Constraint Satisfaction

Finding a way that satisfies given conditions
e.g. How to obtain enough credits without attending Monday classes
- Ways are selections of classes here
- “Without attending Monday classes”
  A subsetting constraint on possible “ways”
- “To obtain enough credits”
  An inequality constraint on obtained credits
- Implicit constraints: “Two classes held in the same time slot cannot be taken”, etc

Formalization of Constraint Satisfaction Problems

Finding a set of values \(<x_1, ..., x_n>, x_i \in D_i\) that satisfies a condition \(C(x_1, ..., x_n)\)
Solutions are not necessarily unique
e.g. Integer \(x\) is greater than 3 and less than 6
  - Domain of the variable \(x\) is integer
  - Constraints: \(3 < x < 6\)
  - Solutions: \(x = 4\) and \(x = 5\)
No newly added information in the solution
⇒ The problem is to modify the constraints into a simpler form
  The definition of being “simple” may vary

Classes of Constraint Satisfaction Problems

- Variable domains
  - Finite domains (one of finite number of candidates)
  - Infinite domains (integer, real, complex, …)
  - Structures (trees, graphs, …) usually decomposed into components
- Kinds of constraints
  - Equality (linear, nonlinear, …)
  - Inequality (total or partial order)
  - Logical combination of multiple constraints
**Integer Domain, Equality Constraints**

Cranes and Turtles

“There are 5 animals, cranes and/or turtles, and the sum of their legs is 14. How many cranes and turtles are there?”

- Variables $x$ and $y$ in integer domain
- Constraints:
  - $x + y = 5$
  - $4x + 2y = 14$
- Solution: $x = 2, y = 3$

**Complex Domain, Equality Constraints**

Solving quadratic equations

- Variable domain is complex numbers
- Constraint is a quadratic equation
  
  \[ ax^2 + bx + c = 0 \]

  Solution:
  \[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

  In general, solutions to constraint satisfaction problems restates the given information in some other forms

**Real Domain, Inequality Constraints**

The region satisfying the constraint

- Below this line
- Above this line

**Integer Domain, Inequality Constraints**

The region satisfying the constraint

- Below this line
- Above this line
Finite Domain, Combinatorial Constraints

Thesis defense: selecting slots, rooms, judges
- Domains of slots, rooms, judges are finite
- Constraints:
  - Each judge has his/her expertise area
  - At one time slot and one room, only one defense can be done
  - Judges can attend one defense at a time
  - All the defenses should be done

Most of combinatorial constraint satisfaction problems are NP-hard with exponential complexity.

P, NP, NP-hard, NP-complete

P: Problems that can be solved within time expressed as a polynomial of input size
NP: Problems for which whether a given value is a solution or not can be told in polynomial time
NP is from Nondeterministic Polynomial time, meaning that nondeterministic Turing machines can solve them in polynomial time
NP-hard: Problems to which any NP problems can be transformed in polynomial time
NP-complete: NP-hard problems that are NP
In short, the hardest subset of NP
Whether P=NP or not is still an open problem.

P, NP, NP-hard, and NP-complete Visually

Eight Queens Problem

Placing eight queens on a chess board so that none of them offend one another
A Simple Strategy for Combinatorial Constraints

Generate and Test
- Generate all possible combinations systematically
- Test all of the generated combinations whether they satisfy the constraints

When the number of variables increases, the number of combinations becomes huge

"Combinatorial Explosion"
- e.g. Ways to put 8 queens on a 8 by 8 board is $\binom{64}{8} = 4.426 \ldots \times 10^9$

Tree Structure of Generating Combinations

Branch factor $b$
- # of selections $d$
- # of combinations $= b^d$

Solving Combinatorial Constraints More Efficiently

Reduction of tree generation: branch pruning
- Generation scheme that reflects the constraints
- Constraint check before complete generation: backtracking
- Checking the possibility of satisfaction during generation: forward checking
- If the constraints enforce a selection, make that decision earlier: determinacy detection
  - Generalization: Make decisions earlier on selections with less alternatives

In Case of Eight Queens
- Generation scheme
  - Only one queen in a row
  - $\binom{64}{8} = 4.426 \ldots \times 10^9 \Rightarrow 8^8 = 16,777,216$
- No more queens on columns already with one
  - $8^8 \Rightarrow 8! = 40,320$
- Backtracking
  - Constraint checks made each time a queen is placed, rather than after placing all the queens
  - More devices are needed for larger-scale problems
Forward Checking
- After several queens are placed, check whether all the remaining rows still have safe columns

Determinacy Detection
- After several queens are placed, if there is a row with a unique safe position, place a queen there first

