الگوریتم رنگ آمیزی گراف

رحمت اله انصاری

درس اصول طراحی الگوریتم

دکتر مسعودی فر

دانشگاه حکیم سبزواری

رشته مهندسی کامپیوتر

تیر 1401

بسم الله الرحمن الرحیم

مشخصات

عنوان پروژه : **پیاده سازی الگوریتم رنگ آمیزی گراف**

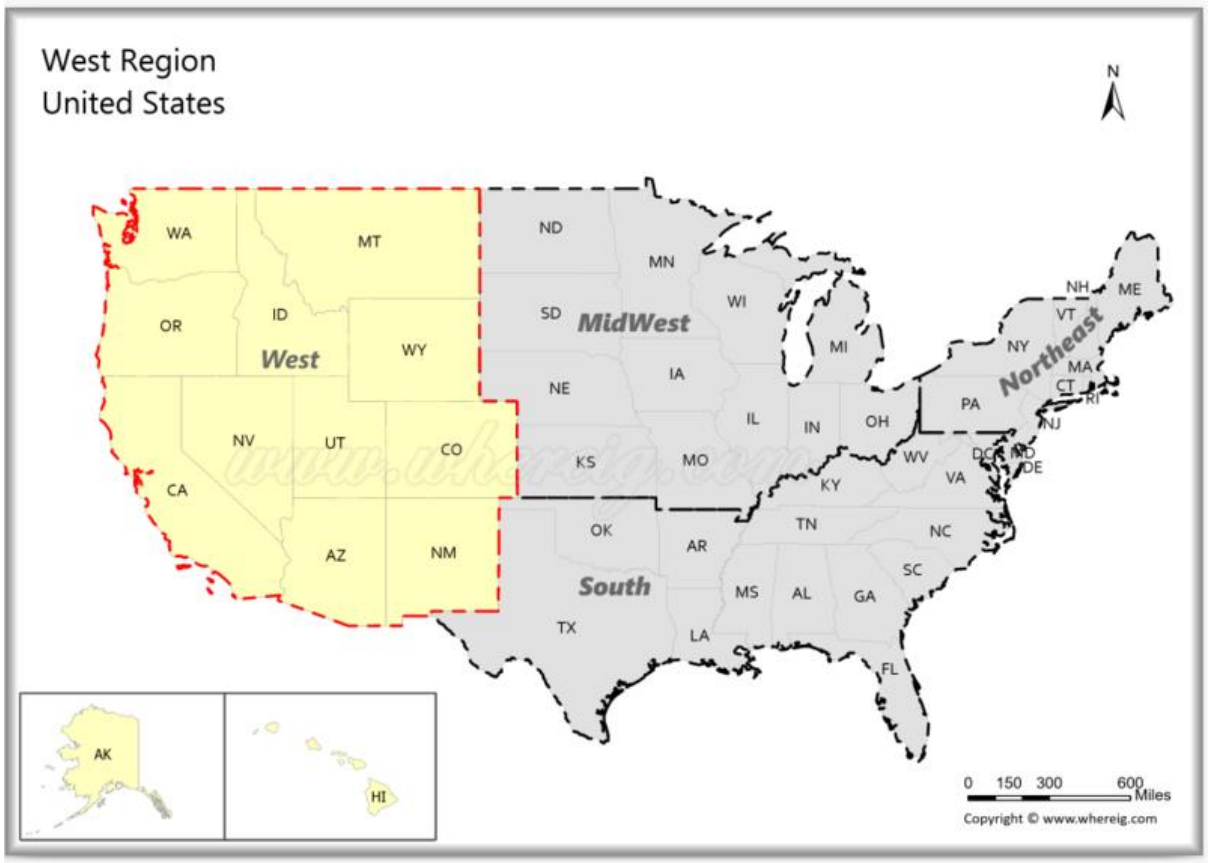
نویسنده:

نام و نام خانوادگی : **رحمت اله انصاری**

شماره دانشجویی : **9912377331**

ایمیل : **Rahmat2022a@gmail.com**

هدف از این پروژه بررسی الگوریتم رنگ آمیزی گراف است.

