CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 11: Graph Coloring

Harvard SEAS - Fall 2024 *Oct. 8, 2024*

1 Announcements

- Midterm Oct 17 (practice problems on Ed).
- Midterm material up to Lecture 9.
- Survey from WiCS (deadline Oct 18) - see Ed post by Salil
- Pset 4 due Wed.
- Pset 5 released next week.

Recommended Reading:

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

2 Motivating graph coloring: register allocation

In Theorem 5.1 of Lecture 7, we saw a way to simulate (Word-)RAM programs that used many variables, with a (Word-)RAM program that used a fixed number c of variables (or registers).[1] This fact is useful because compilers generate code with a huge number of variables, which then needs to be converted into code that can run on our CPUs, which have only a fixed number c of registers. However, the approach we used for that simulation—swapping variables in and out of memory—leads to a big slowdown in practice, since reading and writing from memory is much slower than carrying out operations on values that are already held in CPU registers. We can do much better by exploiting the fact that most of the variables generated by compilers are *temporary* variables, whose value only needs to be maintained for a short time. This allows for the possibility that we can reuse the same register to represent many different variables.

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.

[1] We presented that theorem for RAM programs, but the same result holds for Word-RAM programs.

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

Algorithm 1: Toy Word-RAM program

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

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

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

- Vertices = a subset of the variables of P (other than `input_len`, `output_len`, `output_ptr`, `word_len`) for which we want to do register allocation (e.g. the 'temporary' variables created during compilation)
- Edges = $\{\{(\mathtt{var}, \mathtt{var}')\} : R_{\mathtt{var}} \cap R_{\mathtt{var}'} \neq \emptyset\}$.

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

Definition 2.1. For an undirected graph $G = (V, E)$, a (proper[2]) *k-coloring* of G is a mapping $f : V \to [k]$ such that for all edges $\{u, v\} \in E$, we have $f(u) \neq f(v)$.

The computational problem of finding a coloring is called the Graph Coloring problem:

[2]An *improper* coloring allows us to assign the same color to vertices that share an edge, but for us 'coloring' will always mean 'proper coloring' unless we explicitly state otherwise.

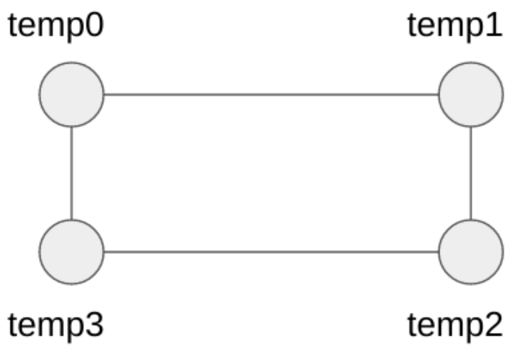

Figure 1: Conflict graph for the Toy Word-RAM program

```
Input    : A graph $G = (V, E)$ and a 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) $\texttt{reg}_{f(\texttt{var})}$, thereby using only the k variables $\texttt{reg}_0, \texttt{reg}_1, \ldots, \texttt{reg}_{k-1}$ in our new (but equivalent) program.

Looking at the conflict graph for our Toy Word-RAM program above, a proper 2-coloring assigns $\texttt{temp}_0$ and $\texttt{temp}_2$ the color 0 and $\texttt{temp}_1$ and $\texttt{temp}_3$ the color 1. This gives us the following new program after register allocation:

