ECE457A Notes

June 15, 2019

1 Introduction

1.1 Intelligence

- **Intelligence**: The ability to acquire and apply knowledge and skills.
- **Artificial Intelligence**: The science of creating intelligent machines, including intelligent computer programs.

1.2 Rational Thinking & Rational Behavior

- **Rational System**: A logical system that optimizes a given set of criteria.
- **Rational Thinking**: A logical system that achieves goals via logical inferencing.
- **Rational Behavior**: A logical system that perceives its environment and acts to achieve goals according to some set of beliefs.

1.3 Agents

- **Agent**: Senses its environment and acts on collected information.
- **Rational Agent**: An agent that acts in a way that is expected to maximize performance on the basis of perceptual history and built-in knowledge.

1.4 Types of Agents

- **Simple Reflex Agents**: Follow a lookup-table approach; needs fully observable environment.
- **Model-Based Reflex Agents**: Add state information to handle partially observable environments.
- **Goal-Based Agents**: Add concept of goals to help choose actions.
- **Utility-Based Agents**: Add utility to decide "good" or "bad" when face with conflicting goals.
- **Learning Agents**: Add ability to learn from experience to improve performance.

1.5 Environments

- **Fully vs. Partially Observable**:
  - *Fully Observable*: Sensors can detect all aspects relevant to the choice of an action.
  - *Partially Observable*: Missing Information or Inaccurate Sensors.

- **Deterministic vs. Stochastic**:
  - *Deterministic*: Environments that are only influenced by their current state and the next action executed by the agent.
  - *Stochastic*: Randomness/Noise.
- **Episodic vs. Sequential**:
  - *Episodic*: The choice of an action in each episode does not depend on previous episodes.
  - *Sequential*: An agent is required to "think ahead".
- **Static vs. Dynamic**: N/A.
- **Discrete vs. Continuous**: N/A.
- **Single Agent vs. Multi Agent**: N/A.

1.6 Cooperative and Adaptive Algorithms

- **Cooperative**: Solve Joint Problems.
- **Adaptive**: Change Behavior While Running.

2 Search Problem Formulation

2.1 Characteristics of Search Problems

- Large, Non-Polynomial Search Space Size
- Large, Non-Polynomial Constraints Size

2.2 Well-Structured vs. Ill-Structured Problems

- **Well-Structured Problems**: Problems for which the existing state and desired state are clearly identified, and the methods to reach the desired state are fairly obvious.
- **Ill-Structured Problems**: Situation in which its existing state and the desired state are unclear and, hence, methods of reaching the desired state cannot be found.
  1. Start & Improve Guess
  2. Search Alternatives
  3. Forward Search from Problem to Answer
  4. Backward Search from Goal to Problem Situation

2.3 Optimization Problems

- **Optimization Problem**: Finding the best solutions from a set of solutions subject to a set of constraints.

2.4 Types of Optimization Algorithms

- **Exact Algorithms**:
  - Find Optimal Solution
  - High Computational Cost
- **Approximate Algorithms**:

  - Find Near-Optimal Solution
  - Low Computational Cost

2.5 Approximate Algorithms

- **Heuristics**: A solution strategy or rules by trial and error to produce acceptable (optimal or sub-optimal) solutions to complex problems in a reasonably practical time.
- **Constructive Methods**: A solution is constructed by iteratively introducing a new component.
- **Local Search Methods**: An initial solution is improved by iteratively applying actions.

2.6 Goal and Problem Formulation

- *Requirements for Search: Goal Formulation + Problem Formulation*
- *Closed World Assumption: All necessary information about a problem domain is available in each percept so that each state is a complete description of the world.*

2.7 Problem Formulation Template

- **State Space**: Complete/Partial Configuration of Problem
  - *Required*: Each State = **UNIQUE**
- **Initial State**: Beginning Search State
- **Goal State**: Ending Search State
- **Action Set**: Set of Possible State Transitions
- **Cost**: Comparison Function between Solutions

2.8 Extra Terminologies

- **State**: Any Possible Agent/Problem Configuration
- **Transition Model**: Action Description

3 Graph Search Algorithms

3.1 Search Tree Terminology

- **Node**: Search Problem State
- **Edge**: Search Problem Action
- **Fringe**: Frontier/Leaves of Search Tree
- **Branching Factor (*b*)**: The maximum number of child nodes extending from a parent node.
- **Maximum Depth (*m*)**: The number of edges in the shortest path from the root node to the furthest leaf node.
- **Optimal Goal Depth (*d*)**: the number of edges in the shortest path from the root node to an optimal goal node.

