SSA-Based Register Allocation

What is Register Allocation?

Register allocation is the process of mapping the variables that a program manipulates to physical locations, either registers or memory. Memory is abundant but significantly slower than registers. Registers, in contrast, are few in number—modern architectures typically have between 16 and 32 general-purpose registers (e.g., 16 in x86-64 and 32 in RISC-V)—but they provide much faster access. To optimize program performance, it is essential to map frequently used variables to registers whenever possible. The figure below illustrates the impact of register allocation on the binary that the compiler produces for a program:

```
int compute(int n) {
    int a = 1, b = 1, c = 2, d = 3, e = 5;
    int f = 8, g = 13, h = 21, i = 34, j = 55;

    for (int k = 0; k < n; ++k) {
        // Fibonacci-like series update
        int next = a + b + c + d + e + f + g + h + i + j;
        a = b + c;
        b = c + d;
        c = d + e;
        d = e + f;
        e = f + g;
        f = g + h;
        g = h + i;
        h = i + j;
        i = j + next;
        j = next; // Keep all variables live
    }

    // Combine results to avoid dead code elimination
    return a + b + c + d + e + f + g + h + i + j;
}
```

```
; without register allocation
LBB0_3:
  movl    36(%esp), %eax
  addl    32(%esp), %eax
  addl    28(%esp), %eax
  addl    24(%esp), %eax
  addl    20(%esp), %eax
  addl    16(%esp), %eax
  addl    12(%esp), %eax
  addl    8(%esp), %eax
  addl    4(%esp), %eax
  addl    (%esp), %eax
  addl    $60, %esp
  popl    %esi
  popl    %edi
  popl    %ebx
  popl    %ebp
  retl
```

```
; opt -mem2reg fib.ll -o reg.ll
LBB0_3:
  addl    %edi, %eax
  addl    12(%esp), %eax
  addl    16(%esp), %eax
  addl    %ebp, %eax
  addl    %ecx, %eax
  addl    (%esp), %eax
  addl    4(%esp), %eax
  addl    %ebx, %eax
  addl    %esi, %eax
  addl    $36, %esp
  popl    %esi
  popl    %edi
  popl    %ebx
  popl    %ebp
  retl
```

The assembly on the right shows the last basic block of function `compute`, assuming an x86-32 target. We can see that without register allocation, all the local variables (`a`, `b`, `c`, etc) are saved on the stack (e.g., `36(%exp)` is a stack address).

With register allocation, the compiler is able to map some of these variables to registers such as `eax`, `ebx` and `ecx`. There are still some variables that end up mapped onto the stack, because the number of registers is limited, as we see with `12(%esp)`. However, most of the variables are now in fast storage!

This task is more than a simple mapping; it combines several complex decision problems with interesting theoretical foundations:

1. **Register Assignment Problem**
   This problem asks whether it is possible to compile a program using `K` registers such that every variable remains in one or more registers throughout its lifetime. This involves ensuring that the allocation does not exceed the available number of registers at any point in the program.

2. **Spilling Problem**
   When the number of live variables exceeds `K`, some variables must be "*spilled*" to memory. The spilling problem asks whether a program can be compiled using `K` registers and no more than `L` spilled variables, with memory accessed using load and store instructions to retrieve and save their values.

3. **Memory Access Problem**
   Beyond determining which variables to spill, this problem focuses on minimizing the performance cost associated with spilling. Specifically, it asks whether a program can be compiled using `K` registers while keeping the number of memory accesses below given thresholds: `S` store instructions and `L` load instructions.

4. **Register Coalescing Problem**
   During program execution, copy instructions like `v0 := v1` are common, where a variable is assigned the value of another. The goal of register coalescing is to assign both variables to the same register whenever possible, eliminating the need for such copy instructions. The problem asks whether it is possible to compile a program using `K` registers while ensuring that no more than `M` move instructions remain in the final code after allocation.

These problems are interrelated and, in many cases, computationally challenging. Solving them efficiently is critical to generating high-performance code. Modern compilers use a mix of heuristics and algorithmic approaches—often based on graph coloring, linear programming, and other optimization techniques—to tackle these challenges and deliver code that makes the most efficient use of the limited registers available.

Is register allocation an important optimization problem?

Register allocation is one of the most critical optimizations in modern compilers, as it has a profound impact on the performance of the generated code. John Hennessy and David Patterson emphasize this in their seminal work, *Computer Architecture: A Quantitative Approach*, describing register allocation as "*one of the most important—if not the most important—of the optimizations*." This statement underscores the central role of register allocation in achieving efficient, high-performing programs.

Why is Register Allocation So Important?

1. **Performance Impact of Registers vs. Memory**
   Registers are orders of magnitude faster than memory. Accessing a register typically takes a single clock cycle, while accessing memory can take tens to hundreds of cycles, depending on cache performance. By ensuring that frequently accessed variables are kept in registers, register allocation significantly reduces the time spent on memory operations, boosting execution speed.

2. **Enabling Other Optimizations**
   Many compiler optimizations, such as instruction scheduling, loop unrolling, and inlining, are most effective when the variables involved are stored in registers. If variables are spilled to memory, the benefits of these optimizations are often diminished or negated due to the increased cost of accessing spilled variables. Register allocation, therefore, acts as a foundation for amplifying the effectiveness of other optimizations.

3. **Limited Register Resources**
   While registers are fast, they are also a scarce resource, especially in architectures with a small number of general-purpose registers, such as x86-64. This scarcity makes efficient register allocation a challenging combinatorial problem, where the compiler must carefully balance the demands of multiple variables to maximize performance while minimizing spills.

4. **Direct Influence on Code Quality**
   Poor register allocation leads to excessive spilling, resulting in additional memory operations that slow down program execution. Conversely, effective register allocation minimizes memory access, reduces instruction count, and optimizes CPU pipeline utilization, directly contributing to faster and more compact code.

5. **Impact on Power Efficiency**
   Beyond performance, register allocation also affects energy consumption. Memory operations are not only slower but also more power-hungry compared to register accesses. Efficient use of registers thus plays a role in reducing the energy footprint of programs, a critical consideration for mobile and embedded systems.

In summary, register allocation is a keystone of compiler design. Its importance stems not only from the performance gains it directly enables but also from its role in unlocking the potential of other optimizations. As Hennessy and Patterson argue, register allocation is foundational to the pursuit of efficient, optimized code—a goal at the heart of every modern compiler.

What are the typical approaches to solve register allocation?

Probably register allocation is the optimization problem in compiler construction that has inspired the most diverse set of solutions among any other compilation challenge. These solutions are grounded in different mathematical models and strategies, each with unique

trade-offs between efficiency and optimality. Below are some of the methods used to tackle register allocation:

1. Linear Scan Register Allocation

This method is particularly well-suited for just-in-time (JIT) compilers and other scenarios where compilation speed is critical. Linear scan treats register allocation as a coloring problem on interval graphs, where the intervals represent the live ranges of variables. The algorithm assigns registers by scanning live ranges linearly, reusing registers whenever possible. While not as optimal as more complex methods, linear scan is fast and straightforward, making it a popular choice in practical compilers like LLVM and Java's HotSpot VM.

2. Graph-Coloring-Based Allocation

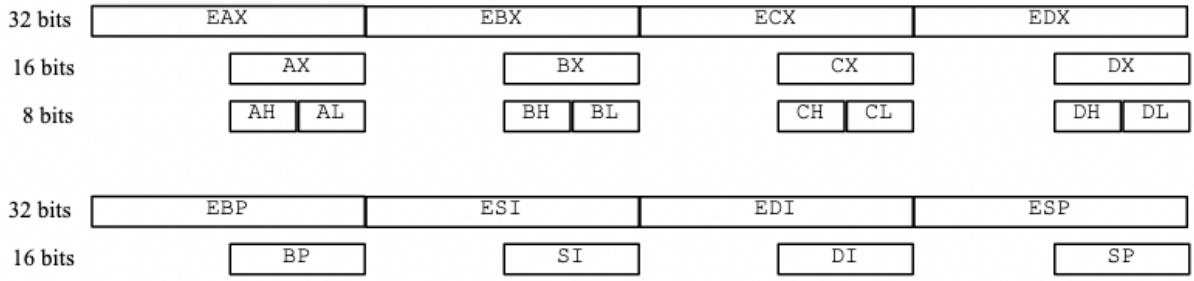
Graph coloring is one of the most well-known approaches to register allocation. Variables' live ranges are represented as vertices in an interference graph, with edges connecting variables whose live ranges overlap. Allocating registers then becomes a graph-coloring problem, where the goal is to assign colors (registers) to vertices without adjacent vertices sharing the same color. This approach, pioneered by Chaitin, remains a cornerstone of register allocation, although it can be computationally expensive due to the NP-hard nature of graph coloring.

3. Integer Linear Programming (ILP) Solutions

ILP-based register allocation formulates the problem as an integer optimization task, where constraints model the relationships between variables, registers, and memory. The objective is to minimize cost functions, such as the number of spills or memory accesses. Although ILP provides optimal solutions, its high computational complexity makes it suitable primarily for scenarios where compilation time is less critical, such as ahead-of-time (AOT) compilation for embedded systems.

4. Register Allocation by Puzzle Solving

This approach conceptualizes register allocation as a puzzle. The live ranges of variables are treated as pieces that must be fit onto a board representing the bank of registers. This method visualizes and resolves allocation challenges as a series of placement decisions, making it particularly appealing for solving complex allocation problems that involve registers with multiple sizes, as in the x86 architecture, where registers such as AH and AL can be combined to form AX, e.g.:



5. Partitioned Boolean Quadratic Programming (PBQP)

PBQP is a combinatorial optimization technique tailored to register allocation. It models the problem as a quadratic programming task where the goal is to minimize a cost function associated with variable assignment. PBQP excels in handling practical constraints like spill costs, coalescing opportunities, and architecture-specific requirements, and it has been implemented successfully in compilers like LLVM.

6. Constraint Solving

Constraint-based solutions, such as the one detailed by Roberto Castaneda Lozano in "Constraint-based Register Allocation and Instruction Scheduling," approach register allocation by encoding the problem as a set of constraints. These constraints capture relationships between live ranges, registers, and memory. The solver then finds an assignment that satisfies all constraints while optimizing performance metrics. This method is particularly versatile and can integrate seamlessly with instruction scheduling for more holistic optimization.

7. Other Notable Approaches

There are, indeed, many other techniques to solve register allocation. More about them can be found in publicly available surveys, some of which we briefly mention below:

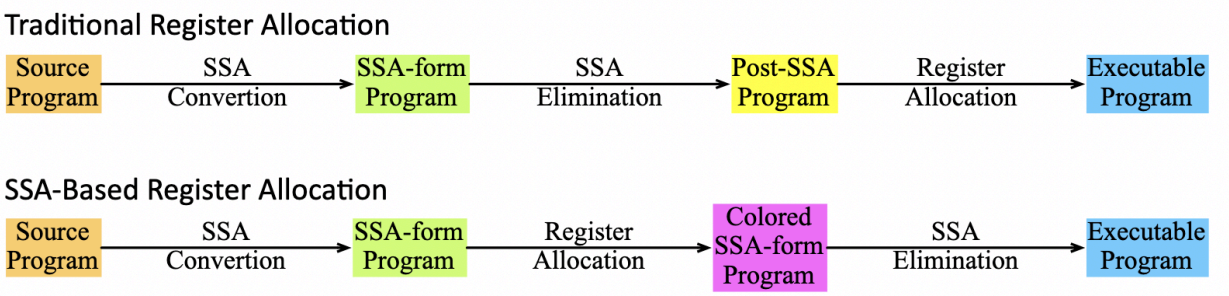
- **Heuristic-Based Allocation**: Some compilers employ heuristics to approximate optimal solutions. These methods are often used in combination with other strategies to achieve a balance between compilation speed and code quality.
- **Machine Learning Techniques**: Emerging approaches use machine learning to predict optimal register assignments based on features extracted from the program and architecture.
- **Simulated Annealing and Genetic Algorithms**: These probabilistic methods explore the search space of register assignments to find near-optimal solutions. While computationally intensive, they are effective for solving highly constrained allocation problems.

A Rich Landscape of Solutions

The diversity of approaches to register allocation reflects its complexity and central importance in compiler design. From theoretical models like graph coloring and ILP to practical methods like linear scan and heuristic-based algorithms, the choice of approach depends on the specific requirements of the compilation process, such as speed, code quality, and target architecture constraints. As research continues, new methods and hybrid solutions will likely emerge, further enriching the toolbox available to compiler developers.

Is register allocation done before or after SSA elimination?

Most compilers adopt **Post-SSA Register Allocation** (also called "Traditional Register Allocation"), performing allocation after SSA elimination, but some explore **Pre-SSA Register Allocation** (also called "SSA-Based Register Allocation"), where allocation happens before eliminating SSA. Each approach has its own advantages, making the choice dependent on the compiler's design goals and the target architecture's characteristics.



Advantages of Post-SSA Register Allocation

1. **Extensive Algorithmic Knowledge**:
   Post-SSA register allocation benefits from a wealth of established research and practical experience.
   - Numerous algorithms for register allocation are described at this level in textbooks.
   - Many high-quality implementations are available in open-source projects and industrial compilers.
   - Developers can leverage this "folk knowledge" for robust and well-understood solutions.
2. **Simpler SSA Elimination**:
   - Performing SSA elimination after register allocation simplifies the elimination process.
   - The algorithm does not need to handle the challenges of mapping registers to the arguments and results of φ functions.
   - Problems like **lost-copy** or **swap issues** (arising when two variables exchange values during SSA elimination) are easier to manage in this approach.

Advantages of Pre-SSA Register Allocation

1. **Polynomial-Time Register Assignment**:
   In SSA form, the interference graph is **chordal**, meaning it has no induced cycles of length greater than three.
   - This property enables efficient, polynomial-time solutions to the register assignment problem when all registers are of similar size.
2. **Shorter Live Ranges**:
   - SSA form inherently shortens variable live ranges, reducing the overlap between live ranges in the interference graph.
   - Shorter live ranges increase the allocator's flexibility, allowing it to map different parts of the same variable more effectively.
3. **Decoupled Allocation and Spilling**:
   - Pre-SSA allocation enables a two-phase strategy:
     - First, spill variables until the program reaches a state where register assignment is feasible.
     - Then, perform register assignment separately, focusing on mapping variables to physical registers without additional spill concerns.

Choosing Between Pre-SSA and Post-SSA Allocation

The decision between pre-SSA and post-SSA allocation depends on the compiler's design priorities:

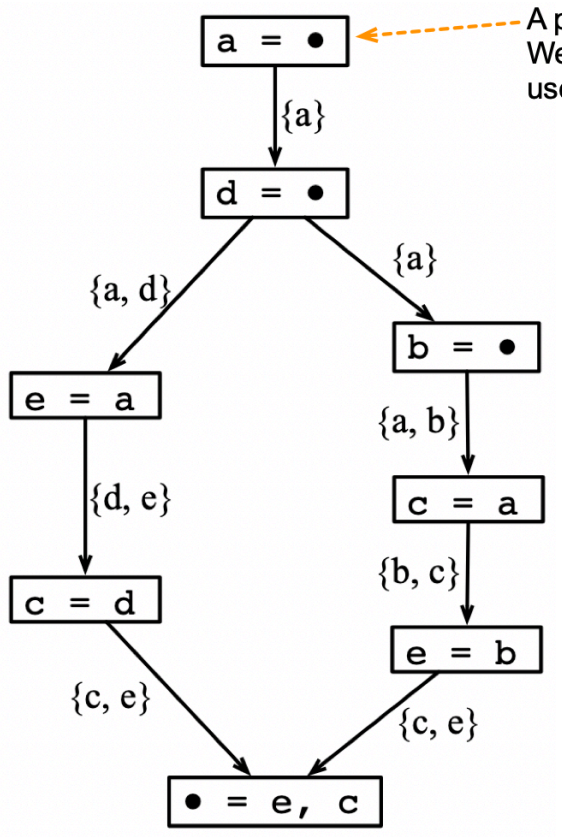
- Post-SSA is more common due to its simplicity and the extensive body of knowledge available.
- Pre-SSA is advantageous in scenarios where the interference graph's chordal nature can be exploited for efficiency or where shorter live ranges are critical.

