Chapter 1

Introduction: Some Representative Problems

1.1 First Problem: Stable Matching

Student Preferences

<table>
<thead>
<tr>
<th>Student</th>
<th>Preferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>h1, h2, h4, h3</td>
</tr>
<tr>
<td>s2</td>
<td>h1, h3, h4, h2</td>
</tr>
<tr>
<td>s3</td>
<td>h2, h1, h3, h4</td>
</tr>
<tr>
<td>s4</td>
<td>h2, h4, h1, h3</td>
</tr>
<tr>
<td>s5</td>
<td>h4, h3, h2, h1</td>
</tr>
<tr>
<td>s6</td>
<td>h3, h1, h4, h2</td>
</tr>
</tbody>
</table>

Hospital Preferences

<table>
<thead>
<tr>
<th>Hospital</th>
<th>Preferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>s5, s2, s6, s3, s1, s4</td>
</tr>
<tr>
<td>h2</td>
<td>s1, s3, s6, s2, s5, s4</td>
</tr>
<tr>
<td>h3</td>
<td>s2, s5, s3, s4, s1, s6</td>
</tr>
<tr>
<td>h4</td>
<td>s6, s4, s5, s3, s1, s2</td>
</tr>
</tbody>
</table>

Six medical students and four hospitals

Matching Residents to Hospitals

Goal: Given a set of preferences among hospitals and medical school students, design a self-reinforcing admissions process.

Unstable pair: applicant x and hospital y are unstable if:

- x prefers y to x’s assigned hospital.
- y prefers x to one of its admitted students.

Stable assignment: Assignment with no unstable pairs.

Natural and desirable condition.

Individual self-interest will prevent any applicant/hospital deal from being made.

Simplification of the Problem:

- Each student will find a hospital (create a no-pay hospital)
- Each student will find a different hospital (duplicate hospitals)
- It becomes a boy-girl matching problem.

Stable Matching Problem

Goal: Given n men and n women, find a “suitable” matching.

- Participants rate members of opposite sex.
- Each man lists women in order of preference from best to worst.
- Each woman lists men in order of preference from best to worst.

Stable matching: perfect matching with no unstable pairs.

Stable matching problem. Given the preference lists of n men and n women, find a stable matching if one exists.
Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?
A. No. Bertha and Xavier will hook up.

Q. How about switch B and C?
A. No. Amy and Xavier (or Yancey) will hook up.

Propose-And-Reject Algorithm


Propose-and-reject algorithm.

\begin{algorithm}
\textbf{Initialize} each person to be free.
\While{\text{some man is free and hasn't proposed to every woman}}
\begin{algorithmic}
\State Choose such a man $m$ with 1st woman on $m$'s list to whom $m$ has not yet proposed
\State if ($w$ prefers $m$ to her fiancé $m'$)
\State assign $m$ and $w$ to be engaged, and $m'$ to be free
\State else
\State $w$ rejects $m$
\EndIf
\EndWhile
\end{algorithmic}
\end{algorithm}

Proof of Correctness: Termination

Observation 1. Men propose to women in decreasing order of preference.
Observation 2. Once a woman is matched, she never becomes unmatched; she only "traded up."

Claim. Algorithm terminates after at most $n^2$ iterations of while loop.

PF Each time through the while loop a man proposes to a new woman. There are only $n^2$ possible proposals.
Proof of Correctness: Perfection

Claim. All men and women get matched.

Pf. (by contradiction)
- Suppose, for sake of contradiction, that Zeus is not matched upon termination of algorithm.
- Then some woman, say Amy, is not matched upon termination.
- By Observation 2, Amy was never proposed to.
- But, Zeus proposes to everyone, since he ends up unmatched.

Proof of Correctness: Stability

Claim. No unstable pairs.

Pf. (by contradiction)
- Suppose A-Z is an unstable pair: each prefers each other to partner in Gale-Shapley matching S*.
  - Case 1: Z never proposed to A.
    - Z prefers his GS partner to A.
    - A-Z is stable.
  - Case 2: Z proposed to A.
    - A rejects Z (right away or later)
    - A prefers her GS partner to Z.
    - In either case A-Z is stable, a contradiction.

Summary

Stable matching problem. Given n men and n women, and their preferences, find a stable matching if one exists.

