Permutation Representations

- Ordering/sequencing problems form a special type
- Task is (or can be solved by) arranging some objects in a certain order
  - Example: sort algorithm: important thing is which elements occur before others (order)
  - Example: Travelling Salesman Problem (TSP) : important thing is which elements occur next to each other (adjacency)
- These problems are generally expressed as a permutation:
  - if there are $n$ variables then the representation is as a list of $n$ integers, each of which occurs exactly once
Permutation representation: TSP example

- **Problem:**
  - Given $n$ cities
  - Find a complete tour with minimal length
- **Encoding:**
  - Label the cities 1, 2, … , $n$
  - One complete tour is one permutation (e.g. for $n=4$, [1,2,3,4], [3,4,2,1] are OK)
- **Search space is BIG:**
  - for 30 cities there are $30! = 10^{32}$ possible tours

Mutation operators for permutations

- Normal mutation operators lead to inadmissible solutions
  - e.g. bit-wise mutation: let gene $i$ have value $j$
  - changing to some other value $k$ would mean that $k$ occurred twice and $j$ no longer occurred
- Therefore must change at least two values
- Mutation parameter now reflects the probability that some operator is applied once to the whole string, rather than individually in each position
Insert Mutation for permutations

- Pick two allele values at random
- Move the second to follow the first, shifting the rest along to accommodate
- Note that this preserves most of the order and the adjacency information

![Insert Mutation Example](image)

Swap mutation for permutations

- Pick two alleles at random and swap their positions
- Preserves most of adjacency information (4 links broken), disrupts order more

![Swap Mutation Example](image)
Inversion mutation for permutations

- Pick two alleles at random and then invert the substring between them.
- Preserves most adjacency information (only breaks two links) but disruptive of order information

\[
1 2 3 4 5 6 7 8 9 \quad \rightarrow \quad 1 5 4 3 2 6 7 8 9
\]

Scramble mutation for permutations

- Pick a subset of genes at random
- Randomly rearrange the alleles in those positions

\[
1 2 3 4 5 6 7 8 9 \quad \rightarrow \quad 1 3 5 4 2 6 7 8 9
\]

(note subset does not have to be contiguous)
Crossover operators for permutations

- “Normal” crossover operators will often lead to inadmissible solutions

\[
\begin{array}{c}
1 \ 2 \ 3 \ 4 \ 5 \\
5 \ 4 \ 3 \ 2 \ 1
\end{array}
\rightarrow
\begin{array}{c}
1 \ 3 \ 2 \ 1
\end{array}
\]

- Many specialised operators have been devised which focus on combining order or adjacency information from the two parents

Order 1 crossover

- Idea is to preserve relative order that elements occur
- Informal procedure:
  1. Choose an arbitrary part from the first parent
  2. Copy this part to the first child
  3. Copy the numbers that are not in the first part, to the first child:
     - starting right from cut point of the copied part,
     - using the order of the second parent
     - and wrapping around at the end
  4. Analogous for the second child, with parent roles reversed
Order 1 crossover example

- Copy randomly selected set from first parent

\[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\hline
8 & 3 & 7 & 8 & 2 & 6 & 5 \end{array} \]

- Copy rest from second parent in order 1,9,3,8,2

\[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\hline
8 & 3 & 7 & 8 & 2 & 6 & 5 \end{array} \]

Informal procedure for parents P1 and P2:
1. Choose random segment and copy it from P1
2. Starting from the first crossover point look for elements in that segment of P2 that have not been copied
3. For each of these \( i \) look in the offspring to see what element \( j \) has been copied in its place from P1
4. Place \( i \) into the position occupied by \( j \) in P2, since we know that we will not be putting \( j \) there (as is already in offspring)
5. If the place occupied by \( j \) in P2 has already been filled in the offspring \( k \), put \( i \) in the position occupied by \( k \) in P2
6. Having dealt with the elements from the crossover segment, the rest of the offspring can be filled from P2.

Second child is created analogously

Partially Mapped Crossover (PMX)
**PMX example**

1. **Step 1**
   - 123456789
   - 937626514
   - 4587

2. **Step 2**
   - 123456789
   - 937626514
   - 245678

3. **Step 3**
   - 123456789
   - 937626514
   - 932456718

---

**Cycle crossover**

**Basic idea:**

Each allele comes from one parent together with its position.

**Informal procedure:**

1. Make a cycle of alleles from P1 in the following way.
   - (a) Start with the first allele of P1.
   - (b) Look at the allele at the same position in P2.
   - (c) Go to the position with the same allele in P1.
   - (d) Add this allele to the cycle.
   - (e) Repeat step b through d until you arrive at the first allele of P1.

