Lecture Note - Heuristic and Metaheuristics

Lan Peng, Ph.D.

School of Management, Shanghai University, Shanghai, China

"Puzzle."

1 Heuristic-Search Procedures

The word "heuristic" stands for "I found it" in Greek. The heuristic algorithm are the procedures that we use some information available about the problem to

- reduce the search space
- or to speed up the search
- but not guarantee to find the optimum.

though there are some shortcomings about heuristic, still, we use heuristic because

- we need to avoid combinatorial explosion, because we need the solution FAST.
- we don't need the "optimal" solution, good approximation will be sufficient.
- The approximations may not be very good in the worst case, but in reality worst cases occur very rarely.
- Trying to understand why a heuristic works/does not work leads to a better understanding of the problem.

1.1 Hill-Climbing: An Irrevocable Strategy

Hill-climbing is a local search algorithm used for optimization problems. It starts with an initial solution and iteratively makes small improvements by selecting a neighboring solution that improves the objective function. The process continues until no better neighboring solutions can be found, often resulting in a local optimum. It is simple and efficient but can get stuck in suboptimal solutions due to its greedy nature.

The generic algorithm is as follows

Algorithm 1 Hill-climbing

```
1: $S \leftarrow$ an initial solution
2: while not stopFlag do
3:     $R \leftarrow$ Neighbor(S.clone())
4:     if OFV($R$) is better than OFV($S$) then
5:         $S \leftarrow R$
6: return $S$
```

We can make this algorithm a little more aggressive: create n neighbors to a candidate solution all at one time, and then possibly adopt the best one. This modified algorithm is called Steepest

Ascent Hill-Climbing , because by sampling all around the original candidate solution and then picking the best, we're essentially sampling the gradient and marching straight up it.

The generic algorithm is as follows

Algorithm 2 Steep Ascent Hill-climbing

```
1: $n \leftarrow$ number of neighbors
2: $S \leftarrow$ an initial solution
3: while not stopFlag do
4:     $R \leftarrow$ Neighbor(S.clone())
5:     for $n - 1$ times do
6:         $W \leftarrow$ Neighbor(S.clone())
7:         if OFV($W$) is better than OFV($R$) then
8:             $R \leftarrow W$
9:     if OFV($R$) is better than OFV($S$) then
10:        $S \leftarrow R$
11: return $S$
```

There are several weakness of the Hill-climbing algorithm:

- They usually terminate at solutions that are only locally optimal.
- There is no information as to the amount by which the discovered local optimum deviates from the global optimum, or perhaps even other local optima.
- The optimum that's obtained depends on the initial configuration.
- In general, it is not possible to provide an upper bound for the computation time.

1.2 Uninformed Search

Uninformed search, also known as blind search, refers to a class of search algorithms that do not use any domain-specific knowledge or heuristic information to guide the search process. These algorithms explore the search space systematically without any additional information about the goal or the structure of the problem beyond the problem definition itself.

1.2.1 Depth-First and Backtracking: LIFO Search Strategies

Depth-First Search (DFS) starts at a selected node (often the root in trees or any node in graphs) and explores as far as possible along each branch before backtracking.

DFS uses a stack (either explicitly or through recursion) to keep track of nodes to visit. It is useful for tasks like solving mazes, topological sorting, and detecting cycles in graphs. However, it may not find the shortest path in unweighted graphs and can get stuck in infinite loops in graphs with cycles if not properly managed.

1.2.2 Breadth-First: A FIFO Search Strategy

Breadth-First Search (BFS) explores all nodes at the present depth level before moving on to nodes at the next depth level.

BFS is useful for finding the shortest path in unweighted graphs, level-order traversal in trees, and solving puzzles like the shortest path in a maze. It ensures that all nodes at a particular depth are explored before moving deeper, making it effective for shortest-path problems. However, it can be memory-intensive due to the queue storing all nodes at the current depth level.

1.2.3 Iterative Deepening Search

Iterative-Deepening Search combines BFS and DFS in an interesting way. IDS starts with a depth limit of 0 and increment it as long as it has not found a goal node. For every depth limit, IDS performs DFS up to the depth limit. This means that, if the current node is at the depth limit, IDS will not generate and add its successors to the frontier — essentially, IDS backtracks when it reaches the depth limit. Every time we increase the depth limit, IDS starts the depth-first search all over again.

IDS is an interesting hybrid of BFS and DFS. It is similar to DFS since it performs DFS for each depth limit. IDS is similar to BFS since it explores the search graph level by level by increasing the depth limit by one every time.