Both approaches demonstrate the flexibility and adaptability of modern compiler design in addressing the challenges of register allocation.

You mentioned that register assignment has a polynomial-time solution for SSA-form programs. How's that possible?

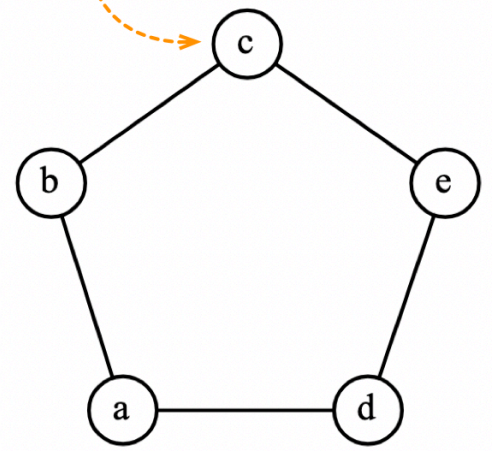
To explain why register assignment for SSA-form programs has a polynomial-time solution, let's break it down step by step:

1. **Defining the Interference Graph**:
   The **interference graph** of a program represents the relationships between variables based on their live ranges:
   - Each vertex in the graph corresponds to a variable.
   - An edge connects two vertices if the corresponding variables have overlapping live ranges (i.e., they are simultaneously "live").

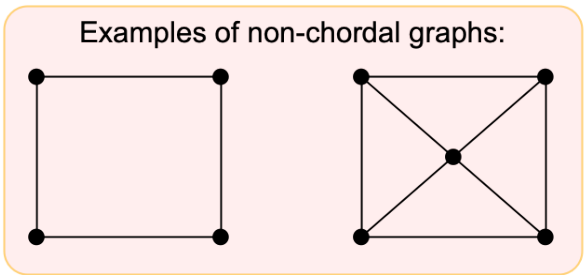


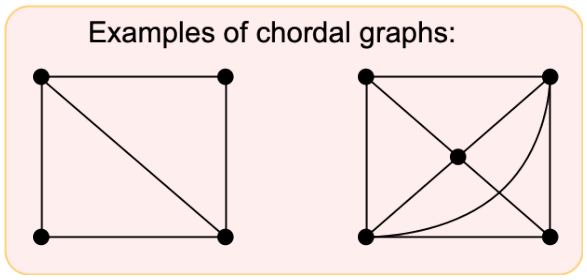
A program formed by nine different instructions. We use the • symbol to imply that a particular use or definition is not important.

The interference graph of this program is a pentagon. This is an interesting graph: its largest click has size two; and yet its chromatic number is three! This graph is not *chordal*.

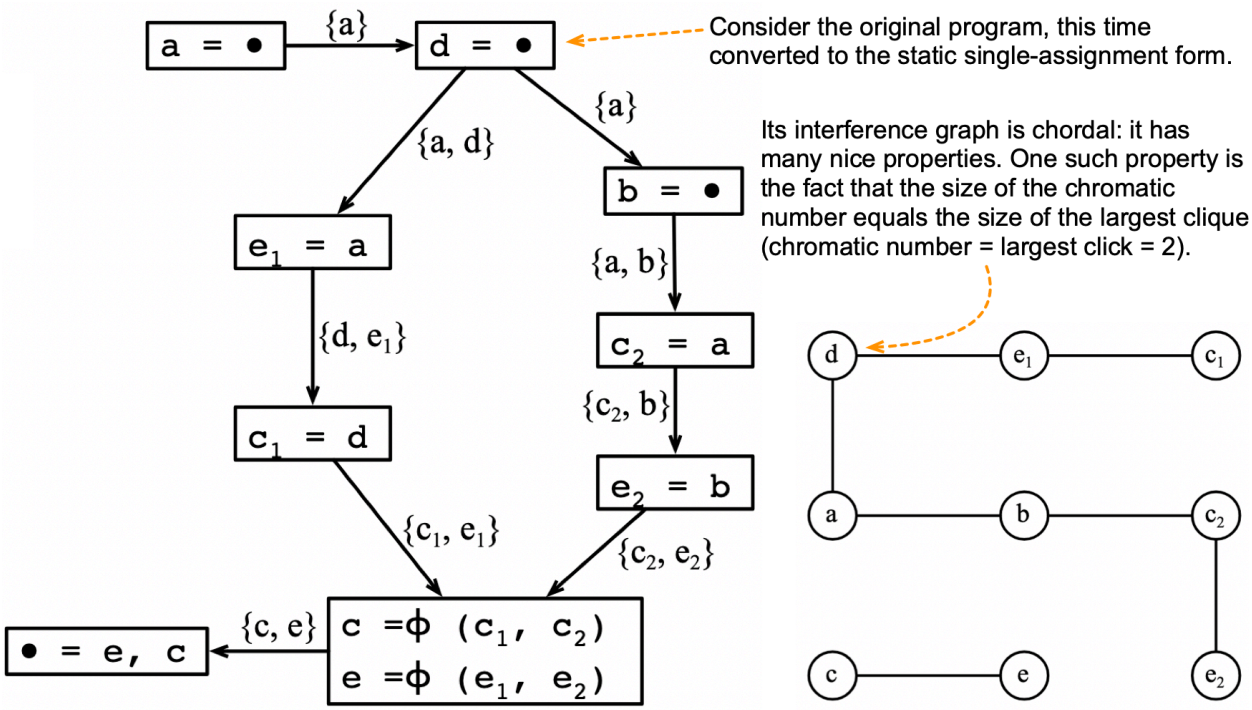


2. **Chordal Property of SSA Interference Graphs**:
   The interference graph of an SSA-form program is chordal.
   - A **chordal graph** is one in which there are no induced cycles of length greater than three.



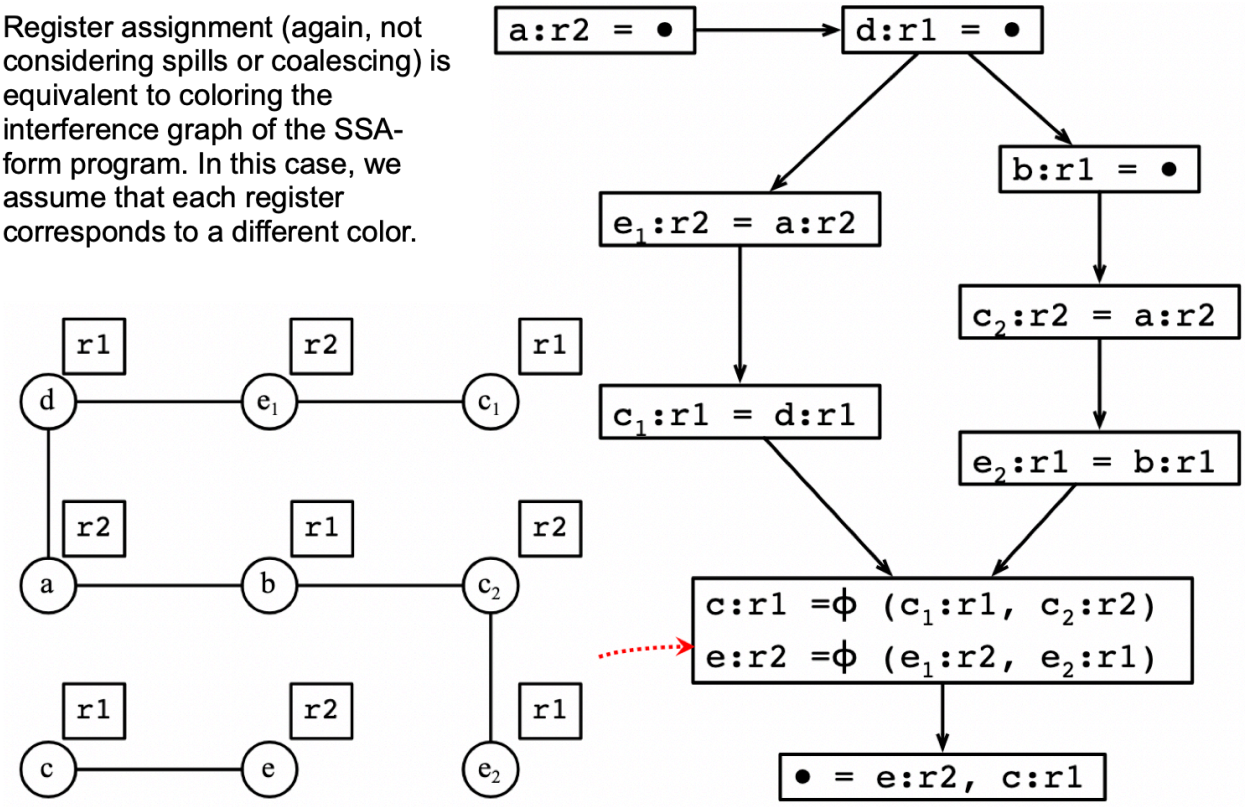


   - An **induced cycle** is a cycle in the graph such that no edges exist between non-consecutive vertices in the cycle.



3. **Efficient Coloring of Chordal Graphs**:
   Chordal graphs have a key property that simplifies the register assignment problem:
   - These graphs can be colored using an algorithm with time complexity O(E), where |E| is the number of edges in the graph.
   - Graph coloring involves assigning "*colors*" (representing registers) to vertices such that no two connected vertices share the same color.
4. **Connection to Register Assignment**:
   The problem of determining whether an SSA-form program can be compiled using K registers (ie, without spilling) corresponds directly to determining whether the interference graph can be colored with K colors:

Register assignment (again, not considering spills or coalescing) is equivalent to coloring the interference graph of the SSA-form program. In this case, we assume that each register corresponds to a different color.



5. **Polynomial-Time Solution**:
   Since graph coloring for chordal graphs has a linear-time solution relative to the number of edges, determining whether an SSA-form program can fit within K registers is efficiently solvable in polynomial time.

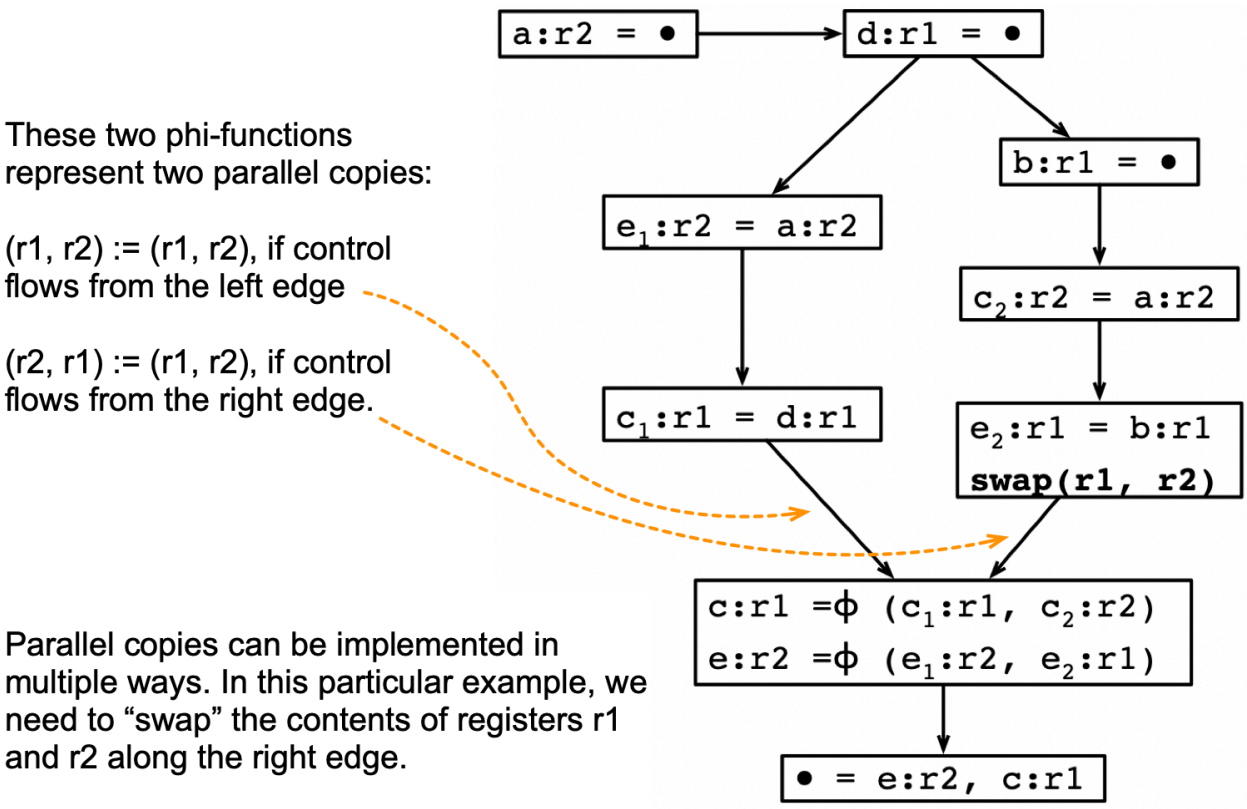
This efficient solution for SSA-form programs makes it a valuable property in register allocation, especially when leveraging the simplicity and structure of SSA. Yet, notice that we are talking only about *register assignment*! Problems like the minimization of spilling or the maximization of coalescing remain NP-complete.

But what do you do with the phi-functions? The phi-related variables are mapped to registers, but we still need to implement these instructions with actual assembly instructions.

Handling φ-functions during SSA-based register allocation is indeed one of the trickiest parts of the whole thing. After register allocation, φ-functions need to be translated into actual assembly instructions, which typically involves implementing **parallel copies**. Here's how this is managed:

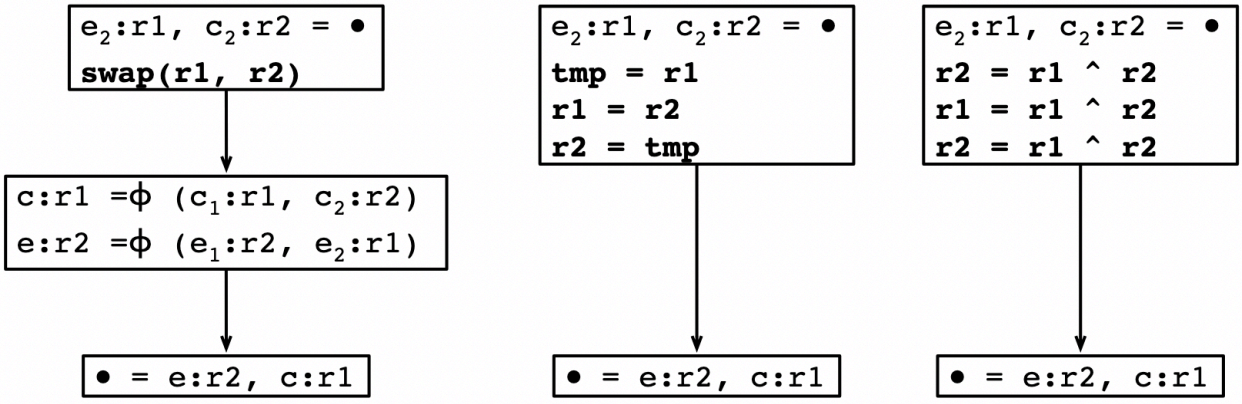
1. **Translating φ-functions to Parallel Copies**:
   A `φ(a, b)` function, for example, means that the value of `a` or `b` should be assigned to a target variable, depending on the control flow path. This requires us to generate **copy**

instructions that move values to their assigned registers. If the parameters of the phi-functions are associated with registers, then we must take these locations into consideration when choosing the right instructions to replace phi-functions:



2. **Algorithms for Efficient Parallel Copies**:
   The process of implementing these parallel copies has been extensively studied. For example, the paper "SSA Elimination After Register Allocation" discusses algorithms to convert φ-functions into a sequence of move instructions while preserving correctness and minimizing additional register usage.