3.2 Properties of Search Algorithms

- **Completeness**: Guarantee Find A Goal Node
- **Optimality**: Guarantee Find Best Goal Node
- **Time Complexity**: # of Nodes Generated
- **Space Complexity**: # of Nodes Stored

3.3 Generic Search

- *Fringe = Queue-Like Data Structure*

1. Choose Node
2. Test Node
3. Expand Node

3.4 Local Search Strategies

- **Uninformed Strategies**: No knowledge of the direction of goal nodes.
  - Breadth-First
  - Depth-First
  - Depth-Limited
  - Uniform-Cost
  - Depth-First Iterative Deepening
  - Bidirectional
- **Informed Strategies**: Domain knowledge of the direction of goal nodes.
  - Hill Climbing
  - Best-First
  - Greedy Search
  - Beam Search
  - A
  - A*

4 Uninformed Search Strategies

4.1 Breadth-First Search

- Expand the shallowest unexpanded nodes, storing the fringe to be expanded in a FIFO queue.

4.2 Properties of Breadth-First Search

- *Completeness*:
  - Yes - If b is Finite
- *Optimality*:
  - Yes - If Cost = Depth

- *Time Complexity*: $O(b^{d+1}) \approx O(b^d)$
- *Space Complexity*: $O(b^{d+1}) \approx O(b^d)$

4.3 Uniform Cost Search

- Expand the lowest cost unexpanded node, storing the fringe to be expanded in a minimum priority queue.
- **Required**: *No Zero/Negative-Cost Edges*

4.4 Properties of Uniform Cost Search

- Let C^* be the path cost to the goal.
- Let ϵ be the minimum cost of all other actions.
- *Completeness*:
  - Yes - If b is Finite
- *Optimality*: Yes
- *Time Complexity*: $O(b^{\frac{C^*}{\epsilon}+1}) \approx O(b^d)$
- *Space Complexity*: $O(b^{\frac{C^*}{\epsilon}+1}) \approx O(b^d)$

4.5 Depth-First Search

- Expand the deepest unexpanded nodes, storing the fringe to be expanded in a LIFO stack.

4.6 Properties of Depth-First Search

- *Completeness*:
  - No - If Search Space with Infinite-Depth/Loops
- *Optimality*: No
- *Time Complexity*: $O(b^m)$
- *Space Complexity*: $O(bm)$

4.7 Depth-Limited Search

- Execute DFS with a maximum search depth as a restriction.
  - Prevents Infinite-Depth Problem
  - Prevents Loops Problem

4.8 Properties of Depth-Limited Search

- Let l be the maximum search depth.
- *Completeness*:
  - Yes - If Solution's Depth $d \leq l$
- *Optimality*: No
- *Time Complexity*: $O(b^l)$
- *Space Complexity*: $O(bl)$

4.9 Iterative Deepening Search

- Iteratively, execute DLS with an increasing maximum search depth l until a solution is found.

4.10 Properties of Iterative Deepening Search

- *Completeness*:
  - Yes - If b is Finite
- *Optimality*:
  - Yes - If Cost = Depth
- *Time Complexity*: $O(b^d)$
- *Space Complexity*: $O(bd)$

4.11 Breadth-First vs. Depth-First Strategies

4.12 Breadth-First Strategies

- High Memory Requirement
- Never Stuck on Infinite Depths
- Find Shortest Path to Goal

4.13 Depth-First Strategies

- Low Memory Requirement
- Stuck on Infinite Depths
- Find Any Path to Goal

4.14 Avoiding Repeated States

- *Increasing Computational Costs*:

1. Do not return to the state your just came from.
2. Do not create paths with cycles in them.
3. Do not generate any state that was ever created before.