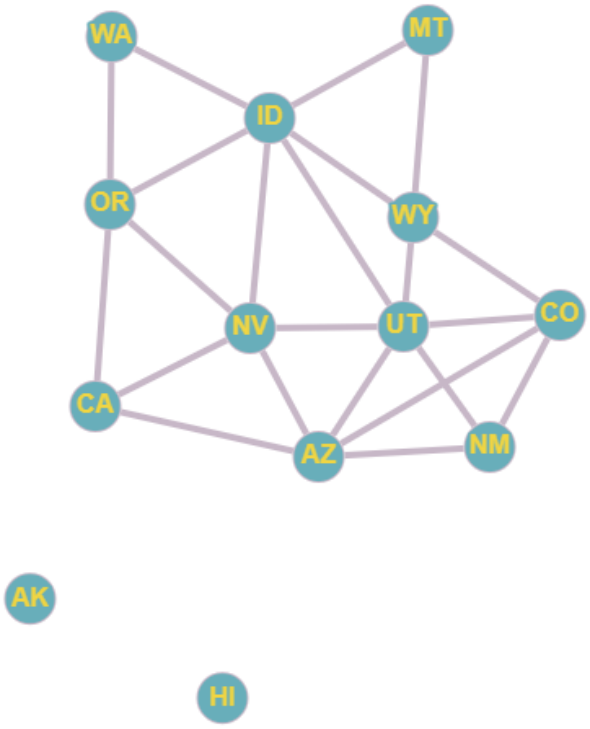


میخواهیم نقشه غرب آمریکا را با استفاده از الگوریتم رنگ آمیزی گراف رنگ آمیزی کنیم.

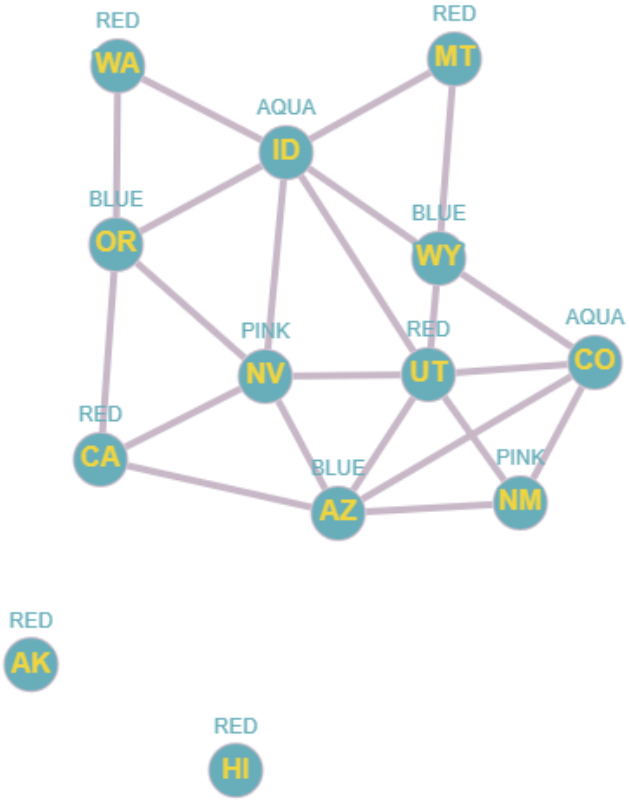
نقشه در بالا آمده است. نام ایالات به طور مخفف آمده است.

نام های کامل آن ها را در شکل زیر شاهد هستیم.

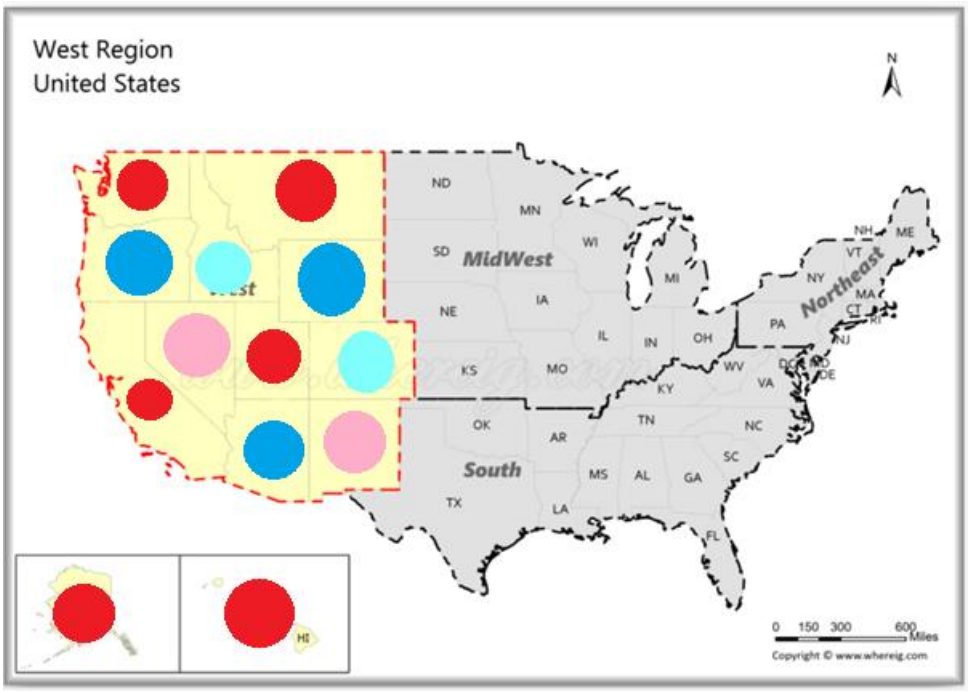
در اینجا ما با استفاده از نام مخفف آن ها این نقشه را به یک گراف تبدیل کرده ایم.