3. **Swapping Register Contents Without Temporaries**:
   A key optimization is to avoid introducing temporary locations (e.g., in memory) when swapping the contents of two registers. Many architectures provide hardware support for this.
   - **Temporary location**: parallel copies can be implemented with a temporary location.
   - **The xor trick**: Three **xor** instructions can swap two register values efficiently without a temporary:
   - **Special Instructions**: Some architectures directly support a `swap` instruction to exchange the values of two registers. For instance, the **x86 architecture** provides a `xchg` instruction for this purpose.



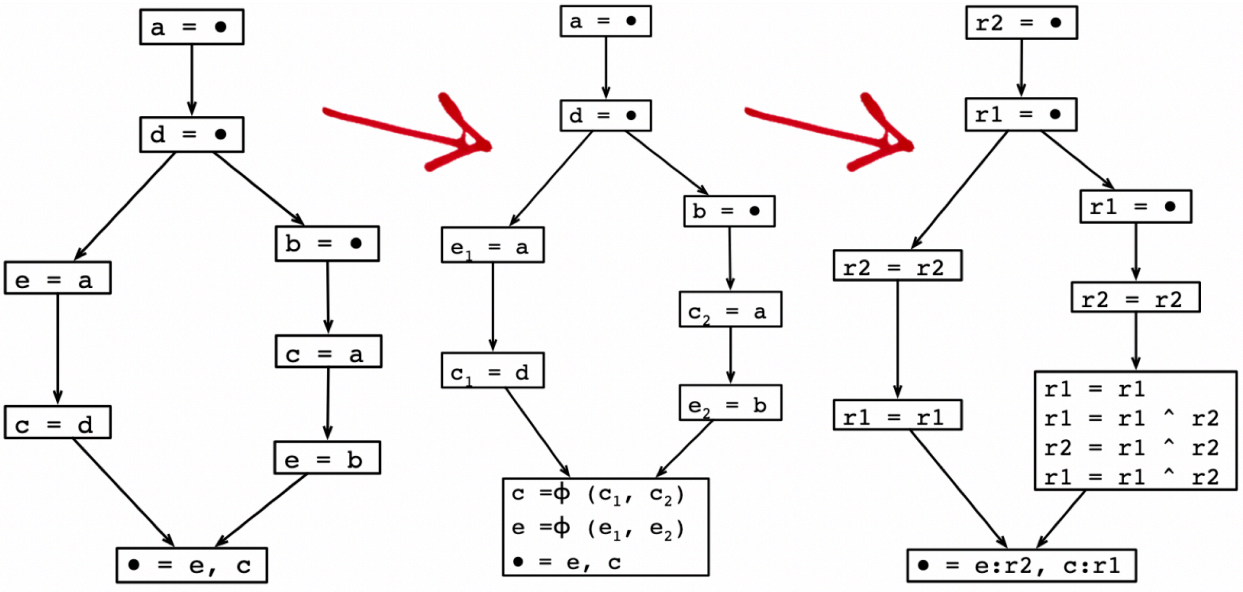
4. **Preserving Register Allocation Results**:
   The goal is to implement these copies without requiring additional registers beyond those assigned during allocation. This ensures the efficiency of the register allocation process is maintained. If the hardware or algorithm cannot guarantee this, a fallback might involve introducing temporary locations. In fact, if swaps are not allowed, then the problem of deciding if a program can be compiled with K registers (without spilling) is NP complete, as seen in the paper [Register Allocation after Classical SSA Elimination is NP-complete].

In practice, handling φ-functions efficiently after register allocation combines clever algorithmic design with low-level architectural features to balance correctness and performance:

Any program can be converted into the static single-assignment format in polynomial time.

An SSA-form program has a chordal interference graph; hence, register assignment can be solved in polynomial time.

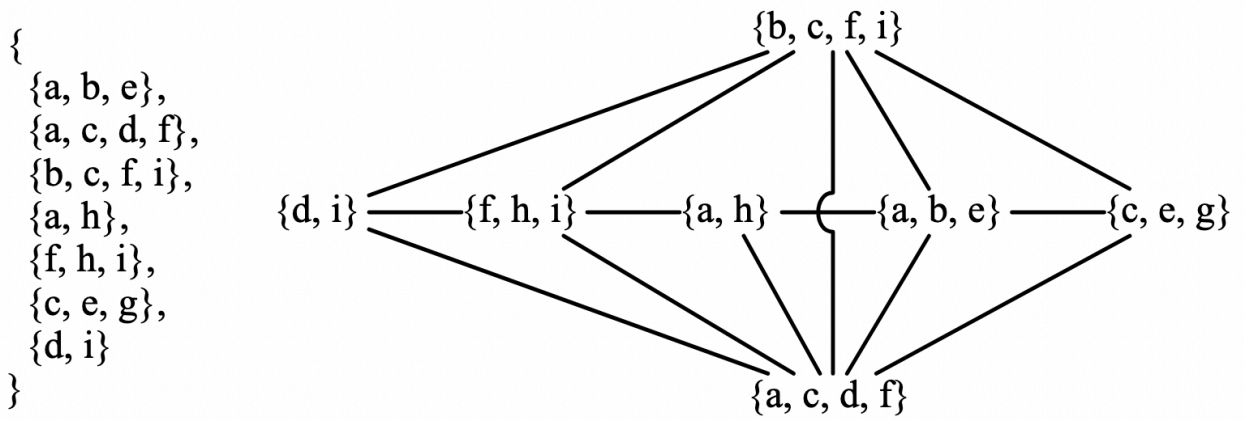
However, eliminating phi-functions becomes more complicated, for we need to implement parallel copies taking the locations of the parameters into consideration.



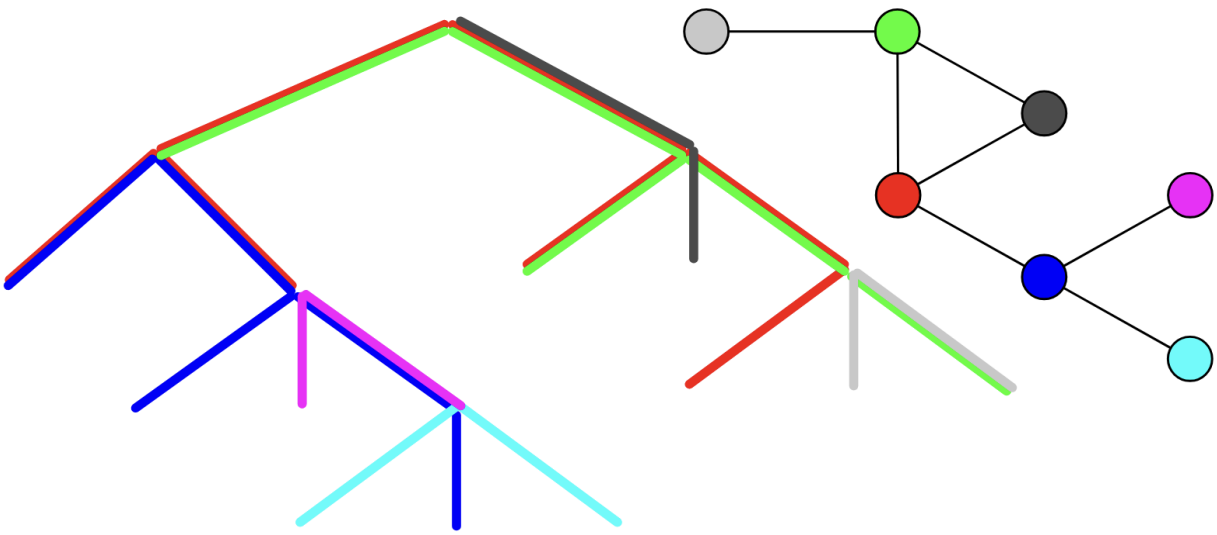
Can you provide some intuition on why the interference graphs of SSA-form programs are chordal?

To understand why the interference graphs of SSA-form programs are chordal, we need to explore the relationship between **live ranges** and **dominance trees** in SSA form:

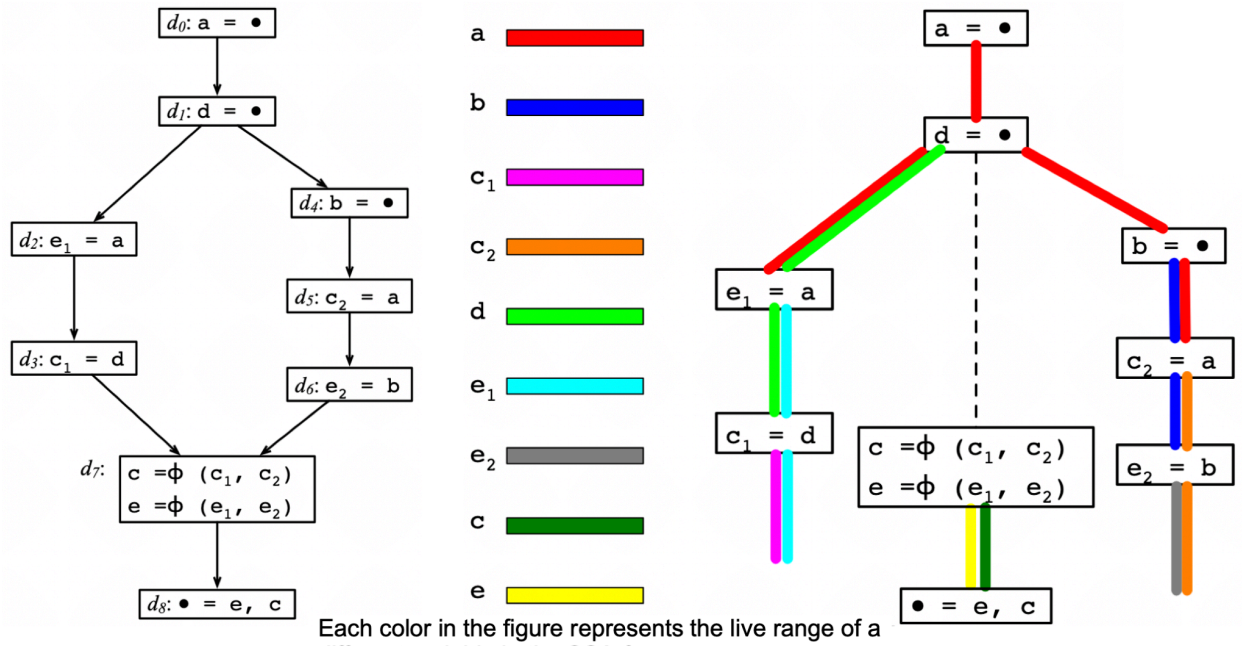
1. **Intersection Graphs**:
   An **intersection graph** is constructed as follows:
   - For a set S of sets, each set $s \in S$ corresponds to a vertex in the graph.
   - An edge exists between two vertices if the corresponding sets intersect, e.g.: $s0 \cap s1 \neq \varnothing$.
   - The figure below shows an example of an intersection graph:



2. **Chordal Graphs**:
   Another way to define chordal graphs is through the **intersection graph of subtrees of a tree**, as demonstrated in the paper "The Intersection Graphs of Subtrees in Trees are Exactly the Chordal Graphs":
   - If S is a set of subtrees of a tree, the intersection graph of these subtrees is chordal.
   - Conversely, if a graph is chordal, then it can be represented as the intersection graph of subtrees in some tree.



3. **SSA Form and Dominance Trees**:
   In SSA form, each variable has a well-defined **live range**, which can be visualized as a connected region of program points where the variable is live. Importantly:
   - The live ranges of variables correspond to subtrees in the **dominance tree** of the program.
   - The dominance tree encodes the control flow hierarchy of the program, ensuring that live ranges respect dominance relationships.
4. **Interference Graph of SSA-form Programs**:
   The interference graph of an SSA-form program represents overlaps between the live ranges of variables:
   - A vertex corresponds to a variable.
   - An edge exists if two variables have overlapping live ranges.

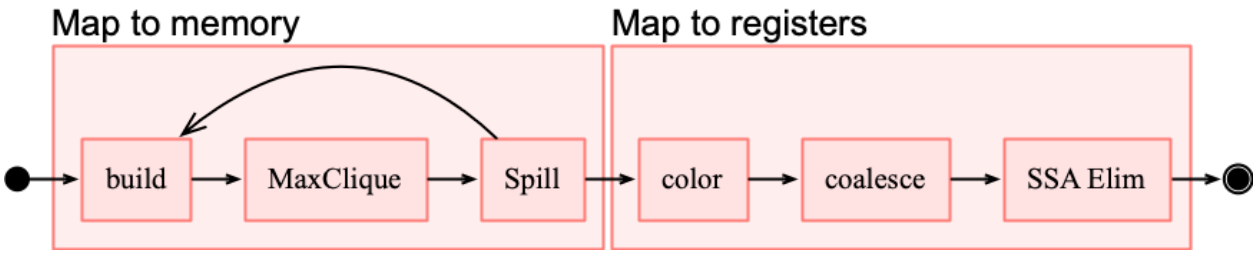


Each color in the figure represents the live range of a different variable in the SSA-form program.

5. Since the interference graph is the intersection graph of live range subtrees on the dominance tree, and the intersection graph of subtrees of a tree is chordal, **the interference graph of an SSA-form program is chordal**.

But how can we use all that to build a register allocator? You mention that there are several problems to be solved...

Let's use the register allocation algorithm described in the paper "Register Allocation via the Coloring of Chordal Graphs". The figure below highlights the key steps of the algorithm.



High-Level View:

This algorithm has been designed to work on chordal interference graphs. It can be applied onto a non-SSA form program, as long as its interference graph is chordal. The algorithm consists of two main phases:

1. **Mapping to memory**: the algorithm decides which variables, if any, will be mapped to memory. This is done by gauging the size of the maximum clique in the interference

graph. Whenever a variable is mapped to memory, it's removed from the program, and accesses to it are replaced with loads and stores.
2. **Mapping to registers**: at this point, the program is guaranteed to be K-colorable, and variables are mapped to "colors" (registers) via a greedy coloring algorithm. Because the maximum clique in the graph has no more than K registers in this phase, spilling is no longer necessary.