5 Informed Search Strategies

5.1 Overview

- Apply domain knowledge in a problem to search the "most promising" branches first.
- Potentially, find solutions faster or cheaper than uninformed search algorithms.

5.2 Heuristics

- A **heuristic function** $h(n)$ can be used to estimate the "goodness" of node n.
  - $\forall n, h(n) \geq 0$
  - $h(n) = 0 \implies n$ is a goal node.

  - $h(n) = \infty \implies n$ is a dead end that does not lead to a goal.

- **Admissible/Optimistic**: If a heuristic function never overestimates the cost of reaching the goal.

5.3 Strong vs. Weak Methods

- **Strong Methods**: *Specific Approach to Some Problems*
- **Weak Methods**: *General Approach to Many Problems*

5.4 Examples of Weak Methods

- **Mean-End Analysis**: A strategy where a representation is formed for the current and goal state, and actions are analyzed that shrink the difference between the two.
- **Space Splitting**: A strategy where possible solutions to a problem are listed, and then classes of these solutions are ruled out to shrink the search space.
- **Subgoaling**: A strategy where a large problem is split into independent smaller ones.

5.5 Best-First Search/Greedy Search

- *~Uniform Cost Search with Priority Queue*
  - Minimize $f(n) \mapsto h(n)$
  - Greedy If $f(n) = h(n)$

5.6 Properties of Best-First Search/Greedy Search

- *Completeness*:
  - No - Stuck in Loops
- *Optimality*:
  - No
- *Time Complexity*: $O(b^m)$
- *Space Complexity*: $O(b^m)$

5.7 Beam Search

- *~Breadth-First Search + Reduced Memory Requirements*
  - Expands Best β (*Beam Width*) Nodes Per Level

5.8 Properties of Beam Search

- *Admissible*:
  - No
- *Completeness*:
  - No

- *Optimality*:
  - No
- *Time Complexity*: $O(\beta b)$
- *Space Complexity*: $O(\beta b)$

5.9 A Search/A* Search

- *A Search: Best-First Search with $f(n) = g(n) + h(n)$*
  - $g(n)$ is the cost from the start to n.
  - $h(n)$ is the cost from n to the goal.
- *A* Search: Constraint $h(n) \leq h^*(n)$*
  - $h^*(n)$ is the actual minimal path cost from n to the goal.

5.10 Properties of A Search

- *Admissible*:
  - No
- *Completeness*:
  - No - If $h(n) \to \infty$
- *Optimality*:
  - No

5.11 Properties of A* Search

- *Admissible*:
  - Yes - If $h(n) \leq h^*(n)$
- *Completeness*:
  - Yes - If b is finite and only fixed positive costs.
- *Optimality*:
  - Yes

5.12 Hill Climbing Search

- *Improvement of Depth-First Search*
- *~Beam Search with $\beta = 1$*
- *~Greedy Search with No Backtracking*
- *Not Complete at Local Minima, Plateaus, Ridges*

1. Start with an arbitrary solution.
2. Attempt to improve the solution by changing a single element at a time.
3. Sort the successors of a node according to their heuristic values, and then adding them to the list to be expanded.
4. Make changes until no further improvements can be found.

5.13 Rule of Hill Climbing Search

- If there is a successor s for node n such that:
  - $h(s) < h(n)$ and
  - $h(s) \leq h(t)$ for all successors t of n.
- True $\implies$ Advance from n to s.
- False $\implies$ Halt at n.

5.14 Heuristics

- **Perfect Heuristic**: If $h(n) = h^*(n)$, then only nodes on the optimal solution are expanded.
- **Null Heuristic**: If $h(n) = 0$, then A&ast behaves like uniform cost search.
- **Better Heuristic**: If $h_1(n) < h_2(n) < h^*(n)$, then h_2 is a better heuristic than h_1.

6 Game Playing as Search

6.1 Overview

- Games involve playing against an opponent, where search problems involve finding a good move, waiting for an opponent's response, and then repeating.
- Time is typically limited in each search.