خروجی ما پس از دادن شکل بالا به برنامه شکل زیر خواهد بود که رنگ هر گره در بالای آن مشخص شده است.

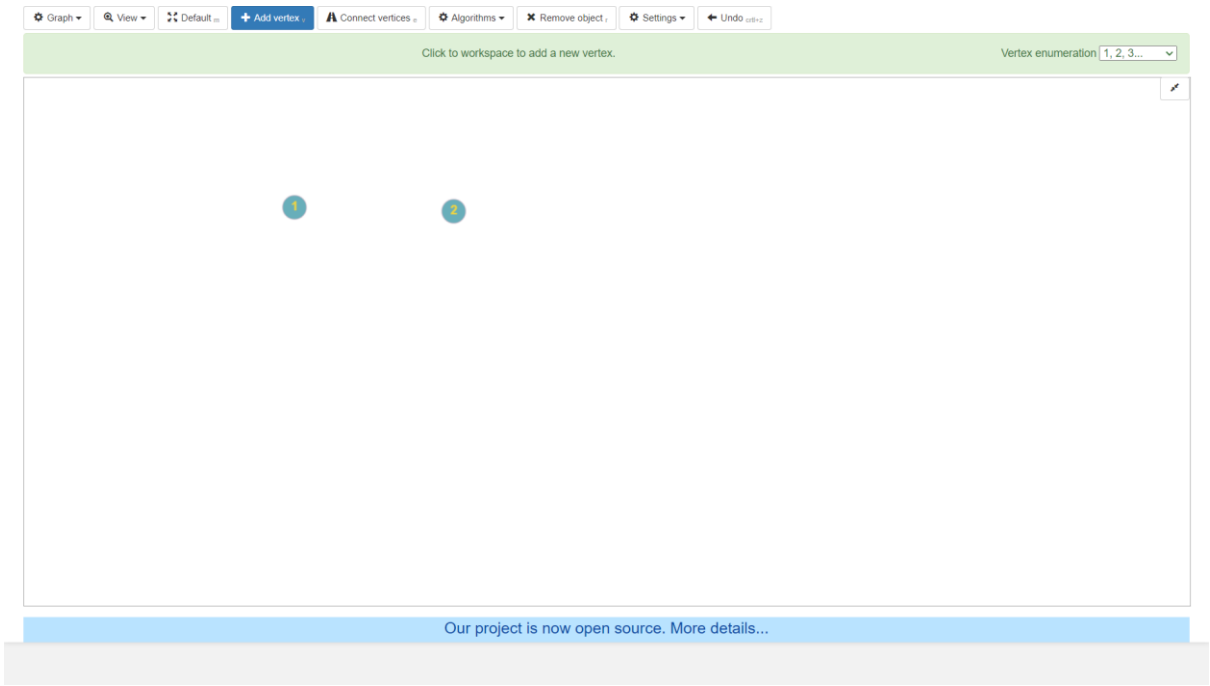


اگر بخواهیم نقشه را رنگ کنیم به صورت زیر خواهد بود:



همانطور که میبینید از ۴ رنگ برای رنگ آمیزی استفاده شده است.