2. Put the alleles of the cycle in the first child on the positions they have in the first parent.

3. Take next cycle from second parent.
Cycle crossover example

- Step 1: identify cycles

- Step 2: copy alternate cycles into offspring

Edge Recombination

- Works by constructing a table listing which edges are present in the two parents, if an edge is common to both, mark with a +

- e.g. [1 2 3 4 5 6 7 8 9] and [9 3 7 8 2 6 5 1 4]

<table>
<thead>
<tr>
<th>Element</th>
<th>Edges</th>
<th>Element</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,5,4,9</td>
<td>6</td>
<td>2,5+7</td>
</tr>
<tr>
<td>2</td>
<td>1,3,6,8</td>
<td>7</td>
<td>3,6,8+</td>
</tr>
<tr>
<td>3</td>
<td>2,4,7,9</td>
<td>8</td>
<td>2,7+, 9</td>
</tr>
<tr>
<td>4</td>
<td>1,3,5,9</td>
<td>9</td>
<td>1,3,4,8</td>
</tr>
<tr>
<td>5</td>
<td>1,4,6+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Edge Recombination 2

Informal procedure once edge table is constructed

1. Pick an initial element at random and put it in the offspring
2. Set the variable current element = entry
3. Remove all references to current element from the table
4. Examine list for current element:
   - If there is a common edge, pick that to be next element
   - Otherwise pick the entry in the list which itself has the shortest list
   - Ties are split at random
5. In the case of reaching an empty list:
   - Examine the other end of the offspring is for extension
   - Otherwise a new element is chosen at random

Edge Recombination example

<table>
<thead>
<tr>
<th>Element</th>
<th>Edges</th>
<th>Element</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5,4.9</td>
<td>6</td>
<td>2.5+7</td>
</tr>
<tr>
<td>2</td>
<td>1.3,0.8</td>
<td>7</td>
<td>3.6,8+</td>
</tr>
<tr>
<td>3</td>
<td>2.4,7.9</td>
<td>8</td>
<td>2.7+, 9</td>
</tr>
<tr>
<td>4</td>
<td>1.3,5.9</td>
<td>9</td>
<td>1.3,4,8</td>
</tr>
<tr>
<td>5</td>
<td>1.4,6+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Choices</th>
<th>Element selected</th>
<th>Reason</th>
<th>Partial result</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>1</td>
<td>Random</td>
<td>$[1]$</td>
</tr>
<tr>
<td>2,5,4.9</td>
<td>5</td>
<td>Shortest list</td>
<td>$[1]$</td>
</tr>
<tr>
<td>4,6</td>
<td>6</td>
<td>Common edge</td>
<td>$[1,5]$</td>
</tr>
<tr>
<td>2,7</td>
<td>2</td>
<td>Random choice (both have two items in list)</td>
<td>$[1,5,6,2]$</td>
</tr>
<tr>
<td>3,8</td>
<td>8</td>
<td>Shortest list</td>
<td>$[1,5,6,2,8]$</td>
</tr>
<tr>
<td>7,9</td>
<td>7</td>
<td>Common edge</td>
<td>$[1,5,6,2,8,7]$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Only item in list</td>
<td>$[1,5,6,2,8,7,3]$</td>
</tr>
<tr>
<td>4,9</td>
<td>9</td>
<td>Random choice</td>
<td>$[1,5,6,2,8,7,3,9]$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Last element</td>
<td>$[1,5,6,2,8,7,3,9,5]$</td>
</tr>
</tbody>
</table>
Multiparent recombination

- Recall that we are not constricted by the practicalities of nature
- Noting that mutation uses 1 parent, and “traditional” crossover 2, the extension to $a>2$ is natural to examine
- Been around since 1960s, still rare but studies indicate useful
- Three main types:
  - Based on allele frequencies, e.g., p-sexual voting generalising uniform crossover
  - Based on segmentation and recombination of the parents, e.g., diagonal crossover generalising n-point crossover
  - Based on numerical operations on real-valued alleles, e.g., center of mass crossover, generalising arithmetic recombination operators

Population Models

- SGA uses a Generational model:
  - each individual survives for exactly one generation
  - the entire set of parents is replaced by the offspring
- At the other end of the scale are Steady-State models:
  - one offspring is generated per generation,
  - one member of population replaced,
- Generation Gap
  - the proportion of the population replaced
  - 1.0 for GGA, 1/pop_size for SSGA
Fitness Based Competition

- Selection can occur in two places:
  - Selection from current generation to take part in mating (parent selection)
  - Selection from parents + offspring to go into next generation (survivor selection)
- Selection operators work on whole individual
  - i.e. they are representation-independent
- Distinction between selection
  - operators: define selection probabilities
  - algorithms: define how probabilities are implemented

