CSE 417
Branch & Bound (pt 3)
"Fast Enough" Exponential Time

UNIVERSITY *of* WASHINGTON

Reminders

- **HW8 due Friday**
  - model the problem of rounding table entries as max flow
    - you are given a library that solves basic max flow
  - don’t forget what you learned in HW7
    - the provided library is just implementing an algorithm you know

Review of previous lectures

- Complexity theory: P & NP
  - answer can be found vs checked in polynomial time
- NP-completeness
  - hardest problems in NP
- Reductions
  - reducing from Y to X proves $Y \leq X$
    - if you can solve X, then you can solve Y
  - X is NP-hard if every Y in NP is $Y \leq X$

Review of previous lectures

more generally, only pay for distance from easy cases

Coping with NP-completeness:

1. Your problem could lie in a special case that is easy
   - example: small vertex covers (or large independent sets)
   - example: independent set on trees

2. Look for approximate solutions
   - example: Knapsack with rounding

3. Look for "fast enough" exponential time algorithms

Next two lectures

3. Look for “fast enough” exponential time algorithms

   - “For every polynomial time algorithm you have,
     there’s an exponential time algorithm I would rather run.”
     — Alan Perlis

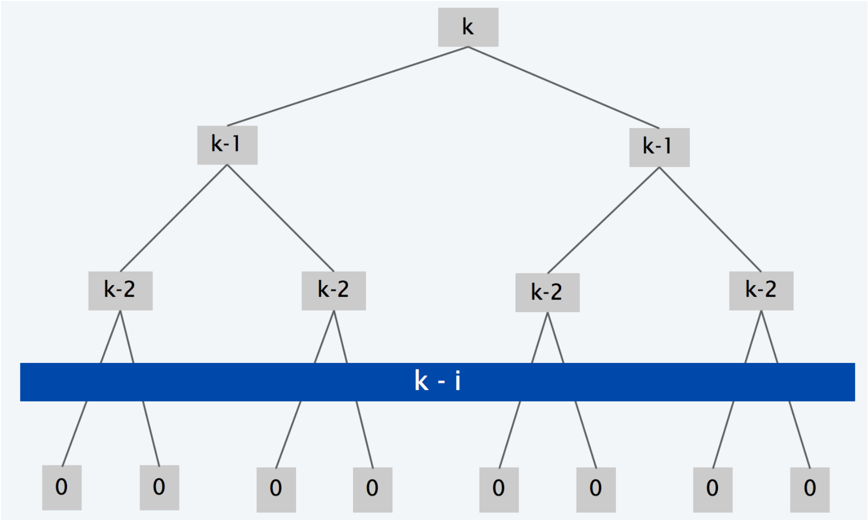
   - In practice, it doesn’t really matter if the algorithm scales exponentially as long as it finishes in a reasonable amount of time on the data you need to run it on.
     - we also have more computing power now than ever before

   - Applies to both decision problems and optimization

Outline for Today

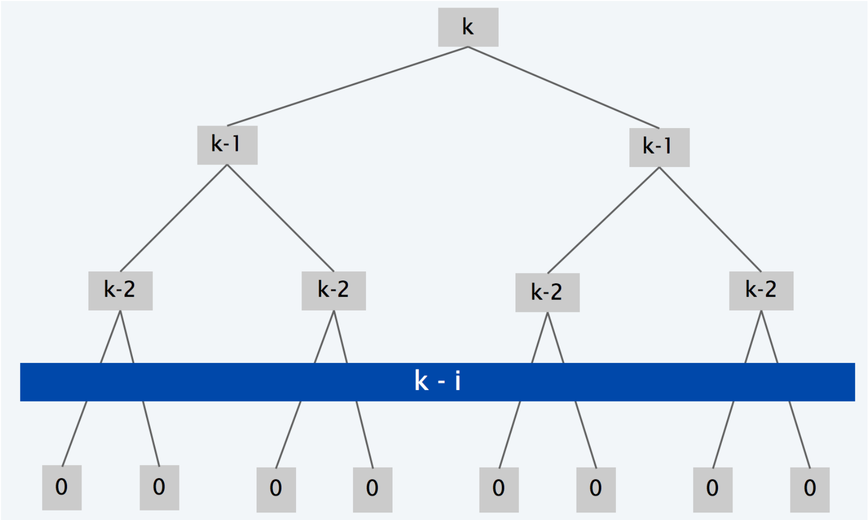
> Search Trees
> 3-SAT
> Knapsack + Vertex Cover
> Register Allocation
> Branch & Bound