For space complexity, IDS performs DFS for every depth limit. Since IDS increases the depth limit by 1 each time, it will terminate at depth d, the depth of the shallowest goal node. The maximum length of the current path is d and each node on the path has at most b siblings.

Thus, the space complexity is $O(bd)$. This is linear in d, the depth of the shallowest goal node. The space complexity is similar to DFS.

In the worst case, IDS will visit all the nodes in the top d levels. Thus, IDS's time complexity is similar to that of BFS. The number of nodes up to depth d is dominated by the number of nodes at depth d, which is b^d. Thus, the time complexity is $O(b^d)$. This is exponential in d, the depth of the shallowest goal node.

Some pros about IDS:

- Similar to DFS, IDS requires linear space only.
- Similar to BFS, IDS is complete and is guaranteed to find the shallowest goal node.
- IDS also has the same time complexity as BFS, although the exact number of nodes visited by IDS is larger than that of BFS.

1.3 Informed Search

Informed search, also known as heuristic search, refers to a class of search algorithms that use problem-specific knowledge or heuristic information to guide the search process. Unlike uninformed search algorithms, which explore the search space blindly, informed search algorithms leverage additional information (such as an estimate of the cost to reach the goal) to make more intelligent decisions about which paths to explore.

For each node, we define the following notations for current node n.

- $h(n)$ as the estimated cost from the current node n to the destination.
- $g(n)$ as the known (exact) actual cost from the start node to current node n
- $f(n) = g(n) + h(n)$ is the heuristic knowledge. Think of $f(n)$ as an estimate of the cost of the cheapest path from the start state to a goal state through the current state n.

1.3.1 A* search

A* search explores the most promising next node, whereas depth first search goes as deep as possible in an arbitrary pattern and breadth first search explores all the nodes on one level before moving to the next. A* search uses a heuristic that provides a merit value for each node, whereas depth-first search and breadth-first search do not.

Algorithm 3 A* Search

```
1: Initialize priority queue Open with $s$
2: Initialize set Close as empty
3: Initialize $f(s) = 0$
4: while Open $\neq \emptyset$ do
5:     $cur \leftarrow$ node in Open with the lowest $f(n)$
6:     if $cur == goal$ then
7:         return path found from $s$ to $cur$
8:     Open $\leftarrow$ Open $\setminus \{cur\}$
9:     Close $\leftarrow$ Close $\cup \{cur\}$
10:    for each neighbor $n$ of $cur$ do
11:        if $n$ not in Close then
12:            $g'(n) \leftarrow g(cur) + cost(cur, n)$
13:            if $n$ not in Open or $g'(n) < g(n)$ then
14:                $g(n) \leftarrow g'(n)$
15:                $f(n) \leftarrow g(n) + h(n)$
16:                if $n$ not in Open then
17:                    Add $n$ to Open
```

1.3.2 Uniform Cost Search

A special case of the A* search, where $h(n) = 0$, use the cost only. Technically this is not an informed search.

1.3.3 Best-first search

Another special case of the A* search, where $g(n) = 0$, use the heuristic only.

2 Single-State Metaheuristic Methods

2.1 Simulated Annealing

Simulated Annealing is a probabilistic optimization technique inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to reduce defects. SA explores the solution space by accepting worse solutions with a certain probability, allowing it to escape local optima.

Simulated Annealing varies from Hill-Climbing in its decision of when to replace the original solution S with a newly found neighborhood R. Specially, if R is better than S, we always replace S with R as usual. But if R is worse than S, we may still replace S with R with a certain probability $P(t, R, S)$:

$$P(t,R,S) = e^{\frac{\texttt{OFV(R)} - \texttt{OFV(S)}}{t}}, t > 0 \tag{1}$$

The generic algorithm for Simulated Annealing is as follows

Algorithm 4 Simulated Annealing

```
1: $t \leftarrow$ initial temperature
2: $S \leftarrow$ initial candidate solution
3: $Best \leftarrow S$
4: while not stopFlag do
5:     $R \leftarrow$ Neighbor(S.clone())
6:     if OFV(R) better than OFV(S) then
7:         $S \leftarrow R$
8:     else if Rnd() $< e^{\frac{\texttt{OFV(R)} - \texttt{OFV(S)}}{t}}$ then
9:         $S \leftarrow R$
10:    Decrease $t$
11:    if OFV(S) better than OFV(Best) then
12:        $Best \leftarrow S$
13: return $Best$
```

2.2 Tabu Search

Tabu Search is a way of using memory in exchange of runtime. It keeps around a history of "recently" considered candidate solutions (known as a tabu list) and refuses to return to those candidates until they are sufficiently far in the past. Thus if we wander up a hill, we have no choice but to wander back down the other side because we are not permitted to stay at or return to the top of the hill. The key to Tabu Search is to maintain the tabu list with some maximum length.