Here's a breakdown of how the steps contribute to the overall process:

1. **Build:** The Build stage is where the **interference graph** is created. Imagine each variable as a node in a graph. If two variables' live ranges overlap (meaning they are "alive" at the same time and thus cannot occupy the same register), an edge is drawn between their corresponding nodes. This graph is the core structure for determining which variables can potentially share registers and which cannot.
2. **MaxClique:** Next, the MaxClique stage does a clever analysis to understand the maximum register pressure. It does this by:
    - Using an algorithm called Maximum Cardinality Search to create a special ordering of the nodes.
    - Then, using that order to identify the size of the largest *clique* (a fully connected sub-graph where every node is connected to every other node) in the graph. This size represents the maximum number of variables simultaneously "live" and thus, potentially competing for registers.
3. **Spill:** If the size of the maximum clique found is greater than the number of available registers (K), then the graph is un-colorable in the number of available registers. The Spill step kicks in. This is where the algorithm removes nodes from the graph that prevent the graph from being K-colorable. To "remove" a variable we insert loads/stores, to map it to memory. This introduces the concept of spilling. The algorithm identifies the best nodes to spill, and introduces code to save these variables to memory, freeing up a register. The algorithm continues spilling nodes and re-building the interference graph in a cycle, until a graph is found where the maximum clique is no greater than K.
4. **Color:** Once the maximum clique size is at or below K, the Color stage begins. Here, the algorithm greedily assigns registers to variables in the order determined in the MaxClique step. Since the graph is now chordal and the maximal clique is of size K or less, the greedy coloring algorithm guarantees a coloring using a maximum of K registers.
5. **Coalesce:** The Coalesce stage is an optimization step. It tries to merge the nodes of variables related by copy instructions (like `v := u`). If `v` and `u` are never live at the same time, they can use the same register. This reduces the number of required registers.
6. **SSA Elimination:** The SSA Elimination step happens towards the end and transforms the phi-functions into parallel copy instructions, which can be directly executed in the machine.

Key Points & Insights:

- **Chordal Graph Advantage:** The algorithm relies heavily on the structure of chordal graphs because they have the nice property that optimal coloring can be achieved in polynomial time.
- **Iterative Approach:** The algorithm is iterative due to the spill process. If the initial interference graph is too dense (more live ranges than registers), the algorithm needs to remove variables (through spilling), rebuild, and try again.

How is it possible to run spilling separate from register assignment in chordal graphs?

This is a nice consequence of the fact that the SSA-form programs have chordal interference graphs. Chordal graphs possess several advantageous characteristics for register allocation:

- **Efficient Chromatic Number Determination:** The chromatic number of a chordal graph (the minimum number of colors needed to color the vertices such that no adjacent vertices share the same color) can be determined in polynomial time. As we have already seen, the chromatic number directly corresponds to the minimum number of registers required to allocate all variables without spilling.

- **Decoupled Approach:** This efficient determination of the chromatic number enables a decoupled approach to register allocation:

    1. **Spilling:** Initially, we can aggressively spill variables to reduce the number of live variables at any given point (MaxLive) until it aligns with the desired register budget (K). This effectively reduces the size and complexity of the interference graph.

    2. **Register Assignment:** Subsequently, we perform register assignment on the reduced interference graph. Since the graph remains chordal, we can efficiently determine the register allocation using techniques like graph coloring algorithms, ensuring optimal register usage within the reduced register budget.

This decoupled approach simplifies the register allocation process by separating the potentially complex and iterative spilling decisions from the register assignment itself. The main advantage of this approach is simplicity: we can do register assignment without worrying about mapping the excess of variables to memory as Florent Bouchez explains in his paper, "A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases".

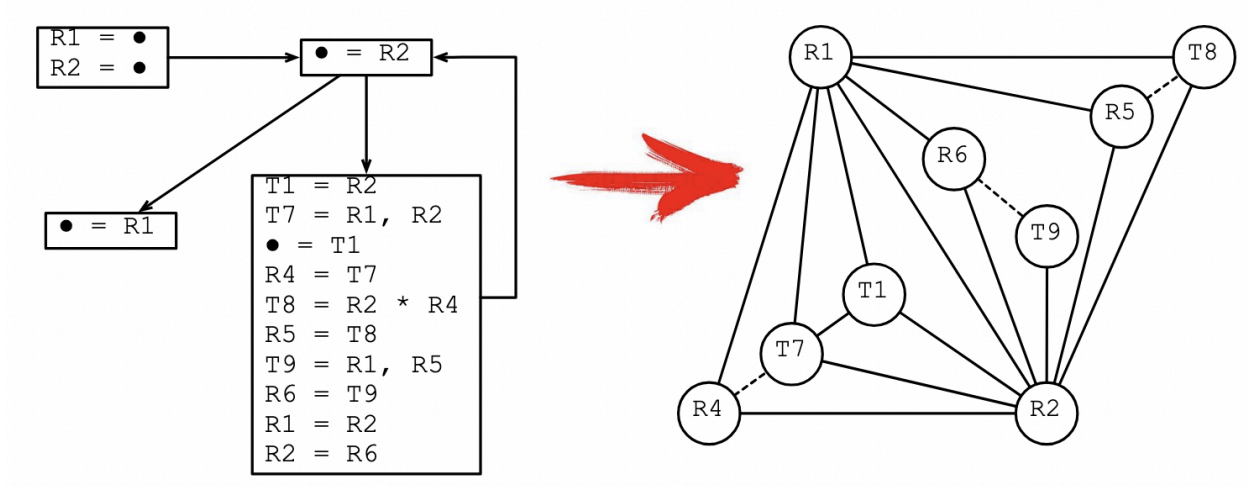
How does the build phase work?

The build phase in register allocation for SSA-form programs involves constructing an **interference graph**. This graph visually represents the constraints on register assignments. Here's a breakdown of the process:

1. **Liveness Analysis:** The initial step is to perform liveness analysis on the SSA-form program. Liveness analysis determines which variables are "live" at each program point. A variable is considered live at a point if its value might be used in the future, as we had seen in our class about data-flow analyses.

2. **Interference Graph Construction:**
    - Traverse the program, examining each program point.
    - At each point, identify all live variables.
    - For every pair of live variables at that point, add an undirected edge between their corresponding nodes in the interference graph. This edge signifies that these variables cannot be assigned to the same register, as they are both potentially in use at that point.

In essence, the build phase translates the liveness information into a graphical representation of register allocation conflicts:



Notice that we can still obtain chordal interference graphs, even if the original program is not in SSA form. The figure above shows an example of such a situation. Indeed, many real-world programs have chordal interference graphs, even after SSA elimination.

If the SSA-form simplifies static analyses, does it simplify liveness analysis too?

It does! Checking if two variables interfere on an SSA-form program can be done very quickly! Additionally, discovering the program points where variables are alive has also a non-iterative solution for SSA-form programs. Liveness analysis in SSA-form programs can be efficiently computed due to the unique properties of the Static Single Assignment (SSA) form.

Key SSA Property:

- Every use of a variable is dominated by its unique definition. This means that if a variable is used at a certain point, its definition must have occurred earlier in the program's execution path.

Algorithm:

We can leverage this property to perform a backward traversal of the Control Flow Graph (CFG) of the SSA-form program.

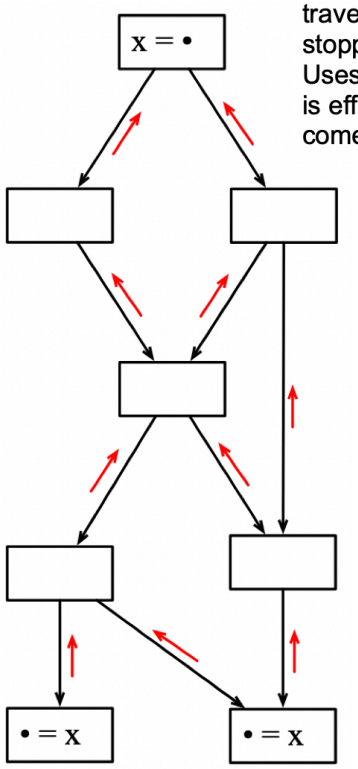
```
def live_analysis(program):
  """
  Computes liveness information for each statement in the
  SSA-form program.

  Args:
    program: The SSA-form program represented as a data structure
             (e.g., a list of statements, where each statement has
             'uses' and 'defs' attributes).

  Returns:
    A dictionary where keys are statements and values are sets of
    live variables at the entry of each statement.
  """

  live_vars = {}
  for statement in program:
    live_vars[statement] = set()

  # Backward traversal
  for statement in reversed(program):
    for use_var in statement.uses:
      live_vars[statement].add(use_var)
      for pred_stmt in statement.predecessors:
        # If not defined in predecessor
        if use_var not in pred_stmt.defs:
          live_vars[pred_stmt].add(use_var)

  return live_vars
```

Explanation:

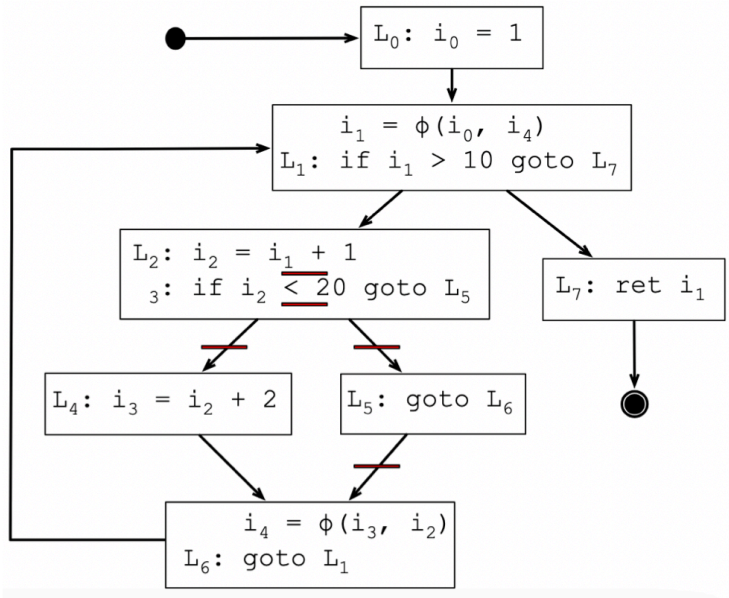
1. **Initialization:**
    - Create an empty dictionary `live_vars` to store the set of live variables at the entry of each statement.
2. **Backward Traversal:**

- Iterate through the program statements in reverse order (from the end to the beginning).

3. **Mark Uses as Live:**
   - For each variable used in the current statement, add it to the `live_vars` set for that statement.
4. **Propagate Liveness:**
   - For each predecessor statement of the current statement:
     - If the used variable is not defined in the predecessor statement:
       - Add the used variable to the `live_vars` set of the predecessor statement. This ensures that the variable remains live along the execution path leading to the current statement.



To find the program points where x is alive in the program on the left, we traverse the control-flow graph backwards, starting at the use sites of x, and stopping at the definition site of x. Phi-functions require some care, though. Uses of variables in phi-functions are considered in a different way. The variable is effectively used in the OUT set of the predecessor block where its definition comes from. In other words, i2 is alive at OUT[L5], but is not alive at IN[L6].

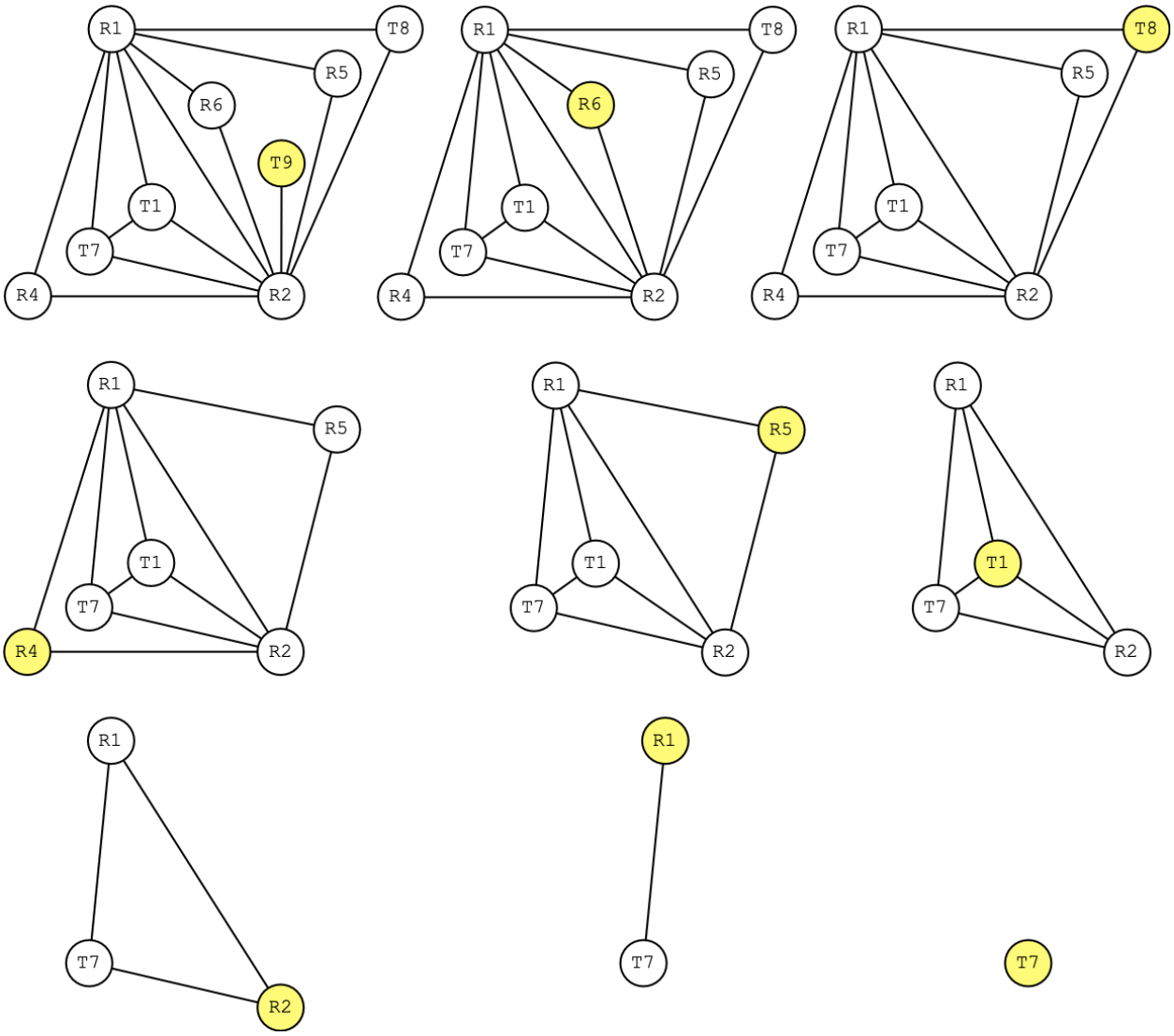


And how do we get the size of the maximum clique in the chordal graph?

In chordal graphs, the size of the maximum clique can be efficiently determined by leveraging the concept of **simplicial vertices** and **simplicial elimination orderings**.

