

Lecture 11: Graph Coloring

Harvard SEAS - Fall 2022

Oct. 11, 2022

1 Announcements

- Pset revision video: target is 3 minutes. Prioritize if necessary. Graders won't watch past 6 minutes.
- Sophomores: CS concentration advising event 10-30 in LL1.232/200, <http://tiny.cc/sophomorecs>
- Be sure to regularly check Ed for clarifications or corrections on psets.

Recommended Reading:

- Lewis-Zax Ch. 18
- Roughgarden III Sec. 13.1

2 Graph Coloring

Motivating Problem: Register allocation.

Goal: more efficiently simulate (Word-)RAM programs with a large number of variables on CPUs with a fixed number c of registers (=new variables) by reusing the same registers for different variables, rather than swapping variables in and out of main memory like we did in Lecture 7 (Thm. 5.1). Specifically, compilers generate code with a huge number of short-lived temporary variables, and it would be very slow if all of these had to be continually swapped in and out of main memory.

Approach: 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.

To do this, 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:

Input	: An array $x = (x[0], x[1], \dots, x[n-1])$
Output	: $(x[0] + 1)^2 + (x[1] + 1)^2 + \dots + (x[n-1] + 1)^2$
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 1: Toy RAM program

Live regions for temp₀, temp₁, temp₂, temp₃:

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

A formal definition of live regions is below for optional reading in case you are interested.

Definition 2.1 (live regions — optional). Let $P = (V, C_0, C_1, \dots, C_{\ell-1})$ be a RAM program. For a variable $\text{var}_0 \in V - \{\text{input_len}, \text{output_len}, \text{output_ptr}\}$, an *assign line for var* is a line C_i of P of one of the following forms:

1. $\text{var} = c$,
2. $\text{var} = \text{var}_0 \text{ op } \text{var}_1$ with $\text{var}_0, \text{var}_1 \neq \text{var}$, or
3. $\text{var} = M[\text{var}_0]$ with $\text{var}_0 \neq \text{var}$.

An *access line for var* is a line C_i of P of one of the following forms:

1. $\text{var}_0 = \text{var}_1 \text{ op } \text{var}_2$ with $\text{var}_1 = \text{var}$ or $\text{var}_2 = \text{var}$,
2. $\text{var}_0 = M[\text{var}]$,
3. $M[\text{var}_0] = \text{var}_1$ with $\text{var}_0 = \text{var}$ or $\text{var}_1 = \text{var}$, or

4. IF `var == 0` GOTO k .

For a line C_i of P we say that `var` is *live* at C_i if line C_i can potentially be executed before the execution of an access line for `var` (inclusive—so `var` is live at every access line) but with no intervening assign line.¹ The *live region* R_{var} is defined to be the set of lines at which `var` is live.

Q: How can we model this problem graph-theoretically?

How can we formulate the problem of finding a valid assignment of live regions to registers?

3 Graph Coloring

Definition 3.1. For an undirected graph $G = (V, E)$, a (proper) k -coloring of G is

An *improper* coloring allows us to assign the same color to vertices that share an edge, but we will work with proper colorings unless we explicitly state otherwise.

Example: If we have a proper k -coloring f of the register interference graph, then we can safely

¹Note that it can be possible for C_i to be executed before C_j even $i > j$ because GOTOs can lead to lines being executed out of order. To determine the live regions, we treat the conditional (`var0 == 0`) in each GOTO line as if it can be either true or false (ignoring how `var0` was computed). That is, we use a *syntactic* definition of live regions, rather than a *semantic* one, which would ask whether there exists an input x to P such that in the computation of P on x , C_i is executed between an assign line and an access line. It turns out that computing the semantic live regions of a program is an *unsolvable* computational problem.

replace each variable `var` with a new register (i.e. variable) `regf(var)`, thereby using only the k variables `reg0`, `reg1`, \dots , `regk-1` in our new (but equivalent) program.

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

Computational Problem Graph Coloring

Alternatively, we are given a graph G and we wish to find a proper coloring using as *few* colors as possible. What problem is this an opposite of?

4 Greedy Coloring

A natural first attempt at graph coloring is to use a *greedy* strategy:

<pre> 1 GreedyColoring(G) Input : A graph $G = (V, E)$ Output : A coloring f of G using “few” colors 2 Select an ordering $v_0, v_1, v_2, \dots, v_{n-1}$ of V; 3 foreach $i = 0$ to $n - 1$ do 4 $f(v_i) = \min \{c \in \mathbb{N} : c \neq f(v_j) \ \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\}.$ 5 return f </pre>

Example:

In general, a *greedy* algorithm is one that makes a sequence of myopic decisions (above, the color of a vertex v), without regard to what choices will need to be made in the future.

Assuming that we select the ordering (Line 2) 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 $O(n + m)$. (However, sometimes we will want to select the ordering in a more sophisticated manner that takes more time.)

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 $d_{\max} + 1$ colors, where $d_{\max} = \max\{d(v) : v \in V\}$.*

Proof.

□

Note that this is an algorithmic proof of a pure graph theory fact: every graph is $(d_{max} + 1)$ -colorable. However, this bound of $d_{max} + 1$ can be much larger than the number of colors actually needed to color G , but this turns out to be tight for greedy coloring in an arbitrary vertex order, even on 2-colorable graphs.

However, the performance of greedy algorithms is 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*:

```
1 BFSColoring( $G$ )
   Input      : A connected graph  $G = (V, E)$ 
   Output     : A coloring  $f$  of  $G$  using “few” colors
2 Fix an arbitrary start vertex  $v_0 \in V$ ;
3 Start breadth-first search from  $v_0$  to obtain a vertex order  $v_1, v_2, \dots, v_{n-1}$ ;
4 foreach  $i = 0$  to  $n - 1$  do
5   |  $f(v_i) = \min \{c \in \mathbb{N} : c \neq f(v_j) \ \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\}.$ 
6 return  $f$ 
```

Example:

Theorem 4.2. *If G is a connected 2-colorable graph, then $\text{BFSColoring}(G)$ will color G using 2 colors.*

Proof. Let f^* be a 2-coloring of G . We may assume that $f^*(v_0) = 0$ without loss of generality (why?). Let f be the coloring of G found by $\text{BFSColoring}(G)$. We argue by (strong) induction on i that $f(v_i) = f^*(v_i)$ for $i = 0, \dots, n - 1$.

□

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

Proof.

□