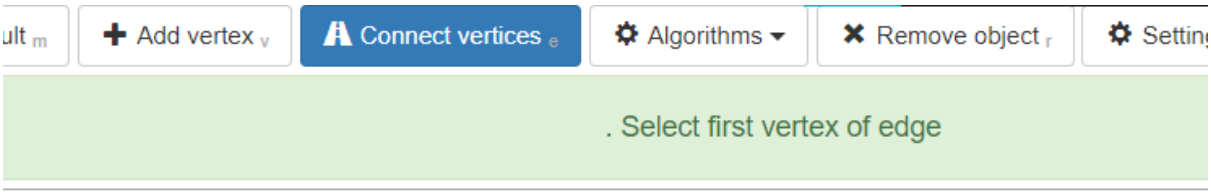
برای ساختن شکل ما از سایت https://graphonline.ru/en/ استفاده کرده ایم.



برای اضافه کردن گره از Add vertex استفاده می کنیم.

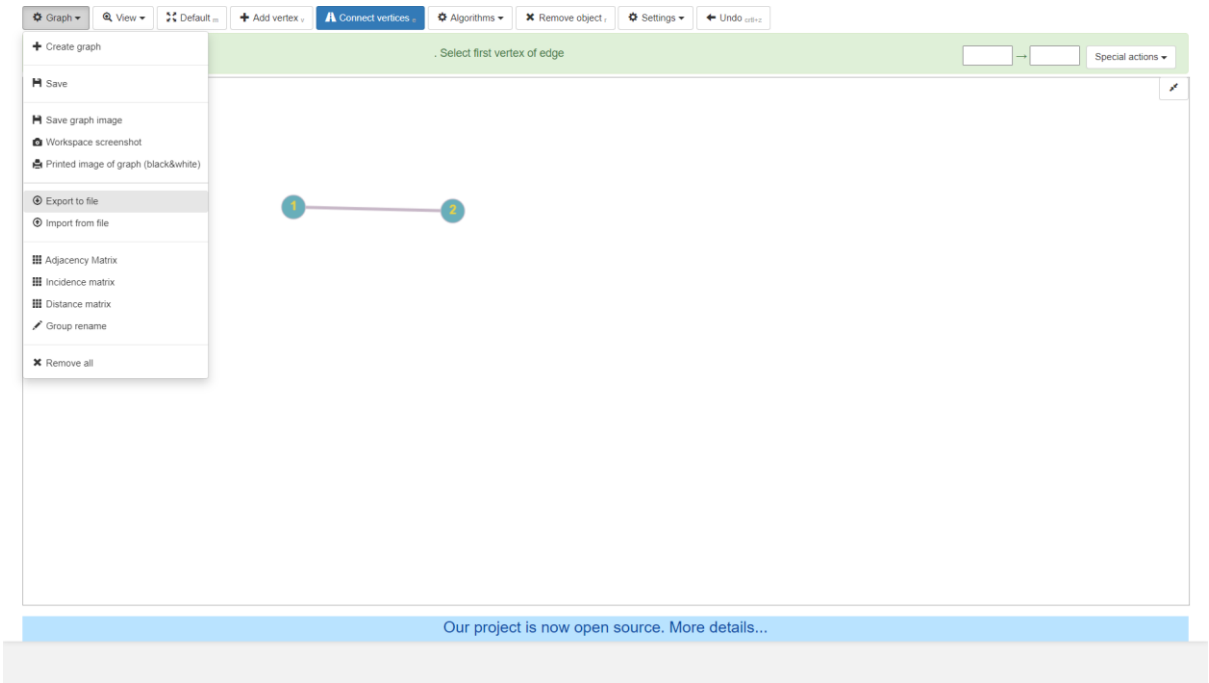
با راست کلیک کردن روی هر گره میتوانیم نام آن را عوض کنیم.

برای اضافه کردن یال از connect vertices استفاده می کنیم.



برای اینکار ابتدا بر روی مبدا یال و سپس مقصد یال کلیک می کنیم. سپس بر روی undirect میزنیم تا یال ما بدون جهت باشد. (وزن آن را که به طور پیشفرض روی بدون وزن است تغییر نمی دهیم.)