Implementation example: SGA

- Expected number of copies of an individual $i$:
  $$E(n_i) = \mu \cdot f(i)/\langle f \rangle$$
  ($\mu =$ pop.size, $f(i) =$ fitness of $i$, $\langle f \rangle =$ avg. fitness in pop.)
- Roulette wheel algorithm:
  - Given a probability distribution, spin a 1-armed wheel $n$ times to make $n$ selections
  - No guarantees on actual value of $n_i$
- Baker’s SUS algorithm:
  - $n$ evenly spaced arms on wheel and spin once
  - Guarantees $\lfloor E(n_i) \rfloor \leq n_i \leq \lceil E(n_i) \rceil$
Fitness-Proportionate Selection

- Problems include
  - One highly fit member can rapidly take over if rest of population is much less fit: Premature Convergence
  - At end of runs when fitnesses are similar, lose selection pressure
  - Highly susceptible to function transposition
- Scaling can fix last two problems
  - Windowing: $f'(i) = f(i) - \beta^t$
    - where $\beta$ is worst fitness in this (last n) generations
  - Sigma Scaling: $f'(i) = \max( f(i) - (f') - c \cdot s_f), 0.0)$
    - where $c$ is a constant, usually 2.0

Function transposition for FPS

![Diagram showing function transposition for FPS](image)
Rank – Based Selection

- Attempt to remove problems of FPS by basing selection probabilities on *relative* rather than *absolute* fitness
- Rank population according to fitness and then base selection probabilities on rank where fittest has rank \( \mu \) and worst rank 1
- This imposes a sorting overhead on the algorithm, but this is usually negligible compared to the fitness evaluation time

Linear Ranking

\[
P_{\text{lin-rank}}(i) = \frac{(2 - s)}{\mu} \left(1 + \frac{2i(s-1)}{\mu(s-1)}\right)
\]

- Parameterised by factor \( s: 1.0 < s \leq 2.0 \)
  - measures advantage of best individual
  - in GGA this is the number of children allotted to it
- Simple 3 member example
A.E. Eiben and J.E. Smith, Introduction to Evolutionary Computing

Exponential Ranking

\[ P_{exp\text{-}rank}(i) = \frac{1 - e^{-i}}{c}. \]

- Linear Ranking is limited to selection pressure
- Exponential Ranking can allocate more than 2 copies to fittest individual
- Normalise constant factor $c$ according to population size

Tournament Selection

- All methods above rely on global population statistics
  - Could be a bottleneck esp. on parallel machines
  - Relies on presence of external fitness function which might not exist: e.g. evolving game players
- Informal Procedure:
  - Pick $k$ members at random then select the best of these
  - Repeat to select more individuals
Tournament Selection 2

- Probability of selecting $i$ will depend on:
  - Rank of $i$
  - Size of sample $k$
    - higher $k$ increases selection pressure
  - Whether contestants are picked with replacement
    - Picking without replacement increases selection pressure
  - Whether fittest contestant always wins
    (deterministic) or this happens with probability $p$
- For $k = 2$, time for fittest individual to take over
  population is the same as linear ranking with $s = 2 \cdot p$

Survivor Selection

- Most of methods above used for parent selection
- Survivor selection can be divided into two approaches:
  - Age-Based Selection
    - e.g. SGA
    - In SSGA can implement as “delete-random” (not recommended) or as first-in-first-out (a.k.a. delete-oldest)
  - Fitness-Based Selection
    - Using one of the methods above or
Two Special Cases

- **Elitism**
  - Widely used in both population models (GGA, SSGA)
  - Always keep at least one copy of the fittest solution so far

- **GENITOR: a.k.a. “delete-worst”**
  - From Whitley’s original Steady-State algorithm (he also used linear ranking for parent selection)
  - Rapid takeover: use with large populations or “no duplicates” policy

Example application of order based GAs: JSSP

Precidence constrained job shop scheduling problem
- J is a set of jobs.
- O is a set of operations
- M is a set of machines
- \( \text{Able} \subseteq O \times M \) defines which machines can perform which operations
- \( \text{Pre} \subseteq O \times O \) defines which operation should precede which
- \( \text{Dur} : \subseteq O \times M \rightarrow IR \) defines the duration of \( o \in O \) on \( m \in M \)

The goal is now to find a schedule that is:
- Complete: all jobs are scheduled
- Correct: all conditions defined by \( \text{Able} \) and \( \text{Pre} \) are satisfied
- Optimal: the total duration of the schedule is minimal
Precedence constrained job shop scheduling GA

- Representation: individuals are permutations of operations
- Permutations are decoded to schedules by a decoding procedure
  - take the first (next) operation from the individual
  - look up its machine (here we assume there is only one)
  - assign the earliest possible starting time on this machine, subject to
    - machine occupation
    - precedence relations holding for this operation in the schedule created so far
- fitness of a permutation is the duration of the corresponding schedule (to be minimized)
- use any suitable mutation and crossover
- use roulette wheel parent selection on inverse fitness
- Generational GA model for survivor selection
- use random initialisation

JSSP example: operator comparison