Optimization Problems
- Multiple solutions satisfying the constraints may have relative merits
- Should choose the best solution with the highest merits
- Merits should have a total order
- With a partial order, the best might not exist
- When more than one aspects are to be considered, they should be combined into one single value with total order

Formalization of Constraint Optimization
- Minimize (or maximize) the objective function $f(x_1, ..., x_n)$ with values $<x_1, ..., x_n>$, $x_k \in D_k$ that satisfies a condition $C(x_1, ..., x_n)$
- An objective function to be minimized is also called a cost function
- A set of values $<x_1, ..., x_n>$ that satisfies the constraint but may or may not give the minimum (or maximum) of the objective function is called a feasible solution
**Objective Function**

- Values of the objective function should be compared in a total order.
- Objective functions which do not have numeric values can be considered.
- Numerical values, however, is desirable as some algorithms require them to work.
- For example, values of objective functions for partial problems may be summed up to obtain the value for the whole problem.

**Linear Programming (LP)**

Minimization of linear objective function under linear inequality constraint.

**Integer Programming (IP)**

Linear Programming in Integer Domain.

**Combinatorial Optimization**

- Maximization/minimization of objective function under combinatorial constraints on a finite domain.
- A variety of constraints.
- No restriction on objective functions.
- Objective functions may be non-linear.
- No efficient general algorithm is known.
- Usually, some smoothness is assumed: Similar arguments result in similar values.
Knapsack Problem: a Typical Combinatorial Optimization

- A knapsack of some given capacity
- Goods of various sizes and values
- Find the highest-valued set of goods that fits in the knapsack

Formulation of the Knapsack Problem

- Variables $x_k$: Boolean variables indicating whether or not the item $k$ is included in the set
- Constraint: Sum of the sizes of items $s_k$ should not exceed the knapsack capacity $c$ (linear inequality)
  \[ \sum_{k=1}^{n} (x_k \cdot s_k) \leq c \]
- Objective function: Sum of the values $v_k$ (linear)
  \[ f(x_1, \ldots, x_n) = \sum_{k=1}^{n} (x_k \cdot v_k) \rightarrow \max \]
- Can be regarded as an $n$-dimensional integer programming problem with 0-1 domain

Travelling Salesman Problem: another typical problem

- A number of cities and distances between them are given
- The shortest route to visit all the cities should be found

Formulation of TSP

- Variables $x_k$: The $k$-th city visited
  The domain is the set of cities (finite domain)
- Constraint: Should visit all of $n$ cities
  \[ \{x_1, \ldots, x_n\} = X \]
- Cost function: Total travel distance
  \[ f(x_1, \ldots, x_n) = \sum_{k=1}^{n-1} d(x_k, x_{k+1}) \rightarrow \min \]
Algorithms for Combinatorial Optimization

- **Strict algorithms**
  - The strictly best solution is to be found, i.e., No other feasible solution is better
  - Often requires large computational cost

- **Approximate algorithms**
  - Find a solution close to the best, i.e., Not necessarily the real best
  - A variety of definitions on how close the obtained solution should be
  - Often decreases the computational cost

Simple Strict Algorithm: Generate and Evaluate

- Generate all feasible solutions and evaluate them
  1. Generate all feasible solutions systematically
     - Efficient algorithms for combinatorial constraint satisfaction may be used
  2. Compute the objective function for each
  3. The solution giving the maximum/minimum value is the optimum

Simple and easy to understand, but inefficient when there are a huge number of feasible solutions, which is frequently the case

Complexity of Generate-and-Evaluate Scheme

- Huge number of feasible solutions may exist
  - When each selection has $b$ choices and the number of selections (tree depth) is $d$, there are $b^d$ leaves (exponential)
  - For a knapsack problem with 10 items, there are 1024 feasible solutions; with 20 items, about one million; with 30, one billion
  - TSP with 10 cities, there are 362,880 feasible solutions; with 100 cities, the number has 156 digits!