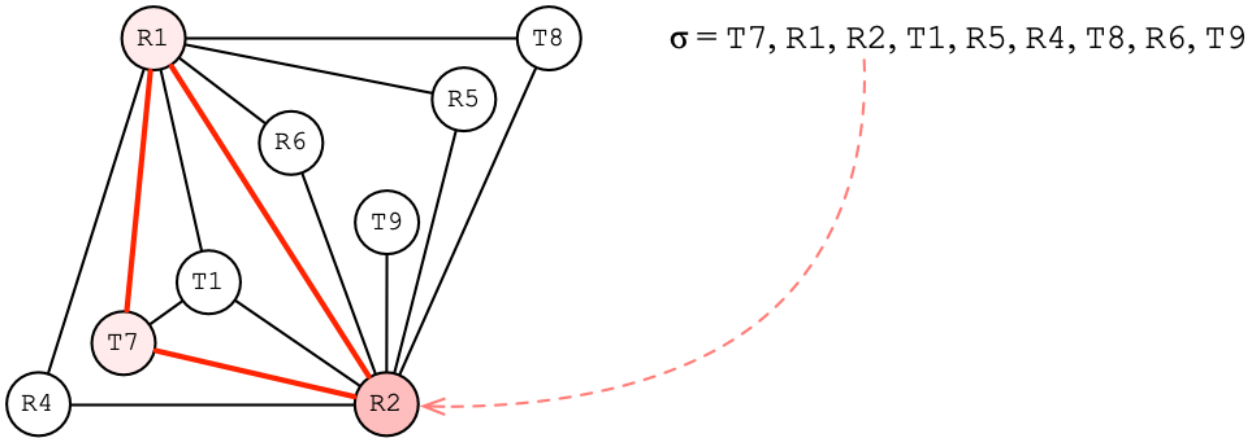
Simplicial Vertex:

- A vertex in a graph is **simplicial** if its neighbors form a complete subgraph (i.e., a clique). As an example, each yellow node in the figure below is simplicial:



Simplicial Elimination Ordering:

- A Simplicial Elimination Ordering of a graph G is a bijection σ: V(G) → {1, …, |V|}, such that every vertex vi is a simplicial vertex in the subgraph induced by {v1, …, vi}.
- An undirected graph without self-loops is chordal if, and only if, it has a simplicial elimination ordering, as seen in the paper "On Rigid Circuit Graphs".
- Continuing with our example, the sequence T7, R1, R2, T1, R5, R4, T8, R6 and T9 forms a simplicial elimination ordering. Consider R2, for instance. The two nodes that come before, T7 and R1, are both neighbors of R2, and both are connected. Thus, {T7, R1, R2} forms a clique.



Finding the Maximum Clique Size:

1. **Determine a Simplicial Elimination Ordering:**
    - One efficient algorithm to find a simplicial elimination ordering is **Maximum Cardinality Search (MCS)**.

```
def maximum_cardinality_search(graph):
    """
    Computes a simplicial elimination ordering using
    Maximum Cardinality Search.

    Args:
        graph: The input graph represented as an adjacency
        list or matrix.

    Returns:
        A list representing the simplicial elimination ordering.
    """
    n = len(graph)
    labels = [0] * n
    ordering = []

    for _ in range(n):
        max_label_vertex = max(range(n), key=lambda v: labels[v])
        ordering.append(max_label_vertex)
        for neighbor in graph[max_label_vertex]:
            labels[neighbor] += 1
        del graph[max_label_vertex]

    return ordering
```

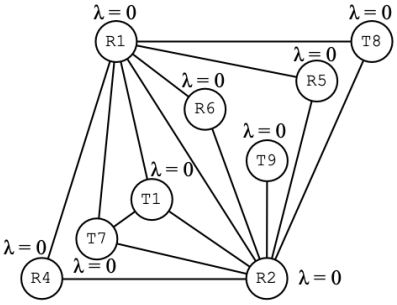
2. **Calculate Maximum Clique Size:**
    - During the MCS, keep track of the maximum label assigned to any vertex.

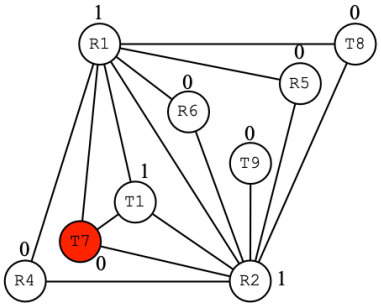
    - **The maximum clique size in the chordal graph is equal to this maximum label plus 1**!

Rationale:

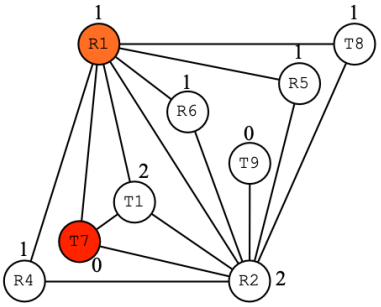
- The MCS algorithm assigns labels to vertices based on the number of already-ordered neighbors.
- In a chordal graph, the maximum label assigned to a vertex during MCS represents the size of the maximum clique that includes that vertex.

The sequence of figures below show how maximum cardinality search will find a simplicial elimination ordering for the interference graph of our running example:

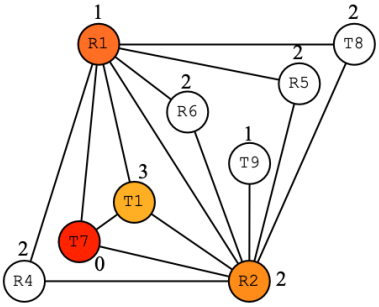




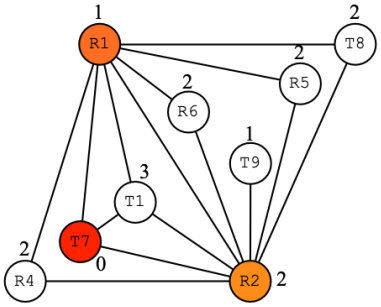
σ = T7,



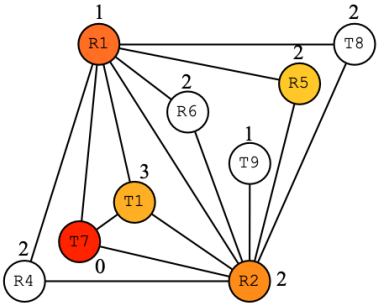
σ = T7, R1,



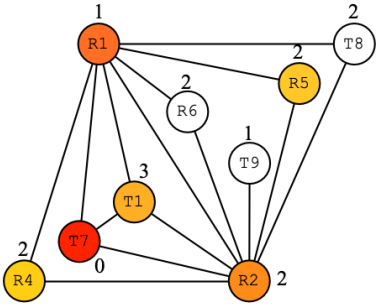
σ = T7, R1, R2, T1,



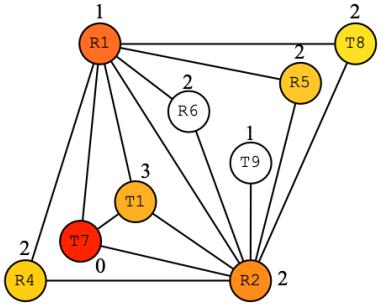
σ = T7, R1, R2,



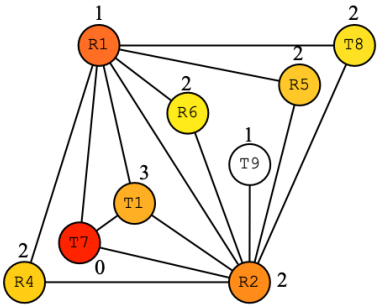
σ = T7, R1, R2, T1, R5,



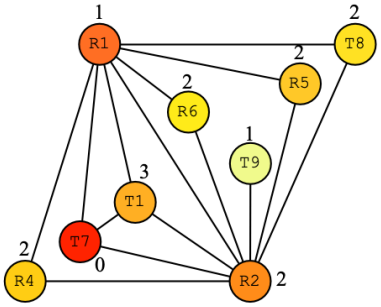
σ = T7, R1, R2, T1, R5, R4,



σ = T7, R1, R2, T1, R5, R4, T8,



σ = T7, R1, R2, T1, R5, R4, T8, R6



σ = T7, R1, R2, T1, R5, R4, T8, R6, T9

Maximum Cardinality Search

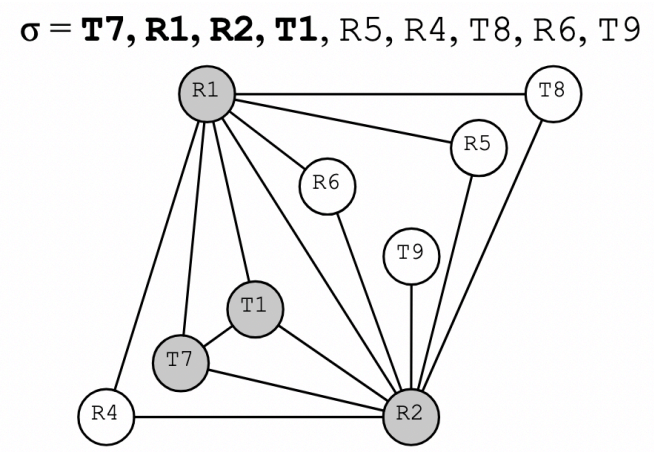
input: G = (V, E)
output: a simplicial elimination ordering $\sigma = v_1, \ldots, v_n$
for all $v \in V$ do $\lambda(v) \leftarrow 0$
for $i \leftarrow 1$ to $|V|$ **do**
  let $v \in V$ be a vertex such that $\forall u \in V, \lambda(v) \geq \lambda(u)$ **in**
    $\sigma(i) \leftarrow v$
    for all $u \in V \cap N(v)$ **do** $\lambda(u) \leftarrow \lambda(u) + 1$
    $V = V \setminus \{v\}$

In this example, the simplicial elimination ordering will be T7, R1, R2, T1, R5, R4, T8, R6 and T9, as we had seen before. If we remove the nodes in the reverse simplicial ordering, then whenever we remove a node, all its neighbors (in the remaining graph) form a clique

Ok, I can find out the size of the largest clique of a chordal interference graph in polynomial time. But how does that help me with spilling?

Sebastian Hack has a cute theorem in his PhD Dissertation that helps with that!

1. **The Clique Theorem in SSA Interference Graphs**:
   As demonstrated in Sebastian Hack's PhD thesis (Page 42), for an SSA-form program P with interference graph G=(V,E):
   - For any clique C={c1,c2,…,cn} in G, there exists a program point L where all the variables represented by the nodes ci are simultaneously live.
   - The size of the maximum clique in G corresponds to the minimum number of registers required to compile P without spilling.
2. **Using the Clique Size to Manage Register Pressure**:
   If the largest clique in the graph has more than K nodes (where K is the number of available registers), spilling is necessary to reduce register pressure. Remember: in the chordal graph, identifying cliques with more than K nodes is a trivial problem: it suffices to find the largest counters created by the maximum cardinality search. In our example, we would have a clique of size four:



3. **Challenges in Spilling**:
   While finding the size of the largest clique is polynomial-time for chordal graphs, **determining the minimal set of variables to spill** to make the graph K-colorable is an **NP-complete problem**, even for chordal graphs. This is known as the "Maximum K-Colorable Subgraph Problem for Chordal Graphs."
   - In practical terms, this means:
     - We can efficiently determine **if spilling is needed** (by comparing MaxLive with K).
     - However, **choosing which variables to spill** to reduce the graph's chromatic number is computationally hard. We need to resort to heuristics.

4. **Iterative Spilling and Graph Reconstruction**:
   Because of this complexity, register allocation typically involves an iterative process:
   - Build the interference graph.
   - Analyze live ranges and cliques to assess register pressure.
   - Perform spilling to reduce pressure.
   - Rebuild the interference graph with reduced live ranges and repeat until MaxLive≤K:

By using the size of the largest clique as a guide, we can strategically reduce register pressure, enabling efficient register allocation while balancing computational constraints.

But if we end up having to spill, which nodes do we choose?

When deciding which nodes to spill in a chordal interference graph, the goal is to minimize the "damage" caused by spilling a variable (i.e., mapping it to memory). To achieve this, we compute the **spill cost** of each variable using heuristics. The **spill cost** estimates the expense of spilling a variable based on its usage in the program.

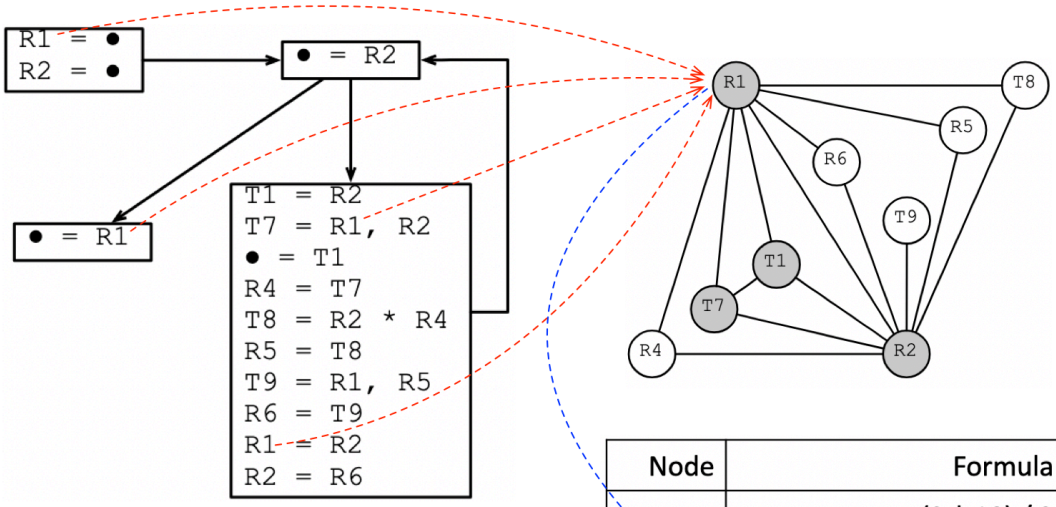
Spill Cost Formula

The following formula, adapted from Andrew Appel's book *Modern Compiler Implementation in Java*, provides a structured way to compute spill costs:

```
def Spill_Cost(v, interference_graph):
    cost = 0
    # Iterate over all blocks where 'v' is defined/used
    for block in v.blocks:
        # Number of defs and uses in the block
        SB = block.num_defs_and_uses
        # Loop nesting factor of the block
        N = block.loop_nesting_factor
        # Degree of 'v' in the interference graph
        D = interference_graph.degree(v)
        cost += (SB * 10**N) / D
    return cost
```

- `SB`: Number of uses and definitions in a given block BB.
- `N`: Loop nesting factor of BB; deeper loops imply higher execution frequency.
- `D`: Degree of the variable vv in the interference graph; higher degrees mean more overlapping live ranges.

Variables with lower spill costs are prioritized for spilling, as they are expected to cause less runtime overhead. The figure below shows how this formula could be applied to our running example:



Let's assume we are targeting an architecture with three registers. We have a clique of size four; hence, we must spill. We spill the node with the lowest spilling cost. In our example, that would be `R1`.

