Graph Coloring and Compilers

Alex Broihier

Outline

Updates!

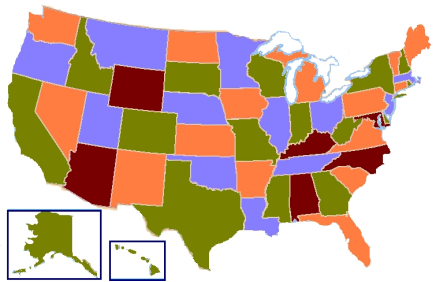
Weekly updates:

- Last meeting on graph theory
- Make meetings

Section 1

Graph Coloring

Map Coloring



- Given some map, how can we color each state / nation?
- We want to color the map such that two bordering states do not have the same color

Formalizing the Map Coloring Problem

- We have a relation between states – they cannot be the same color if they share a border
- We can represent this as a graph
  - $G = (V, E)$
  - V is the set of all states
  - E is the set of edges between any two states which share a border
  - This type of graph, with edges between vertices we do not want to have the same property, is called an interference graph

Formalizing the Map Coloring Problem (Continued)

- With our graph $G = (V, E)$, we want to assign colors to each vertex such that each vertex has a different color than all of its neighbors
- Question: how many colors do we need to color the map / graph?
  - The minimum number of colors we need is the graph's **chromatic number** (sometimes written as $\chi(G)$)
  - A graph is k-colorable if you can color it with k colors
- Furthermore, how can we assign colors optimally?

Graph Coloring Algorithm

- One easy recursive algorithm to optimally color a graph (and find its chromatic number) comes to mind:
  - For $k = 1, 2, \ldots$, assign colors to the vertices in all possible ways
  - If we find an arrangement that satisfies k-colorability, return k
  - Otherwise move on to the next k

Graph Coloring Algorithm Analysis

- We have an exponential time complexity algorithm
- Can we do any faster (that is not still exponential time complexity)?
- Probably not! Graph coloring is an NP-Complete problem
  - In fact, determining whether a graph is 3-colorable is NP-Complete
  - Determining whether a graph is 2-colorable can be done in $O(|V| + |E|)$ time

Other Graph Coloring Results

- Kenneth Appel and Wolfgang Haken proved in 1976 at the University of Illinois at Urbana-Champaign that any map can be colored using at most 4 colors
- Arbitrary graphs are much more general than maps and can have arbitrary large chromatic numbers

Questions?

Section 2

Compilers and Register Allocation

A More Applicable Problem

- Graph coloring is boring and not applicable to most programming tasks
- Compilers are very applicable to programming tasks
  - A compiler translates source code into a different code representation; often we consider compilers that translate to assembly / executable machine code
  - Compilers provide many abstractions that have become common place in code (classes, functions / closures, pass by reference, ...)
  - Compilers can outperform humans at writing assembly / machine code
  - Compilers allow you to easily target different architectures with different assembly languages

Registers

- Compilers that create machine code have to deal with registers
- A register is a part of the processor that can store a set amount of bits (ex: a 32 bit register)
- The CPU typically performs operations (like addition) on values stored in registers
- Problem: there are a very limited amount of registers on a processor

Introduction of Our Driving Example

```
FUNC:
    int a = 1;
    int b = 2;
    int c = a;

    while b ≠ 0 {
        b = b - 1;
    }

    int d = a + b;
    c = c + d;

    return c;
```

```
FUNC:
    $a = 1;
    $b = 2;
    $c = $a;

    while $b ≠ 0 {
        $b = $b - 1;
    }

    $d = $a + $b;
    $c = $c + $d;

    $r = $c;
    ret;
```

Registers and Main Memory

- We can also store values in main memory
- You can fetch a value from main memory into a register and later write another value back to main memory

A Basic Approach to Register Allocation

- Given we have a limited number of registers, store each value in main memory
- For each operation $c = a + b$, first fetch a and b from main memory into arbitrary registers $\$r$ and $\$s$
- Perform addition on $\$r$ and $\$s$ and store the result in $\$r$
- Write $\$r$ to c in main memory

Basic Approach Example

```
FUNC:
    int a = 1;
    int b = 2;
    int c = a;

    while b ≠ 0 {
        b = b - 1;
    }

    int d = a + b;
    c = c + d;

    return c;
```

```
FUNC:
    $a = 1;
    $b = 2;
    $c = $a;

    while $b ≠ 0 {
        $b = $b - 1;
    }

    $d = $a + $b;
    $c = $c + $d;

    $r = $c;
    ret;
```

```
FUNC:
    $r = 1;
    Mem[$sp - 4] = $r;
    $r = 2;
    Mem[$sp - 8] = $r;
    $r = Mem[$sp - 4];
    Mem[$sp - 12] = $r

    ⋮

    $r = Mem[$sp - 12];
    ret;
```

Questions?