برای گرفتن خروجی و دادن این شکل به برنامه به صورت زیر عمل می کنیم:

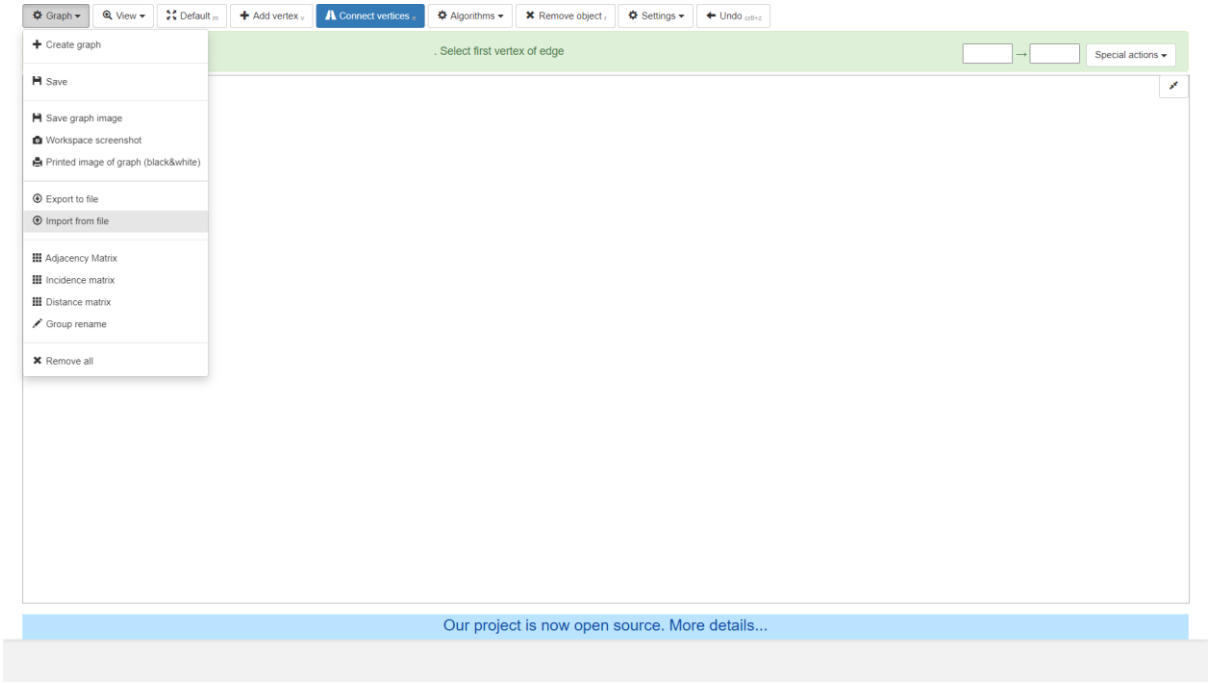


از تب Graph بر روی Export to file میزنیم تا فایل را داشته باشیم. این فایل در پوشه دانلود ها می باشد. این فایل را می بایست به کنار برنامه خود منتقل کنیم. (البته راه های دیگری هم هست.)

این فایل یک فایل xml با پسوند graphml می باشد. درون آن اطلاعات گراف می باشند.

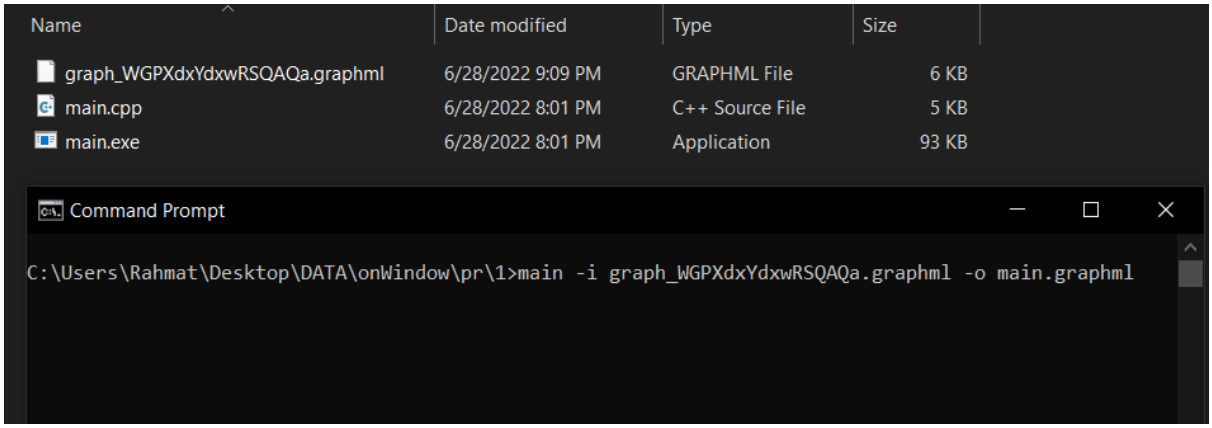
تنها تغییری که برنامه بر روی این فایل انجام می دهد این است که نوشته روی گره ها را به رنگ آن تغییر می دهد.