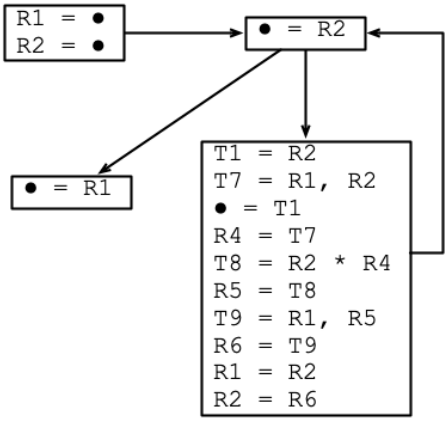
Node	Formula	Spilling Cost
T7	(2 * 10) / 3	6.66
R1	(1 + 1 + 3 * 10) / 7	4.57
R2	(1 + 1 + 5 * 10) / 8	6.5
T1	(2 * 10)/3	6.66

Spilling and Code Transformation

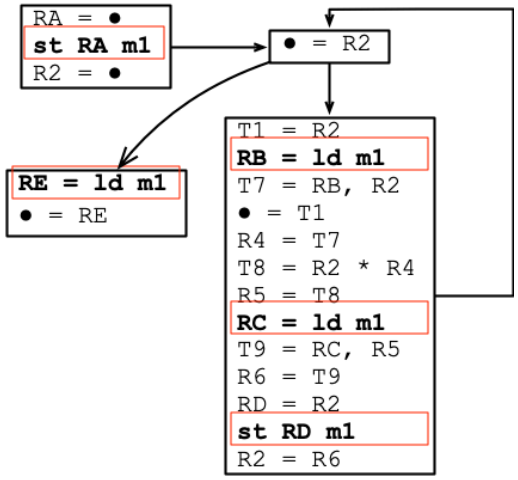
Once a variable is chosen for spilling:

1. **Insert Loads and Stores**:
    - Replace the variable's usage with memory loads and definitions with memory stores.
    - Ensure the semantics of the program remain consistent.
2. **Rebuild the Interference Graph**:
    - With live ranges altered by the newly inserted loads and stores, the interference graph changes.
3. **Restart Register Allocation**:
    - Check the size of the largest clique in the updated graph.
    - Spill more variables if necessary, and repeat the process until the graph is K-colorable. The figure below shows this process of spilling and rebuilding the interference graph of our running example:

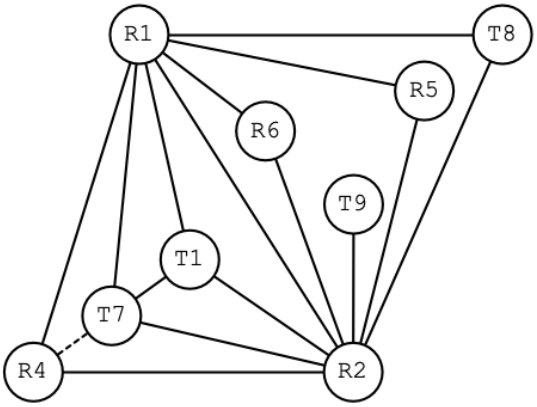
Program before spilling `R1`.



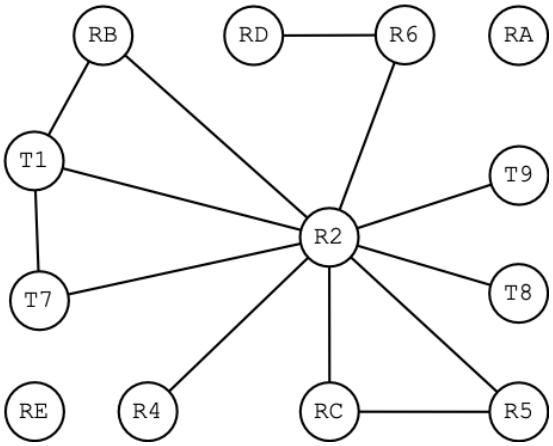
Program after spilling `R1`.



Interference graph before spilling.



Interference graph after spilling.



Iterative Process

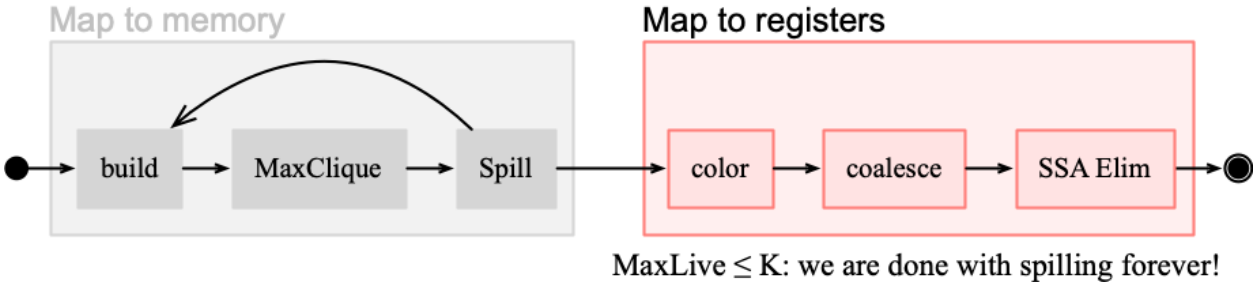
Spilling is part of an iterative process:

- Compute spill costs for all variables.
- Select the variable(s) with the lowest spill cost for spilling.
- Transform the program with appropriate loads and stores.
- Rebuild the interference graph and check if further spilling is needed.

By systematically choosing spill candidates with minimal impact, we ensure efficient register allocation while minimizing the runtime cost of spilling.

Once we are done with spilling, how do we assign registers to the variables?

The beauty of the decoupled approach is that once we lower the size of the largest clique to the number of available registers, we know we don't need to worry about spilling anymore!



To assign registers to variables, we can use greedy coloring! Here is a mock Python implementation of the greedy coloring algorithm:

```
def Greedy_Coloring(G, S):
    """
    Assign colors to vertices in a graph G using a sequence S.
    Parameters:
      G: tuple (V, E), where V is the set of vertices and
         E is the set of edges
      S: list, sequence of vertices in V
    Returns:
      m: dictionary mapping vertices to colors
    """
    V, E = G
    # Initialize all vertices with no color (None)
    m = {v: None for v in V}

    for v in S:
        # Find colors used by neighbors of v
        neighbor_colors = {m[neighbor] for neighbor in V \
          if (v, neighbor) in E or (neighbor, v) in E}
        # Assign the lowest available color
        m[v] = next(c for c in range(len(V)) \
          if c not in neighbor_colors)

    return m
```

Why Greedy Coloring Works Optimally for Chordal Graphs

For **chordal graphs**, if we follow a **Simplicial Elimination Ordering (SEO)**, greedy coloring yields an optimal coloring.

- **Simplicial Node**: A node whose neighbors form a clique.
- **SEO**: An ordering of the graph's vertices such that each vertex, when removed along with its incident edges, leaves a subgraph where its remaining neighbors form a clique.

Key Insights for Chordal Graphs

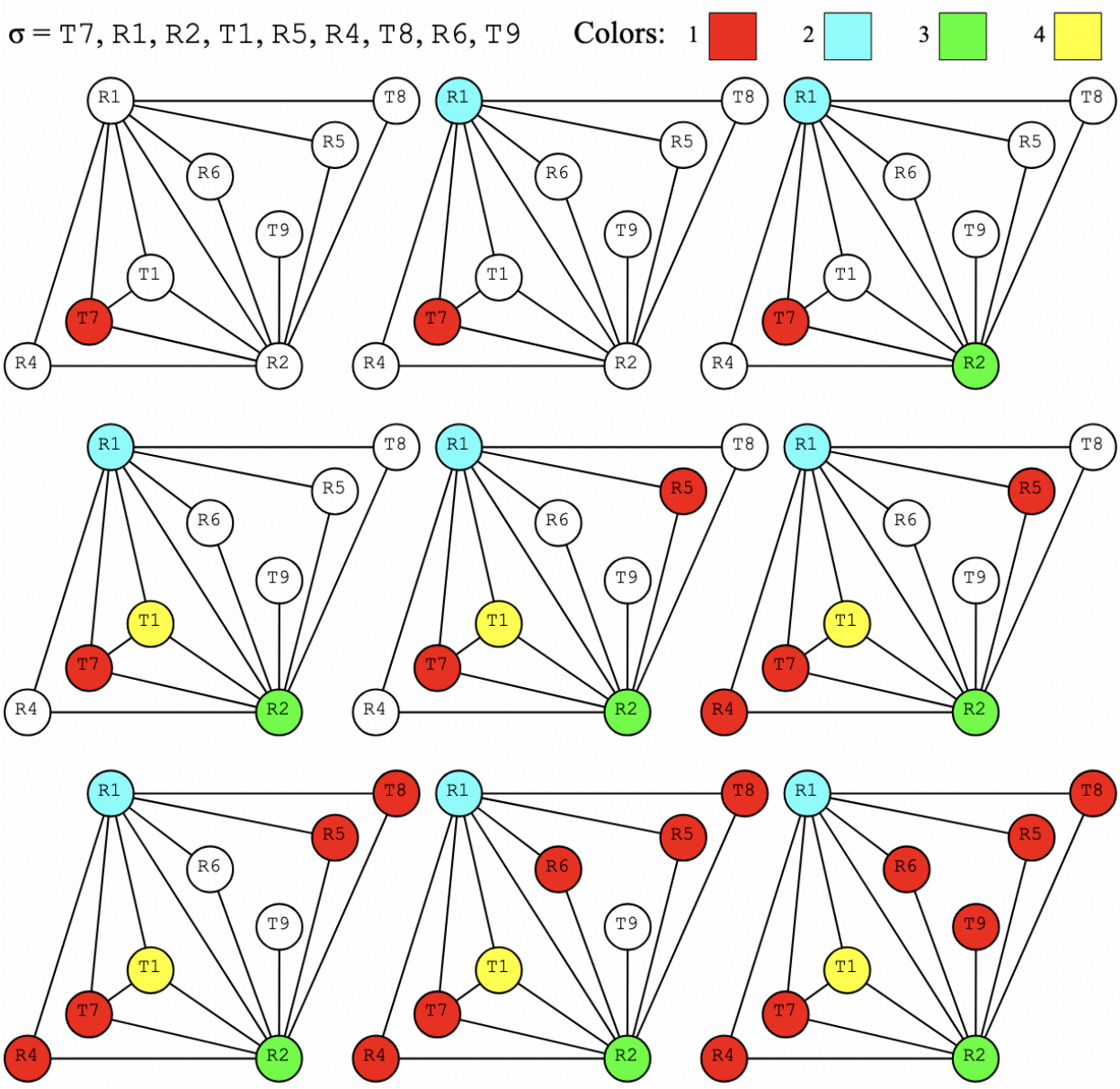
1. **Clique Properties**:
    - At any point in the SEO, the neighbors of a node v already colored form a clique.
    - All nodes in a clique must receive different colors.
    - Therefore, if v has M already-colored neighbors, it will be assigned color M+1.
2. **Chromatic Number**:
    - For chordal graphs, the chromatic number (minimum number of colors needed) equals the size of the largest clique.

Applying Greedy Coloring to Register Allocation

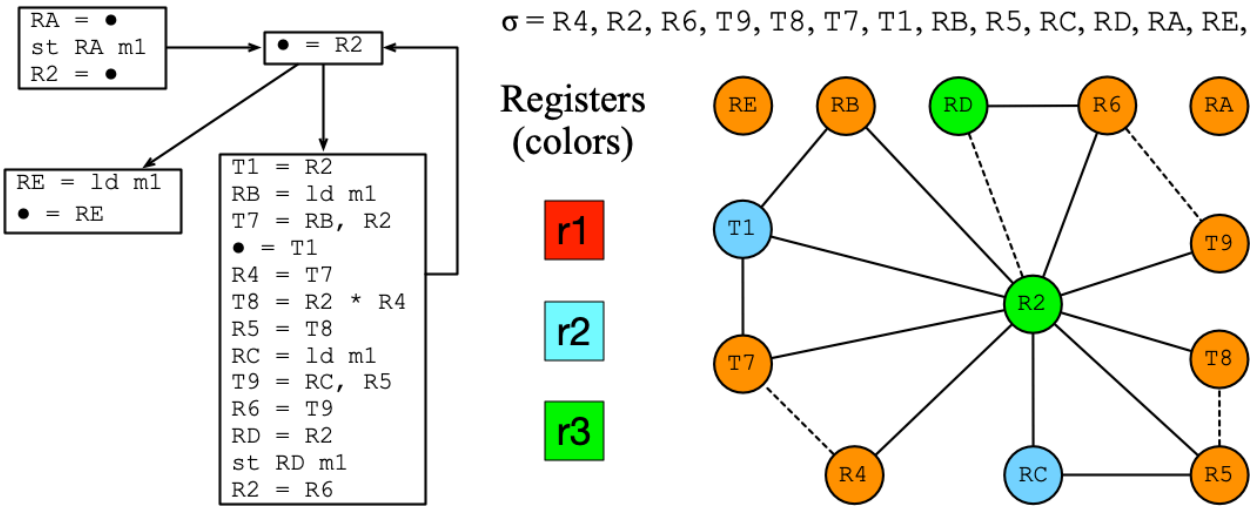
After spilling, the interference graph is guaranteed to be K-colorable, where K is the number of available registers. Using greedy coloring:

- Each color corresponds to a register.
- Variables are assigned registers such that no two variables interfering with each other share the same register.

This process ensures a valid and efficient register assignment for the program without additional spilling. Thus, notice that it's vital to reduce the size of the largest clique to at most K, the number of registers in the target architecture. For instance, if we get back to the interference graph of our original example, you will see that we will need at least four different colors:



However, if we try the same greedy coloring algorithm on the graph that is produced after the spilling phase (assuming three registers), then we are guaranteed to find a coloring with at most three colors, e.g.:



How can we implement coalescing in this register allocation algorithm?

Coalescing in this register allocation algorithm aims to eliminate unnecessary copy instructions by assigning the same register to the source and destination of a copy. This optimization is achieved while ensuring that the interference graph remains K-colorable, where K is the number of available registers.

Post-Assignment Coalescing

We adopt a **best-effort coalescing** strategy, as described in "Register Allocation via the Coloring of Chordal Graphs." This method leverages the flexibility of unused colors in local neighborhoods of the interference graph to coalesce copy instructions.

The algorithm modifies the interference graph GG and attempts to merge nodes corresponding to variables involved in copy instructions while maintaining colorability.

