CS120: Intro. to Algorithms and their Limitations	Hesterberg & Vadhan

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 $x = (x[0], x[1], \ldots, x[n-1])$
Output : $(x[0]+1)^2 + (x[1]+1)^2 + \cdots + (x[n-1]+1)^2$
Variables: $\texttt{input\_len}, \texttt{output\_len}, \texttt{output\_ptr}, \texttt{temp}_0, \texttt{temp}_1, \texttt{temp}_2, \texttt{temp}_3$

0 $\texttt{output\_ptr} = \texttt{input\_len}$;
1 $\texttt{output\_len} = 1$;
2 $\texttt{temp}_3 = 0$;
3   IF $\texttt{input\_len} == 0$ GOTO 15;
4     $\texttt{temp}_0 = 1$;
5     $\texttt{temp}_0 = \texttt{temp}_0 + \texttt{temp}_3$;
6     $\texttt{input\_len} = \texttt{input\_len} - \texttt{temp}_0$;
7     $\texttt{temp}_1 = M[\texttt{input\_len}]$;
8     $\texttt{temp}_1 = \texttt{temp}_1 + \texttt{temp}_0$;
9     $\texttt{temp}_1 = \texttt{temp}_1 \times \texttt{temp}_1$;
10     $\texttt{temp}_2 = M[\texttt{output\_ptr}]$;
11     $\texttt{temp}_2 = \texttt{temp}_2 + \texttt{temp}_1$;
12     $\texttt{temp}_3 = 0$;
13     $M[\texttt{output\_ptr}] = \texttt{temp}_2$;
14   IF $\texttt{temp}_3 == 0$ GOTO 3;
15 HALT ;   `/* not an actual command */`

Algorithm 1: Toy RAM program

Live regions for $\texttt{temp}_0, \texttt{temp}_1, \texttt{temp}_2, \texttt{temp}_3$:

$$
\begin{aligned}
R_0 &= \{4, 5, 6, 7, 8\} \\
R_1 &= \{7, 8, 9, 10, 11\} \\
R_2 &= \{10, 11, 12, 13\} \\
R_3 &= \{2, 3, 4, 5, 12, 13, 14\}
\end{aligned}
$$

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

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

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

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

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

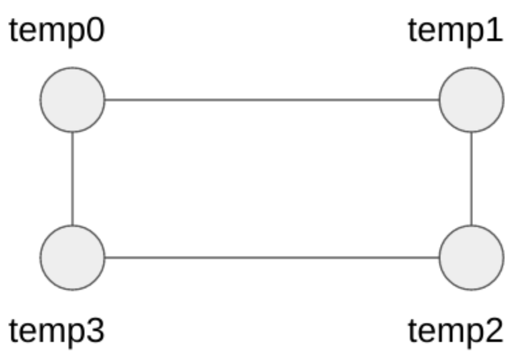
4. IF $\texttt{var} == 0$ GOTO k.

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* $R_{\texttt{var}}$ is defined to be the set of lines at which `var` is live.

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

Define a *conflict* graph (aka the "register interference graph"):

- Vertices = a subset of the variables of P (other than $\texttt{input\_len}, \texttt{output\_len}, \texttt{output\_ptr}$) for which we want to do register allocation (e.g. the 'temporary' variables created during compilation)
- Edges = $\{\{(\texttt{var}, \texttt{var}')\} : R_{\texttt{var}} \cap R_{\texttt{var}'} \neq \emptyset\}$.



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 a mapping $f : V \to [k]$ such that for all edges $\{u, v\} \in E$, we have $f(u) \neq f(v)$

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

[1]Note 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 ($\texttt{var}_0 == 0$) in each GOTO line as if it can be either true or false (ignoring how $\texttt{var}_0$ 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) $\texttt{reg}_{f(\texttt{var})}$, thereby using only the k variables $\texttt{reg}_0, \texttt{reg}_1, \ldots, \texttt{reg}_{k-1}$ in our new (but equivalent) program.

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

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?

Coloring is the opposite of connected components!

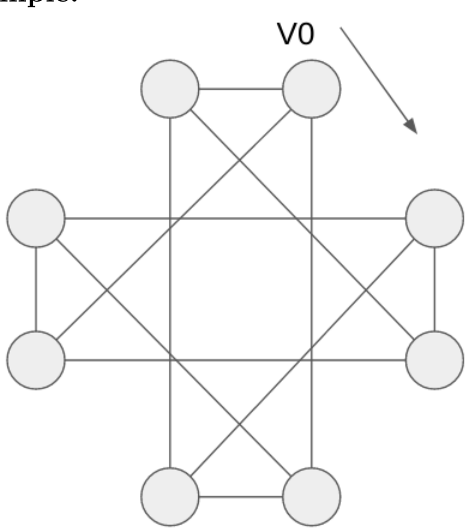
- Coloring: partition V into as few sets as possible such that there are no edges within each set.
- Connected components: partition V into as many sets as possible such that there are no edges crossing between different sets.