سپس فایل خروجی که آن هم باید با پسوند graphml می باشد را باید به درون سایت بیاوریم.

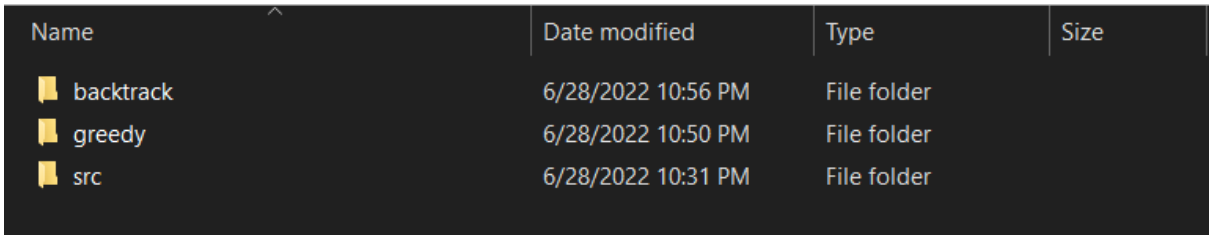


برای اینکار باز هم به تب Graph می رویم و در نهایت import from file را انتخاب می کنیم و فایل خروجی را انتخاب می کنیم.

برای ساخت خروجی از برنامه به نحوه زیر استفاده می کنیم.



برای اینکار ابتدا نام برنامه را مینویسم و سپس از دو سوییچ i و o استفاده می کنیم. اولی برای دادن آدرس فایل ورودی برنامه و دومی برای دادن آدرس (و نام) فایل خروجی ما.



ساختار فایل های ما به شکل بالاست.

در پوشه greedy از الگوریتم حریصانه برای حل استفاده کرده ایم. در این روش پیچیدگی زمانی ما O(V^2 + E) می باشد. یعنی ما برای حل نقشه وارده در بالا به تعداد زیر عملیات نیاز داریم:

V ^ 2 + E = 13 ^ 2 + 21 = 190

در روش بک ترک اما دارای پیچیدگی زمانی O(V^m) هستیم که اگر ندانیم m را چند بگیریم کارمان سخت خواهد بود. (بدترین حالت یعنی m = v) ما در اینجا کوچکترین حالت ممکن یعنی ۴ را میگیریم. اگر 3 بگیریم برنامه یه ما خواهد گفت که چنین چیزی ممکن نیست.

V ^ m = 13 ^ 4 = 28,561

همانطور که می بینید زمان روش بک ترک فوق العاده بیشتر است.

البته نقطه ضعف الگوریتم حریصانه این است که ممکن است از کمترین میزان رنگ استفاده نکند.

فایل src هم شامل فایلی است که تبدیل فایل xml را بر عهده دارد.

```cpp
#include <iostream>
#include <fstream>
#include <list>
using namespace std;
#include "../src/graphic.hpp"

class Graph {
    int V;
    list<int> *adj;
public:
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; }
    void addEdge(int v, int w) {
        adj[v].push_back(w);
        adj[w].push_back(v);
    }
    void greedyColoring(int result[]) {
        result[0] = 0;
        for(int u = 1; u < V; u++)
            result[u] = -1;
        bool available[V] = {0};
        for(int u = 1; u < V; u++) {
            list<int>::iterator i;
            for(i = adj[u].begin(); i != adj[u].end(); ++i)
                if (result[*i] != -1)
                    available[result[*i]] = true;
            int cr;
            for(cr = 0; cr < V; cr++)
                if(available[cr] == false) break;
            result[u] = cr;
            for(i = adj[u].begin(); i != adj[u].end(); ++i)
                if(result[*i] != -1)
                    available[result[*i]] = false;
        }
    }
};

```

کد بالا مربوط به روش حریصانه است.

در این روش از دیتا استرکچری مخصوص استفاده کرده ایم.

اسم این دیتا استراکچر را Graph گذاشتیم.