6.2 Problem Formulation of Games

- **Initial State**: *Initial Position + Whose Move It Is*
- **Operators**: *Legal Player Moves*
- **Goal (Terminal Test)**: *Is Game Over?*
- **Utility (Payoff)**: *Measures Outcome/Desirability*

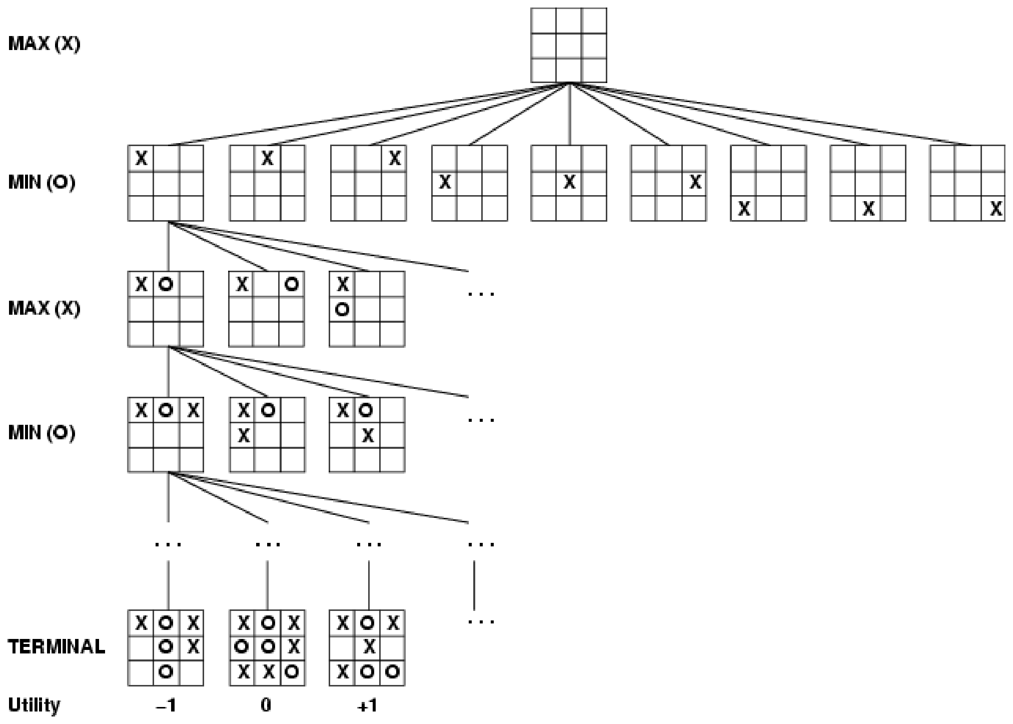
6.3 Types of Games

- **Perfect Information**: Each player has complete information on the opponent's state and available choices.
- **Imperfect Information**: Each player does not have complete information on the opponent's state and available choices.

6.4 Max Min Strategy

- With perfect information and two players, a game tree can be expanded to describe all possible moves of the player and the opponent in the game.
- **Zero Sum Games**: *Player Win* $\implies$ *Opponent Loss*
- **Minimax Principle**: *Minimize the maximum losses that occur.*

6.5 Minimax Algorithm



Example of Minimax Algorithm

- **Important Note**: *Bottom-Up*

1. Generate the game tree labeling each level with alternating MAX($player$) and MIN($opponent$) labels.
2. Apply the utility function to each terminal state (leaf) to get its minimax value.
3. Extrapolate these minimax values to determine the utility of the nodes on level higher in the search tree.
    - For a MAX($player$) level, select the maximum minimax value of its successors.
    - For a MIN($opponent$) level, select the minimum minimax value of its successors.
4. From the root node, select the move which leads to the highest minimax value.

6.6 Limited Depth

- For complicated games, a limited depth of the game tree should be explored.
- An **evaluation function** $f(n)$ is used to measure the "goodness" of a game state.