Search Trees



- Vertex Cover algorithm showed an example of a search tree
  - tree of recursive calls
  - VertexCover(G, k) calls VertexCover(G – {u}, k – 1) and VertexCover(G – {v}, k – 1) for some edge (u,v)
- Each node corresponds to a set of choices about what sort of solution to look for
  - each node looks G – {u_1, u_2, ..., u_k}
  - removed those are the ones are going to use (so don't need cover)

Search Trees



> Vertex Cover algorithm showed an example of a search tree
  – easily implemented recursively

> Each node corresponds to a set of choices about what sort of solution to look for

> Running time is O(#nodes · time per node)
  – #nodes is exponential in the worst case
  – key point: work hardest on reducing #nodes not time per node

Outline for Today

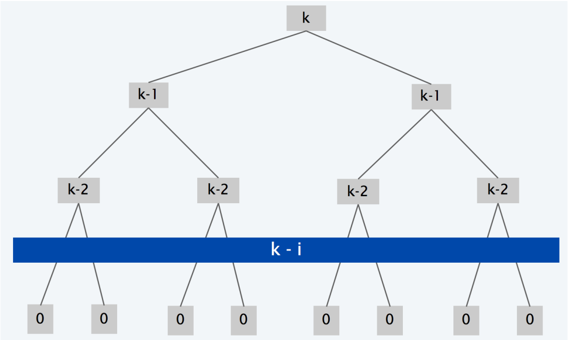
> Search Trees
> 3-SAT
> Knapsack + Vertex Cover
> Register Allocation
> Branch & Bound

Recall: SAT and 3-SAT

- **SAT**: Given a logical formula on variables x_1, ..., x_n using only **and**, **or**, & **not**, determine whether there is a setting of the variables to T/F so that the formula evaluates to T

- **3-SAT**: As above, but formula is of the form "t_1 **and** t_2 ... **and** t_m", where each t_i is of the form "f_{i1} **or** f_{i2} **or** f_{i3}", where each f_{ij} is either "x_k" or "**not** x_k" for some k
  - e.g.: ((not x_1) or x_2 or x_3) and
    (x_1 or (not x_2) or x_3) and
    ((not x_1) or (not x_2) or (not x_3))

Brute Force Algorithm



> Search tree with nodes for {x_1=T/F, ..., x_k=T/F}
  – root node has empty set {} of assignments
  – two children of node with assignments {x_1=T/F, ..., x_k=T/F} are
    > {x_1=T/F, ..., x_k=T/F} + {**x_{k+1} = T**} AND
    > {x_1=T/F, ..., x_k=T/F} + {**x_{k+1} = F**}

> #nodes is $O(2^n)$

> time per node is O(m) in leaves
  – leaf nodes have T/F value for every variable
  – evaluate each clause, see if all are satisfied

Improved Algorithm

- Look at individual clause "f_1 or f_2 or f_3" and consider how it could be satisfied…
- Either have f_1 = T or f_2 = T or f_3 = T
  - (or rather, cannot have all three being F)
- Each f_i is either x_j or not x_j,
  so setting f_i = T is setting x_j = T or x_j = F

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Improved Algorithm

> Look at individual clause "f_1 or f_2 or f_3"
  – suppose these correspond to variable x_i, x_j, and x_k
  – suppose those are satisfied by setting $x_i = b_i$, $x_j = b_j$, and $x_k = b_k$, resp.