The generic algorithm for Tabu Search is as follows

2.3 Iterated Local Search

Iterated Local Search (ILS) tries to search through the space of local optima in a more intelligent fashion: it tries to stochastically hill-climb in the space of local optima. That is, ILS finds a local optimum, then looks for a "nearby" local optimum and possibly adopts that one instead, then finds a new "nearby" local optimum, and continue. The heuristic here is that you can often find a better local optima near to the current one, and walking from local optimum to local optimum in this way often outperforms just trying new locations entirely random.

There are two tricks in ILS, first, ILS does not pick a new restart location entirely at random, it maintains a "home base" local optimal and selects new restart locations that are near to the "home base" but not too close. We want the restart to be far away enough from current home base to wind up in a new local optimum, but not too far to be total random restart.

Second, when ILS discovers a new local optimum, it decides whether to retain the current "home base" local optimum, or to adopt the new local optimum as the "home base". There are two extremes on the spectrum, if we always accept the new local optimum no matter what, we are doing a random walk, if we always accept the local optimum when it is better than our current one, we are doing hill-climbing. ILS is something in-between those two extremes.

The generic algorithm for ILS is as follows

Algorithm 5 Tabu Search

```
1: $l \leftarrow$ desired maximum tabu list length
2: $n \leftarrow$ number of new neighbor generate from current solution
3: $S \leftarrow$ an initial solution
4: $Best \leftarrow S$
5: $L \leftarrow \{\}$ as a tabu list of length $l$
6: $L.enqueue(S)$
7: while not stopFlag do
8:     while $L.length > l$ do
9:         $L.pop()$
10:    $R \leftarrow$ Neighbor(S.clone())
11:    for $n$ times do
12:        $W \leftarrow$ Neighbor(S.clone())
13:        if $W \notin L$ and OFV($W$) better than OFV($R$) or $R \in L$ then
14:            $R \leftarrow W$
15:    if $R \notin L$ then
16:        $S \leftarrow R$
17:        $L.enqueue(R)$
18:    if S better than Best then
19:        $Best \leftarrow S$
20: return $Best$
```

Algorithm 6 Iterated Local Search with Random Restarts

```
1: $T \leftarrow$ possible time intervals
2: $S \leftarrow$ an initial solution
3: Update "home base" $H \leftarrow S$
4: $Best \leftarrow S$
5: while not stopFlag do
6:     $t \leftarrow$ random time in the near future, sample from $T$
7:     while not run out to time $t$ do
8:         $R \leftarrow$ Neighbor(S.clone())
9:         if OFV($R$) better than OFV($S$) then
10:            $S \leftarrow R$
11:        if OFV($S$) better than OFV($Best$) then
12:            $Best \leftarrow S$
13:        $H \leftarrow$ New "home base" according to $S$
14:        $S \leftarrow$ Perturb($H$)
15: return Best
```

2.4 Greedy Randomized Adaptive Search Procedures

The Greedy Randomized Adaptive Search Procedures (GRASP) is a component-oriented method, designed for certain kinds of spaces consist of combinations of components drawn from a typically fixed set. It's the presence of this fixed set that we can take advantage of in a greedy, local fashion by maintaining historical "quality" values of individual components rather than complete solutions.

GRASP is built on the notions of constructing and searching neighbors for feasible solutions, but it does not use any notion of component-level "historical quality". The overall algorithm is as follows: we create a feasible solution by constructing from among highest value (lowest cost) components and the do hill-climbing on the solution.