6.7 Properties of Minimax Algorithm

- *Completeness*:
    - Yes - If game tree is finite
- *Optimality*:

    - Yes - If opponent is optimal

- *Time Complexity*: $O(b^d)$
- *Space Complexity*: $O(bd)$

6.8 α-β Pruning

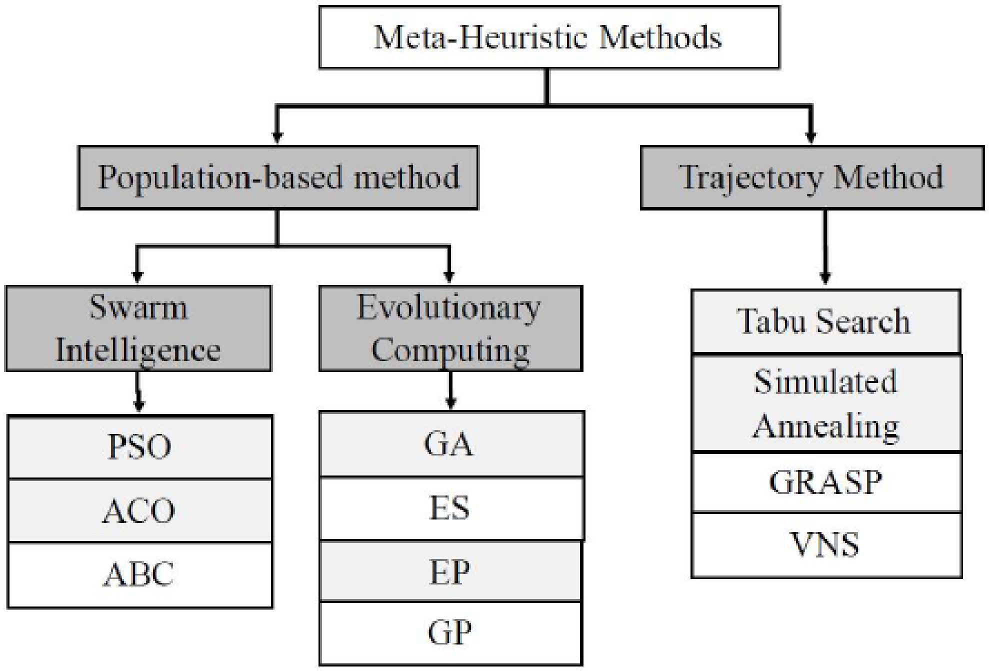
- *Branch and Bound: Reduce # of Generated/Evaluated Nodes*
    - *Avoid Processing Subtrees* $\neq$ *Affecting Result*
- **Alpha (α)**: The best value for MAX seen so far.
    - Used in MIN nodes
    - Assigned in MAX nodes
    - Never Decreases
- **Beta (β)**: The best value for MIN seen so far.
    - Used in MAX nodes
    - Assigned in MIN nodes
    - Never Increases
- **Alpha Cutoff** (*Lower Bound*): When the value of a minimum position is less than or equal to the alpha-value of its parent, stop generating further successors.
- **Beta Cutoff** (*Upper Bound*): When the value of a maximum position is greater than the beta-value of its parent, stop generating further successors.

6.9 α-β Minimax Algorithm Revisions

1. Search discontinued below any MIN with $\beta \leq \alpha$ of one of its ancestors.
    - Set final value of the node to be this β value.
2. Search discontinued below any MAX with $\alpha \geq \beta$ of one of its ancestors.
    - Set final value of the node to be this α value.

7 Metaheuristics

7.1 Overview



Overview of Metaheuristic Methods

- **Metaheuristics**: High-level heuristics designed to select other heuristics to solve a problem by exploring and exploiting its search space.
  - *Approximate Solutions*
  - *Nondeterministic Solutions*

7.2 Properties

- Mechanisms to avoid getting trapped in confined areas of the search space.
- Not problem-specific; may use domain-specific knowledge from heuristics controlled by upper-level strategy.
- Search history to guide the search.
- Hybrid search models where the search identifies neighborhoods where a goal may lie, and then the search is intensified in that area.

7.3 Population-Based Methods

- Population-based methods are metaheuristic approaches that apply multiple agents to a search space and can handle multiple simultaneous solutions.