> CanSatisfy(P) iff
CanSatisfy(P, $\{x_i=b_i\}$) or CanSatisfy(P, $\{x_j=b_j\}$) or CanSatisfy(P, $\{x_k=b_k\}$)
  – one of those must work if F is satisfiable

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Improved Algorithm

- CanSatisfy(P) iff
  CanSatisfy(P, $\{x_i=b_i\}$) or CanSatisfy(P, $\{x_j=b_j\}$) or CanSatisfy(P, $\{x_k=b_k\}$)
  - one of those must work if P is satisfiable
- Running time satisfies $T(n) = 3\,T(n - 1) + O(m)$
  - solution is $O(m\,3^n)$
  - that's actually worse than brute force!

Improved Algorithm

> Improve it with this observation:
   we only care about satisfying P with $x_j = b_j$ if
   there is no way to satisfy it with $x_i = b_i$

> In other words, if there is no solution where f_1 is satisfied, then we should look for solutions where f_2 is T and f_1 is F
  - no point in considering f_1 = T anymore
  - we already showed there is no solution with that property

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Improved Algorithm

- If there is no solution where f_1 is satisfied, then we should look for solutions where f_2 is T and f_1 is F
  - no point in considering f_1 = T anymore

- CanSatisfy(P) iff
  CanSatisfy(P, $\{x_i = b_i\}$) or
  CanSatisfy(P, $\{x_i = \text{not } b_i, x_j = b_j\}$) or
  CanSatisfy(P, $\{x_i = \text{not } b_i, x_j = \text{not } b_j, x_k = b_k\}$)

Improved Algorithm

- CanSatisfy(P) iff
  CanSatisfy(P, $\{x_i = b_i\}$) or
  CanSatisfy(P, $\{x_i = \text{not } b_i, x_j = b_j\}$) or
  CanSatisfy(P, $\{x_i = \text{not } b_i, x_j = \text{not } b_j, x_k = b_k\}$)

- Running time satisfies: $T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(m)$

- Solution is $O(m\ 1.84^n)$
  - not hard to check that this holds
    - use fact that ~1.84 is largest root of $r^3 = r^2 + r + 1$

More Algorithms

- There is a 3-SAT algorithm that runs in $O(1.334^n)$ time
- In practice, SAT solvers work surprisingly well
  - can solve problems with >10k variables and >1m clauses
- Reduction to 3-SAT lets you use this solver to solve your problem
  - note: that does not prove your problem is NP-complete
    - need to reduce from 3-SAT to prove that
  - (Cook proved every NP problem reduces to 3-SAT but the reduction is very inefficient)

Outline for Today

> Search Trees
> 3-SAT
> Knapsack + Vertex Cover
> Register Allocation
> Branch & Bound

Knapsack + Vertex Cover

- **Problem**: Given a set of items $\{(w_i, v_i)\}$, a weight limit W, and a collection of pairs $\{(i, j)\}$, find the subset of items with largest total value subject to the constraints that:
  - total weight is under the limit
  - for each pair (i, j), either i or j (or both) is included

- HW6 was a special case of Knapsack + Independent Set

Knapsack + Vertex Cover

- Saw a recursive strategy earlier
  - efficient if the vertex cover is small
  - it may not be here…

- Alternative strategy: hope that opt solutions are often covers
  - in HW6, opt solution often did not violate independence constraints
  - this strategy will also work well if the vertex cover is small

Knapsack + Vertex Cover

> Recall our algorithm for Vertex Cover:

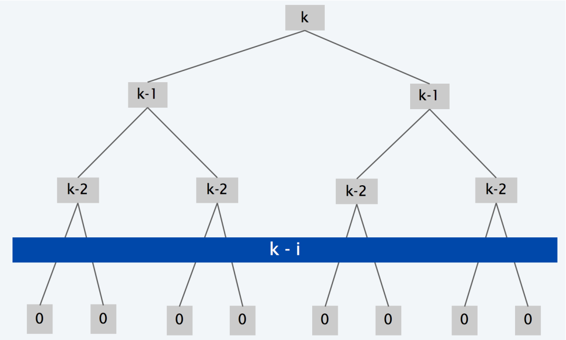
```
VertexCover(G, k):

  if k > 0:
    pick an edge (u,v)
    return VertexCover(G – {u}, k-1) ||
           VertexCover(G – {v}, k-1)
  else:
    return true iff G has no edges
```

