New Project 1 deadline: Monday, April 6.

New Homework #3 deadline: Wednesday, April 8.

Video of last lecture available on sakai/Resources
What is a hard compiler problem?

Answer: Problems that are NP-complete (non-deterministic polynomial)

Definition: A problem $X$ is NP-complete iff
(1) $X$ is in NP, and
(2) Every problem $Y$ in NP can be reduced in polynomial time to $X$
Hard compiler problems (cont.)

How to prove that a (decision) problem X is in NP-complete?

(1) Show that you can verify in polynomial time that a given solution/witness of X is valid (polynomial time verification)
(2) Show a polynomial time reduction of any instance of a existing/known NP-complete problem to an instance of X

Example “classical” NP-complete problems
• 3 SAT (3 Conjunctive Normal Form Satiability Problem)
• Traveling salesman
• Graph coloring
• Integer programming

Example “compiler” NP-complete problems
• Register allocation
• Instruction scheduling
• Automatic data layout
Example proof outline (Kremer 1993/1995)

**Theorem:** Minimal cost *dynamic data layout problem* is NP-complete

**Problem statement:**
- Program consists of multiple phases
- Phases access multi-dimensional arrays
- Each array has a finite set of candidate layouts (by row, by column, blocked, ...)
- Cost of array access in phase computation depends on array’s layout
- Array remapping (e.g.: from row to column) may be performed between phases
- Remapping is not free, i.e., has a cost

Determine the layout of each array in each phase such that the overall computation and remapping costs are minimized.
“Any advanced / interesting compiler optimization problem is NP-complete” (Keith Cooper, Rice University)

So what to do?

**Option 1** (current wisdom): *Use a heuristic*
This can work well since problem instances that occur in practice may have structure, i.e., exhibit properties that may not lead to exponential cost.

**Option 2**: *Use state-of-the-art integer programming tools*
This will produce the optimal solution (no computation approximation). If there is structure in the problem, the solver will exploit it. If the optimal solution takes too long to compute, return the best feasible solution within a specified time budget (heuristic solution, if needed).

WE WILL USE OPTION 2. ⇒ **GUROBI - MIXED INTEGER PROGRAMMING TOOL**
Mathematical optimization problem

minimize/maximize: \( f(x_1, x_2, \ldots, x_n) \)

subject to: \( g_i(x_1, x_2, \ldots, x_n) \begin{cases} \leq \\ \geq \\ = \end{cases} b_i, \ \forall i \in \{1, \ldots, m\} \)

\( x_j \geq 0, \ \forall j \in \{1, \ldots, n\} \)

- \( x_j \) are the decision variables.
- \( g_i(x_1, x_2, \ldots, x_n) \begin{cases} \leq \\ \geq \\ = \end{cases} b_i \) are structural constraints.
- \( x_j \geq 0 \) are nonnegativity constraints.
- \( f(x_1, \ldots, x_n) \) is the objective function.
- A feasible solution, \( x = (x_1, \ldots, x_n) \) satisfies all constraints.
- The feasible region is the set of all feasible solutions.
- The objective function ranks the feasible solutions.
LP: Decision variables $x_i$ have continuous values (e.g. floats)
MIP: Some decision variables $x_i$ must have integral values (integers)
ILP: All decision variables $x_i$ have integral values (integers)
0-1 ILP: All decision variables $x_i$ are binary, i.e. 0 or 1

$$z^* = \minimize \quad c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$
subject to: $$a_{i1} x_1 + a_{i2} x_2 + \cdots + a_{in} x_n \begin{cases} \geq \ n \ \ \ \ \ \ \ i = 1, \ldots, m \\ = \end{cases} b_i$$
$$x_1, x_2, \ldots, x_n \geq 0$$

- $a_{ij}$, $c_j$, and $b_i$ are data.
- Find $x^*$ satisfying $c_1 x_1^* + \cdots + c_n x_n^* \leq c_1 \hat{x}_1 + \cdots + c_n \hat{x}_n$ for all feasible $\hat{x}$.
- A linear program (LP) is a special type of math program with:
  - $f(x_1, \ldots, x_n) = c_1 x_1 + \cdots + c_n x_n$
  - $g_i(x_1, \ldots, x_n) = a_{i1} x_1 + \cdots + a_{in} x_n, \quad i = 1, \ldots, m$
Example: Food Diet Problem

Want to find a combination of foods (in ounces) that satisfies my nutrient requirements while keeping the overall cost minimal.