Branch and Bound

- The tree of feasible solutions is expanded in a depth-first order
  - During tree expansion, if a node is known never to have leaves better than an already known solution, no further expansion below the node is required
  - If the merit upper bound of possible further choices is known, the following condition can be used to prune the branches below the node
    - $[\text{sum of merits of already chosen part}] + [\text{upper bound of sum of merits of further choices}] \leq [\text{merit of already known solution}]$
Branch and Bound

```
Branch and Bound

known
solution

merit upper
bound

red > green + merit upper bound ?
```

B&B for Knapsack Problems

- Decide whether to put an item into the knapsack in the descending order of value per size, \( v_k/s_k \)
- The total value of the items in first \( m \) choices is
  \[
  \sum_{k=1}^{m} (x_k \cdot v_k)
  \]
- For all remaining \( n - m \) items, value per size never exceeds \( v_{m+1}/s_{m+1} \) and thus an upper bound of the sum can be given as
  \[
  v_{m+1}/s_{m+1} \times \left( S - \sum_{k=m+1}^{n} s_k \right)
  \]

Relaxation of Constraints

- Some problems can be more efficiently solved if constraints are relaxed
  e.g., Widening the domain of integer programming from integer to real makes it an LP problem, for which we have efficient algorithms
- The optimum of the original problem can never be better than that of the relaxed problem
  \( \Rightarrow \) A relaxed problem gives an upper bound

Relaxing IP to LP

```
Relaxing IP to LP

optimum of integer programming

optimum of linear programming

x
```
Solving Relaxed Problems to Estimate Upper Bounds

- During tree search, choices not made yet can be considered to be a subproblem.
- The solution of a relaxed subproblem gives an upper bound of the original subproblem.
- This information can be used for search pruning.
- Useful when:
  - Subproblems can be cleanly extracted,
  - Relaxed subproblems have optimum not too much different from the original, and
  - Efficient algorithms are known.

Relaxation of Knapsack Problem to LP

- Original Knapsack:
  \[ \sum_{k=1}^{n} (x_k \cdot v_k) \leq S \]
  \[ \forall x_k \in \{0, 1\} \]
  \[ f(x_1, \ldots, x_n) = \sum_{k=1}^{n} (x_k \cdot v_k) \rightarrow \text{max} \]

- Relaxed Knapsack:
  \[ \sum_{k=1}^{n} (x_k \cdot v_k) \leq S \]
  \[ \forall x_k \in [0, 1] \]
  \[ f(x_1, \ldots, x_n) = \sum_{k=1}^{n} (x_k \cdot v_k) \rightarrow \text{max} \]

- This is an LP problem and can be solved in polynomial time.

Principle of Optimality

- "The optimum solution of a subproblem forms a part of the optimum solution of the whole problem."
- If there exists a better solution to a subproblem, replacing the part with that solution improves the solution of the whole problem.
- Note: This is applicable only when subproblems are independent.
- E.g., in TSP, optimum subroute for a subset of cities might not be a part of the optimum route for all the cities; Start and end points have to agree to merge the subroutes into one.

Dynamic Programming (DP)

- An algorithm derived from the principle of optimality.
  1. Solve small subproblems.
  2. Gradually combine solutions of subproblems.
  3. Record optimum solutions of subproblems to avoid recomputation: memoization.

- Conditions to apply dynamic programming:
  - Subproblems should be mutually independent.
  - The number of subproblems is small enough so that their optimum solutions can be recorded.
Optimization on State Transitions

- A set of state $S = \{ s_0, s_1, \ldots, s_n \}$
- Cost of transition from $s_i$ to $s_j$ is $c_{ij}$
- At time $t$, the state is $x_t$
- Minimize the total cost of transitions from the initial state $x_0$ to the final state $x_n$

$$f(x_0, \ldots, x_n) = \sum_{t=0}^{n-1} c_{x_t, x_{t+1}}$$

Optimum Solution Expressed as Recurrence Equations

- The cost sum after $t = u$ can be defined as
  $$f_u(x_u, \ldots, x_n) = \sum_{t=u}^{n-1} c_{x_t, x_{t+1}}$$
- The following recurrence formula is observed
  $$f_u(x_u, \ldots, x_n) = f_{u+1}(x_{u+1}, \ldots, x_n) + c_{x_u, x_{u+1}}$$
  $$\min_{x_u, x_{u+1}} \left\{ f_{u+1}(x_{u+1}, \ldots, x_n) + c_{x_u, x_{u+1}} \right\}$$
  $$f_0(x_0, \ldots, x_n) = f(x_0, \ldots, x_n)$$

Computing Edit Distance

- Edit a character string to make another
- A given finite set of primitive editing operations
  - Deletion: cost 3 *ABCD* → *ABD*
  - Insertion: cost 4 *ABD* → *AYBD*
  - Replacement: cost 5 *AYBD* → *XYBD*
- Find a sequence of operations with the least cost to obtain the target string
- A problem that completely fits the DP framework
- States: Strings being edited
- Costs of primitive operations are independent

Editing a Sequence

- Record in each node the minimum cost to reach there
- In practice, only a single row of memory is required at a time
  \(O(n^2)\) time and \(O(n)\) space
Matrix Chain Multiplication: A More Complex DP