Knapsack + Vertex Cover

> Algorithm for Knapsack + Vertex Cover…
  – change leaf nodes to solve Knapsack
    > all items in the cover are included… let knapsack choose the rest

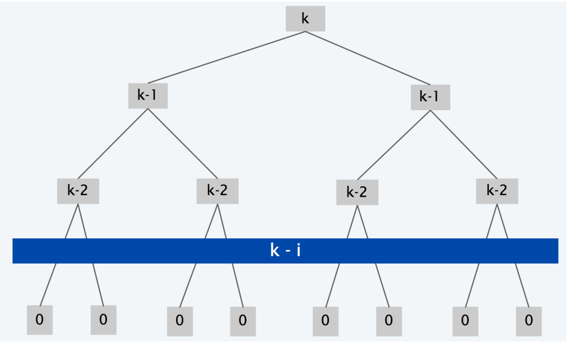
```
KVC(G, S):
  if G has an edge:
    pick an edge (u,v)
    return max(KVC(G – {u}, S + {u}),
               KVC(G – {v}, S + {v})
  else:
    return Knapsack(items – S, W – (weight of S))
             + (value of S)
```

Knapsack + Vertex Cover



- Algorithm for Knapsack + Vertex Cover...
  - change leaf nodes to solve Knapsack
    - all items in the cover are included... let knapsack choose the rest
- So far, this will search through all possible set covers
  - exponentially many: potentially 2^m in worst case
  - fast if the graph is small
- We can do better if best solutions are usually covers...

Knapsack + Vertex Cover



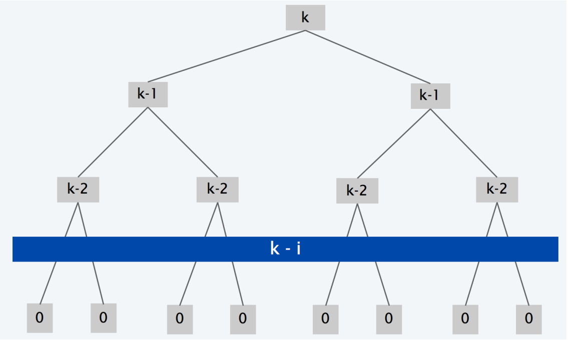
- We can do better if best solutions are usually covers...

- Try solving knapsack at internal nodes of search tree also
  - if knapsack solution is a vertex cover, then no need to recurse further
    - that must be the optimal solution
      - it is optimal amongst all knapsack solutions
      - even those that are not vertex covers
  - if knapsack has no solution, then no need to recurse further
    - there is no solution
  - otherwise, recurse as usual

Knapsack + Vertex Cover

```
KVC(G, S):
  solve knapsack on (items – S) with W – (weight of S)

  if there is no solution:
    return –infinity                                  (no point searching further)

  else if some edge (u,v) is not covered by solution:
    return max(KVC(G – {u}, S + {u}),
               KVC(G – {v}, S + {v}))

  else:
    return knapsack value + (value of S)
```

Knapsack + Vertex Cover



- Try solving knapsack at internal nodes of search tree also
  - stop recursion if we find a solution or there is no solution
- If knapsack solutions are usually covers, then this will be **much** faster
  - ideally, we will solve knapsack only once
  - (this was the case in HW6)
- If knapsack solutions are usually not covers, then this will be slower, but not by much
  - only a factor of 2 slower in the worst case

Principles

> Important lessons about exponential time searches...

1. Slow (poly time) work in each node can easily pay for itself
   - intuition may suggest you want fast checks in each node
     BUT expensive checks often pay for themselves by shrinking tree
   - (this comes up frequently in branch & bound...)

