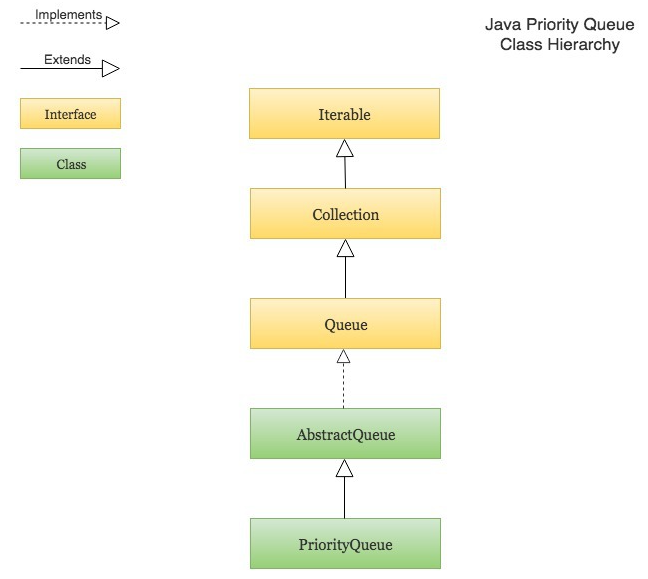
**Java Tips**

* LinkedList:
  1. Part of the *Collection* framework in *java.util.package*
  2. Linear data structure – elements are not stored in contiguous locations
  3. Every element is a separate object with a data part and address part.
  4. The elements are linked using pointers and addresses.
  5. Suitable for dynamic manipulation, such as insertion/deletion
  6. **Limitation**: elements cannot be accessed directly. Instead, we need to start from the head and follow through the link to reach the node we wish to access.
* PriorityQueue:
  1. Special type of queue wherein all the elements are **ordered** according to their natural ordering or based on a custom *Comparator* supplied at the time of creation.
  2. The **front**: least element; the **rear**: the greatest element.



|  |
| --- |
|  |
|  |

**Search Algorithm**

|  |  |  |
| --- | --- | --- |
|  | **Advantage** | **Disadvantage** |
| **DFS** | * Require less space: O(d) | * Delay search for node close to the root but not belonging to the first few deep substrees |
| **BFS** | * Good for local search | * Require more space: O(n) |

1. Iterative deepening depth-first search (IDDFS) is invented to inherit the advantages of both DFS and BFS.
   1. The algorithm visits the search tree in the same order as DFS
   2. The cumulative order in which nodes are first visited is BFS
2. A\* Search
   1. Find path that minimizes *f(x)*:
   2. *F(x) = g(x) + h(x)*
   3. *G(x)* is the cost function, *h(x)* is the estimated cost from the current to destination nodes.

**Heuristics vs meta-heuristics**

1. Both heuristics and meta-heuristics aim to get reasonably good optimal solution (best possible solution is not guaranteed) in a short time.
2. However, heuristics is specific to the underlying problem at hand, whereas meta-heuristics are problem independent and therefore could be applied to any kinds of problems.

**Tabu Search**

1. Local search algorithm such as hill climbing is gradient based approach. The algorithm is easily stuck in local optimum and it takes repetitive initializations of the algorithm to detect the global maximum.
2. Tabu Search is designed to help the optimum-searching algorithm escape from the local optimum by prohibiting returning to the previously visited state.
3. When a move is prohibited, aka made tabu, it is added to a list called **tabu list**, usually implemented as a queue. New item inserted into the list could be store for as many iterations as the size of the queue, usually referred to as **the tabu tenure**.
4. **Aspiration** criteria - Sometimes, we do allow tabu moves for example, when the given move allows a new global best solution.