```
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 : input_len, output_len, output_ptr, word_len, $\texttt{reg}_0$, $\texttt{reg}_1$
0  output_ptr = input_len;
1  output_len = 1;
2  $\texttt{reg}_1 = 0$;
3      IF input_len == 0 GOTO 15;
4      $\texttt{reg}_0 = 1$;
5      $\texttt{reg}_0 = \texttt{reg}_0 + \texttt{reg}_1$;
6      input_len = input_len $-$ $\texttt{reg}_0$;
7      $\texttt{reg}_1 = M$[input_len];
8      $\texttt{reg}_1 = \texttt{reg}_1 + \texttt{reg}_0$;
9      $\texttt{reg}_1 = \texttt{reg}_1 \times \texttt{reg}_1$;
10     $\texttt{reg}_0 = M$[output_ptr];
11     $\texttt{reg}_0 = \texttt{reg}_0 + \texttt{reg}_1$;
12     $\texttt{reg}_1 = 0$;
13     $M$[output_ptr] = $\texttt{reg}_0$;
14     IF $\texttt{reg}_1$ == 0 GOTO 3;
15 HALT ;                                    /* not an actual command */
```

Algorithm 2: Toy Word-RAM program after register allocation

Remark: A variant of the Graph Coloring problem is to find a proper coloring using as *few* colors as possible, for a given a graph G. This problem is opposite of a problem we recently looked at: 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.

3 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):

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

An example of this algorithm is depicted in Figure 2.

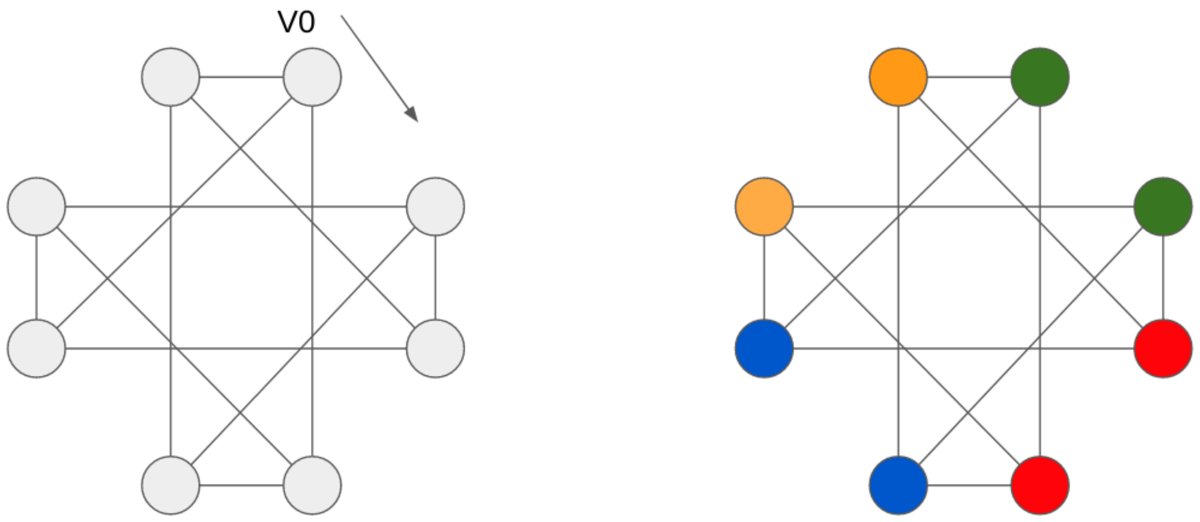

Figure 2: `GreedyColoring` algorithm on the graph (left hand side), with the ordering of vertices in clockwise direction starting from v_0, produces a 4-coloring (right hand side).

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

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

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. Note that while the BFS algorithm in Lecture 10 was for directed graphs, an undirected graph can be viewed as a special case of a directed graph in which $\{u, v\} \in E$ if and only if $\{v, u\} \in E$.