7.4 Trajectory Methods

- Trajectory methods are metaheuristic variants of local search that apply memory structure to avoid getting stuck at local minima, and implement an explorative strategy that tries to avoid revisiting nodes.

8 Simulated Annealing

8.1 Physical Annealing Analogy

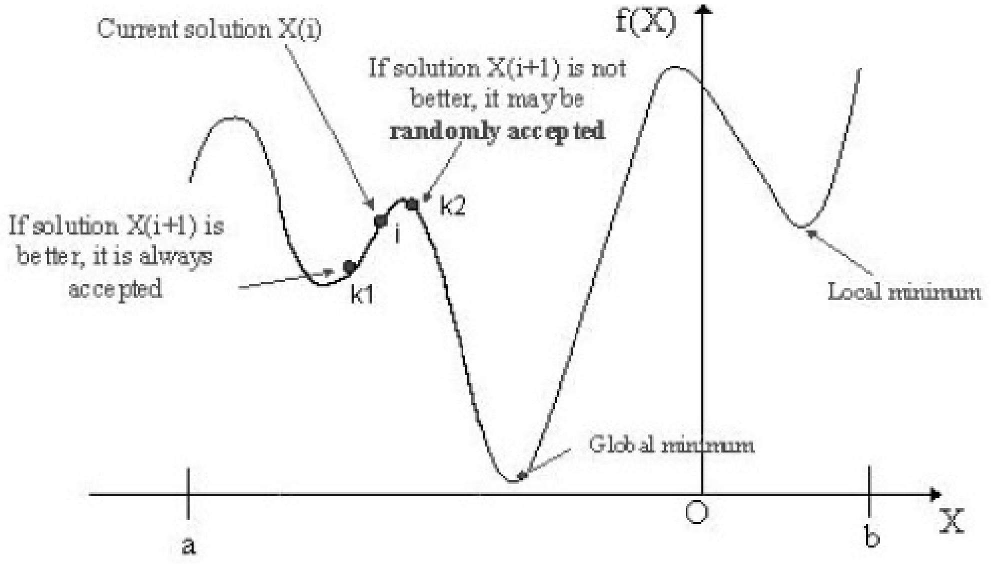
- Physical annealing involves heating a substance (e.g. a metal) and then letting it cool to increase its ductility and reduce hardness.
- The goal is to make the molecules in a cooled substances arrange themselves in a low-energy structure, and the properties of this structure are inuenced by the temperatures reached and the rate of cooling.
- A sequence of cooling times and temperatures is referred to as an annealing or cooling schedule.

8.2 Simulated Annealing Algorithm

- Let $s = s_0$ be a current solution initialized to s_0.
- Let $t = t_0$ be a current temperature initialized to t_0.
- Let α be a temperature reduction function.

```
1. Repeat,
    1. Repeat,
        1. Select a solution $s_i$ from the neighborhood $N(s)$.
        2. Calculate the change in cost $\Delta C$.
        3. If $\Delta C < 0$, then accept the new solution: $s = s_i$.
        4. Else, generate a random number $x \in (0, 1)$.
        5. If $x < \exp(\frac{-\Delta C}{t})$, then accept the new solution: $s = s_i$.
    2. Until maximum number of iteration for $t$.
    3. Decrease $t$ using $\alpha$.
```

8.3 Strategy


Current solution X(i)
If solution X(i+1) is not better, it may be **randomly accepted**
f(X)
If solution X(i+1) is better, it is always accepted
k1
i
k2
Local minimum
Global minimum
a
O
b
X

Simulated Annealing Strategy

- Simulated annealing always accepts better solutions.
- Simulated annealing randomly accepts worse solutions.
- At higher temperatures, explore parameter space.
- At lower temperatures, restrict exploration.

$$\text{Low Temperature} \wedge \text{High Change in Cost} \implies \text{Low Acceptance Probability}$$

8.4 Annealing Schedule

- **Annealing Schedule**: *Adjusts Temperature*
  - *Initial Temperature*
  - *Final Temperature*
  - *Temperature Decrement Rule*
  - *Temperature Iterations*

8.5 Initial Temperature

- The initial temperature should be high enough to allow exploration to any part of the search space.
- If the initial temperature is too hot, simulated annealing would behave too randomly.
- The maximum change of a cost function should be considered when setting the initial temperature.
- **General Rule**: Set the initial temperature to accept around 60% of worse solutions.