Gale-Shapley algorithm. Guarantee to find a stable matching for any problem instance.

Q. How to implement GS algorithm efficiently?
Q. If there are multiple stable matchings, which one does GS find?

Efficient Implementation

Efficient implementation. We describe O(n^2) time implementation.

Representing men and women.
- Assume men are named 1, ..., n.
- Assume women are named 1', ..., n'.

Engagements.
- Maintain a list of free men, e.g., in a queue.
- Maintain two arrays wife[m] and husband[w].
  - set entry to 0 if unmatched
  - if m matched to w then wife[m] = w and husband[w] = m

Men proposing.
- For each man, maintain a list of women, ordered by preference.
- Maintain an array count[m] that counts the number of proposals made by man m.

Women rejecting/accepting.
- Does woman w prefer man m to man m'?
  - For each woman, create inverse of preference list of men:
    - wpref[w][j] = m iff women w’s jth preference is m
    - inverse[w][m] = j
  - Constant time access for each query after O(n^2) preprocessing.

Propose-and-Reject Algorithm

Initialize Mpref[ ][], inverse[ ][], stack, count[];
while (stack != empty) {
  m = pop(stack);
  w = Mpref[m][++count[m]];
  if (husband[w] == 0) {
    wife[m] = w; husband[w] = m;
  } else if (inverse[w][m] < inverse[w][husband[w]]) {
    push(husband[w], stack); wife[husband[w]] = 0;
    wife[m] = w; husband[w] = m;
  } else {
    push(m, stack);
  }
}

Initialize each person to be free.
while (some man is free and hasn’t proposed to every woman) {
  Choose such a man m
  w = 1st woman on m’s list to whom m has not yet proposed
  if (w is free)
    assign m and w to be engaged
  else if (w prefers m to her fiancé m’)
    assign m and w to be engaged, and m’ to be free
    also w rejects m
  else
    w rejects m
}

Initialize Mpref[ ][], inverse[ ][], stack, count[];
while (stack != empty) {
  w = Mpref[stack][count[stack]];
  if (husband[w] == 0) {
    w = wpref[w][count[w]];
    if (husband[w] == 0) {
      husband[w] = w;
      if (inverse[w][m] < inverse[w][husband[w]]) {
        push(husband[w], stack);
        wife[husband[w]] = 0;
        wife[w] = w;
        husband[w] = m;
      } else {
        push(m, stack);
      }
    } else {
      push(w, stack);
    }
  }
}
Understanding the Solution

Q. For a given problem instance, there may be several stable matchings. Do all executions of Gale-Shapley yield the same stable matching? If so, which one?

An instance with two stable matchings.
- A-Y, B-X, C-Z.
- A-X, B-Y, C-Z.

Def. Man m is a valid partner of woman w if there exists some stable matching in which they are matched.

Man-optimal assignment. Each man receives best valid partner.

Claim. All executions of GS yield man-optimal assignment, which is a stable matching!
- No reason a priori to believe that man-optimal assignment is perfect, let alone stable.
- Simultaneously best for each and every man.

Stable Matching Summary

Stable matching problem. Given preference profiles of n men and n women, find a stable matching.
- Gale-Shapley algorithm. Finds a stable matching in O(n^2) time.

Man-optimality. In the version of GS where men propose, each man receives best valid partner.
- A is a valid partner of w if there exists some stable matching where a and w are paired.

Q. Does man-optimality come at the expense of the women?

Woman-Pessimality

Woman-pessimal assignment. Each woman receives worst valid partner.

Claim. GS finds woman-pessimal stable matching S*.
- Suppose A-Z matched in S*, but Z is not worst valid partner for A.
- There exists stable matching S in which A is paired with a man, say Y, whom she likes less than Z.
- Let B be Z’s partner in S.
- Z prefers A to B.
- Thus, A-Z is unstable in S.

Man Optimality

Claim. GS matching S* is man-optimal.