```
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 $v_1, v_2, \ldots, v_{n-1}$;
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 $f$
```

Running the `BFSColoring` algorithm on the same example as in Figure 2, we obtain the coloring in Figure 3.

Now, we show that for 2-colorable graphs, `BFSColoring` finds a 2-coloring efficiently.

Theorem 3.2. *If G is a connected 2-colorable graph, then* `BFSColoring` *(G) will color G using 2 colors.*

Proof. Since G is 2-colorable, fix one such coloring $f^* : V \to \{0, 1\}$. Let W be the set of vertices that are assigned the color 0 - that is $W = \{v \in V : f^*(v) = 0\}$ - and W' be the set of vertices that are assigned the color 1. Line 2 of the `BFSColoring` algorithm starts with picking up an arbitrary vertex v_0. We may assume that $f^*(v_0) = 0$ without loss of generality, that is $v_0 \in W$. Note that are no edges between the vertices in W and similarly no edges between the vertices in W', by the definition of 2-coloring (see the example depicted in Figure 4).

Let f be the coloring returned by the algorithm in Line 6. Let F_d be the set of frontier vertices in d-th iteration of the BFS algorithm (Algorithm 1 from Lecture 10). In Line 3, the vertex order from BFS is as follows: $F_0 = \{v_0\}$, followed by all the vertices in F_1, followed by all the vertices in F_2 and so on. We will show the following statement:

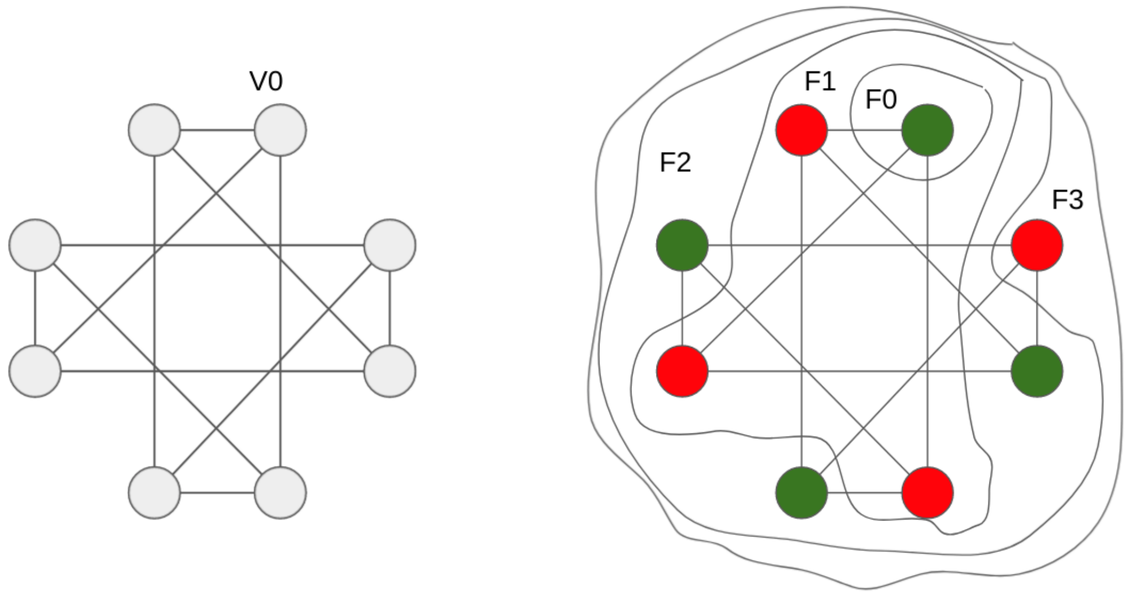

Figure 3: `BFSColoring` algorithm on the graph (left hand side) produces a 2-coloring (right hand side).

"(*)*For even d, $F_d \subseteq W$ and for odd d, $F_d \subseteq W'$. Furthermore, Lines 4-5 of* `BFSColoring` *ensure that $f(v) = 0$ for a vertex $v \in F_d$ with even d and $f(v) = 1$ for a vertex $v \in F_d$ with odd d.*"

The proof will complete once (*) is established for all d, since the frontier vertices $F_0, F_1, \ldots$ are disjoint and every vertex in v belongs to some frontier vertex in a connected graph. Then (*) will ensure that a proper 2-coloring is achieved - in fact, f and f^* are the same coloring.

To prove (*), we use strong induction. Base case with $d = 0$ holds since $F_0 = \{v_0\} \subset W$ and $f(v_0) = 0$. Assume that the statement holds for all $d' \leq d$; we will show that it also holds for $d+1$. Suppose d is even. Then for $d+1$, note that every vertex v in F_{d+1} has an edge with some vertex in $F_d \in W$ (this follows from the way the BFS algorithm chooses F_{d+1}). The vertex v must belong to W' (it can't be in W as no two vertices in W share an edge). Thus, $F_{d+1} \subseteq W'$. Finally, we argue that for any vertex $v \in F_{d+1}$, $f(v) = 1$. Recall from the vertex order that the vertices in F_{d+1} are considered in Lines 4-5 after the vertices in $F_0 \cup F_1 \cup \ldots F_d$ are assigned a color. When Line 5 is executed for the vertex v, $f(v)$ is chosen to be the smallest c that is not assigned to any of its neighbours (among the vertices that have been assigned a color so far). However, such neighbours of v only belong to $F_0 \cup F_2 \cup \ldots F_{d-2} \cup F_d$, which are assigned 0 by the induction hypothesis. Thus, $f(v) = 1$ for any vertex $v \in F_{d+1}$. The argument for odd d holds similarly.

□

Corollary 3.3. *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 $O(n + m)$. □

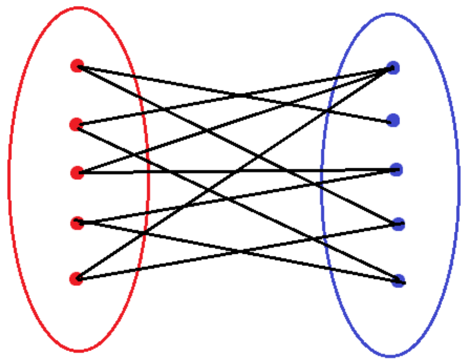

Figure 4: The vertices of a 2-colorable graph can be divided into two parts such that there are no edges within each part. They are also called bipartite graphs. As the BFS algorithm is run on such a graph, the frontier vertices switch between the two parts. This is the main idea behind Theorem 3.2

4 Formal definition of live regions in a Word-RAM programs

This section is an optional reading in case you are interested.

Definition 4.1. Let $P = (V, C_0, C_1, \dots, C_{\ell-1})$ be a (Word-)RAM program. For a variable $\texttt{var}_0 \in V - \{\texttt{input\_len}, \texttt{output\_len}, \texttt{output\_ptr}, \texttt{word\_len}\}$, 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 $\texttt{var} \notin \{\texttt{var}_0, \texttt{var}_1\}$, or

3. $\texttt{var} = M[\texttt{var}_0]$ with $\texttt{var} \neq \texttt{var}_0$.

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 some access line C_j for `var` (inclusive—so `var` is live at every access line) but with no intervening assign line for `var` between C_i and C_j.[3] The *live region* $R_{\texttt{var}}$ is defined to be the set of lines at which `var` is live.

[3]Note that it can be possible for C_i to be executed before C_j even if $i > j$ 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.