2. Try to limit exponential search to hard constraints only
   - without VC constraints, last problem was efficiently solvable
   - try to only pay exponential time for difficulty of those constraints

Outline for Today

- Search Trees
- 3-SAT
- Knapsack + Vertex Cover
- Register Allocation ⇐
- Branch & Bound

Graph Coloring

- **Problem**: Given a graph G and a number k, find an assignment of colors to nodes such that, for every edge (u,v) in G, u and v are assigned *different* colors.

- Properties:
  - easy when k = 2
    - graph is bipartite iff it is 2-colorable
  - NP-complete when k ≥ 3

3-SAT ≤ 3-Coloring

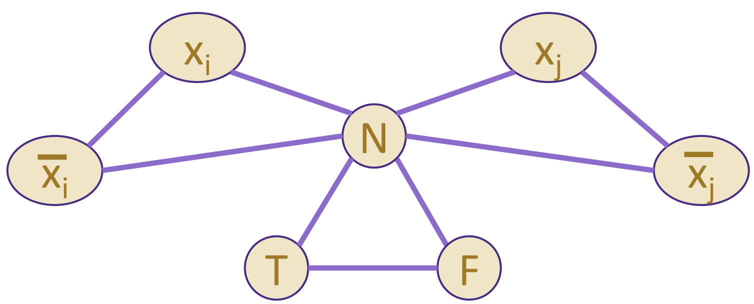
> Given a formula such as

$$\Phi = (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4)$$

> Need to find a graph that is 3-colorable iff the formula is satisfiable

3-SAT ≤ 3-Coloring

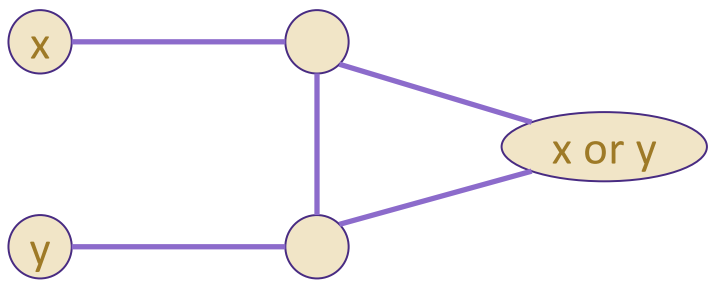
- Create triangles {N, T, F} and {N, x_i, not x_i} for each variable xi
  - all three nodes in a triable **must** get different colors
  - color of T indicates true and color of F indicates false
  - each "x_i" and "not x_i" node is assigned T or F
    - cannot be assigned N color due to triangle



(triangle for each of x_1, x_2, ..., x_n)

3-SAT ≤ 3-Coloring

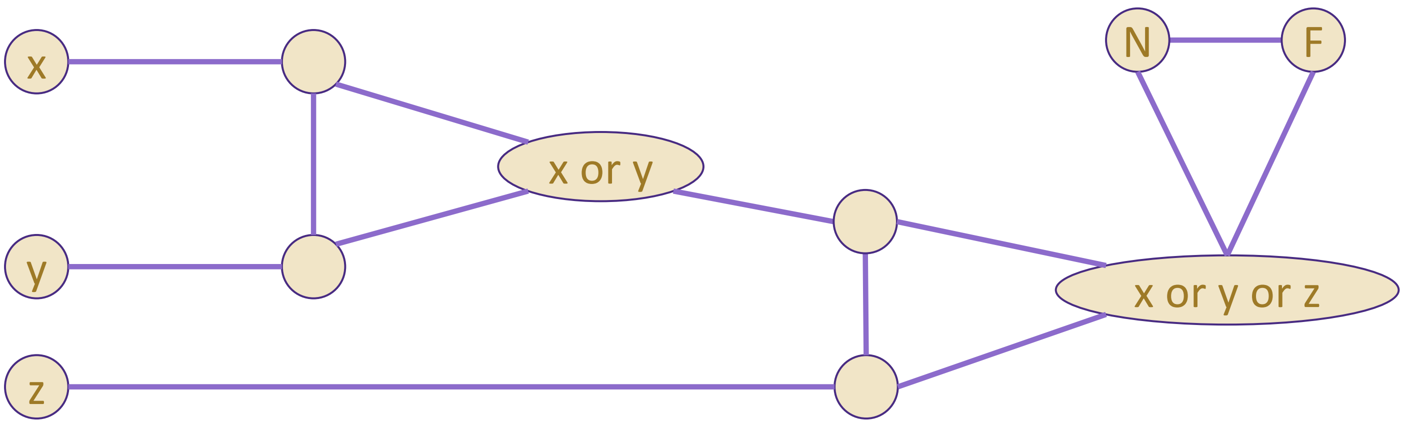
- Represent "x or y" by a triangle:
- Can check that:
  - x = y = T means "x or y" = T
  - x = y = F means "x or y" = F
  - x = T and y = F (or vice versa) means "x or y" is arbitrary