Algorithm 7 Greedy Randomized Adaptive Search Procedures (GRASP)

```
1: $C \leftarrow \{C_1, \cdots, C_n\}$ as a set of components
2: $p \leftarrow$ percentage of components to include each iteration
3: $m \leftarrow$ length of time to do hill-climbing
4: $Best \leftarrow$None
5: while not stopFlag do
6:     $S \leftarrow \{\}$ as candidate solution
7:     while $S$ is not a feasible solution do
8:         $C' \leftarrow C_i \in C \setminus S$ which could be added to $S$ without being infeasible
9:         if $C' = \emptyset$ then
10:            $S \leftarrow \{\}$
11:        else
12:            $C'' \leftarrow$ the $p\%$ highest value components in $C'$
13:            $S \leftarrow S \cup \{$component chosen uniformly at random from $C''\}$
14:    for $m$ times do
15:        $R \leftarrow$ Neighbor(S.clone())                    ▷ $R$ need to remain feasible
16:        if OFV($R$) better than OFV($S$) then
17:            $S \leftarrow R$
18:    if $B ==$None or OFV($S$) better than OFV($Best$) then
19:        $Best \leftarrow S$
20: return $Best$
```

3 Population-based Metaheuristic Methods

3.1 Evolution Strategies

The family of evolution algorithms are based on a simple procedure for selecting individuals called truncation selection, and only use mutation as neighborhood searching operator. The simplest implementation is the (μ, λ) algorithm. We begin with a population of λ number of randomly generated individuals. Each iteration we access the fitness of each individual and remove all but the μ fittest ones. Each of the μ fittest individuals get to produce descendants to repopulate the group until the number of individuals recover to λ number.

The (μ, λ) algorithm has three knobs with which we may adjust exploration versus exploitation.

- The size of λ. This essentially controls the sample size for each population, and is basically the same thing as the n variable in Steepest-Ascent Hill Climbing With Replacement. At the extreme, as λ approaches ∞, the algorithm approaches exploration (random search).

Algorithm 8 The (μ, λ) Evolution Strategy

```
1: $\mu \leftarrow$ number of parents selected
2: $\lambda \leftarrow$ number of children generated by the parents
3: $P \leftarrow \{\}$
4: while $P.size < \lambda$ do
5:     $P \leftarrow P \cup \{$new random individual$\}$
6: $Best \leftarrow$ the best individual among $P$
7: while not stopFlag do
8:     for Each $P_i \in P$ do
9:         if OFV($P_i$) better than OFV(Best) then
10:            $Best \leftarrow P_i$
11:    $Q \leftarrow \mu$ individuals in $P$ whose $OFV()$ are better
12:    $P \leftarrow \{\}$                          ▷ For $(\mu + \lambda)$, change it to $P \leftarrow Q$
13:    for Each $Q_i \in Q$ do
14:        for $\frac{\lambda}{\mu}$ times do
15:            $P \leftarrow P \cup \{$ Mutate(Q.clone()) $\}$
16: return $Best$
```

- The size of μ. This controls how selective the algorithm is; low values of μ with respect to λ push the algorithm more towards exploitative search as only the best individuals survive.
- The degree to which Mutation is performed. If Mutate has a lot of noise, then new children fall far from the tree and are fairly random regardless of the selectivity of μ

3.2 Genetic Algorithm

The Genetic Algorithm is similar to the (μ, λ) Evolution Strategy in many respects: it iterates through fitness assessment, selection and breeding, and population reassembly. The primary difference is in how selection and breeding take place: whereas Evolution Strategies select all the parents and then create all the children, the Genetic Algorithm little-by-little selects a few parents and generates a few children until enough children have been created.To breed, we begin with an empty population of children. We then select two parents from the original population, copy them, cross them over with one another, and mutate the results. This forms two children, which we then add to the child population. We repeat this process until the child population is entirely filled.

3.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a stochastic optimization technique somewhat similar to evolutionary algorithms but different in an important way. It's modeled not after evolution per se, but after swarming and flocking behaviors in animals. Unlike other population-based methods, PSO does not resample populations to produce new ones: it has no selection of any kind. Instead, PSO maintains a single static population whose members are Tweaked in response to new discoveries about the space. The method is essentially a form of directed mutation.

The key differential between the GA and PSO are threefold:

- first, in GA, each individual moves in discrete space, each represents a discrete solution, while in PSO, each particle moves in continue space, the location of each particle represents

Algorithm 9 Genetic Algorithm

```
1: $p \leftarrow$ size of population
2: $P \leftarrow \{\}$
3: while $P.size < p$ do
4:     $P \leftarrow P \cup \{$new random individual$\}$
5: $Best \leftarrow$ the best individual among $P$
6: while not stopFlag do
7:     for Each $P_i \in P$ do
8:         if OFV($P_i$) better than OFV(Best) then
9:             $Best \leftarrow P_i$
10:    $Q \leftarrow \{\}$
11:    for $p/2$ times do
12:        Select a Parent $P_a \leftarrow$ Select(P)
13:        Select another parent $P_b \leftarrow$ Select(P)
14:        Breed children $C_a, C_b \leftarrow$ Crossover($P_a$, $P_b$)
15:        $Q \leftarrow Q \cup \{C_a, C_b\}$
16:    $P \leftarrow Q$
17: return $Best$
```

a continuum option.