8.6 Final Temperature

- The final temperature should be quite low but not neccessarily have to reach zero.
- A search using simulated annealing can be stopped once no better moves are being found and no worse moves are being accepted.

8.7 Temperature Decrement Rule

- **Linear**: $t = t - \alpha$
- **Geometric**: $t = t \times alpha$
- **Slow Decrease**: $t = \frac{t}{1+\beta t}$

8.8 Temperature Iterations

- Enough iterations should be allowed at every temperature for the system to be stable at that temperature.
- If the search space is very large, a large number of iterations may be required.
- If the slow decrease rule is used, one iteration per temperature should be used.

8.9 Convergence

- Simulated annealing is guaranteed to eventually converge to a solution at a constant temperature, assuming some sequence of moves leads to the goal state.
- When temperature is not constant, convergence can still be guaranteed but only under conditions that result in very slow temperature reduction and an exponential increase in the number of iterations at each temperature.

8.10 Advantages and Disadvantages

8.11 Advantages

- Easy
- Widely Applicable

8.12 Disadvantages

- Time Complexity
- Many Tunable Parameters

8.13 Adaptation

- **Adaptation** refers to adapting the critical parameters of the simulated annealing algorithm.

8.14 Initial Temperature

- Finding the right temperature is very problem-specific, and different search algorithms can be applied in finding this temperature.

8.15 Cooling Schedule

- Some cooling schedules require that only the cooling rate α is specified, and the remainder of parameters are automatically determined using a linear random combination of previously accepted states and parameters to estimate new steps and parameters.

8.16 Probability of Acceptance

- Some attempted adaptations concerning the probability of acceptance include using a lookup table for the relevant calculations (to decrease computation time) or using a different, non-exponential probability formula.

8.17 Cost Function

- Cost functions that return similar values for many different states tend to not lead to effective search.
- As an alternative, a cost function can have a penalty term associated with certain types of states, and weighting of these penalty terms can vary dynamically.

8.18 Cooperation

- Cooperative simulated annealing involves multiple concurrent runs of a simulated annealing search algorithm on a search space.
- Potential solutions are produced by somehow combining the value of a run with the value of a random previous run of the algorithm.
- This exchange of information from other solutions is known as cooperative transition, and is a concept borrowed from genetic algorithms.

9 Tabu Search

9.1 Overview

- **Tabu Search**: A general trajectory-based metaheuristic strategy for controlling inner heuristics.
  - *Combination of Local Search Strategy + Search Experience Model* $\Longrightarrow$ *Escape Local Minima + Explorative Strategy*

9.2 Local Search Strategy

1. Start with an initial feasible solution.
2. Repeat,
   1. Generate a neighboring solution by applying a series of local modifications.
   2. If the new solution is better, then replace the current solution.

9.3 Local Search Strategy Challenges

1. It can be costly to consider all possible local modifications.
2. It can get stuck in local optimum.

9.4 Basic Ideas

- Penalize moves that take a solution back to a previously visited (tabu) state.
- Accepts non-improving solutions in order to escape from local optima and eventually find a better solution.

9.5 Use of Memory

- **Short-Term Memory** (*Recency of Occurrence*): Prevent the search from revisiting recently visited solutions.
  - **Tabu List**: A short-term memory structure that stores recent moves applied to the current solutions or their attributes.
  - **Tabu Tenure**: The number of iterations T for which a certain move or its attributes are kept in the list.
  - Complete solutions are rarely used because of the space requirement.
- **Long-Term Memory** (*Frequency of Occurrence*): Prevent the search from revisiting frequently visited solutions.

9.6 Neighborhood

- When selecting a new state, consider neighbors that are not on the tabu list.

$$N(s) - T(s)$$

- The neighborhood structure $N(s)$ can be reduced and modified based on history and knowledge.