- If we force "x or y" = T,
  then we must have either x = T or y = T or both

3-SAT ≤ 3-Coloring

- Force “x or y or z” to be true like this:
  - triangle with N/F forces “x or y or z” = T
  - that forces at least one of {x, y, z} to be T
    - see previous slide



3-SAT ≤ 3-Coloring

- This is an example of a “gadget” proof
  - triangle connected to x and y is an “OR gadget”
    - represents SAT “or” operator within the context of coloring
- Similar techniques are used in many other reductions
  - depend on careful understanding of details of the problem
    - (that’s why we’re not going to study them carefully…)

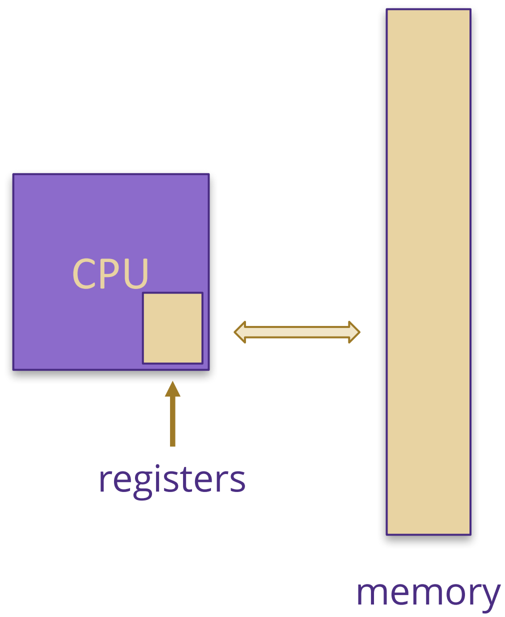
Graph Coloring

> Next: at an important application of graph coloring in compilers

> A little background first...

Computer Architecture

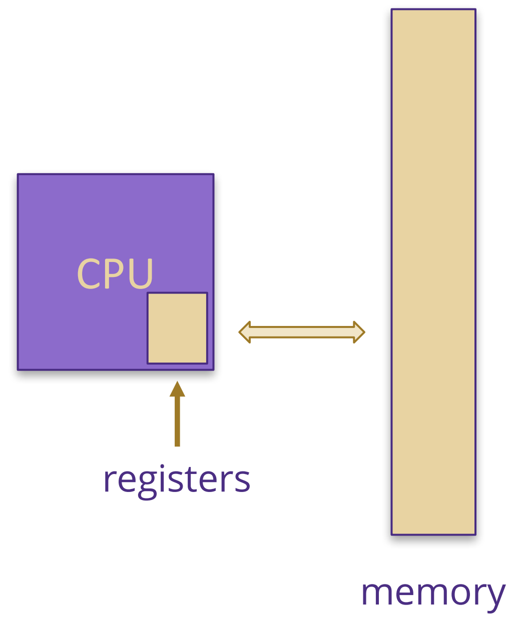
- Typical processor instructions:
  - load from memory to registers
  - store from registers to memory
  - operations on registers:
    - arithmetic
    - comparisons
    - etc.