A Basic Approach: Problems

- We may be continuously fetching / writing the same few values from main memory
- Reading / writing to main memory is significantly slower than using registers
- Goal: access main memory as infrequently as possible and use registers as much as possible
  - This is the problem of register allocation – how do we assign registers to variables?
  - Register allocation is an NP-Complete problem
  - We can still write a fast register allocation algorithm that is close to optimal

Better Register Allocation Example

```
FUNC:
    int a = 1;
    int b = 2;
    int c = a;

    while b ≠ 0 {
        b = b - 1;
    }

    int d = a + b;
    c = c + d;

    return c;
```

```
FUNC:
    $a = 1;
    $b = 2;
    $c = $a;

    while $b ≠ 0 {
        $b = $b - 1;
    }

    $d = $a + $b;
    $c = $c + $d;

    $r = $c;
    ret;
```

```
FUNC:
    $a = 1;
    $b = 2;
    $c = $a;

    while $b ≠ 0 {
        $b = $b - 1;
    }

    $b = $a + $b;
    $c = $c + $b;

    $r = $c;
    ret;
```

Better Register Allocation Example

```
FUNC:
  $a = 1;
  $b = 2;

  while $b ≠ 0 {
      $b = $b - 1;
  }

  $b = $a + $b;
  $a = $a + $b;

  $r = $a;
  ret;
```

```
FUNC:
  $r = 1;
  $b = 2;

  while $b ≠ 0 {
      $b = $b - 1;
  }

  $b = $a + $b;
  $r = $r + $b;

  ret;
```

```
FUNC:
  $r = 2;
  ret;
```

Questions?

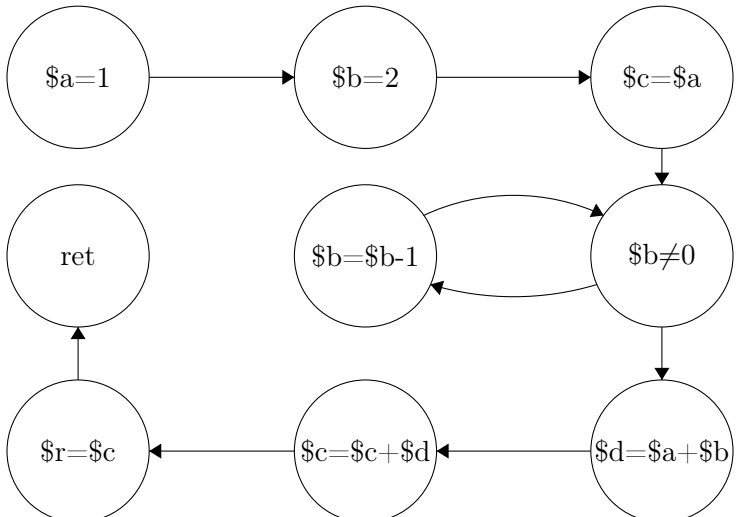
Section 3

Dataflow Analysis

Improved Register Allocation: First Steps

- We say a variable is live if it has been defined and will be used later in the program
- Two variables can not use the same register if they are live at the same point in time
- We analyze liveness in a control flow graph
  - A control flow graph is a graph of instructions with directed edges from an instruction to any instructions that immediately follow it

Control Flow Graph Example



Liveness Analysis

- To determine the liveness of a variable, from its last use run a (backwards) depth first search
- If a node defines the variable, stop searching from that node, but continue searching other paths per depth first search
- The edges traversed are where the variable is live
- Again: two variables can be assigned to the same register if they are not live at the same point in time
- We construct an interference graph from the liveness data: two variables have an edge between them if they are live at the same time