<table>
<thead>
<tr>
<th>Food</th>
<th>#1</th>
<th>#2</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Nutrient requirements:
#1: 21, #2: 12

Decision Variables
\[ x_j = \text{# of ounces of food type } j = 1, 2, \ldots, 5 \]

Objective Function
\[ \text{minimize } \quad 20x_1 + 10x_2 + 31x_3 + 11x_4 + 12x_5 \]

Structural Constraints
\[ 2x_1 + 0x_2 + 3x_3 + x_4 + 2x_5 \geq 21 \]
\[ 0x_1 + x_2 + 2x_3 + 2x_4 + x_5 \geq 12 \]

Nonnegativity constraints
\[ x_j \geq 0, j = 1, 2, \ldots, 5 \]
Gurobi - state-of-the-art mixed integer programming tool

Installed on ilab machines, but need to set path:
  if you are running BASH: (most people), type this on the terminal:
      source /koko/system/gurobi901/gurobi.bash

To run a python gurobi interpreter, type: gurobi.sh

You may obtain individual educational/academic licenses directly at gurobi.com for your own laptop. There are online tutorials at gurobi.com. and youtube. Search for “Getting started with gurobi” (part 1 through 3).

On the ilab machines, you can find gurobi “program” examples at /koko/system/gurobi901/linux64/examples
Example: Food Diet Problem

Minimize
20 food.1 + 10 food.2 + 31 food.3 + 11 food.4 + 12 food.5
Subject To
nutrient.1: 2 food.1 + 3 food.3 + food.4 + 2 food.5 >= 21
nutrient.2: food.2 + 2 food.3 + 2 food.4 + food.5 >= 12
Bounds
food.1 >= 0
food.2 >= 0
food.3 >= 0
food.4 >= 0
food.5 >= 0
Integers
food.1 food.2 food.3 food.4 food.5
End

Gurobi lp format:
file FoodDietProblem.lp
**Example: Food Diet Problem**

\[
\text{Minimize} \quad 20 \text{food}.1 + 10 \text{food}.2 + 31 \text{food}.3 + 11 \text{food}.4 + 12 \text{food}.5 \\
\text{Subject To} \\
\text{nutrient}.1: \quad 2 \text{food}.1 + 3 \text{food}.3 + \text{food}.4 + 2 \text{food}.5 \geq 21 \\
\text{nutrient}.2: \quad \text{food}.2 + 2 \text{food}.3 + 2 \text{food}.4 + \text{food}.5 \geq 12 \\
\text{Bounds} \\
\text{food}.1 \geq 0 \\
\text{food}.2 \geq 0 \\
\text{food}.3 \geq 0 \\
\text{food}.4 \geq 0 \\
\text{food}.5 \geq 0 \\
\text{Integers} \\
\text{food}.1 \text{food}.2 \text{food}.3 \text{food}.4 \text{food}.5 \\
\text{End}
\]

**gurobi.sh (python gurobi shell):**

```python
m = read('FoodDietProblem.lp')
m.optimize()
m.write('FoodDietProblem.sol')
```

**Gurobi Ip format:**

```
file FoodDietProblem.lp

Minimize
  20 food.1 + 10 food.2 + 31 food.3 + 11 food.4 + 12 food.5
Subject To
  nutrient.1: 2 food.1 + 3 food.3 + food.4 + 2 food.5 \geq 21
  nutrient.2: food.2 + 2 food.3 + 2 food.4 + food.5 \geq 12
Bounds
  food.1 \geq 0
  food.2 \geq 0
  food.3 \geq 0
  food.4 \geq 0
  food.5 \geq 0
Integers
  food.1 food.2 food.3 food.4 food.5
End
```
Example: Food Diet Problem

Minimize
\[ 20 \text{food.1} + 10 \text{food.2} + 31 \text{food.3} + 11 \text{food.4} + 12 \text{food.5} \]

Subject To
\[ \text{nutrient.1}: 2 \text{food.1} + 3 \text{food.3} + \text{food.4} + 2 \text{food.5} \geq 21 \]
\[ \text{nutrient.2}: \text{food.2} + 2 \text{food.3} + 2 \text{food.4} + \text{food.5} \geq 12 \]

Bounds
\[ \text{food.1} \geq 0 \]
\[ \text{food.2} \geq 0 \]
\[ \text{food.3} \geq 0 \]
\[ \text{food.4} \geq 0 \]
\[ \text{food.5} \geq 0 \]

Integers
\[ \text{food.1 food.2 food.3 food.4 food.5} \]

End

gurobi.sh (python gurobi shell):

```python
m = read('FoodDietProblem.lp')
m.optimize()
m.write('FoodDietProblem.sol')
```

Gurobi lp format:

file FoodDietProblem.lp

Minimize
\[ 20 \text{food.1} + 10 \text{food.2} + 31 \text{food.3} + 11 \text{food.4} + 12 \text{food.5} \]
Subject To
\[ \text{nutrient.1}: 2 \text{food.1} + 3 \text{food.3} + \text{food.4} + 2 \text{food.5} \geq 21 \]
\[ \text{nutrient.2}: \text{food.2} + 2 \text{food.3} + 2 \text{food.4} + \text{food.5} \geq 12 \]
Bounds
\[ \text{food.1} \geq 0 \]
\[ \text{food.2} \geq 0 \]
\[ \text{food.3} \geq 0 \]
\[ \text{food.4} \geq 0 \]
\[ \text{food.5} \geq 0 \]
Integers
\[ \text{food.1 food.2 food.3 food.4 food.5} \]
End