Register Allocation

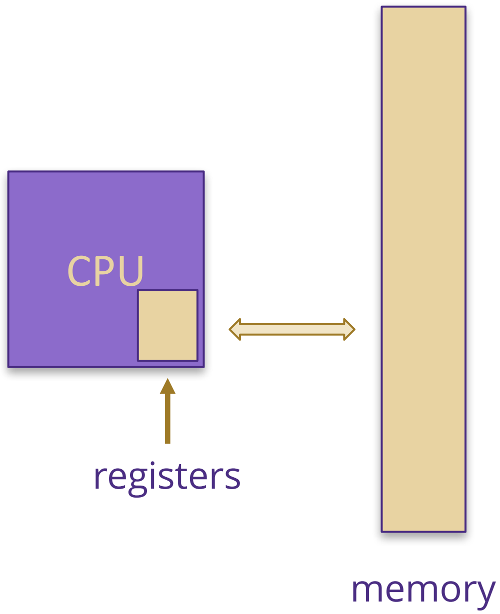
- Compilers translates source code (e.g., Java) to processor instructions

- To do so, it must choose how to assign local variables to registers
  - CPUs have a fixed number (e.g., 32) of registers
  - any two variables needed at the same time should be assigned to different registers
  - those not needed can be "spilled" to memory
    - i.e., written to memory and later read back
    - this has a cost



Register Allocation

- To do so, it must choose how to assign local variables to registers
  - CPUs have a fixed number (e.g., 32) of registers
  - any two variables needed at the same time should be assigned to different registers
- Model as graph coloring:
  - nodes for local variables
  - each color indicates a register
  - edges between local variables used at the same time (cannot be in same register)



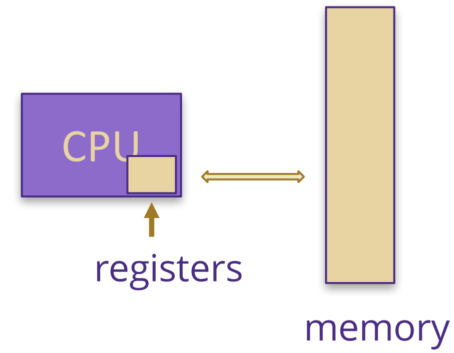
Graph Coloring

- Can speed up the exponential search considerably...
- Idea: simplify the graph by removing all nodes with <k neighbors
  - (neighbors are nodes directly connected to it by edges)
- Any such node can be easily colored no matter what colors are chosen for the other nodes
  - just pick one of the colors not used by any of its neighbors
  - since it has <k neighbors, some color is not used

Graph Coloring

- Idea: simplify the graph by removing all nodes with <k neighbors
  - any such node can be easily colored no matter the colors of the other nodes
  - this can be repeated: removing a node takes away neighbors of other nodes
  - sometimes (not always) this solves the problem
    - simplifies all the way down to an empty graph
- Rather than doing an exponential search over resulting graph, we will change the problem slightly
  - allow (u,v) to have both u and v assigned the same color BUT doing so has an associated <u>cost</u>
  - (cost relates to expense of moving variables in/out of memory)

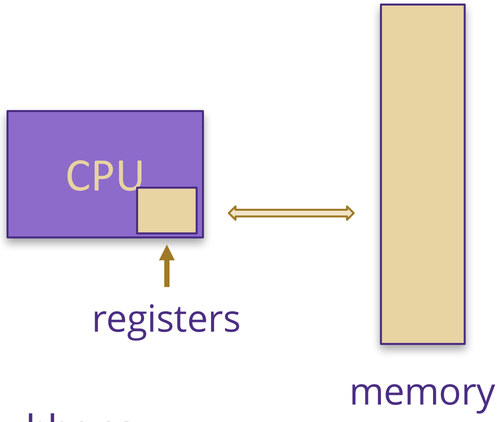
Register Allocation



> Model as variant of graph coloring:
  – given weighted graph G, find a coloring of the nodes *minimizing* sum of costs on conflicting edges
  – (edge (u,v) is conflicting if u and v are assigned same color)

> In particular, we will restrict to colorings produced by the process described before
  – i.e., remove least cost set of edges so that the resulting graph can be colored simply by repeatedly removing nodes with <k neighbors
  – (should still be NP-complete)

