CS1200: Intro. to Algorithms and their Limitations	Prof. Salil Vadhan
Lecture 12: Graph Coloring	
Harvard SEAS - Fall 2025	*2025-10-09*

1 Announcements

- Keep up the in-class participation! Apologies for missing the questions on Ed; Alisona will sit in a more visible spot.

Recommended Reading:

- Hesterberg–Vadhan Ch. 13
- Roughgarden III, 13.1
- Lewis & Zax 18

2 Loose Ends: Finding the Paths in BFS

Q: How to actually find shortest *path* from s to t, not just the distances?

A:

Input: A digraph $G = (V, E)$ and a source vertex $s \in V$
Queries: For any query vertex t, return a shortest path from s to t (if one exists).

Abstract Data Type Single-Source Shortest Paths

Theorem 2.1. Single-Source Shortest Paths *has a data structure that, for digraphs with n vertices and m edges, has preprocessing time $O(n+m)$ and the time to answer a query t is $O(\mathrm{dist}_G(s,t))$ if $\mathrm{dist}_G(s,t) < \infty$, and $O(1)$ if $\mathrm{dist}_G(s,t) = \infty$.*

Given this, we deduce an algorithm for the Shortest Walk problem we started with, by running the preprocessor and then querying the data structure once with our given destination vertex t:

Corollary 2.2. Shortest Walk *(equivalently,* Shortest Path*) can be solved in time $O(n+m)$.*

It is natural and very useful (e.g. in Google maps) to have data structures for variants of Shortest Path on *dynamic graphs*. There is a rich collection of methods for dynamic graph data structures, which are beyond the scope of this course.

Another extension of BFS is to handle *weighted graphs*. This is Djikstra's algorithm, and is covered in CS 1240.

3 Motivating graph coloring: register allocation

When covering RAM simulations, we saw how to simulate programs that many use many variables with ones that use a fixed number c of registers, by swapping variables in and out of memory.

New Approach. Avoid swapping in and out of memory, by reusing the same registers for different variables. (Most variables generated by compilers are temporary.)

This approach proceeds as follows: at each line of code, every 'live' temporary variable is assigned to one of the c registers. We need to ensure that no register is assigned to more than one live variable at a time. For each temporary variable `var`, we define a *live region* R, which are the lines of code in which the value of `var` needs to be maintained.

Example 3.1. Consider the (Word-)RAM program in Algorithm 3.1.

```
    SquaredDistanceToNegativeOne(x):
    Input      : An array x = (x[0], x[1], ..., x[n-1])
    Output     : (x[0]+1)^2 + (x[1]+1)^2 + ... + (x[n-1]+1)^2
    Variables  : input_len, output_len, output_ptr, temp_0, temp_1, temp_2, temp_3
 0  output_ptr = input_len;
 1  output_len = 1;
 2  temp_3 = 0;
 3      IF input_len == 0 GOTO 15;
 4      temp_0 = 1;
 5      temp_0 = temp_0 + temp_3;
 6      input_len = input_len - temp_0;
 7      temp_1 = M[input_len];
 8      temp_1 = temp_1 + temp_0;
 9      temp_1 = temp_1 × temp_1;
10      temp_2 = M[output_ptr];
11      temp_2 = temp_2 + temp_1;
12      temp_3 = 0;
13      M[output_ptr] = temp_2;
14      IF temp_3 == 0 GOTO 3;
15  HALT ;                                  /* not an actual command */
```

Algorithm 3.1: Toy (Word-)RAM program, which interprets the input as the coordinates of a point in n dimensions and calculates its squared distance to the point $(-1, -1, \ldots, -1)$.

In this program, the live regions for the temporary variables $\texttt{temp}_0, \texttt{temp}_1, \texttt{temp}_2, \texttt{temp}_3$ are:

$$\begin{aligned} R_0 &= \\ R_1 &= \\ R_2 &= \\ R_3 &= \end{aligned}$$

We can model this problem graph-theoretically by defining a *conflict* graph (aka the “register interference graph”):

Graph coloring is the way to formulate the problem of finding a valid assignment of live regions to registers.