Finding the best order of computing matrix multiplication chain
- Needs the product of many matrices
- Associativity allows any computation order
  \((A_1 \times A_2 \times A_3) \times A_4 = A_1 \times (A_2 \times (A_3 \times A_4)) = \ldots\)
- The order affects the computational cost
- **The number of scalar multiplications should be minimized**
- The number of possible ordering increases exponentially proportional to the number of matrices

Recurrence Equation on Computational Cost

With \(m_{i,j}\) being the least number of scalar multiplications to obtain \(A_i\)
- As \(A_{i,j}\) is obtained by \(A_{i,k} \times A_{k,j}\) for some \(k\),
  we have the following recurrence equation

\[
m_{i,j} = \begin{cases} 
0 & ; i = j \\
\min_{i < k < j} (m_{i,k} + m_{k,j} + p_{i,k}p_{j}) & ; i \neq j
\end{cases}
\]

Matrix Chain Multiplication

- Size of the matrix \(A_i\) is \(p_{i-1} \times p_i\)
- Multiplying a \(p \times q\) matrix by a \(q \times r\) matrix requires \(p \times q \times r\) scalar multiplications
- By \(A_{i,j}\) we mean the product \(A_i \times \ldots \times A_j\)
  \(A_{i,j}\) has the size \(p_{i-1} \times p_j\)

Matrix Chain Computation as DP

- Subproblems: Computing \(A_{i,j}\) (\(1 \leq i < j \leq n\))
  These subproblems are independent
- The number of subproblems is \(O(n^2)\) where \(n\) is the number of matrices
  \(\Rightarrow\) Small enough to record
  1. Make a cost table of \(m_{i,j}\)
     With initial values of infinity (cost unknown)
  2. Compute \(m_{i,j}\) starting with \(m_{i,i+1}\) and gradually widening the range
  3. When \(m_{1,n}\) is reached, trace back the way it is computed
Matrix Chain Computation

Approximating Algorithms

Approximating Algorithms

Algorithms, usually for NP-hard problems, that give solutions not necessarily optimal but hopefully acceptable (suboptimal)
- Polynomial-time approximation
- Algorithms that repeatedly search for better solutions

Polynomial-Time Approximation

By permitting \((1 + \varepsilon)\) times the cost of the optimum solution, computational time might be made proportional to a polynomial of the problem size

- Computational complexity can be polynomial for fixed \(\varepsilon\) but rapidly becomes larger for smaller \(\varepsilon\), for example, \(O(n^{1/\varepsilon})\), or even \(O(n^{\exp(1/\varepsilon)})\)
- Thus, the algorithms are not practical for problems requiring small \(\varepsilon\)

Iterative Improvement Methods

1. Find an initial feasible solution somehow, which should satisfy the constraint but may be far from optimal
2. The solution is modified a bit without violating the constraints, making the next feasible solution (neighbor solution)
3. Repeat the process until some termination condition is reached

Small modifications are expected to lead to a little better feasible solutions

Iterative Improvement Methods
Simple Iterative Improvement

- In the step 2 of the previous page, always choose the best among the neighbor solutions
- Simple and efficient
- Several names
  - Local Search
  - Greedy Search
  - Hill Climbing

Local Search

1. If there exists a better feasible solution in the neighborhood of the current solution, make that solution current
2. Repeat this until there is no better solutions in the neighborhood

Neighborhood: A set of feasible solutions that can be easily derived from the current solution
- Usually, some of the variables comprising the solution are modified
- Neighborhoods too wide may incur high cost
- Should be able to cover all the feasible solutions

K-Opt method: A Neighborhood Construction Scheme for TSP

- \( k \) paths in the solution are cut and
- With \( n \) cities, neighborhood size is \( O(n^k) \)
- Commonly used \( k \) are 2 and 3

Or-Opt method: Another Neighborhood for TSP

- \( s \) cities in the route are removed and inserted again at a different position
- Neighborhood size is \( O(n^s) \)
- Commonly used \( s \) are 1 through 3
Convergence to Local Optima

Local search may result in a **locally optimal solution** which is far from the global optimum.

- **cost**
- **solution space**
- **initial solution**
- **one of local optima**
- **The global optimum**

Summary

Constraint satisfaction and constraint optimization problems

- Combinatorial optimization
  - Strict algorithms
    - Pruning, forward checking, lower bound
    - Dynamic programming
  - Approximate algorithms
    - Algorithms with known approximation errors
    - Iterative Improvement
      - But, local search may lead to a local optimum ...

Next Week

- Iterative improvement methods that can escape from local optima
- Are there general methods that are not specific to problem domains?
  - Metaheuristics