PF. (by contradiction)
- Suppose some man is paired with someone other than best partner.
- Men propose in decreasing order of preference ⇒ some man is rejected by valid partner.
- Let Y be first such man, and let A be first valid woman that rejects him.
- Let S be a stable matching where A and Y are matched.
- When Y is rejected, A forms (or reaffirms) engagement with a man, say Z, whom she prefers to Y.
- Let B be Z’s partner in S.
- Z not rejected by any valid partner at the point when Y is rejected by A. Thus, Z prefers A to B.
- But A prefers Z to Y.
- Thus A-Z is unstable in S.

Extensions: Matching Residents to Hospitals

Ex: Men = hospitals, Women = med school residents.

Variant 1. Some participants declare others as unacceptable.

Variant 2. Unequal number of men and women.

Variant 3. Limited polygamy.

Def. Matching S unstable if there is a hospital h and resident r such that:
- h and r are acceptable to each other; and
- either r is unmatched, or r prefers h to her assigned hospital; and
- either h does not have all its places filled, or h prefers r to at least one of its assigned residents.
1. Extension: Stable Roommate Problem

Q. Do stable matchings always exist?
A. Not obvious a priori.

Stable roommate problem:
- 2n people; each person ranks others from 1 to 2n-1.
- Assign roommate pairs so that no unstable pairs.

Observation. Stable matchings do not always exist for stable roommate problem.

3. 1.2 Five Representative Problems

Independent Set
Input. Graph.
Goal. Find maximum cardinality independent set.

* Subset of nodes such that no two joined by an edge.

Exhaustive Search:
- \( G = (V, E), n = |V| \)
- For \( k = n \) downto 2
  - lookForIndependentSet(G, k)
    - for all subset \( S \) of \( V \), \( |S| = k \)
      - checkIndependent(G, S)

Complexity of lookForIndependentSet:
- \( O(2^n k^2) \), where \( O(n, k) = n!/((n-k)!k!) \)
- \( O(n^2 k^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- One of NP-complete problems
- \( O(n^2) \) solution. Enumerate all subsets.

- \( S^* \leftarrow S \)
- foreach subset \( S \) of nodes { check whether \( S \) in an independent set
  - if \( S \) is largest independent set seen so far
    - update \( S^* \leftarrow S \)
  }

- \( O(n^2) \) solution. Enumerate all subsets of \( k \) nodes.

- Complexity of lookForIndependentSet:
  - \( O(2^n k^2) \), where \( O(n, k) = n!/((n-k)!k!) \)
  - \( O(n^2 k^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \) solution. Enumerate all subsets of \( k \) nodes.

- Complexity of lookForIndependentSet:
  - \( O(2^n k^2) \), where \( O(n, k) = n!/((n-k)!k!) \)
  - \( O(n^2 k^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \) solution. Enumerate all subsets of \( k \) nodes.

- Complexity of lookForIndependentSet:
  - \( O(2^n k^2) \), where \( O(n, k) = n!/((n-k)!k!) \)
  - \( O(n^2 k^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \) solution. Enumerate all subsets of \( k \) nodes.

- Complexity of lookForIndependentSet:
  - \( O(2^n k^2) \), where \( O(n, k) = n!/((n-k)!k!) \)
  - \( O(n^2 k^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \) solution. Enumerate all subsets of \( k \) nodes.

- Complexity of lookForIndependentSet:
  - \( O(2^n k^2) \), where \( O(n, k) = n!/((n-k)!k!) \)
  - \( O(n^2 k^2) \), where \( C(n, k) = n!/((n-k)!k!) \)

- \( O(n^2) \), where \( C(n, k) = n!/((n-k)!k!) \)
Interval Scheduling

**Input.** Set of jobs with start times and finish times.

**Goal.** Find maximum cardinality subset of mutually compatible jobs.

### Interval Scheduling: Greedy Algorithm (ch. 4.1)

**Greedy algorithm.** Consider jobs in increasing order of finish time.

- Take each job provided it’s compatible with the ones already taken.

**Implementation.** $O(n \log n)$.

- Remember job $j^*$ that was added last to $A$.
- Job $j$ is compatible with $A$ if $s_j \geq f_{j^*}$.

### Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Proof (by contradiction).**

- Assume greedy is not optimal, and let’s see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

**Greedy:**

- Solution still feasible and optimal but contradicts maximality of $r$.

### Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of mutually compatible jobs.

**Can be reduced to Independent Set of Maximum Weight.**

### Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of mutually compatible jobs.