Definition 3.2. For an undirected graph $G = (V, E)$, a (proper[1]) *k-coloring* of G is

The computational problem of finding a coloring is called the GRAPH COLORING problem:

Input: A graph $G = (V, E)$ and a natural number k
Output: A k-coloring of G (if one exists)

Computational Problem GRAPH COLORING

Returning to the REGISTER ALLOCATION problem, if we have a proper k-coloring f of the conflict graph, then we can safely replace each variable `var` with a new register (i.e. variable) $\mathtt{reg}_{f(\mathtt{var})}$, thereby using only the k variables $\mathtt{reg}_0, \mathtt{reg}_1, \ldots, \mathtt{reg}_{k-1}$ in our new (but equivalent) program.

Using a 2-coloring of the conflict graph for our Toy (Word-)RAM program above, we obtain a new program after register allocation as follows:

Remark. A common variant of the GRAPH COLORING problem is to find a coloring using as *few* colors as possible, for a given a graph G. This problem is in some sense the opposite of another problem we recently looked at:

[1] An *improper* coloring allows us to assign the same color to vertices that share an edge, but in graph theory, 'coloring' means 'proper coloring' unless explicitly stated otherwise.

4 Greedy Coloring

A natural first attempt at graph coloring is to use a *greedy* strategy (in general, a *greedy* algorithm is one that makes a sequence of myopic decisions, without regard to what choices will need to be made in the future):

`GreedyColoring`(G):
Input : A graph $G = (V, E)$
Output : A coloring f of G using "few" colors
0 Select an ordering $v_0, v_1, v_2, \ldots, v_{n-1}$ of V;
1 **foreach** $i = 0$ *to* $n - 1$ **do**
2 $\quad f(v_i) =$ ____.
3 **return** f

Algorithm 4.1: A greedy coloring algorithm.

An example of this algorithm is depicted in Figure 1.

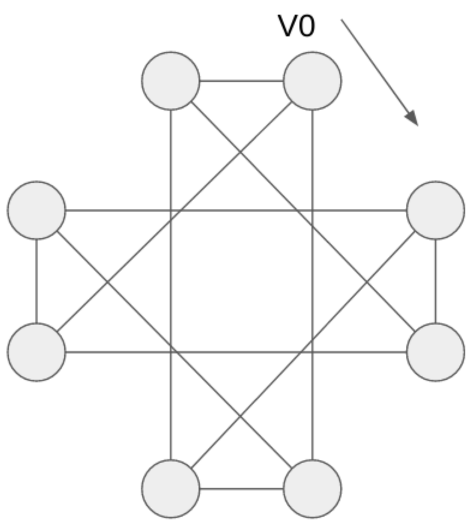

Figure 1: `GreedyColoring` algorithm with the ordering of vertices in clockwise direction starting from v_0

Assuming that we select the ordering (Line 0) in a straightforward manner (e.g. in the same order that the vertices are given in the input), `GreedyColoring`(G) can be implemented in time $O(n + m)$.

By inspection, `GreedyColoring`(G) always outputs a proper coloring of G. What can we prove about how many colors it uses?

Theorem 4.1. *When run on a graph* $G = (V, E)$ *with any ordering of vertices,* `GreedyColoring`(G) *will use at most* $\deg_{max}+1$ *colors, where* $\deg_{max} = \max\{\deg(v) : v \in V\}$.

Proof.

□

Remark. Note that this is an algorithmic proof of a pure graph theory fact: every graph is $(\deg_{max} + 1)$-colorable. However, this bound of $\deg_{max} + 1$ can be much larger than the number of colors actually needed to color G.

The performance of greedy algorithms can be very sensitive to the order in which decisions are made, and often we can achieve much better performance by picking a careful ordering. For example, we can process the vertices in *BFS order*, as described below.

`BFSColoring`(G):
Input : A connected graph $G = (V, E)$
Output : A coloring f of G using "few" colors

Fix an arbitrary start vertex $v_0 \in V$;
Start breadth-first search from v_0 to obtain a vertex order $v_1, v_2, \ldots, v_{n-1}$;
foreach $i = 0$ *to* $n - 1$ **do**
    $f(v_i) = \min \{c \in \mathbb{N} : c \neq f(v_j) \;\; \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\}$.
return f

Algorithm 4.2: Greedy coloring in BFS order.

Running the `BFSColoring` algorithm on the same example as above, we obtain

Now, we show that, in general, for 2-colorable graphs, `BFSColoring` finds a

2-coloring efficiently.

Theorem 4.2. *If G is a connected 2-colorable graph, then* `BFSColoring`(G) *will color G using 2 colors.*

Proof sketch.

$\square$

Corollary 4.3. Graph 2-Coloring *can be solved in time $O(n+m)$.*

Proof.

$\square$