دیتا ممبرهای این کلاس یکی تعداد رئوس است که در سازنده کلاس هم دریافت می شود. دیگری آرایه ای از لیست هاست. به ازای هر راس یک لیست داریم که درون آن همسایه های راس هستند.

```
8       int V;
9       list<int> *adj;
```

این کلاس دارای مخرب و یک سازنده است. در سازنده تعداد رئوس دریافت می شود و آرایه ای پویا با کمک اشاره گر به تعداد رئوس ساخته می شود. در مخرب هم این آرایه پاک می شود.

```
11      Graph(int V) { this->V = V; adj = new list<int>[V]; }
12      ~Graph() { delete [] adj; }
```

دو تابع دیگر هم در این کلاس وجود دارد.

```
13      void addEdge(int v, int w) {
14          adj[v].push_back(w);
15          adj[w].push_back(v);
16      }
```

یکی برای افزودن یال که در آن دو راس (عدد آن ها) دریافت می شود و هر یک را به لیست همسایه ی دیگری اضافه می کنیم.

دومین تابع تابع اصلی ما یعنی greedyColoring است. این تابع یک آرایه دریافت می کند. این آرایه همان خروجی اصلی ما یا رنگ های ما هستند. تعداد خانه های ما به تعداد رئوس است و به ازای هر راس یک خانه در این آرایه داریم. هر عدد هم به عنوان یک رنگ در این خانه قرار می گیرد.

```
17      void greedyColoring(int result[]) {
18          result[0] = 0;
19          for(int u = 1; u < V; u++)
20              result[u] = -1;
21          bool available[V] = {0};
22          for(int u = 1; u < V; u++) {
23              list<int>::iterator i;
24              for(i = adj[u].begin(); i != adj[u].end(); ++i)
25                  if (result[*i] != -1)
26                      available[result[*i]] = true;
27              int cr;
28              for(cr = 0; cr < V; cr++)
29                  if(available[cr] == false) break;
30              result[u] = cr;
31              for(i = adj[u].begin(); i != adj[u].end(); ++i)
32                  if(result[*i] != -1)
33                      available[result[*i]] = false;
34          }
35      }
```

ابتدا اولین عنصر این خانه را صفر میکنیم. یعنی اولین خانه را با اولین رنگ رنگ میزنیم.

سپس به مقادیر دیگر این آرایه مقدار منفی یک را می دهیم به این معنا که رنگی ندارند.

```
18              result[0] = 0;
19              for(int u = 1; u < V; u++)
20                  result[u] = -1;
```

یک آرایه از جنس boolean به طور کمکی استفاده می کنیم که ابتدا به تمام مقادیرش false میدهیم. تعداد اعضای این تابع به تعداد رئوس ماست یعنی حداکثر تعداد رنگ ممکن.

```
21              bool available[V] = {0};
```

حالا مسئله را برای v – 1 عنصر رنگ نخورده خود ادامه می دهیم. در واقع ما میخواهیم از رنگی که کمترین استفاده را شده و در یال های مجاور قرار ندارد برای رنگ کردن یک راس استفاده کنیم. اگر تمام رنگ ها در مجاورها استفاده شده بود از یک رنگ جدید استفاده میکنیم. از راس بعدی شروع می کنیم و مقدار آرایه کمکی خود که نامش available است را به ازای

رئوسی که رنگ شده اند true قرار می دهیم. مقدار درست available[cr] به این معنی است که رنگ cr در یکی از یال های مجاور استفاده شده.

```
23              list<int>::iterator i;
24              for(i = adj[u].begin(); i != adj[u].end(); ++i)
25                  if (result[*i] != -1)
26                      available[result[*i]] = true;
```

در حلقه بعدی به دنبال اولین رنگ در دسترس میرویم. (بدیهی است اگر تمام رنگ های قبلی در دسترس نبودند یک رنگ جدید میسازیم.

```
27              int cr;
28              for(cr = 0; cr < V; cr++)
29                  if(available[cr] == false) break;
```

این مقدار را به عنوان رنگ این راس بر می گزینیم.

```
30                  result[u] = cr;
```

در نهایت دوباره مقادیر این تابع کمکی را برای مقادیری که true شده بودند false میکنیم و به گره بعد می پردازیم. این کار را تا اتمام گره ها ادامه میدهیم.

```
31              for(i = adj[u].begin(); i != adj[u].end(); ++i)
32                  if(result[*i] != -1)
33                      available[result[*i]] = false;
```