**TSP with Genetic Algorithm**

1. State representation
   1. Suppose we have *N* cities. Genome representation of a solution is an *N*-dimensional array permutation of integers from 0 to *N*-1.
   2. Fitness value is attached to a state as the total cost of the path.
2. Parameters involved in the Genetic Algorithm (GA) of the TSP:
   1. Generation size: number of genomes in the population
   2. Genome size: length of the genome (in the working example, the starting city is purposely separated from the genome. Therefore, the genome size is equal to the number of cities minus 1.
   3. Reproduction size: number of genomes which will be selected to reproduce to make the next generation
   4. Maximum iteration: maximum number of generation that the algorithm will evolve before terminating
   5. Mutation rate: frequency of mutations when creating a new generation.
   6. Travel prices: matrix of the prices of travel between two cities – 0 on the diagonal and symmetric (assuming symmetrical bi-directional cost)
   7. Starting city: index of the starting city
   8. Target fitness: the fitness the best genome has to reach according to the objective function for the program to terminate early.
   9. Selection type: either *tournament* or *roulette*:
      1. *Roulette*: The selection probability is proportional to the fitness of the genome
      2. *Tournament*: A few individuals are selected and the fittest genome will be selected for cross-over
   10. Tournament size: size of the tournament for tournament selection
3. Steps involved in Genetic Algorithm (GA):
   1. Initialization of population
   2. Selection of genomes from the population for cross-over with one of the two selection schemes: tournament or roulette selection
   3. Cross-over:
      1. Because each genome is a permutation of the list of cities, we cannot crossover two parents conventionally:
         1. Two parents (4 1 2 | 5 6 3) and (1 3 4 | 2 5 6)
         2. If we cross over at the vertical dash, we get (4 1 2 | 2 6 3) and (1 3 4 | 5 5 6) as children, which violates the permutation.
         3. Solve this issue with **Partially Mapped Crossover (PMX)**
   4. Mutation:
      1. If a genome passes the probability check, randomly swap two genes in the genome.
      2. Otherwise, return the original genome (no mutation).
   5. Generate next children generation
   6. Termination when:
      1. Fitness of below/above the target fitness for minimization/maximization problem
      2. When maximum iteration is exceeded, return the last least/most fit genome found.
4. Partially Mapped Crossover (PMX):
   1. Randomly select a crossover point
   2. Swap i-th element of one of the parents with the element equivalent to value of the i-th element of the other
   3. Example:
   4. P1: ( 2 4 3 | 1 6 5), P2: (4 6 5 | 3 1 2)
      1. For child 1, C1:
         1. 1st element P1 is 2, that corresponds to 4 in P2
         2. Swap 2 and 4 in P1 to get P1 = (4 2 3 | 1 6 5)
         3. 2nd element in P1 is now 2, that corresponds to 6 in P2
         4. Swap 2 and 6 in P1 to get P1 = (4 6 3 | 1 2 5)
         5. 3rd element in P1 is not 3, that corresponds to 5 in P2
         6. Swap 3 and 5 in P1 to get P1 = (4 6 5 | 1 2 3)
      2. For child 2. C2:
         1. 1st element in P2 is 4, that corresponds to 2 in P1
         2. Swap 4 and 2 in P2 to get P2 = (2 6 5 | 3 1 4)
         3. 2nd element in P2 is 6, that corresponds to 4 in P1
         4. Swap 6 and 4 in P2 to get P2 = (2 4 5 | 3 1 6)
         5. 3rd element in P2 is 5, that corresponds to 3 in P1
         6. Swap 5 and 3 in P2 to get P2 = (2 4 3 | 5 1 6)
5. Information of the search space:
   1. Travel cost matrix that relates the travelling cost of two connected cities
6. Extra Java knowledge:
   1. A class can implement *Comparable* in Java to make any two class objects comparable. To use the class, one needs to override the class method *compareTo*.
   2. Use *StringBuilder* library to build string that consists of a few continuous chunks in a more organized and sequential manner.
   3. *enum* is a special ‘class’ that represents a group of constants

**Swarm Intelligence**

1. It is a collective behaviour of de-centralized, self-organized systems (“emergence”). Independent entities (‘agents’) behaviour following very simple rules can produce something quite complex.
2. Agents interact with each other locally and the environment. Agent on its own is not intelligent, but the population of agents as a whole is intelligent.
3. Example: Conway’s game of life
4. Examples of swamp intelligence algorithm:
   1. Particle Swarm Optimization
   2. Ant Colony Optimization
5. Particle Swarm Optimization (PSO)
   1. Optimizes a problem by iteratively improve a candidate solution based on a measure of quality
   2. Steps in the algorithm:
      1. Create artificial particles
      2. Move them around in the search space according to simple mathematical formulae that describe the particles’ position and velocity
      3. Each particle’s movement is influenced by its local best known position
      4. Each particle is guided towards the best known positions in the search space, which are updated as better positions are found by other particles
      5. This is expected to move the swam towards the best solutions
   3. Advantage:
      1. Do not need to evaluate the gradient of the measure
      2. The problem needs not be differentiable
   4. Algorithms:
      1. For every particle *i = 1, 2, 3,……S*, initialize the position x\_i and velocity v\_i for every particle.
      2. Also initialize the best known position of every particle p\_i
      3. While (before termination condition):
         1. For every particle i
            1. For every dimension d

Generate random numbers r\_p and r\_g

Update the velocity

V\_id = w\*v\_id + c1\*r\_p(p\_id – x\_id) + c2\*r\_g\*(g\_d – x\_id)

Update position

X\_i = x\_i + v\_i

If f(x\_i) < f(p\_i), p\_i = x\_i

If f(p\_i) < f(g), g = p\_i

* + - * 1. W: inertia weight – defines the memory of the system, how significant is the last step
        2. C1: define the tendency to return to the particle’s best previous solution “attractor”
        3. C2: global weights – defines the tendency to move towards the neighbourhood’s best previous position
        4. If the inertia weight is big, then the solution tends towards a global search. Otherwise, local search is more dominant