis amigos son mis amigos) que todos los vértices están conectados entre sí.

16 - Camino al cole

No ha sido posible resolver este ejercicio de manera satisfactoria debido a que la funcionalidad que da la librería para los caminos cortos sólo comprende la devolución de la distancia y padres anteriores en lo referido a vértices.

Lo más próximo que hemos podido ajustar la resolución del ejercicio es la devolución de uno de los caminos mínimos del grafo para el vértice destino.

Hemos usado dijkstra_shortest_paths para la devolución del camino y hemos usado los mapas donde guarda la información para un vértice predecessor_map & distance_map para mostrar a cuánta distancia está el vértice origen del destino.

Podría resolverse el ejercicio correctamente si creamos clases personalizadas, trabajamos con visitor y sobrecargando funciones puesto que boost da la estructura de base donde trabajar, pero este trabajo resulta muy complejo para decir que la librería da la facilidad para hacerlo.

17 - Repartiendo paquetes

Puede resolverse satisfactoriamente usando dijkstra_shortest_paths para hallar cuales son los caminos mínimos para repartir los paquetes desde la oficina hasta el destino. Se invocará esta función de manera iterativa hasta que todos los paquetes hayan sido entregados. Se ha añadido código adicional para mostrar cuál es la distancia desde el origen a todos los vértices (informativo)