Algorithm: Best Effort Coalescing

Below is the algorithm in mock Python:

```
def Best_Effort_Coalescing(L, G, K):
    """
    Perform best-effort coalescing on the interference graph.
    Parameters:
      L: list of copy instructions (e.g., ["x = y", ...])
      G: tuple (V, E), where V is the set of variables and E is the
         set of edges
      K: number of available colors (registers)
    Returns:
      G_prime: the coalesced interference graph
```

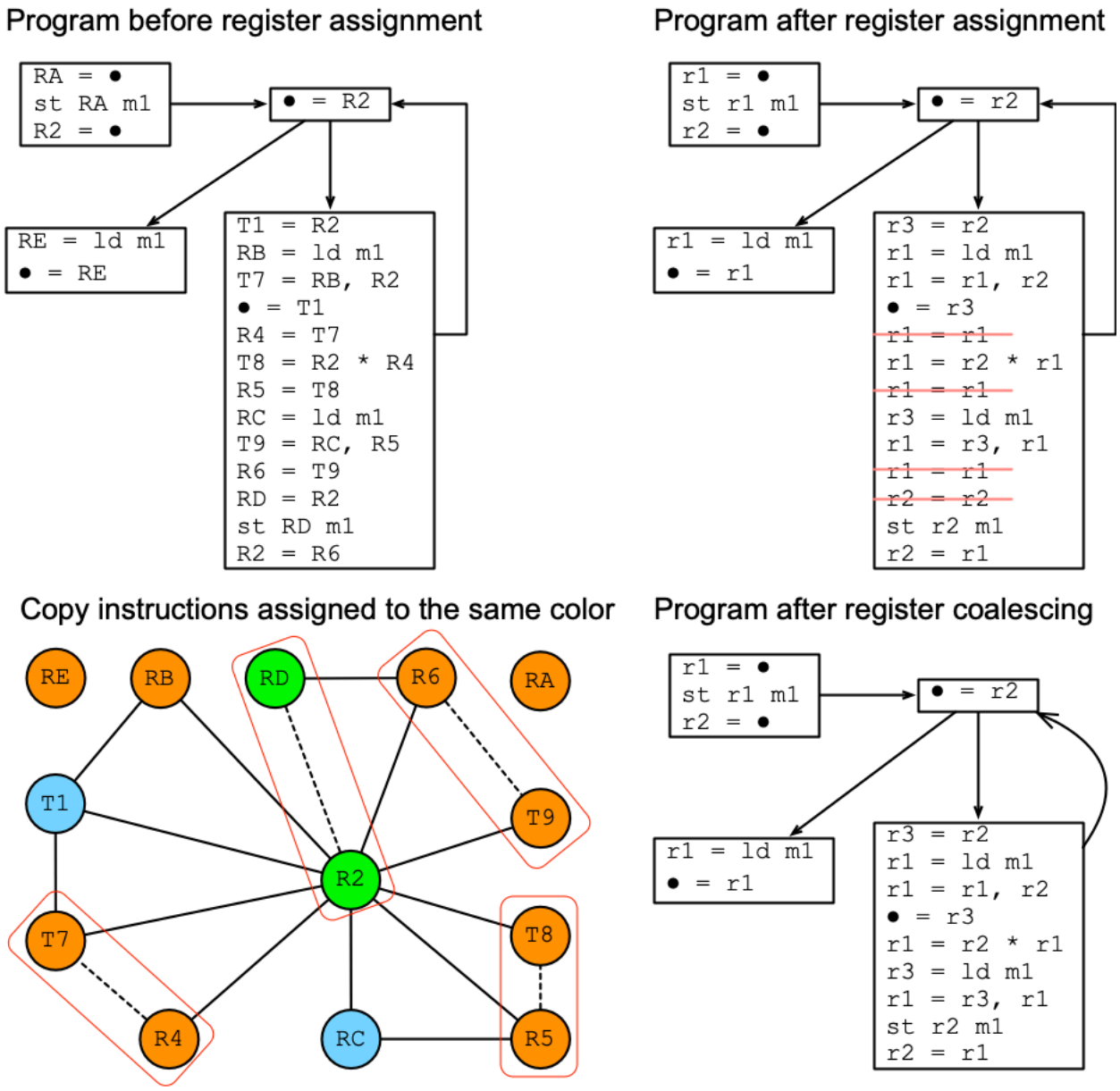
```
    """
    V, E = G
    # Start with a copy of G
    G_prime = (V.copy(), E.copy())

    for copy in L:
        # Extract variables from "x = y"
        x, y = parse_copy_instruction(copy)

        # Sets of colors used in the neighborhoods of x and y
        Sx = {color_of(v, G_prime) for v in neighbors(x, G_prime)}
        Sy = {color_of(v, G_prime) for v in neighbors(y, G_prime)}

        # Find an unused color in the neighborhood of x and y
        for c in range(K):
            if c not in Sx.union(Sy):
                # Merge nodes x and y into a new node xy with color c
                xy = f"{x}_{y}"  # Name the new node

                # Add xy to the graph
                G_prime[0].add(xy)
                G_prime[1].update({(xy, v) for v in \
                  neighbors(x, G_prime).union(neighbors(y,
G_prime))})

                # Replace x and y in copy instructions with xy
                L = [instr.replace(x, xy).replace(y, xy) \
                  for instr in L]

                # Remove x and y from the graph
                G_prime[0].discard(x)
                G_prime[0].discard(y)
                G_prime[1] = {(u, v) for u, v in G_prime[1] \
                  if u != x and u != y and v != x and v != y}

                break  # Move to the next copy instruction

    return G_prime
```

Explanation of the Algorithm

1. **Input**: The list of copy instructions L, the interference graph G, and the number of registers K.

2. **Finding Unused Colors**: For each copy instruction x=y, compute the colors used by the neighbors of x and y in the graph. If there exists a color c that is unused in these neighborhoods, proceed with coalescing.
3. **Coalescing**:
    - Create a new node xy to represent the merged variables.
    - Assign xy a color cc and update the graph to reflect this merge.
    - Replace occurrences of x and y in the copy instructions with xy.
    - Remove x and y from the graph.
4. **Repeat**: Process all copy instructions in L.

The figure below shows the effect of eliminating copies in the program after register assignment:



Benefits and Trade-offs

- **Benefits**: This approach is simple and efficient. It improves runtime performance by reducing the number of copy instructions executed.
- **Limitations**: Unlike more complex methods (e.g., Hack and Goos's graph recoloring algorithm), best-effort coalescing does not propagate changes throughout the graph, potentially missing further coalescing opportunities.

We are implementing coalescing in a way that will never cause spilling. Does that mean that avoiding spilling is more important than removing copy instructions?

Yes, prioritizing the avoidance of spilling over removing copy instructions reflects a deliberate trade-off in the coalescing strategy. Here's why this decision is significant:

1. **Performance Impact of Spilling**:
    - Spilling introduces memory accesses (loads and stores) to manage variables that cannot fit into registers. These accesses are much more expensive than register-to-register operations.
    - Excessive spilling can degrade runtime performance significantly, particularly in loops or frequently executed code.
2. **Guaranteeing K-Colorability**:
    - The coalescing algorithm must ensure the interference graph remains KK-colorable, where KK is the number of available registers.
    - If coalescing causes the graph to exceed KK colors, it may force additional spills, undermining the purpose of register allocation.
3. **Minimal Impact of Copy Instructions**:
    - Copy instructions (e.g., `x=y`) are cheaper than memory operations since they involve only registers.
    - While reducing copy instructions improves performance, the relative cost savings are much smaller compared to avoiding spills.

Balancing the Trade-Off:

The strategy of **best-effort coalescing** reflects this balance:

- It opportunistically removes copy instructions only when it is safe to do so (i.e., without risking spills).
- It avoids propagating changes that might destabilize the interference graph or require further modifications.

Conclusion:

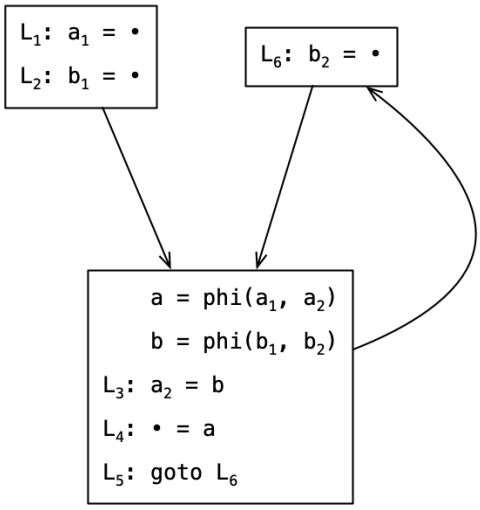
Yes, avoiding spilling is considered more important than removing copy instructions in many different academic works, such as Bouchez' "On the Complexity of Register Coalescing". This approach ensures stable and predictable performance by minimizing expensive memory accesses while still trying to reduce copy instructions when feasible. It reflects a practical compromise between aggressive optimization (copy removal) and maintaining efficient register allocation.

But, can't spilling happen during the SSA elimination phase? What if the arguments of phi-functions end up mapped to different stack slots?

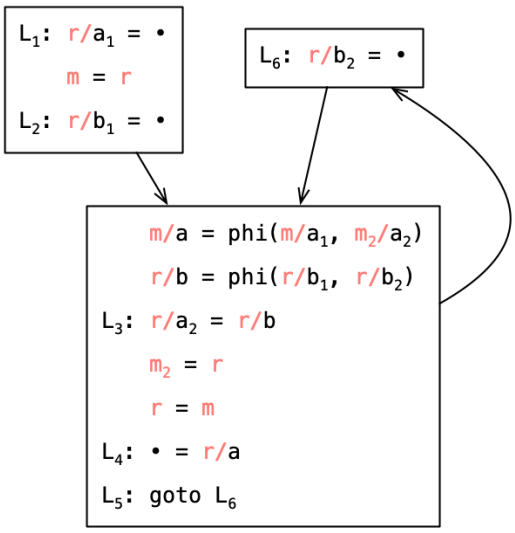
It depends on the program's representation! This issue is discussed in detail in the paper "SSA Elimination after Register Allocation". Here's the breakdown:

1. **Non-CSSA Programs**:
    - If the program is not in CSSA form, spilling may be necessary during SSA elimination.
    - Consider the example in the figure below, where φ-related variables like a and a2 have overlapping live ranges. These variables must be assigned different stack slots during spilling, which complicates SSA elimination.

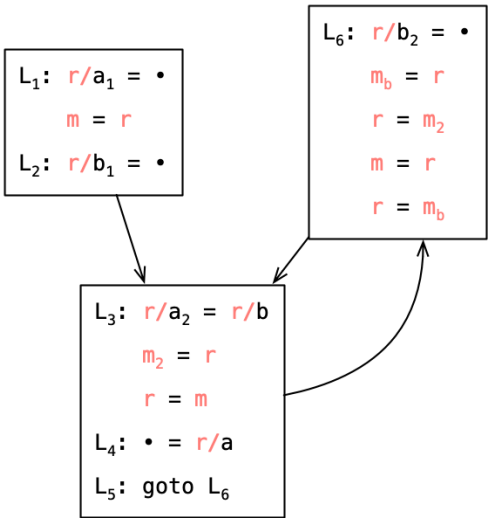
This program is not in conventional static single-assignment form, because the live ranges of two phi-related variables, a and a_2, overlap.



If we have to spill a and a_2 then we must map these variables to different stack slots, say: m and m_2, as their live ranges overlap.



This complicates SSA elimination, because we must implement the copy m = m_2. But if the target architecture does not have instructions to move values directly between memory slots, then we will need a temporary register. Now, if all the registers are already occupied, then we would have to spill again! In this example, if we have only one register r, then will have to spill b_2 (e.g., to a new memory slot m_b) to make room for a temporary register.

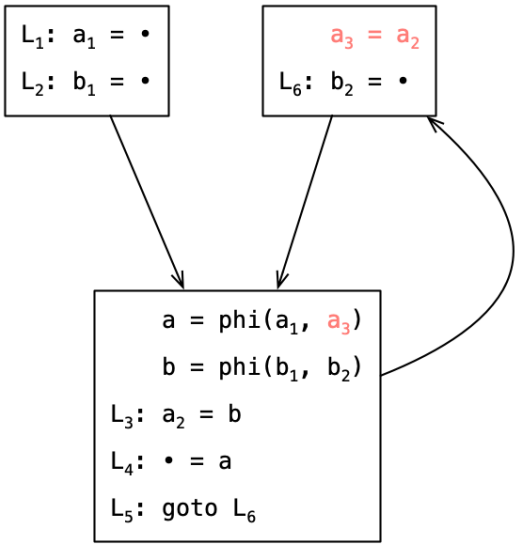


- If the target architecture does not support direct memory-to-memory transfers, temporary registers are required to implement these copies. If all registers are already occupied, additional spilling becomes necessary to free a temporary register. If necessary to avoid the additional spilling, one can use a stack slot as the scratch location, a bit like CraneLift does.

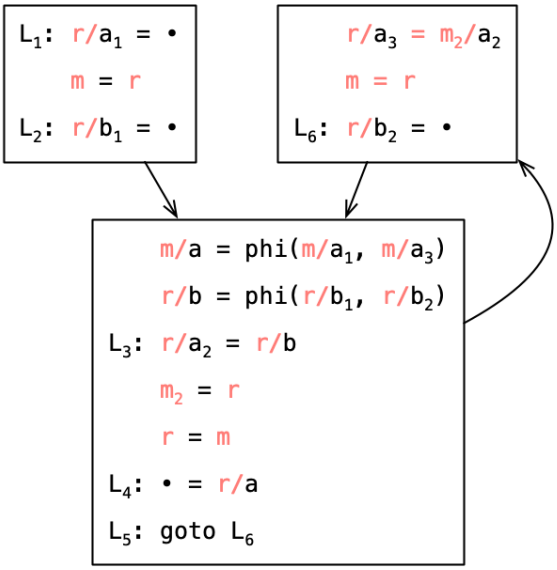
2. **Conventional Static Single Assignment (CSSA) Form**:
   - If the program is in CSSA form, spilling during SSA elimination is unlikely.
   - In CSSA, the arguments of a φ-function can either be mapped to different registers or to the same memory slot. For example, as shown in the figure below,,

splitting the live range of a variable (e.g., inserting a copy like a3 = a2) ensures that φ-related variables are handled cleanly.

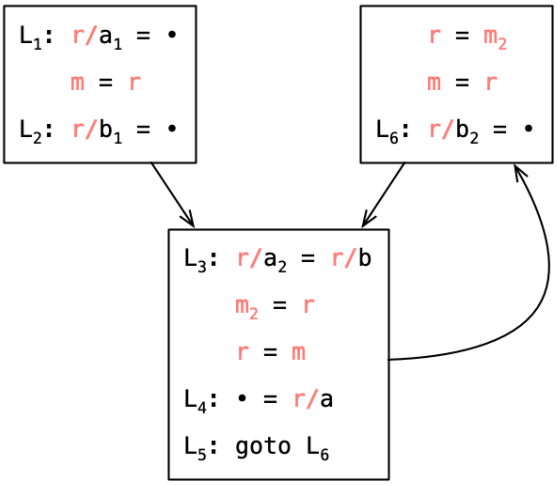
We can convert our program back to the CSSA representation by splitting the live range of a_2, via a new copy instruction $a_3 = a_2$. Now, a_2 and a are no longer phi-related.



During the spilling phase, a different stack slot will be naturally assigned to a_2 and the phi-related variables {a, a_1, a_3}.



The beauty of CSSA is that now the entire phi-function m/a = phi(m/a_1, m/a_3) is mapped to the same slot! So, during spilling, we can simply cut off the phi-function, without having to insert memory transfer instructions in the program.



- The result is that all variables related to a φ-function can be naturally assigned distinct slots, and no extra memory transfer instructions are needed during SSA elimination.

Why Spilling May Occur in Non-CSSA Programs

1. **Overlapping Live Ranges**:
   Variables with overlapping live ranges may require different stack slots, necessitating additional moves.
2. **Temporary Registers for Memory Transfers**:
   Architectures that lack direct memory transfer instructions may require temporary registers to implement copies. If no registers are free, spilling is unavoidable.

Benefits of CSSA for SSA Elimination

CSSA simplifies SSA elimination by ensuring that:

- Arguments of φ-functions are naturally assigned the same stack slot if they are ever spilled.
- Memory transfers can thus be entirely avoided, reducing the need for further spilling.

In the end, by converting a program to CSSA form before spilling, the complexity and overhead associated with SSA elimination can be greatly reduced.