Linux prompt:

```bash
[uli@ls data]$ more FoodDietProblem.sol
# Objective value = 131
food.1 0
food.2 0
food.3 0
food.4 1
food.5 10
[uli@ls data]$ 
```
**Option 1** (current wisdom): **Use a heuristic**
This can work well since problem instances that occur in practice may have structure, i.e., exhibit properties that may not lead to exponential cost.

**Option 2:** **Use state-of-the-art integer programming tools**
This will produce the optimal solution (no computation approximation). If there is structure in the problem, the solver will exploit it. If the optimal solution takes too long to compute, return the best feasible solution within a specified time budget (family of heuristic solutions).

Solution strategy for hard problem X:
Reduce / Map instance of problem X to a linear programming problem and solve it via gurobi
Solution strategy for hard problem X:
Reduce / map instance of problem X to a linear programming problem and solve it via gurobi

Many compiler / runtime problems can be expressed as minimal / maximal solutions to weighted graph problems.

automatic data layout problem

Soda can selection problem under resource constraints and quality weights (service robot)
Solution strategy for hard problem X: Reduce / map instance of problem X to a linear programming problem and solve it via gurobi.

Reduction example: Reduce instance of a data layout problem to a 0-1 linear integer programming problem (0-1 problem).

0-1 linear integer programming problem:
- Decision variables are binary (0 or 1)
- Constraints and objective functions are linear
Solution strategy for hard problem X:
Reduce / map instance of problem X to a linear programming problem and solve it via gurobi

Reduction example: Reduce instance of a data layout problem to a 0-1 linear integer programming problem (0-1 problem).

Construction:
- One variable $x_i$ for each node and $x_j$ for each edge
- Objective function: $\sum cost_i * x_i + \sum cost_j * x_j$
- Two types of constraints:
  - layout and remapping constraints
- There are many possible reductions (translations)
- Particular constraint formulations can result in significantly different solution times
Solution strategy for hard problem X:
Reduce / map instance of problem X to a linear programming problem and solve it via gurobi

**Layout constraints:**
Ensure that a single layout has to be selected for each phase

<table>
<thead>
<tr>
<th>Layout</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout 1:</td>
<td>$x_{11} + x_{12} = 1$</td>
</tr>
<tr>
<td>Layout 2:</td>
<td>$x_{21} + x_{22} = 1$</td>
</tr>
<tr>
<td>Layout 3:</td>
<td>$x_{31} + x_{32} = 1$</td>
</tr>
</tbody>
</table>

Automatic data layout problem

Layout constraints for first three phases
Solution strategy for hard problem X:
Reduce / map instance of problem X to a linear programming problem and solve it via gurobi

**remapping constraints:**
Ensure that if an edge’s source and sink node is selected, the incident edge has to be selected as well

Remapping edge for array “a” between P₂ and P₄ represented by decision variable \( x_{22} \).

Remap: \( x_{22} \times x_{41} = x_{22}^{22} \)

automatic data layout problem
Solutions to hard compiler/runtime problems

Solution strategy for hard problem X:
Reduce / map instance of problem X to a linear programming problem and solve it via gurobi

remapping constraints:
Ensure that if an edge’s source and sink node is selected, the incident edge has to be selected as well

automatic data layout problem

Remapping edge for array “a” between $P_2$ and $P_4$ represented by decision variable $x_{22}^{241}$

remap: $x_{22}^4 \times x_{41} = x_{22}^{241}$

DOES NOT WORK
SINCE NOT A LINEAR CONSTRAINT!
Solution strategy for hard problem X:
Reduce / map instance of problem X to a linear programming problem and solve it via gurobi

remapping constraints:
Ensure that if an edge’s source an sink node is selected, the incident edge has to be selected as well

This works but is not efficient (long solution times)!
Summary

• By reducing a hard compiler / runtime system problem to an integer programming problem, you can exploit decades in R&D in solving problems through integer programming. Many research groups use this technology.

• The particular translation, i.e., linear programming formulation is typically not unique. Efficiency of solution depends on formulations, with possibly orders of magnitude difference in solution time.

• Number of variables and constraints (size of the problem formulation) does not directly reflect the time needed to solve the problem. (*

• Tools such as gurobi can generate families of heuristics such as
  - return first feasible solution found
  - return best feasible solution within a time constraint (e.g.: 2 secs)
  - return first solution within x% of optimal

(*) In my experience, it often takes longer to read / communicate the problem to gurobi than actually solving it.