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.
- Survey from WiCS - see Ed post by Salil

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 : $\texttt{input\_len}, \texttt{output\_len}, \texttt{output\_ptr}, \texttt{word\_len}, \texttt{temp}_0, \texttt{temp}_1, \texttt{temp}_2, \texttt{temp}_3$

```
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 Word-RAM program

Live regions for $\texttt{temp}_0, \texttt{temp}_1, \texttt{temp}_2, \texttt{temp}_3$ 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 2.1. For an undirected graph $G = (V, E)$, a (proper[2]) *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 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 : $\texttt{input\_len}, \texttt{output\_len}, \texttt{output\_ptr}, \texttt{word\_len}, \texttt{reg}_0, \texttt{reg}_1$

```
0  output_ptr = input_len;
1  output_len = 1;
2  reg_1 = 0;
3      IF input_len == 0 GOTO 15;
4      reg_0 = 1;
5      reg_0 = reg_0 + reg_1;
6      input_len = input_len − reg_0;
7      reg_1 = M[input_len];
8      reg_1 = reg_1 + reg_0;
9      reg_1 = reg_1 × reg_1;
10     reg_0 = M[output_ptr];
11     reg_0 = reg_0 + reg_1;
12     reg_1 = 0;
13     M[output_ptr] = reg_0;
14     IF 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:

[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.

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, ..., v_{n-1} of V;
3 foreach i = 0 to n - 1 do
4 |  f(v_i) = min {c ∈ ℕ : c ≠ f(v_j)  ∀j < i s.t. {v_i, v_j} ∈ E}.
5 return f
```

An example of this algorithm is depicted in

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.

□

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

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

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 1).

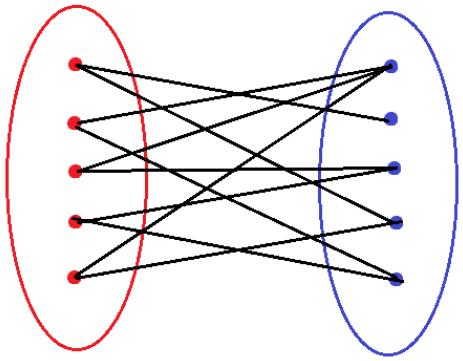

Figure 1: The vertices of a 2-colorable graph can be divided into two parts such that there are no edges within each part. They are also called bipartite graphs. As the BFS algorithm is run on such a graph, the frontier vertices switch between the two parts. This is the main idea behind Theorem 3.2

Let f be the coloring returned by the algorithm in Line 6.

□

Corollary 3.3. *Graph 2-Coloring can be solved in time $O(n + m)$.*

Proof.

□