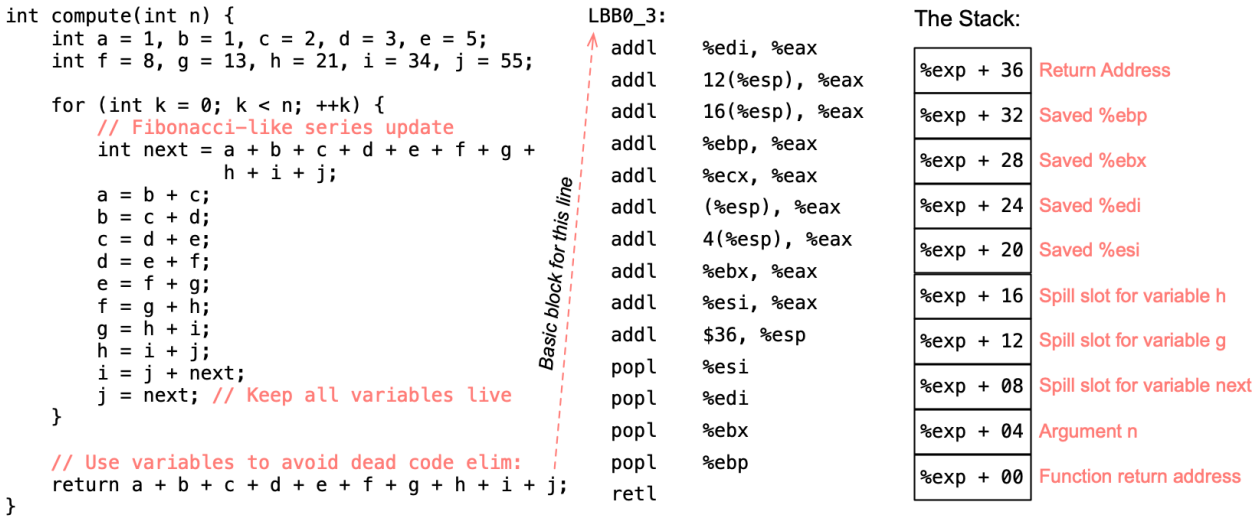
What are these stack slots you mentioned? I mean: where are spilled variables stored?

Spilled variables are mapped to **stack memory**. Spilling occurs when there are not enough registers to hold all the live variables at a given point in the program. The compiler generates additional load and store instructions to move the spilled variable's value between the stack and registers as needed.

1. Can a local variable (within a function's scope) be mapped to the stack or stay in a register?

Both mappings are possible! This is determined by the optimization level and by the number of available registers:

- **At -O0 (no optimization)**:
  - Compilers like GCC and Clang prioritize simplicity and debugging over performance.
  - All local variables are typically mapped to the **stack**, even if they are never spilled. This ensures predictable memory locations, making debugging easier.
- **At -O1 and higher**:
  - The compiler performs optimizations, including **register allocation**.
  - Frequently used variables or those with short live ranges are kept in **registers**, while less critical variables may remain in **stack memory** or be spilled during register pressure. The figure below shows a snapshot of a typical stack on x86-32:



2. What about global variables?

Global variables are not spilled to the stack. Instead, they are stored in **static memory**, which is allocated in the program's **data segment**. Here's why:

- Global variables have a lifetime that spans the entire execution of the program.
- They are accessible from anywhere in the program, so they are placed in a fixed memory location (e.g., `.data` or `.bss` segments).

Finding a spill location for a global variable is unnecessary because the memory location of global variables is already predetermined. We can still operate with them in register, e.g., bring them to registers and keep them in registers for as long as possible. But ultimately, they need to be stored back into their original memory location. In other words, any affectation on the global variable must change the state of the static memory that holds it.

4. What about heap-allocated variables?

Heap-allocated variables are managed differently:

- These variables are explicitly allocated and freed by the programmer (e.g., using `malloc`/`free` in C or `new`/`delete` in C++).
- Their values reside in the **heap**, and only pointers to them are usually manipulated in registers.

It doesn't make sense to talk about **spilling** a heap-allocated variable itself because spilling refers to moving a **register-held value** to memory. However, the **pointer** to a heap-allocated variable could be spilled if it is stored in a register and register pressure is too high.

Summary

If we assume that we are using a compiler like `gcc` or `clang`, than that's what happens:

1. **Local Variables**:
    - At `-O0`: Mapped to the stack for simplicity.
    - At `-O1` and higher: May stay in registers unless spilled to the stack.
2. **Global Variables**: Stored in static memory, never spilled.
3. **Heap-Allocated Variables**: Reside in the heap; only pointers to them may be spilled to the stack.

Spilling is a concept closely tied to register allocation and applies to values temporarily held in registers, not to variables that are permanently in memory (like globals or heap-allocated variables).

Are There Real-World Register Allocators That Run on SSA-form Programs?

While most real-world register allocators operate on **post-SSA programs**—where φ-functions are replaced by actual assembly instructions (e.g., copies and swaps)—there are notable examples of allocators that work directly on SSA-form programs. One such example is the **Go compiler's register allocator**. There is a nice description of the Go Register Allocator in this online article by Vladimir Makarov, who is one of the maintainers of the GCC Register Allocator.

The Go Compiler’s Register Allocator

The Go compiler performs register allocation directly on SSA-form programs. The implementation can be observed in the regalloc.go file. However, the algorithm used differs from the chordal graph-based methods described in these lecture notes. Instead, the Go compiler employs an approach that resembles **linear-scan register allocation with second-chance bin-packing**, as described by Traub et al. in the paper "Quality and Speed in Linear-Scan Register Allocation". The key distinction is that Traub’s algorithm works on non-SSA programs, while Go’s allocator operates directly in SSA form.

Handling φ-functions in the Go Compiler

The Go compiler handles φ-functions similarly to the approach discussed in these lecture notes:

- It ensures that all operands of a φ-function can be mapped to the same physical location, whether in a register or memory.
- If the program is not in **Conventional Static Single Assignment (CSSA)** form, the Go compiler splits the live ranges of φ-related variables to resolve potential conflicts.

Spill Management and Stack Allocation

For spilled variables, the Go compiler uses the stackalloc.go file to assign stack slots. A few notable aspects include:

- **Shared Stack Slots**: The Go compiler optimizes stack usage by allowing multiple SSA values to share the same stack slot, provided their lifetimes do not overlap.
- **Interference Graph for Optimization**: To minimize stack allocation costs, the Go compiler leverages the interference graph of SSA variables, ensuring that stack slots are reused efficiently.

Further Reading

For a detailed explanation of the Go compiler's register allocation process, Vladimir Makarov's article offers an excellent overview. Makarov highlights the unique aspects of Go's SSA-based allocation strategy and its practical implications.

How Did People Start Doing Register Allocation on SSA-form Programs?

The history of register allocation on SSA-form programs is a fascinating interplay between theory and practice, with critical contributions over several decades.

1. Chaitin's Foundational Work (1981)

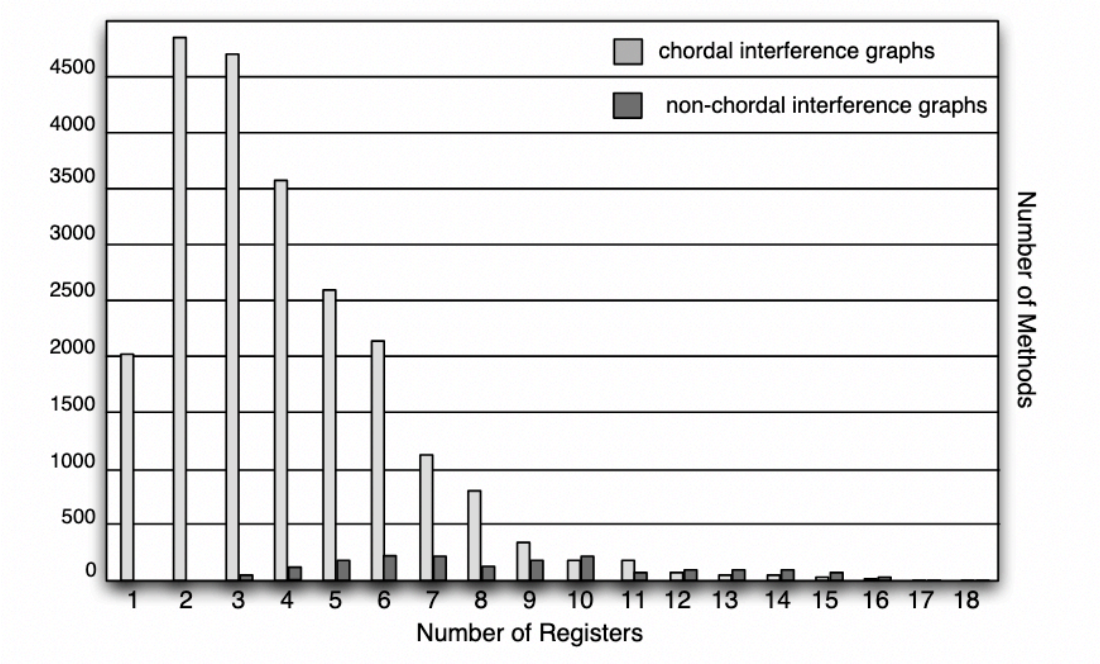
In 1981, Gregory Chaitin introduced the idea of modeling register allocation as a **graph coloring problem** in his seminal paper, *Register Allocation via Coloring*.

- He demonstrated that the general register allocation problem is **NP-complete**, setting the stage for decades of research into approximations and optimizations.

2. Observations on Real-world Interference Graphs (2003–2005)

- **Christian Andersson (2003)**:
  Andersson observed that many interference graphs from real-world programs are **Perfect Graphs**. Perfect graphs are significant because their chromatic number equals the size of their largest clique, simplifying the graph coloring process. He published his results in the paper "*Register Allocation by Optimal Graph Coloring*".

- **Fernando Pereira and Jens Palsberg (2005)**:
  In their paper, *Register Allocation via the Coloring of Chordal Graphs*, Pereira and Palsberg noted that most interference graphs of real-life programs tend to be **Chordal Graphs**.
  - For instance, **95%** of methods in the Java 1.5 library had chordal interference graphs when compiled with the JoeQ compiler.



- However, at the time, they did not yet show that **SSA-form programs inherently produce chordal interference graphs**.

3. The Breakthrough: SSA Interference Graphs Are Chordal (2005–2006)

- **Philip Brisk et al. (2005)**:
  In their paper, *Polynomial-time Graph Coloring Register Allocation*, Brisk and his collaborators proved that **strict SSA-form programs** have **Perfect Interference Graphs**.
- **Sebastian Hack's Contributions**:
  - Hack made the critical breakthrough in understanding the structure of SSA interference graphs. In his technical report *Interference Graphs of Programs in SSA-form* (2005), he proved that strict SSA-form programs have **Chordal Interference Graphs**, a stronger result than Brisk's.
  - A **strict SSA-form program** ensures that every path from the program's start to any use of a variable passes through a definition of that variable.
  - Hack's results were published in 2006 in the paper *Register Allocation for Programs in SSA-form*, and later formalized in *Optimal Register Allocation for SSA-form Programs in Polynomial Time*.
- **Florent Bouchez's Independent Discovery**:
  Around the same time, Florent Bouchez and his collaborators independently proved that SSA interference graphs are chordal in his technical report, *Register Allocation and Spill Complexity under SSA*.
  - Bouchez's team also explored related complexities, such as coalescing and spilling, publishing insights in papers like *Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove?*. The fact is that G.

> Chaitin's proof of NP-completeness for register allocation only works for general programs: it would not work for SSA-form programs.

Significance of These Discoveries

1. **Chordal Graphs and Efficient Coloring**:
   Chordal graphs can be optimally colored in polynomial time, using a **simplicial elimination ordering**. This discovery meant that register allocation for SSA-form programs, previously considered NP-complete in general, could be performed efficiently.

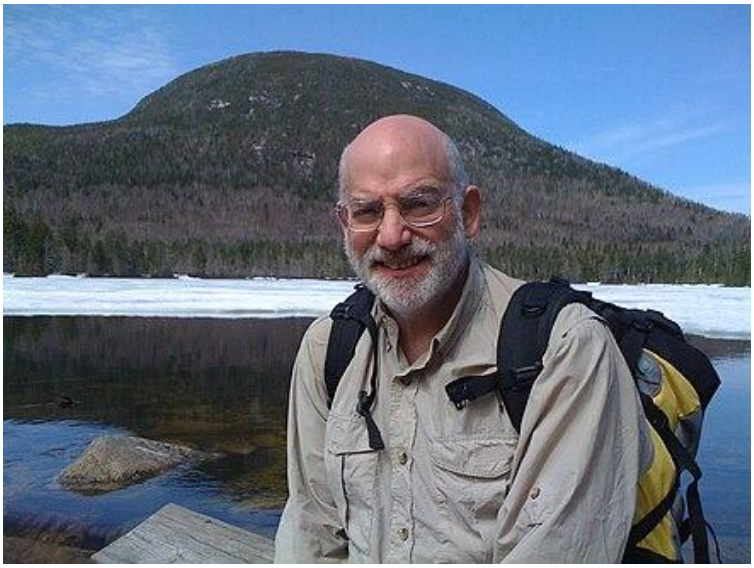
2. **Impact on Modern Compilers**:
   These insights laid the foundation for many modern register allocation techniques, significantly improving compilation times and reducing spill costs.

By connecting theoretical breakthroughs with practical observations, the history of SSA-based register allocation highlights the deep synergy between compiler design and graph theory. Would you like me to expand on any specific part or add more details?

So, it all started with the paper of Gregory Chaitin, in 1981?

Well, in a way. That was one of the first works to connect register allocation with graph coloring. And after that paper, there was a flurry of research and algorithms on register allocation. In this sense, Chaitin could be considered the father of graph coloring-based register allocation. Here's a short bio and summary of Gregory Chaitin's contributions to computer science (and science in general):

Gregory Chaitin is a pioneering figure in both theoretical computer science and mathematics. Born in 1947 in the United States, Chaitin demonstrated exceptional talent in mathematics and computer science from a young age. He is widely recognized for his groundbreaking work in **algorithmic information theory** **(AIT)**, a field that blends computation with information theory to explore the limits of what can be computed and described. Over the years, Chaitin has made contributions to both theoretical and practical aspects of computer science.



Chaitin spent much of his career at IBM Research, where his ideas influenced numerous areas of computer science. In addition to his technical achievements, he is a prolific author and engaging speaker, bringing abstract concepts to broader audiences through books and lectures.

Contributions to Science

1. **Algorithmic Information Theory (AIT)**:
    - Chaitin's work in AIT builds on ideas from Alan Turing and Claude Shannon. He introduced the concept of **Chaitin's constant (Ω)**, a real number representing the probability that a randomly generated program will halt.
2. **Register Allocation via Graph Coloring**:
    - In his seminal 1981 paper, *Register Allocation via Coloring*, Chaitin introduced the idea of modeling the register allocation problem in compilers as a **graph coloring problem**.
    - His approach demonstrated how interference between variables in a program could be represented as a graph, with variables as nodes and edges representing conflicts. Many real-world compilers currently follow his approach. For an example, take a look into the implementation of the interference graph, used in the HotSpot compiler.
    - Chaitin showed that the general register allocation problem is **NP-complete**, but his algorithm provided practical solutions that have been foundational to modern compiler design.
3. **Philosophy and Metamathematics**:
    - Chaitin has extended his work beyond computation into the philosophy of mathematics, addressing fundamental questions about the nature of mathematical truth and the incompleteness of formal systems. His books, such as *Meta Math! The Quest for Omega*, make these ideas accessible to broader audiences.

Legacy

Gregory Chaitin's work has left an indelible mark on both theoretical computer science and practical software engineering. His contributions to register allocation have shaped compiler technology, making programs run faster and more efficiently. Simultaneously, his works in AIT have deepened our understanding of computation, randomness, and mathematical truth, inspiring many researchers.