9.7 Termination Conditions

- *Sample Conditions*:
  - No feasible solution in the neighborhood of current solution.
  - Reached the maximum number of iterations allowed.
  - The number of iterations since the last improvement is larger than a specified number.
  - Evidence shows that an optimum solution has been obtained.

9.8 Candidates and Aspiration

- A **candidate list** stores the potential solutions in a neighborhood to be examined.
  - Isolate regions of a neighborhood with desirable features, aiming to find a solution more efficiently.
  - At times, it can be desirable to include a move in a candidate list even if it is tabu in order to prevent **stagnation**.
  - **Aspiration Criteria**: *Approaches for Canceling Tabus*

9.9 Tabu Search Algorithm

- Let $s = s_0$ be a current solution initialized to s_0.
- Let $N(s)$ be the neighborhood of the current solution.
- Let $T(s)$ be the tabu list of the current solution.
- Let $A(s)$ be the aspiration list of the current solution.

1. Repeat,
    1. Select the best solution s' from $N^*(s) = N(s) - T(s) + A(s)$.
    2. Memorize s' if it is improves the best known solution.
    3. Let $s = s'$.
    4. Update $T(s)$ and $A(s)$
2. Until termination criteria are met.

9.10 Selecting Tabu Restrictions

- *Sample Restrictions*:
    - Not picking a move that involves the same exchange of positions of a tabu move.
    - Not picking a move that results in positions that previously appeared in a tabu move.

9.11 Selecting Tabu Tenure

- *Sample Strategies*:
    - Statically assigning T to be a constant: $\sqrt{n}$ where n is the problem size.
    - Dynamically letting T vary between a T_{min} and a T_{max}.
        * *Advantages: Better Limits Cycles*
        * *Disadvantages: May Contain Cycles Longer Than Tabu Tenure*

9.12 Selecting Aspiration Criteria

- *Sample Strategies*:
    - By default, where a tabu move becomes admissible if it yields a better solution than any found so far.
    - By objective, where a tabu move becomes admissible if it yield a solution better than an aspiration value.
    - By search direction, where a tabu move becomes admissible if the direction of the search remains constant.

9.13 Intensification

- **Intensification**: The process of exploiting a small portion of the search space (e.g. penalizing solutions far from the current solution).
- *General Idea*: Locally optimize a best known solution while trying to preserve the general components of that solution (based on short-term memory).

9.14 Diversification

- **Diversification**: The process of forcing the search into unexplored areas (e.g. penalizing solutions close to the current solution).
- *General Problem*: Tabu search can miss some good solutions in unexplored search space areas.

9.15 Diversification Strategies

- **Restart Diversification**: Where components rarely appearing in solutions are forced into new solutions.
- **Continuous Diversification**: Where the evaluation of possible moves is biased by a term related to component frequency.

9.16 Adaptation

- **Adaptation** refers to a series of techniques for varying the tabu tenure.
  - If the tabu tenure is too small, then cycles in the search are likely.
  - If the tabu tenure is too big, then many moves could be prevented at each iteration.

9.17 Adaptation Techniques

- Randomly select a new tenure from a pre-computed range every predetermined number of iterations.
- Set the tenure to one if a best-so-far solution is found, decreasing it in an improving phase, and increasing it in a worsening phase.

9.18 Cooperation

- Cooperative tabu search involves multiple concurrent runs of a tabu search algorithm on a search space.
- **Synchronous Communication**: Search agents exchange information every fixed number of iterations.
- **Asynchronous Communication**: Search agents relay their best-so-far results to a central memory.
- **Forced Diversification**: A search agent can replace its own best-so-far solution with an incoming solution.
- **Conditional Import**: A seach agent can replace its own best-so-far solution with an incoming solution only if the incoming solution is better.

9.19 Observations on Tabu Search Cooperation

- Increasing the number of search agents improves the solution up to a certain point.
- Increasing the number of synchronization messages increases the computation time due to message passing overhead.
- Conditional imports are almost always preferable to forced diversification.