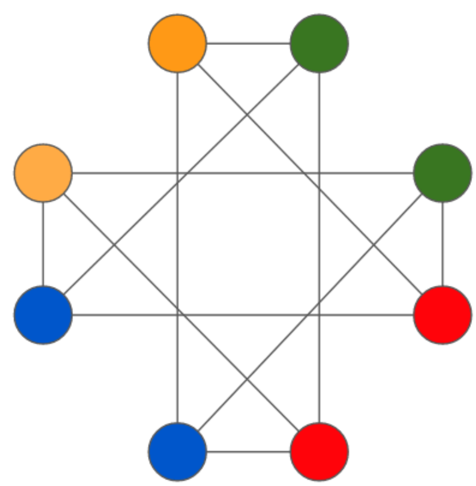
4 Greedy Coloring

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

GreedyColoring(G)
Input : A graph $G = (V, E)$
Output : A coloring f of G using "few" colors
Select an ordering $v_0, v_1, v_2, \ldots, v_{n-1}$ of V;
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

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 $d_{max} = \max\{d(v) : v \in V\}$.* d=degree (number of connections)

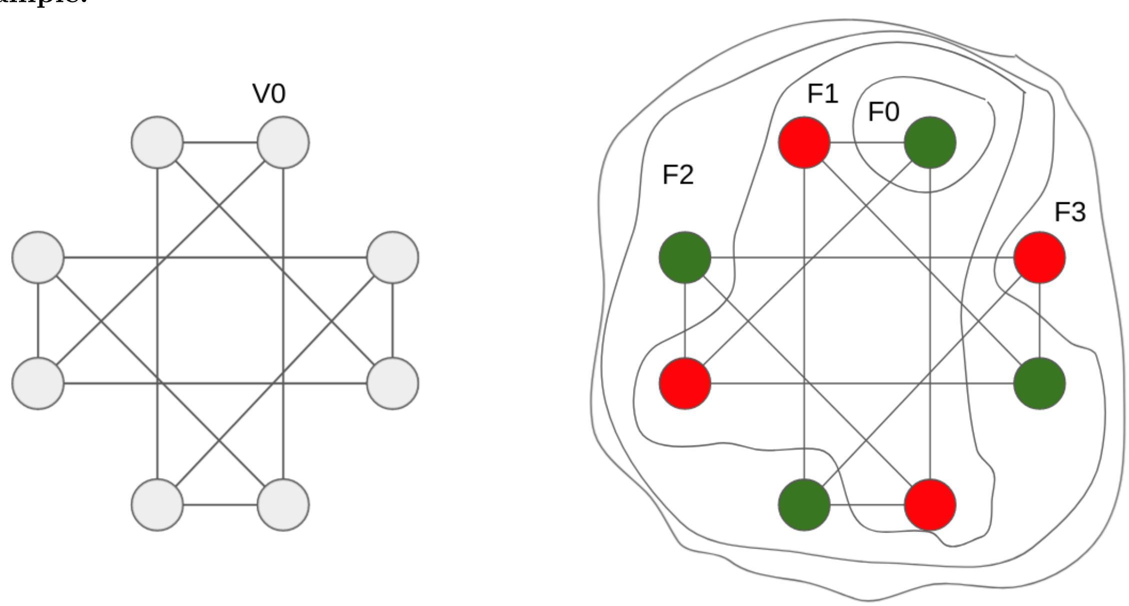
Proof. The set $\{c \in \mathbb{N} : c \neq f(v_j) \quad \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\}$ of size at most $d(v_j) \leq d_{max}$, so cannot include all of the colors $0, 1, 2, \ldots, d_{max}$. Thus when we assign $f(v_i)$ to be the minimum element of the set, we will have $f(v_i) \in [d_{max} + 1]$. □

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*:

```
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 ∈ V;
Start breadth-first search from v_0 to obtain a vertex order v_1, v_2, ..., v_{n-1};
foreach i = 0 to n − 1 do
 |  f(v_i) = min {c ∈ ℕ : c ≠ f(v_j)  ∀j < i s.t. {v_i, v_j} ∈ E}.
return f
```

Example:



Theorem 4.2. *If G is a connected 2-colorable graph, then* `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 `BFSColoring`(G). We argue by (strong) induction on i that $f(v_i) = f^*(v_i)$ for $i = 0, \ldots, n-1$.

For $i = 0$, we observe that `BFSColoring`(G) sets $f(v_0) = 0$. Now for $i > 0$, we will argue that f^* satisfies the same rule used to construct f, namely:

$$f^*(v_i) = \min \{c \in \mathbb{N} : c \neq f^*(v_j) \ \ \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\}. \tag{1}$$

In other words, the value of f^* at v_i is "forced" by its values at the previously assigned vertices v_j. Since f^* is a valid 2-coloring, the value $c = f^*(v_i)$ satisfies the condition $c \neq f^*(v_j)$ for all $j < i$ such that $\{v_i, v_j\} \in E$ automatically holds. If $f^*(v_i) = 0$, then it is certainly the minimum value of c satisfying this condition. If $f^*(v_i) = 1$, we note that that by the definition of BFS, there is a previous vertex v_j (with $j < i$) with an edge to v_i. Since f^* is a valid 2-coloring, we must have $f^*(v_j) = 0$. So $c = 0$, does not satisfy the condition in Equation (1), and hence $c = 1$ must be the minimum value satisfying the condition.

By the definition of `BFSColoring`(G), we have

$$f(v_i) = \min \{c \in \mathbb{N} : c \neq f(v_j) \ \ \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\} \tag{2}$$

By our (strong) induction hypothesis, the right-hand sides of (1) and (2) are equal, and thus $f(v_i) = f^*(v_i)$. □

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

Proof. We can partition G into connected components in time $O(n+m)$. Then, for each connected component we can use BFSColoring on each component, which takes total time $O(n + m)$. □