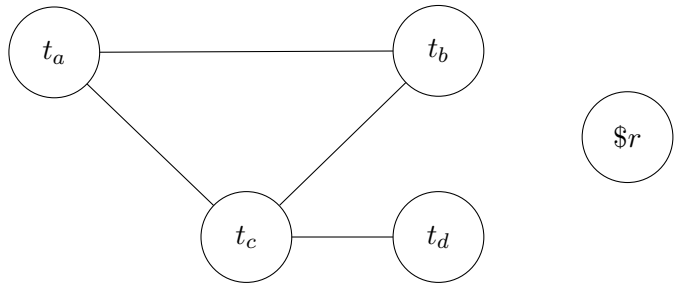
Interference Graph of the Code Example

```
FUNC:
  $a = 1;
  $b = 2;
  $c = $a;

  while $b ≠ 0 {
      $b = $b - 1;
  }

  $d = $a + $b;
  $c = $c + $d;

  $r = $c;
  ret;
```



Dataflow Analysis in Compilers

- Liveness analysis is a form of dataflow anaylsis
- Dataflow analyses are used everywhere in compiler optimizations
  - Figure out if one program statement affects a later program statement
  - Detect and eliminate dead code
  - Constant and copy propagation

Questions?

Section 4

Graph Coloring Algorithms

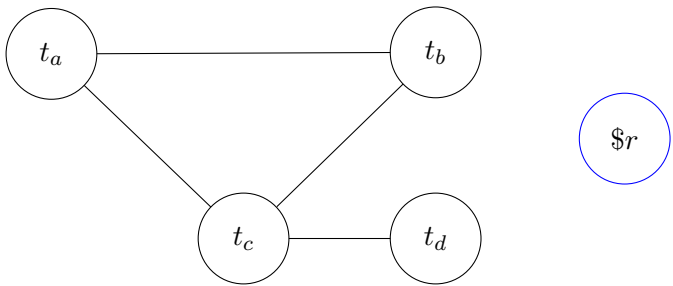
Where we are for Register Allocation

- We have an interference graph representing relationships between variables
- We can allocate registers by "coloring" variables with a specific register out of k registers total
- Register allocation is *nearly* equivalent to graph coloring except that we need store some variables in main memory
- But graph coloring is NP-Complete
  - Imagine that adding one variable to your code doubles the compilation time
  - **Optimal** graph coloring is NP-Complete
  - Let's devise an incorrect graph coloring algorithm that is optimal for some but not all cases

Graph Coloring Approximation: Selection

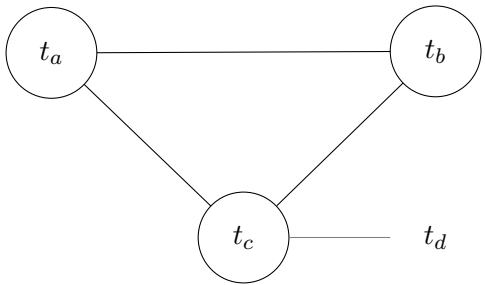
- Given a graph we want to k-color, which vertices can we always assign a color to?
- Any vertex with less than k neighbors
  - Then this vertex is neighboring less than k colors
- Idea: we can remove vertices with less than K neighbors, add them to a stack, and repeat the process on the reduced graph
- We call this the selection phase
- Some vertices (known registers) start with their corresponding color and are not involved in selection

Selection Example



Selection Example

Stack:

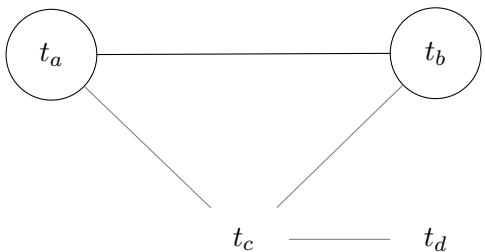


Graph Coloring Approximation: Spilling

- What if we end up with a graph where all vertices have at least k neighbors?
  - Can the graph be k-colorable?
  - If not, can we salvage the graph and k-color most of the vertices?
- Pick an arbitrary vertex. Mark it as "spillable," remove it from the graph, add it to the stack, then try to simplify further
  - "Spillable" means it may "spill" into main memory

Spilling Example

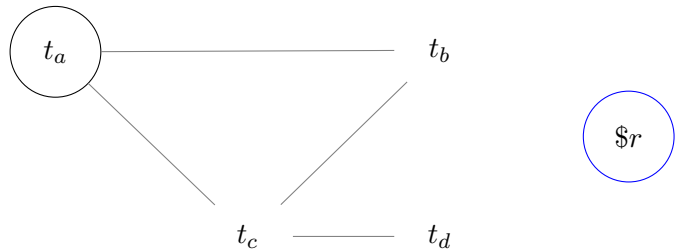
Stack:



Spilling Example

Stack:

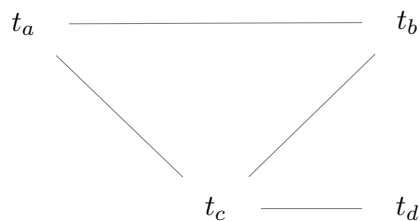
$$\begin{bmatrix} \\ t_b \\ t_c \\ t_d \end{bmatrix}$$



Spilling Example

Stack:

$$\begin{bmatrix} t_a \\ t_b \\ t_c \\ t_d \end{bmatrix}$$



Graph Coloring Approximation: Selection

- Now we have a stack of vertices, some of which may potentially spill (had $\geq k$ neighbors when we removed them)
- Let's reconstruct the original graph, but now with color!
- There are three cases per vertex we try to re-add to the graph

Graph Coloring Approximation: Selection Cases

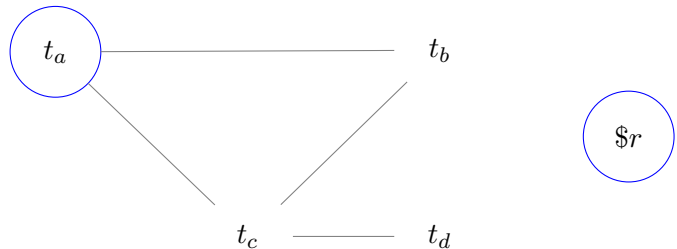
Vertex was not marked as "spillable"

- Then we know there is a valid color we can assign it; add it back to the graph and color it

Selection Case 1

Stack:

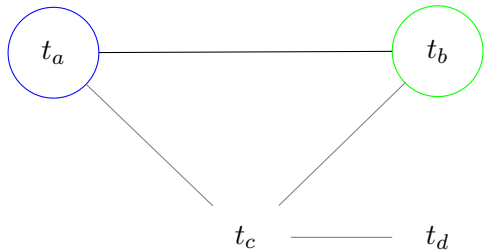
$$\begin{bmatrix} \\ t_b \\ t_c \\ t_d \end{bmatrix}$$



Selection Case 1

Stack:

$$\begin{bmatrix} \\ \\ t_c \\ t_d \end{bmatrix}$$



$\$r$

Graph Coloring Approximation: Selection Cases (Continued)

Vertex was marked as "spillable" but has a valid coloring

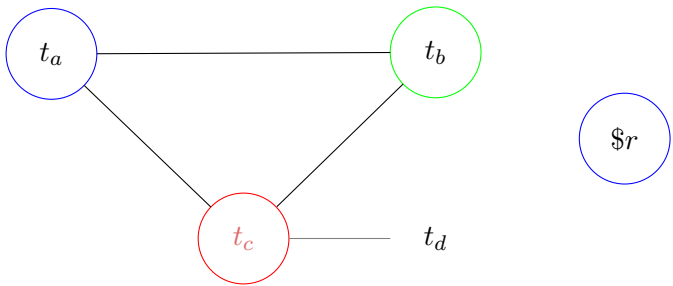
- Then re-add the vertex and assign it a valid color
- This will not impact other vertices not marked as "spillable"

Graph Coloring Approximation: Selection Cases (Continued)

- Vertex was marked as "spillable" and does not have a valid coloring
  - We will generate instructions to store/fetch this variable to/from main memory

Selection Cases 2 and 3

Stack:



Graph Coloring Approximation: Spilled Nodes

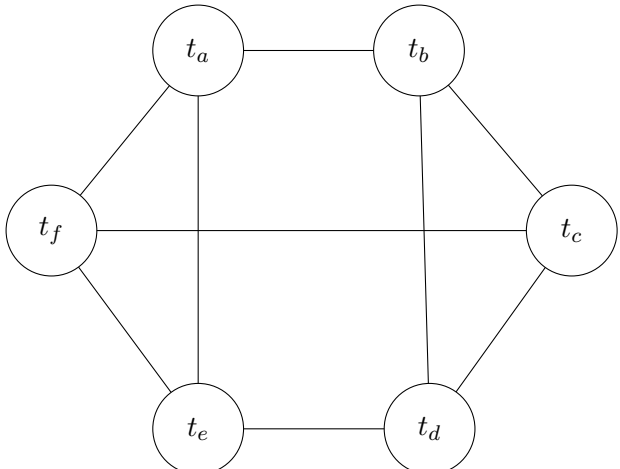
- Can we continue running the algorithm?
- No! The instructions to store/fetch the node to/from main memory use temporary variables
- We rebuild the interference graph and rerun the graph coloring algorithm
- Problem: we introduced new temporary variables, what if the new graph can't be colored?
- The new temporary variables have a very short time they are live; we will eventually converge on a k-coloring with spilling

Questions?

Graph Coloring Approximation Optimality

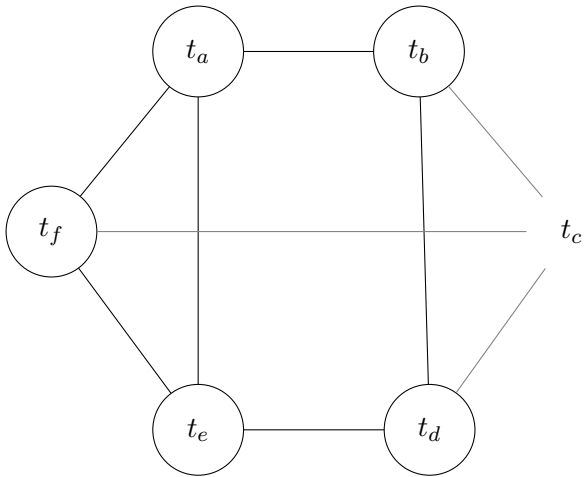
- This algorithm takes $O(n)$ time in the best case, $O(n^2)$ time in the worst case
- This graph coloring algorithm seems to work well, but is **not** optimal

Optimality Counter Example



Optimality Counter Example

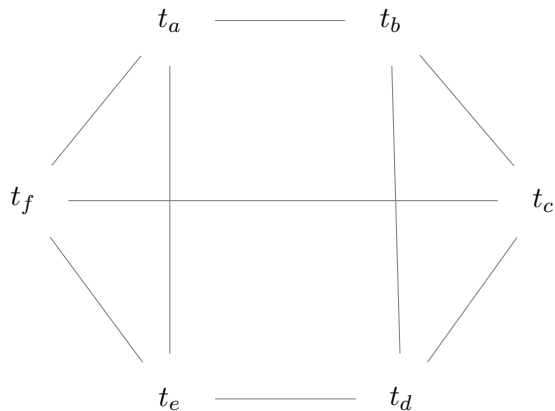
Stack:



Optimality Counter Example

Stack:

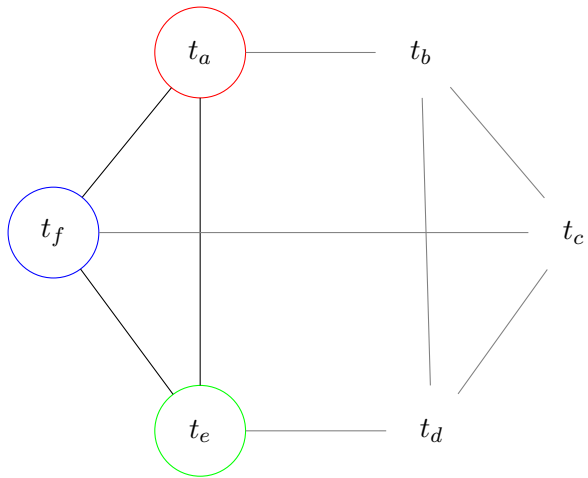
$$\begin{bmatrix} t_f \\ t_e \\ t_a \\ t_d \\ t_b \\ t_c \end{bmatrix}$$