- second, in GA, individuals will be removed due to selection, but in PSO, the particles never die since there is no selection
- third, GA improves the population mainly by crossover, however, PSO improves the population by mutation.

3.4 Ant Colony Optimization

ACO is population-oriented. But there are two different kinds of "populations" in ACO. First, there is the set of components that make up a candidate solutions to the problem. In the Knapsack problem, this set would consist of all the blocks. In the TSP, it would consist of all the edges. The set of components never changes: but we will adjust the "fitness" (called the pheromone) of the various components in the population as time goes on.

Each generation we build one or more candidate solutions, called ant trails in ACO parlance, by selecting components one by one based, in part, on their pheromones. This constitutes the second "population" in ACO: the collection of trails. Then we assess the fitness of each trail. For each trail, each of the components in that trail is then updated based on that fitness: a bit of the trail's fitness is rolled into each component's pheromone.

4 Tuning the evolutionary algorithms

Almost every practical heuristic search algorithm is controlled by some set of parameters. E.g.,

- Genetic algorithm: population size, cross over rate, mutation rate, runtime limit, etc.
- Tabu search: tabu list length, tabu age, number of neighborhoods, runtime limit, etc.

Algorithm 10 Particle Swarm Optimization

```
 1: $s \leftarrow$ size of swarm
 2: $\alpha \leftarrow$ proportion of velocity to be retained
 3: $\beta \leftarrow$ proportion of personal best to be retained
 4: $\delta \leftarrow$ proportion of global best to be retained
 5: $\epsilon \leftarrow$ jump size of a particle
 6: $P \leftarrow \{\}$
 7: while $P.size < s$ do
 8:     $P \leftarrow P \cup \{$new random individual $(\mathbf{x}, \mathbf{v})\}$
 9: $Best \leftarrow$ the best individual among $P$
10: while not stopFlag do
11:     for Each $P_i \in P$ do
12:         if OFV(P$_i$) better than OFV(Best) then
13:             $Best \leftarrow P_i$
14:     for $\mathbf{x_i} \in P$ with velocity $\mathbf{v_i}$ do
15:         $\mathbf{x}^* \leftarrow$ location of previous best solution
16:         $\mathbf{x_i^+} \leftarrow$ location of $i$'s previous best location
17:         Update speed $\mathbf{x_i} \leftarrow \mathbf{x_i} + \epsilon(\alpha\mathbf{v_i} + \beta(\mathbf{x}^* - \mathbf{x_i}) + \delta(\mathbf{x^+}_i - \mathbf{x}_i))$
18: return $Best$
```

Algorithm 11 Ant Colony Optimization (ACO)

```
 1: $C \leftarrow \{C_1, \cdots, C_n\}$ as a set of components
 2: $p \leftarrow$ size of population
 3: $\mathbf{p} \leftarrow (p_1, p_2, \cdots, p)$ as the pheromones of the components, initially 0
 4: $Best \leftarrow$ None
 5: while not stopFlag do
 6:     $P \leftarrow p$ trails built by iteratively selecting components based on pheromones and costs or
        values
 7:     for $P_i \in P$ do
 8:         $P_i \leftarrow$ HillClimb($P_i$)
 9:         if $B ==$None or OFV($S$) better than OFV($Best$) then
10:             $Best \leftarrow S$
11:     Update $\mathbf{p}$ the pheromones for each $P_i \in P$
12: return $Best$
```

- Simulated annealing: initial temperature, length staying at a temperature, accept rate, cooling rate, runtime limit, etc.
- ⋯

These parameters are applied to

- the representation
- the evaluation function
- the neighborhood searching operators
- the population size
- the stopping criteria
- ⋯

Since there are too many combinations of deciding the parameters, the trial-and-error method becomes very tedious and time-consuming, we need a theory to guide how to set some of the parameters of evolutionary algorithms.

In general, there are two types of parameter settings:

- Parameter tuning: offline parameter search - before the GA is running
  - Parameter sweep (try everything)
  - Meta-GA ontop of GA
  - Racing strategy to find best parameters
- Parameter control: online parameter search - during the GA is running
  - Deterministic genetic operators
  - Adaptive probabilities for mutation and crossover
  - Self-adaptive population size