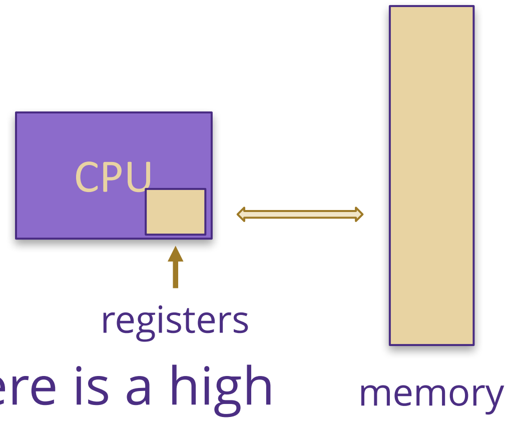
Register Allocation




```
Color(G, k):
  try to solve by repeatedly removing nodes with <k neighbors

  if it works:
    return 0

  else:
    leastCost = infinity
    for every edge (u,v) in resulting Graph:
      cost = Color(G - (u,v), k) + (cost of (u,v))
      leastCost = min(cost, leastCost)
    return leastCost
```

(no edges removed, so no cost)

Register Allocation

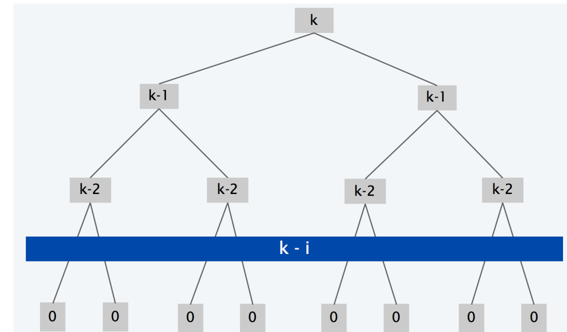


- As with Knapsack, can run very quickly when there is a high likelihood that graph will be colored quickly
  - exact algorithm can still be fast if it usually only takes a few edge removals to get a graph that can be colored
  - unlike K+VC example, it mixes approximation with exponential time search
- This idea is commonly used in real compilers
  - however, they often only solve it approximately (not exactly)
    - sometimes use fixed strategy for which <u>one</u> edge should be removed
    - others perform some amount of search
  - extremely fast (often roughly linear time) in practice

Outline for Today

> Search Trees
> 3-SAT
> Knapsack + Vertex Cover
> Register Allocation
> Branch & Bound

Branch & Bound



- 3-SAT and graph coloring examples were decision problems
  - can stop searching when we find **any** solution
- For optimization, we need to find the **best** solution
  - one approach: solve decision version + binary search
  - usual approach: return the best solution found in subtree
    - root of entire tree returns the best overall solution
    - example: K+VC, min cost graph coloring

Branch & Bound

- For optimization, we need to find the **best** solution
  - usual approach: return the best solution found in subtree
- Can still stop searching a subtree IF
  we can **prove** that it cannot contain the best solution
  - keep track of best value v seen so far (anywhere in the tree)
  - stop if we can prove opt in subtree is worse than v
    - note: do not have to compute opt in subtree to do this!
- Branch (search tree) &
  Bound (eliminate subtree using lower/upper bounds)

Branch & Bound

- Can still stop searching a subtree IF
  we can prove that it cannot contain the best solution
  - keep track of min value v seen so far (anywhere in the tree)
  - stop if we can prove opt in subtree is worse than v

- Bound opt in subtree by <u>removing constraints</u>
  - solving the problem without that constraint can only improve solution
  - if that is still worse than v, then opt in subtree is worse than v as well
    - found opt in a subset of solutions that includes subtree opt

Branch & Bound

- Bound opt in subtree by removing constraints
  - solving the problem without that constraint can only improve solution
- Example: Knapsack + Vertex Cover
  - removing the vertex cover constraints gives knapsack problem
  - if opt solution to knapsack w/out vertex cover constraints is $< v$, then stop
- In particular, want to remove some hard constraints
  - then you get a problem we can solve efficiently
  - reduce your exponential search to just satisfying those
  - only be exponential in distance from easy instances