Optimality Counter Example

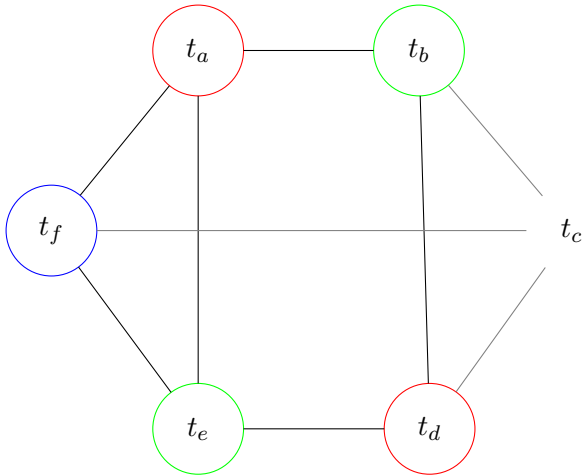
Stack:

$$\begin{bmatrix} \\ \\ \\ t_d \\ t_b \\ t_c \end{bmatrix}$$

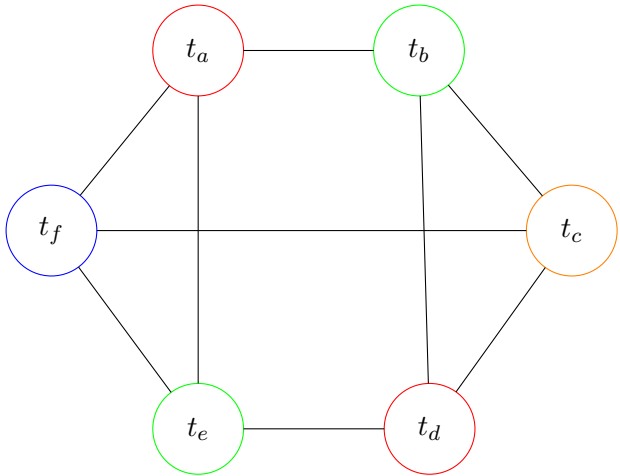


Optimality Counter Example

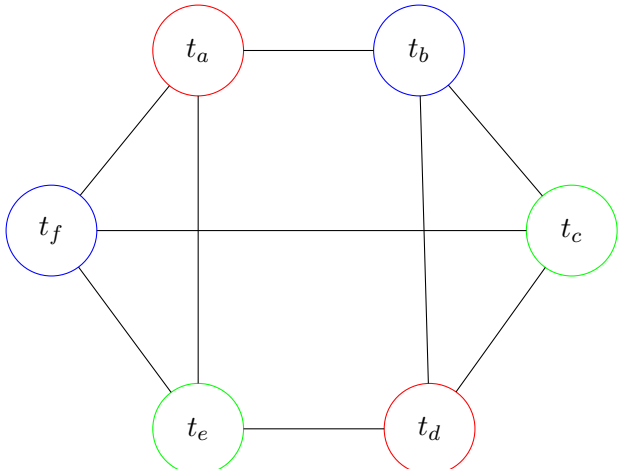
Stack:



Optimality Counter Example



Optimal Solution



Register Allocation: Further Optimizations

- Many instructions of the form of $t_a = t_b$ can be eliminated such that t_a shares the same register or main memory address as t_b – this is move coalescing
- Allows us to do this

```
FUNC:
    $a = 1;
    $b = 2;
    ⋮
    $a = $a + $b;

    $r = $a;
    ret;
```

```
FUNC:
    $r = 1;
    $b = 2;
    ⋮
    $r = $r + $b;

    ret;
```

Register Allocation: Further Optimizations (Continued)

- We can also enforce more strict rules for creating edges in the interference graph after dataflow analysis
- Simplify-Spill-Select turns into Simplify-Coalesce-Freeze-Spill-Select
- Many places where we can introduce heuristics into the graph coloring algorithm
- If we have to restart the algorithm due to spilling, we can reuse a lot of work
- Optimal register allocation is possible for expression trees

Section 5

Conclusion

Takeaways

- Graph coloring is an NP-Complete problem
- Graph coloring is vitally important to optimizing compilers
- Given compilers implement a lot of graph algorithms, if P=NP we could potentially make a lot more optimizations
- However, heuristic approximations to NP hard graph problems already yield an optimal or close to optimal solution in most cases

Questions?

Brainteaser

A plane with 100 seats is to be boarded by 100 passengers, each with an assigned seat. The first person to board forgot their assignment, so will choose any seat at random. The remaining passengers all sit in their assigned seat, unless it is taken, in which case they chose one of the remaining seats at random. What is the probability the last passenger will sit in their assigned seat.

Bibliography I

Andrew Appel.

Modern Compiler Implementation in ML.

Cambridge University Press, 2004.

Wikipedia contributors.

Graph coloring.

August 2024.

Accessed: 09-22-2024.