**Weighted interval scheduling problem.**

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don’t overlap.
- Goal: Find maximum weight subset of mutually compatible jobs.
Unweighted Interval Scheduling Review
Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Weighted Interval Scheduling Review
Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Weighted Interval Scheduling (ch 6.1)
Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. $p(j)$ is the largest index $i < j$ such that job $i$ is compatible with job $j$.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.

Dynamic Programming: Binary Choice
Notation. $OPT(j)$ is the value of an optimal solution to the problem consisting of job requests 1, 2, ..., $j$.

Case 1: $OPT$ selects job $j$.
- Collect profit $v_j$.
- Can’t use incompatible jobs $\{p(j) + 1, p(j) + 2, \ldots, j - 1\}$.
- Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $p(j)$.

Case 2: $OPT$ does not select job $j$.
- Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $j - 1$.

Input: $n$, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $v_1, \ldots, v_n$.
Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$.
for $j = 1$ to $n$
    $M[j] = \emptyset$
for $j = 1$ to $n$
    $M[0] = 0$
Compute-Opt($j$) {
    if ($M[j]$ is empty)
        $M[j] = \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1))$
    return $M[j]$
}

Weighted Interval Scheduling: Brute Force
Brute force algorithm.

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithm.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.

Input: $n$, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $v_1, \ldots, v_n$.
Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$.
for $j = 1$ to $n$
    $M[j] = \emptyset$
for $j = 1$ to $n$
    $M[0] = 0$
Compute-Opt($j$) {
    if ($M[j]$ is empty)
        $M[j] = \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1))$
    return $M[j]$
}
Weighted Interval Scheduling: Running Time

Claim. Memorized version of algorithm takes O(n log n) time.
- Sort by finish time: O(n log n).
- Computing \( p(j) \): O(n log n) via sorting by start time.
- \( \text{M-Compute-Opt}(j) \): each invocation takes O(1) time and either
- (i) returns an existing value \( M[j] \)
- (ii) fills in one new entry \( M[j] \) and makes two recursive calls
- Progress measure \( \Phi \) = \# nonempty entries of \( M \):
  - initially \( \Phi = 0 \), throughout \( \Phi \leq n \).
  - (i) increases \( \Phi \) by 1 \( \Rightarrow \) at most 2n recursive calls.
- Overall running time of \( \text{M-Compute-Opt}(n) \) is O(n).
- \# of recursive calls < \( n \) \Rightarrow O(n).

Remark. O(n) if jobs are pre-sorted by finish times.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithm computes optimal value. What if we want the solution itself?
A. Do some post-processing:
- Run \( \text{M-Compute-Opt}(n) \)
- Run \( \text{Find-Solution}(n) \)

\( \text{Find-Solution}(j) \) {
  if (j = 0)
    output nothing
  else if (\( v_j + M[p(j)] \) > \( M[j-1] \))
    print j
    \( \text{Find-Solution}(p(j)) \)
  else
    \( \text{Find-Solution}(j-1) \)
}

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)
Sort jobs by finish times so that \( f_1 \leq f_2 \leq \cdots \leq f_n \).
Compute \( p(1), p(2), \ldots, p(n) \)

**Iterative-Compute-Opt** {
  \( M[0] = 0 \)
  for \( j = 1 \) to \( n \)
    \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}

Bipartite Matching

Input. Bipartite graph.
Goal. Find maximum cardinality matching.

Matching can be reduced to Independent Set:
Given a graph \( G(V, E) \), create another graph \( G'(E', E') \), where \( E' = \{(x, y) | x \in V, y \in V \text{ and } x \neq y \} \) and \( E' \) shares one endpoint in \( G \), then \( M \) is a maximum matching of \( G \) if \( M \) is a maximum independent set of \( G' \).

5. Competitive Facility Location

Input. Graph with weight on each node.
Game. Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.
Goal. Select a maximum weight subset of nodes. (ch. 9)

Second player can guarantee 20, but not 25.
Five Representative Problems

Variations on a theme: independent set.
Interval scheduling: n log n greedy algorithm.
Weighted interval scheduling: n log n dynamic programming algorithm.
Bipartite matching: mn max-flow based algorithm.
Independent set: NP-complete.
Competitive facility location: PSPACE-complete.